

Introduction to Software Security

Chapter 3.1: Pointers and Strings

Loren Kohnfelder
loren.kohnfelder@gmail.com

Elisa Heymann
elisa@cs.wisc.edu

Barton P. Miller
bart@cs.wisc.edu

Revision 1.1, February 2019.

Objectives (3)

- Understand the security problems associated with pointers and strings.
- Understand the causes of those problems.
- Learn how to mitigate security problems associated to pointers and strings.

Buffer Overflow vulnerabilities (4,5)

The classic buffer overflow is perhaps the best known category of security flaw. The first significant attack exploiting a buffer overflow (the Morris worm) occurred in the late 1980s, but still today it is hardly a solved problem and very much still an ongoing concern that continues to expose systems to real harm.

A buffer overflow bug is code that permits accessing locations of a buffer (typically an array or string) outside of its allocated boundaries. Writing outside of a buffer results in unintended modification of memory. Since many languages manage memory in a heap or stack, these define two subtypes of buffer overflow with stack allocations often being more predictable and hence providing better reproducibility for attackers. Reading outside of a buffer is also a threat in that it may inadvertently disclose information.

Buffer overflow problems arise in C style languages that provide programmers with pointers to allocated memory, entrusting the programmer to write code that reliably stays within the allocated bounds. In hindsight, it is an extremely optimistic approach to what has proven to be a very tall order that continues to this day to be a daunting problem for code of any complexity. In C/C++ null-terminated strings are an important special case of array allocation and pointer use that is liable to buffer overflows, and the problem is aggravated by the *laissez faire* design of the original UNIX standard string library functions.

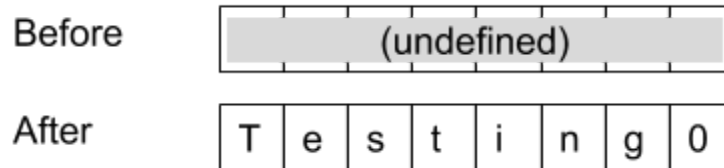
Before getting to the security aspects, here is a quick review of C-style character strings that will help set the stage. Consider the following simple code:

```
01:  char id[8];  
02:  id = "Testing";
```

Line 1 allocates an 8 byte buffer of characters on the stack (recall that local variables are on the stack so that recursive functions work properly, with each call instance getting its own private memory in a new stack frame). Character strings are variable-length ASCII-encoded character sequences and the C run-time

keeps track of the current string length by placing a null-terminator (one byte of zero) following the last character. This is a clever and simple solution, but misuse has led to countless instances of bugs — and often serious security vulnerabilities as we will explore in detail.

The following diagram shows what the memory looks like before and after line 2 executes in the code above. For simplicity, “0” denotes the byte value zero, not the ASCII character code for the digit zero.



In this example, there is no buffer overflow, but you can easily see how it might happen if the string were too long. The seven character “Testing” fits exactly into 8 bytes with the null terminator in the last available byte, but a longer string would overwrite into whatever followed in memory, potentially wreaking havoc. Also, note that before assigning a value to the string, the allocated memory is undefined, so we have no idea what are the contents of this memory: this is a potential read access buffer overflow (unless one of the eight bytes just happened to be zero). Copying the contents of `id` as a string would behave like copying an arbitrary chunk of memory until it happened to get to an incidental null terminator.

Consider these common causes of buffer overflows as patterns to avoid when writing in a language where your code is responsible for staying within the bounds of allocation, as opposed to a language like Java that automatically enforces bounds checking.

