Chapter 9
Pointers and Strings

Revision 3.2, November 2025.

Objectives

- 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 string

9.1 Buffer Overflow Vulnerabilities

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 writing to locations of memory (typically an array) outside of its allocated boundaries. Similarly, a *buffer overrun* is reading locations 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 languages like C and C++, which provide programmers with pointer type variables, trusting the programmer to write code that stays within the bounds of the memory allocated. In hindsight, it was optimistic to assume that programmers could do this well. It continues to be a serious source of security vulnerabilities even after all these decades.

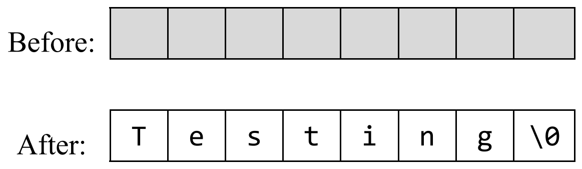
In C/C++ null-terminated strings are an important special case of array allocation and pointer use that is liable to buffer overflows, and the problem is aggravated by the *laissez faire* design of the original UNIX standard string library functions.

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:  strcpy(id, ”Testing”);
```

Line 1 allocates an 8-byte a of array characters on the stack (recall that local variables are on the stack 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 (a zero byte) 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 this chapter.

The following diagram shows what the memory for array `id` looks like before and after line 2 executes in the code above. The gray boxes denotes undefined memory and "\0" denotes the byte value zero, not the ASCII character 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 string "Testing" fits exactly into eight 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. Here, we summarize some of the key problem areas and then expand on them in later sections.

9.1.1 Strings

The use of C-style strings is a frequent source of weaknesses (and often resulting vulnerabilities). Unlike strings in interpreted languages or the use of the String class in C++, the length is encoded in the data; there must be a null byte within the allocated memory to terminate the string.

C-style provide many 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` and `strncpy`), care must be taken to use them properly as explained below.

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.

When handling internationalized multibyte character representations, such as UTF-8[1], there easily can be confusion between number of bytes and number of characters. This situation is made more complex by the fact that UTF-8 character representations varying between one and four bytes.

9.1.2 Pointers

Pointer arithmetic is a favorite tool of experienced C and C++ programmers. *Any* pointer arithmetic requires careful planning, analysis, and bounds checking to ensure safe access.

Memory 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. The confusion can arise when doing pointer arithmetic since incrementing a pointer adds a value to the pointer that is equal to the size the type to which it points. For example, `cp++` where the declaration is "`char *cp`", typically increments the pointer by one. `ip++` where the declaration is "`int *ip`", typically increments the pointer by 4 or 8 (depending on the size of int on your system and compiler).

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. Passing a (buffer pointer, length) pair emulates how strings are implemented in languages (and classes) that protect against overflow/overrun.

9.1.3 Fixed and Variable Memory Allocations

On one hand, you must 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.

On the other hand, dynamic memory allocation also has its hazards. For example, `malloc` may return a zero pointer value in case of an error or for

[1] `https://en.wikipedia.org/wiki/UTF-8`

allocation size of zero. It is critical to always check the return value of `malloc` (and its relatives, such as `calloc` and `realloc`) since this case 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.

In general, 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.

9.2 Buffer Overflow of User Data Affecting Flow of Control

To understand the mechanics of a buffer overflow and what it looks like in a program, we will walk through a simple example in C and show how it can result in unauthorized modification of the flow of control. This code reads a user ID from the input and executes a privileged operation if and only if a valid user ID is presented.

The first two lines declare a character array `id`, which will be used as a character buffer, and `validId`, which is a 32-bit integer initialized to zero. The variable validId is initialized to zero to indicate that a valid user ID has not yet been seen.

Since `id` and `validId` are local variables, they are allocated on the stack. The character array `id` can contain up to eight characters, though the null terminator byte counts as part of that eight, so you effectively can handle seven input characters. If you tried to store an eight character string 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, however 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 */
```

id:								validId:			
								\0	\0	\0	\0

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 `gets` has no way to control the length of input, it will overwrite as much memory as it needs, potentially writing beyond the end of the provided buffer. and easily overwrite the extent of the buffer provided. `gets` is difficult to use safely in any scenario. And, as we have discussed in Chapter 3, `gets` has had a long security mishaps, dating back to the first Internet worm.

In this example, we are going to use the buffer overflow of the buffer to cause the program to do something quite different than intended. We enter the string “evillogin”, where the first eight characters go into the `id` but `gets` keeps going and puts the ninth character (‘n’, which is 110 decimal in the ASCII character representation) outside the bounds of the variable `id`. This ninth byte happens to be the first byte of the integer variable `validId`.

id:								validId:			
e	v	i	l	l	o	g	i	n	\0	\0	\0

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 skips the assignment of `validId` to one. The author assumed that skipping the assignment would leave it as the initialized zero.