- C-style strings are fraught with opportunities to go wrong, either *ad hoc* code manipulation or when using the old (`string.h`) library functions. Note that even when using the new “safer” string functions (such as `strncat`, `strncpy`, etc.), care must be taken to use them properly as explained below.
- Any pointer arithmetic always requires careful bounds checking to ensure safe access.
- Off-by-one errors can result in severe bugs, especially losing the null terminator of a string.
- Similarly, any code that might overwrite a string’s null terminator potentially allows the string to grow to unbounded lengths over arbitrary memory beyond.
- Bounds checking can become more confusing when pointers are recast, such as when you assign a pointer of type `char*` to one of type `int*`. When such recasts are done, it is easy to confuse the length of the data calculated in bytes with the length calculated in integers (probably in units of 4 or 8 bytes).
- When handling multibyte character representations, such as UTF-8, there can easily be confusion between number of bytes and number of characters.
- Beware of code that relies on a fixed-size buffer allocation; this is often based on mistakenly thinking that a large buffer size will always exceed the length of any possible need.
- Any time you pass a pointer without also passing an accompanying variable that contains its allocated length, you increase the risk of failure to enforce memory boundaries.

- Dynamic memory allocation (such as via `malloc`) may return a zero pointer value in case of an error or for allocation size of zero. It is critical to always check for this case since it potentially affords an attacker to access any chosen memory address.

The simplest and most reliable way to ensure buffer boundaries are respected is to always pass a pointer to the beginning of the allocated memory, along with a count of the allocated buffer size. However, sometimes you need to work with pointers that contain an offset into the middle of a buffer. Such pointer use increases the chances that you will incorrectly calculate the bounds of the buffer.

As a rule of thumb, code should never use a pointer to a buffer of unknown allocation size as it is impossible to properly bounds check these accesses. Code can be written with implicit knowledge of the size of the buffer corresponding to a given pointer, but in practice this is difficult to get right, and can be fragile when later modifications are made to the code by someone unfamiliar with the convention.

Buffer overflows often can be harmless or even undetectable in practice so it is easy to have these bugs lurking in your code and fail to notice them for years. Nonetheless, if discovered by a skilled adversary, such flaws might be escalated into significant attacks.

To dive deeper:

To learn about crafting attacks, see the classic paper on how this is done, ["Smashing the Stack for Fun and Profit"](#).

Buffer Overflow of User Data Affecting Flow of Control (6)

To understand the mechanics of a buffer overflow and what it looks like in a program, let's walk through a simple example in C and show how it can result in unauthorized modification of the flow of control. This simplified code reads a user ID from the input and executes a privileged operation if and only if a valid user ID is presented. (Details of the privileged operation and what constitutes a valid user ID are omitted from this example; here we focus only on illustrating the buffer overflow itself.)

The first two lines declare variables `id` and `validId` to be allocated on the stack. The fixed-size declaration of the string `id` can contain up to eight characters, though you have to be careful here to account for the null terminator byte. If you tried to store an eight character user ID in `id`, then you would have its null terminator as the ninth byte, outside the allocated space for the buffer. Note that in our example, writing a single null terminator outside the buffer into the variable `validId` would be harmless (see explanation below); nonetheless, any access outside of a buffer is potentially dangerous and always should be avoided.

```
01:  char id[8];
02:  int validId = 0; /* not valid */
03:  gets(id);      /* reads "evillogin"*/
04:  /* validId is now 110 decimal */
05:  if (IsValid(id)) validId = 1; /* not true */
06:  if (validId)   /* is true */
```

```
07:          DoPrivilegedOp(); /* gets executed */
```

The code begins execution on line 3, where it reads a single line of text input, writing the value as a null terminated string into the variable `id`. Since the user is free to type many more than eight characters worth of input, `gets` will write as much as it takes and easily overwrite the extent of the buffer provided. Note that `gets` is virtually impossible to use safely since no buffer can be allocated in advance that can be guaranteed to be sufficient for an arbitrarily long input line that has yet to be entered. (Note that `gets` has had a long and tarnished record as a security weak point in code, dating back to the first Internet worm that we mention at the start of this module.)

To cause the buffer to overflow we enter the string “evillogin”: the first eight characters go into the variable `id` but `gets` keeps going and the ninth (‘n’ which is 110 decimal in the ASCII character code representation) lands outside the bounds of the variable `id`, which happens to be the first byte of the integer variable `validId`. (In this example we assume a little endian¹ architecture so the value of `validId` changes to become 110.)

Next, at line 5, the code evaluates the newly entered user ID to determine if it is valid. If the input is valid, the code then sets the value of `validId` to one. In this case “evillogin” is too long to be valid, so the function returns false and the assignment of `validId` to one is skipped. The author assumed that skipping would leave it as the initialized zero, or false, value.

On line 6, when the code tests `validId` for the result of the validity check, it now incorrectly sees the nonzero value 110 as indicating a true result and wrongly allows the execution of `DoPrivilegedOp` on line 7. Thus, with the overly long input, the attacker has succeeded in managing to execute privileged code.

Before leaving this code example, it also can be used to understand how a buffer overflow easily can happen without any observable effects. As mentioned above, had the text input instead been “badlogin” (exactly eight characters long), then while the text just fits within the allocated buffer, the null terminator would be written past the bounds. While this violates bounds checking, in this case it would be harmless since typically the memory allocation for `validId` would immediately follow and writing one null terminator (one byte of zero) has no effect as the value is already zero.

This example is based on a buffer overflow of a character buffer (array) that was meant to contain a C-style string. C strings are just one special use of an array. Note that all the same principles apply to any array or malloc (heap allocated) structure as well. The standard C library includes many functions that take pointers but do not perform bounds checking, relying entirely on the calling code to anticipate and prevent possible buffer overflows, or on the presence of a null byte to indicate the limits of valid data. See, for example, almost every function in `string.h`, including `memcpy`, and `memmove`.

¹ Little endian and big endian refer to the order in which the bytes that make up a multibyte integer are stored in memory: storing the bytes starting from the most significant byte to least significant byte (big endian), and storing the bytes from least significant to most significant (little endian). The term “endian” comes from famous book, “Gulliver's Travels”, where it jokingly divides people by whether they crack open their hard boiled eggs from the little end or big end first.

Buffer Overflow Danger Signs: Missing Buffer Size (7)

As we have seen, when code is working with a pointer to a buffer of unknown size it is impossible to properly check the bounds and avoid overflow (which would result in improper access to memory). The original C libraries have several commonly used functions like this that, as a rule, should simply be avoided. While in hindsight it is hard to understand how the brilliant designers of C and UNIX could have gotten this so wrong, remember that this was all designed before the Internet (or its predecessor, ARPANET) existed and that the pioneering use of computers was purely the domain of academics.

The following table shows inherently unsafe standard library functions that use string buffers where the caller has no explicit way of ensuring the buffer is going to be large enough for a given value. The table also includes somewhat safer alternatives to these functions.

Unsafe	Safer
<code>gets(s)</code>	<code>fgets(s, sLen, stdin)</code>
<code>getwd(s)</code>	<code>getcwd(s, sLen)</code>
<code>scanf("%s", s)</code>	<code>scanf("%100s", s)</code>

The safer alternatives shown in the right hand column use parameters that explicitly limit the buffer length (e.g., `sLen`), so accesses will stay within the bounds of the buffer. Of course, these functions are safe only when the correct buffer length is *reliably* provided by the programmer — these functions are by no means guaranteed to be safe. Having a compiler-provided check to ensure that your accesses stay within bounds is a much more reliable mechanism; however, C provides no such facility though other languages such as Java and Python do.

For fixed length buffers, `scanf` provides syntax to specify a maximum length modifier to ensure against possible overflow by long strings. Remember that the buffer must have room for the null terminator in addition to the maximum number of characters specified. Note that if a given `scanf` statement will operate on buffers of different sizes, the constant-specified limit (the “100” in the example in the table above) will not be sufficient. A possible, but awkward solution to this problem is to dynamically create a format-string with the proper length (perhaps using `sprintf`).