At 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, we can also see how a buffer overflow can happen without any observable effects. As mentioned above, had the text input instead been 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 overwriting the zero-initialized variable `validId` with zero has no effect on the program's logic.

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 declared array or heap allocated data structure. 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. Such dangerous functions include almost every function in `string.h`, including `memcpy`#and `memmove`.

9.3 Missing Buffer Sizes

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 lists a few 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**
`gets(s)`	`fgets(s,sLen,stdin)`
`getwd(s)`	`getcwd(s,sLen)`
`scanf("%s",s)`	`scanf("%100s",s)`

The safer alternatives shown in the right hand column use parameters that explicitly limit the buffer length (e.g., `sLen`), so these functions can be used in a way that stays 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 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.

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 it is best to avoid this function altogether.

9.4 strcat, strcpy, sprintf, vsprintf – Handle with care

Next, we examine 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 be used with correct bounds checking, which, in practice, is 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. 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 that performs a safe concatenation operation. Here, the destination string `dst` is already allocated with `dstSize` bytes, so the `if` statement first checks that there is space for the concatenated result and its null terminator before doing the actual concatenation.

Concatenate s1 and s2 into dst:

```
01: if (dstSize < strlen(s1) + strlen(s2) + 1)
02:   {ERROR("buffer overflow");}
03: else {
04:   strcpy(dst, s1);
05:   strcat(dst, s2);
06: }
```

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.

9.5 Trying to Operate on Strings Safely

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 is 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 fragile, and the situation is exacerbated by the fact that they are used in so many places. 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. This case would happen if `n ≥ (dstSize-strlen(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#can specify 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 (strlen(src) ≥ n) then the resulting destination string 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.

Here are a more unsafe functions with their safer alternatives:

Unsafe	Safer
strlen(s)	strnlen(s,sLen)
strcmp(s1,s2)	strncmp(s1,s2,maxLen)
strdup(s)	strndup(s,sLen)
wcscpy(dest,src)	wcsncpy(dest,src,maxLen)
wcslen(s)	wcsnlen(s, sLen)

9.6 Safer String Handling: C-library functions

There are some even better choices when dealing with functions that operate on C style strings. By using these functions properly, you can both avoid overflow and detect truncation.

`snprintf(buf,bufSize,fmt, …)` and `vsnprintf(buf,bufSize,fmt, …)`

So long as the correct buffer size is specified, these safer versions of vsulqwi# 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 a safer version of strcpy and strcat as shown below.

Concatenate s1 and s2 into dst:

```
01: r = snprintf(dst, dstSize,"%s%s",s1,s2);
02: if (r >= dstSize)
03:   {ERROR("truncation");}
```

9.7 Attacks on Code Pointers

Anytime that you use a pointer in C and C++ (which is most of the time), you must be careful. Note that these languages do not just support pointers to data but also to code. Weaknesses based on misusing code pointers are quite powerful and have been frequently exploited in the real world.

The most obvious type of code pointer in C and C++ is when you explicitly declare a pointer to function. Such pointers are commonly used to design code that is general purpose. However, there are several other places where the compiler will generate code that contains pointers to code. Any weakness that allows an overwrite of any of these types of code pointers can cause serious problems.

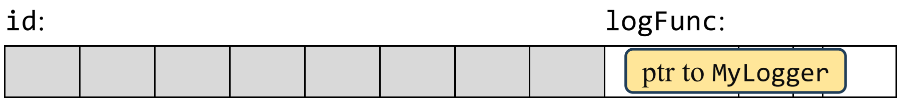
- `switch`/`case` statements are often implemented as a table of jump addresses
- Return addresses for function invocations are stored on the stack
- C++ virtual function tables are often implemented as a table of function pointers
- Library functions that manage execution context, e.g., `jmp_buf` (C++) 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. We saw an example of stack smashing in Chapter 3 when we described the Morris worm.

Here is another example of stack smashing where, in this example, our goal is to overflow a buffer and overwrite an adjacent variable. This overwrite affects the logic of the program in such a way that it gives unauthorized access to the system.

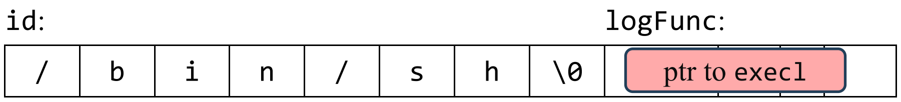
The first two lines declare variables on the stack: a small, fixed length buffer `id` that will be subject to buffer overflow and, adjacent to it, a function pointer `loggingFunc` that points to the function `MyLogger`.

```
01: char id[8];
02: int  (*loggingFunc)(char*) = MyLogger;
03: gets(id);
04: loggingFunc(id);
```



As we mentioned before, the `gets` used on line 3 accepts arbitrarily long input so the attackers can provide an overly long input line containing characters of the attacker's choice. The first eight characters are stored the variable `id` as they should, but the rest of the input continues to be written into memory past the bounds of the buffer `id`. In this case, the characters overwrite the `logFunc` function pointer. If these additional characters contained bytes that formed a valid pointer to code in memory, then the function call on line 4, which calls the function pointed to by loggingFunc, could be redirected anywhere the attacker wished. An attractive possibility is to force to the `execl` system call in the C library, allowing the attacker to control what program is executed.