There is no standard safer alternative for the function `getpass(s)` — a variant of `gets(s)` that suppresses echoing of input to protect the confidentiality of password inputs — so instead you need to modify the console input mode directly to suppress automatic echoing of keyed input.

`strcat`, `strcpy`, `sprintf`, `vsprintf` — Handle with care (8)

Next, consider the collection of library functions that are possible to use safely, but where it is impossible for the function itself to detect potential buffer overflow; these standard library functions entirely rely on the caller to be vigilant. While these functions can be used safely, each call to the function must get the bounds check right, which, in practice, is extremely difficult to do all of the time.

We urge the use of safer library functions that do the same job in an inherently safer way as shown below. If, for some reason, you need to use the more dangerous functions directly, then code reviews or other disciplines will be necessary to increase the chances that bounds checks always are handled properly.

Alternatively, consider wrapping these functions in your own function that ensures that proper bounds checking always occurs, as illustrated by the following code snippet. Here the destination string `dst` is already allocated with `dstSize` bytes, so the `if` statement checks that there is room for the concatenated result and its null terminator before taking action.

Proper usage: concat s1, s2 into dst	
01:	<code>if (dstSize < strlen(s1) + strlen(s2) + 1)</code>
02:	<code> {ERROR("buffer overflow");}</code>
03:	<code>else {</code>
04:	<code> strcpy(dst, s1);</code>
05:	<code> strcat(dst, s2);</code>
06:	<code>}</code>

The critical bounds check is performed on line 1: the length of the resulting concatenated string will be the combined length of the two strings `s1` and `s2`, plus one more byte for the null terminator. If the allocated destination buffer length (`dstSize` bytes) is too small, then an error function is invoked on line 2. The `else` clause on line 3 is important to ensure that in the error case no attempt is made to construct a result as this would exceed the bounds of the allocated target buffer. Only if the bounds check succeeds does the code proceed to construct the result at lines 4 and 5.

Buffer Overflow Danger Signs: Difficult to Use and Truncation (9)

The “n” versions of the C string functions, for example, `strncpy`, were created so that the programmer could put explicit limits on the memory that they would reference. Even these more safer versions of the string functions need to be used correctly to avoid buffer overflow. These improved versions help with safety but only when used properly, so continued vigilance is required. These versions of the string functions do not eliminate the possibility of trouble, so we describe them as “safer” rather than truly “safe”. Think of them as a child seat: used properly they provide considerable increased safety yet never absolve the driver (programmer) of responsibility.

It’s worth drilling down on exactly how the “safer” string functions help as well as their limitations. The unsafe string functions rely entirely on the calling code to handle all bounds checking: once invoked the code within the function blindly does its work without any way of bounds checking its own accesses. This dependency on finding a null byte in the right position is extremely fragile, and the situation is exacerbated by the fact that there are countless millions of invocations of these common functions. By contrast, with the safer functions, so long as the caller provides an accurate buffer length, the bounds checking inside the function is easy to get right — and it is far more efficient to do it there as well. Thus, we strongly urge the use of the safer form for secure coding, but at the same time it is critical to understand that these are by no means foolproof since correctness depends on the caller providing accurate buffer length information.

strncat(dst, src, n)

The improved safety of `strncat` comes from the guarantee that it will never append more than `n` bytes to the destination `dst`. However, since the function does not know the size of the buffer pointed to by `dst`, `strncat` is still dependent on the caller to use it safely. That is, even with the safer form of this function, it still can overflow if the number of bytes to concatenate exceeds the available buffer size beyond the end of the initial string value — $n \geq (\text{dstSize} - \text{strlen}(\text{dst}))$ — since it will try to append `n` bytes to the destination buffer of length `dstSize`. If the string pointed to by `src` is longer than `n-1` characters, then the result will be truncated but the resulting string always will have a null terminator. Note that the only way to determine if truncation occurred is by checking the available space, such as shown above, *before* invoking the function since afterwards the destination string will be modified.

strncpy(dst, src, n)

Even though the caller to `strncpy` specifies a maximum number of bytes to write, this function has several ways that it is dangerous to use. In compliance with the POSIX standard, `strncpy` copies up to `n` bytes from the buffer pointed to by `src` to the buffer pointed to by `dst`. If the null terminator is reached in the source, then it fills the remaining destination memory with zeros until the full count of `n` is reached. So long as `n` does not exceed the destination buffer size, no overflow occurs. However if the source is too long ($\text{strlen}(\text{src}) \geq n$) then the resulting destination strings will not be null terminated. Even worse, the function does not report this condition back to the caller. Omitting the null terminator can easily result in a read-type buffer overflow when subsequently using the destination string.

++ Also be aware of similar commonly used more precise versions of other standard library functions, including:

Unsafe	Safer
<code>strlen(s)</code>	<code>strnlen(s, sLen)</code>
<code>strcmp(s1, s2)</code>	<code>strncmp(s1, s2, maxLen)</code>
<code>strdup(s)</code>	<code>strndup(s, sLen)</code>
<code>wscpy(dest, src)</code>	<code>wcsncpy(dest, src, maxLen)</code>
<code>wcslen(s)</code>	<code>wcsnlen(s, sLen)</code>

Safer String Handling: C-library functions (10)

`snprintf(buf, bufSize, fmt, ...)` and `vsprintf(buf, bufSize, fmt, ...)`

So long as the correct buffer size is specified, these safer versions of `sprintf` ensure that the bounds will not be exceeded and that the resulting string will be properly null terminated. Truncation can, of course occur, but it is easy to check: the function return value is the full untruncated number of bytes of formatted output produced had there been infinite space. Code can easily check if truncation occurred by

testing if the return value exceeds the `bufSize` parameter value. Note that the equal case means that one byte was truncated to make room for the null terminator that must always be provided.

Since `snprintf` guarantees a properly terminated string result, and truncation is easily detected, it can be used as safer version of `strcpy` and `strcat` as shown below.

Proper usage: concat s1, s2 into dst	
01:	<code> r = snprintf(dst, dstSize, "%s%s",s1, s2);</code>
02:	<code> if (r >= dstSize)</code>
03:	<code> {ERROR("truncation");}</code>

Attacks on Code Pointers (11)

We have looked at buffer overflow flaws using strings and other types of arrays, but both C and C++ also have pointers to functions and other variables that contain addresses of executable code; these code pointers present potential risks.

Some common examples of pointers to functions or addresses in code are listed below. Any buffer overflow allowing modification of any of these instances represents a serious problem.

- `switch/case` statements, which are often implemented as a table of jump addresses
- Return addresses for function invocations, stored on the stack
- Function pointers
- C++ virtual function tables, which are often implemented as a table of function pointers
- Library functions that manage execution context, e.g., `jmp_buf` (C++) and `atexit` (Python)
- Tables of addresses for exception and signal handlers

Stack Smashing is perhaps the best known example where a buffer overflow can modify the return location that a function jumps to upon conclusion.

Here is an example of Stack Smashing in detail. The first two lines declare variables on the stack: a small fixed length buffer `id` that will be subject to buffer overflow, and a pointer to the function `MyLogger`.

01:	<code> char id[8];</code>
02:	<code> int (*logFunc)(char*) = MyLogger;</code>
03:	<code> gets(id); /* reads "/bin/sh\0XXXX" */</code>
04:	<code> /* equivalent to system(id) */</code>
05:	<code> logFunc(id);</code>

As we mentioned before, `gets` accepts arbitrarily long input so it is trivial to provide an overly long input line containing characters of the attacker’s choice. The first eight characters go into the variable `id` as they should, but the rest of the input continues to be written into memory past the bounds of the buffer. In this case, the characters overwrite the `logFunc` function pointer. The “XXXX” on line 3 would actually be

carefully chosen byte values corresponding to the binary address of a system call the attacker wants to invoke instead of the harmless `MyLogger`.