Noted that function that is called on line 4 takes `id` as a parameter. If the first 8 characters of the attacker's input contained the string "/bin/sh", the program might be forced to start a shell running in its place, giving the attacker arbitrary remote code execution (a very serious kind of exploit).



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. Over the years, the UNIX documentation has gotten increasingly strident on the advice to not use `gets`. For example, the latest gets manual page for Linux says:

> Never use. 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. Use `fgets()` instead.

However, it is puzzling that the designers of Linux still provide this function. It is also annoying that replacing calls to `gets` with calls to `fgets` is not always straightforward. First, `fgets` required an input file specified where the input file in `gets` is implicitly standard input. Second, and perhaps more annoying is that `gets` removes any newline character from the input that is placed in the buffer, however `fgets` removes the newline character.

It is worth noting that operating system and compiler designers have worked to make 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. We describe ASLR in Chapter 26. In addition, modern compilers include mechanisms to detect buffer overflows on the variables. We describe these mechanisms in Chapter 27. As a result, it is easier to cause crashes than to cause a specific attack.

9.8 Even C# Can Be Unsafe

Having seen the problems with buffer overflow in C and C++, 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");
}
```

Any use of the `unsafe` keyword in C# should be a warning flag and such code should be carefully reviewed before being accepted into the code repository. In fact, some organizations will configure the compiler to forbid any use of this feature.

9.9 Heartbleed: A Frightening Buffer Overflow

Buffer overflows and buffer overruns have caused some traumatic global vulnerabilities. One of the most notable vulnerabilities occurred in 2014 when much of the world’s secure internet communication was rendered insecure by a simple buffer overflow. This buffer overflow was the root of the infamous Heartbleed[2] vulnerability, a great example of how a buffer overrun can be devastating; unintended modification is not the only problem.

Transport Layer Security (TLS) is the protocol behind secure socket communication and is used to provide secure browsing via HTTPS. This 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[3], the world's most popular TLS implementation, as a result of the request message lying about the length of the payload buffer that 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, the maximum 64 KB size, the other side returned not just the correct payload but also the contents of its memory that was adjacent to and beyond the buffer. This adjacent memory happened to contain secret keys and digital certificates. The exposure of such private information left the program using TLS open to having its encrypted messages read by the attacker and possibly have false messages injected into the communication stream.

It is worth looking at the actual fix, 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

[2] http://heartbleed.com/

[3] https://www.openssl.org/

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 (`frqvw`) variables (depending on the language). Embedding magic numbers in your code is error prone and confusing.
- In fact, avoid at all costs the use of constants embedded in your code. Such constants create fragile code where the size of the buffer allocation or data structure size is separately specified from its use. Instead, anywhere the numeric value is used, instead used a symbolic value declared something like:

  ```
  const int PayloadLength=2;
  ```

  Note that the OpenSSL code does include a declared `padding` value, but inexplicably it was not used here.
- Include a reference to Common Vulnerability Enumeration (CVE)[4] name that identifies this particular code flaw in the National Vulnerability Database (NVD)[5]. CVE's and the NVD are invaluable resources. For Heartbleed, see CVE-2014-0160[6].

9.10 Even More on Buffer Overflows and Overruns

The simple examples that 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

[4] `https://cve.mitre.org/`
[5] `https://nvd.nist.gov/`
[6] `https://nvd.nist.gov/vuln/detail/CVE-2014-0160`

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, 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 and often difficult to track.

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 buffer sizes of 4096 (or possibly 4097) are always safe? Of course not. For starters, code that calls 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.

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[7] (described in later chapters) 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.

In addition, memory checking tools, as described in Chapter 27, can help you to find many of these types of errors, including those that have no visible effect on the program's behavior.

9.11 Summary

Pointers and strings are a common source of buffer overflows that can easily result in security vulnerabilities. For many decades, 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.

[7] `https://en.wikipedia.org/wiki/Fuzzing`

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., just using the `grep` utility to look for known dangerous functions such as `strcpy` can be quite effective).

9.12 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[8].) 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

[8] `https://nvd.nist.gov/vuln/detail/cve-2015-1781`
`https://sourceware.org/legacy-ml/libc-alpha/2015-04/msg00256.html`

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: 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?