In this example, an extremely effective attack would be if `XXXX` pointed to the `execl` kernel call, where the parameter to the call (based on the string stored in `id`) would cause a shell to be executed. Instead of performing logging as intended, the attacker has managed to invoke a different system function, such as `exec`, then from the shell they can do just about whatever they like from here: game over.

Note that, by design, `gets` will read null bytes from input and place them in the buffer. These null bytes be confused by the program as null terminators for a string. In fact, the authors had to confirm this by trying it out since it is unexpected yet not explicitly documented. One consequence of this behavior is that the buffer overflow may happen even though `strlen` of the result string is less than the allocated buffer size. This is because string length only counts up to the first, possibly embedded null, although `gets` may continue writing more bytes up to newline or EOF.

We are using the deprecated `gets` function for simplicity in this example, but this kind of flaw exist in code using any number of other ways. While the documentation clearly says that use of `gets` should be avoided, so long as it exists for back compatibility, and so long as over-confident programmers ignore documentation, these flaws persist as a real threat.

It is worth noting that operating system designers have made Stack Smashing attacks more difficult by randomly selecting the address of the stack and of each library each time that a program is run. This technique, called address space layout randomization (ASLR), makes it difficult for the attacker to know what addresses to use in their attacks. In addition, modern operating systems include mechanisms to detect buffer overflows on the stack (*Stack Canaries*) and on heap (*Heap Guards*). As a result, it is easier to cause crashes than to cause a specific attack. Of course, attackers have been working on new ways to defeat such defenses, but discussion of these attacks and defenses are out of the scope of this chapter (but will be discussed in more detail in another chapter). In conclusion, ensure that these extra protections are enabled, but you should never rely on them and get lazy about preventing buffer overflows.

To dive deeper:

- Read: [“Bypassing Stack Cookies, SafeSeh, SEHOP, HW DEP and ASLR”](#)
- Read: [“ASLR/DEP bypass whitepaper”](#) (PDF)

Even C# can be unsafe (13)

Having seen the horrors of buffer overflow in C and C++, and with an understanding of the nuances of properly using even the “safer” library functions provided to help avoid the problem, it is tempting to think that simply by using a more modern language such as C#, which has compiler-provided array bounds checking, you can stop worrying about buffer overflows. Unfortunately, this is only partially true.

The C# language includes the `unsafe` keyword, which results in code that does not contain the normal safety checks — that is, code that the compiler and runtime cannot ensure is always going to be safe and

free of buffer overflows. As a safeguard, the compiler does require the `/unsafe` command line flag to accept the `unsafe` keyword.

```
unsafe static void bufferOverflow(string s){
    char* ptr = stackalloc char[10];

    foreach (var c in s) {
        *ptr++ = c; // Buffer overflow if s.Length > 10
    }
}

unsafe static void Main() {
    bufferOverflow("A-long-string");
}
```

Heartbleed: a recent and frightening buffer overflow (14-16)

If you still think buffer overflows are a relic of the past, consider that in 2014 much of the world's secure browsing technology was rendered insecure by a simple buffer overflow. This buffer overflow was the root of the infamous [Heartbleed](#) vulnerability, a great example of how read-access buffer overflows can be devastating; unintended modification is not the only problem.

Transport Layer Security (TLS) is the protocol behind secure browsing HTTPS. This complex cryptographic communication protocol includes something called the Heartbeat Extension that is used to keep long-lasting connections between peers alive over periods of inactivity. In essence, when a heartbeat request message is sent by one peer, the other sends back a corresponding response. The request contains a payload buffer together with a field indicating its length; the response echoes back the payload of the request.

The buffer overflow happened in [OpenSSL](#), the world's most popular TLS implementation, as a result of the request message lying about the length of the payload buffer it sent. On the receiving end, the code assumed that the payload size field value would be accurate and simply copied back that many bytes of the payload in response.

```
memcpy(bp, pl, payload);
```

When the attacker sent a message that contained a short payload but claimed that it was a much longer payload, one of the maximum 64KB size, the other side returned not just the correct payload but also a snapshot of the contents of its memory that happened to lie adjacent to and beyond the buffer. Since the security of TLS relies on each of the peers maintaining some secret keys as private, you do not need to be a cryptographer to imagine how this could easily lead to undermining the security of the entire protocol.

It is worth looking at the actual fix ([git commit](#)), which was quite simple: it just checks that the payload size provided actually matches the payload in the request. Here is a slightly simplified version of the fix:

```
if (1 + 2 + payload + 16 > s->s3->rrec.length)
    return 0; /* silently discard per RFC 6520 sec. 4 */
```

This fix checks to see if the request specifies a payload length in excess of the actual payload provided. If the length is too long, then the request is simply ignored, as permitted by the specification. This fix works and obviously it was critical to repair the code as quickly as possible while much of the world's HTTPS infrastructure was broken and vulnerable to attack.

When fixing a security bug it is imperative to make the code crystal clear so as to ensure against possible regression in the future. While the authors attempted to make this fix clear by expressing the length as an expression, `1 + 2 + payload + 16`, the use of embedded constants, and providing no comments as to their meaning, violates good software engineering practices. Generally speaking, some important details worth reflecting in code in situations like this include:

- Add a comment clearly stating that this is a critical check. Without such a comment, this change to the code looks like a minor detail.
- Document the meaning of mysterious constants like 1, 2, and 16 in the check. In this case they correspond to the fixed size in bytes of the message type (1), payload length (2), and final padding field (16) as specified in the specification. **Even better**, use defined symbolic constants or constant (`const`) variables (depending on the language). Embedding magic numbers in your code is error prone and confusing. Note that the OpenSSL code does include a declared `padding` value, but inexplicably it was not used here.
- Their comment does reference the applicable RFC specification but gives no clue about the important security implications of this check.
- Names should provide clarity of meaning: rather than `'payload'` which sounds like the buffer containing the payload, that variable might better be named `'payloadLength'`.
- Include a reference to [Common Vulnerability Enumeration \(CVE\)](#) name that identifies this particular code flaw in the [National Vulnerability Database \(NVD\)](#). CVE's and the NVD are invaluable resources. For Heartbleed, see [CVE-2014-0160](#).

Even more on buffer overflows

The simple examples we have examined thus far had the bounds check immediately preceding the memory reference or function call that potentially does the out of bounds reference. However, in real code this often is not the case and the out-of-bounds access may be nowhere near the intended bounds check.

Function definitions that require the caller to provide parameters that contain the buffer length along with the pointers are the best way to ensure against buffer overflows. Inclusion of the length allows bounds checking to be done wherever an access occurs. Even better is to define objects or structures that contain both the pointer and length so that all operations on the data have access to the actual buffer bounds.

In legacy code, you often have a pointer without any size information in sight, and then it gets tricky: to fix such code, do you find all callers to the function and add a length parameter? Of course, such a

change potentially requires that you go up the call graph to ensure that bounds checking is somehow happening at the layers above, which can be laborious. One compromise that may work is to enforce an arbitrary length limit in the code at hand and then reject or truncate anything longer that might possibly be passed in. A truncated result or even an unfriendly error condition is almost surely a better prospect than having an open buffer overflow in your code. (Use an approach such as this one for short-term repairs or as a last resort needed to close vulnerabilities. They should be temporary, until a more complete fix can be developed.)

Avoid attempts to infer what the longest possible string or array might be. For example, many UNIX-based systems define a maximum file path name length, `PATH_MAX`, as 4096, but does that mean you can confidently copy any `char *pathname` string value with impunity? Of course not. For starters, code that calls into your function may not know the correct limit or may be passing through values that originated from untrusted inputs. Also, you need to beware of off-by-one errors. `PATH_MAX` includes one byte for the null terminator, so with the 4096 value the string itself may not exceed 4095 characters.

After you have designed your code to taken all the necessary precaution, testing is a critical part of ensuring against buffer overflows. For example, code that handles file pathnames should be tested with paths of `PATH_MAX-1` characters (the longest valid Unix path), `PATH_MAX` (the shortest path that is too long, for those off-by-one errors), and `2*PATH_MAX` or some other excessively long input value. Ensure that a valid error is raised or, if the code should gracefully truncate and keep working, that it does so correctly. [Fuzz random testing](#) is a great way to exercise the limits of code: be sure to configure the test cases to use sufficiently long inputs to trigger buffer overflows and chose values to maximize the likelihood that a crash or other easily detectable error will result. Fuzz testing will be discussed in more detail in Chapter 7.

Summary (17)

Pointers and strings are a common source of buffer overflows that can easily result in security vulnerabilities. For over thirty years, researchers and software engineers have been warning of these problems, but, as long as we continue to use languages that allow unchecked memory accesses, we will need to be extremely vigilant when we write in such languages.

Any out-of-bounds access to memory is playing with fire and must strictly be avoided. Although, in practice, many buffer overflows may appear harmless, attackers can often manipulate them into powerful weapons to modify protected state (write-type data buffer overflow), modify flow of control (overwriting code pointers), or to exfiltrate private information (read-type buffer overflow).

To mitigate these problems, we recommend the use of safer functions, or consider writing your own wrapper functions that reliably ensure all bounds checking is performed. While it is possible to get bounds checking right in every instance of calling a potentially dangerous function, it is far safer to only use functions that ensure against buffer overflow so long as they are provided with accurate inputs.

Secure coding standards have been published that provide complete details about preventing buffer overflows. Where possible use compiler flags (e.g., use canaries and guards, disallow `C# /unsafe`) or scanning tools (e.g., even “grep” for known dangerous functions such as `strcpy` can be quite effective).

- <https://www.securecoding.cert.org/confluence/display/c/SEI+CERT+C+Coding+Standard>
- <https://misra-c.org/>

Exercises

1. Search for instances of some of the unsafe library functions covered in your own C/C++ code, or in an open source code base of your choice. For each of these uses, determine the following:
 - a. Are buffer overflow defenses present in the code, and are they thorough enough to always prevent an overflow?
 - b. Write test cases that try to trigger a buffer overflow or confirm that it cannot happen.
2. Find public fixes to buffer overflows like Heartbleed and study the modifications needed to fix. (For example, see [CVE-2015-1781](#).) For each of these cases:
 - a. What signs in the code might have tipped you off that there was a problem?
 - b. Did the fix resolve the problem entirely?
 - c. Can you provide a concrete argument (or even a proof) that the fix is resilient to the reported attack?
 - d. Can you think of a new attack that overflows the buffer even with the fix applied?
3. Experimentation: Your goal is to write code that overflows a buffer. Using either pointers or array subscripts, write code that intentionally overflows a buffer. Try this experiment in a variety of scenarios, such as:
 - a. Use C or C++, where pointers can be used freely and there is no array bounds checking.
 - b. Use the C++ String or Array classes, that enforce bounds checking.
 - c. Use Java or C#, which have more checking built into the language.
 - d. Use a scripting language, such as Python or Ruby, which have stronger checking built in.
 - e. Use a more recent language, such as Rust or Go.
4. Research: Study an implementation of *Heap Guards* and *Stack Canaries*. How do these mechanisms try to reduce the impact of buffer overflows? What are the limitations of the protection provided? Can you write code that sneaks an attack past the guard or canary?
5. Research: Design and implement an improved version of one or more of the “safer” string functions that addresses some or all of the remaining pitfalls. Can you make it perfectly safe without imposing performance costs? Can you make it fully or nearly compatible with the standard version it replaces?