Aside A case of mandatory alignment

For most x86-64 instructions, keeping data aligned improves efficiency, but it does not affect program behavior. On the other hand, some models of Intel and AMD processors will not work correctly with unaligned data for some of the SSE instructions implementing multimedia operations. These instructions operate on 16-byte blocks of data, and the instructions that transfer data between the SSE unit and memory require the memory addresses to be multiples of 16. Any attempt to access memory with an address that does not satisfy this alignment will lead to an *exception* (see Section 8.1), with the default behavior for the program to terminate.

As a result, any compiler and run-time system for an x86-64 processor must ensure that any memory allocated to hold a data structure that may be read from or stored into an SSE register must satisfy a 16-byte alignment. This requirement has the following two consequences:

- The starting address for any block generated by a memory allocation function (`alloca`, `malloc`, `calloc`, or `realloc`) must be a multiple of 16.
- The stack frame for most functions must be aligned on a 16-byte boundary. (This requirement has a number of exceptions.)

More recent versions of x86-64 processors implement the AVX multimedia instructions. In addition to providing a superset of the SSE instructions, processors supporting AVX also do not have a mandatory alignment requirement.

```
  int    g;
  char   *h;
} rec;
```

A. What are the byte offsets of all the fields in the structure?

B. What is the total size of the structure?

C. Rearrange the fields of the structure to minimize wasted space, and then show the byte offsets and total size for the rearranged structure.

3.10 Combining Control and Data in Machine-Level Programs

So far, we have looked separately at how machine-level code implements the control aspects of a program and how it implements different data structures. In this section, we look at ways in which data and control interact with each other. We start by taking a deep look into pointers, one of the most important concepts in the C programming language, but one for which many programmers only have a shallow understanding. We review the use of the symbolic debugger GDB for examining the detailed operation of machine-level programs. Next, we see how understanding machine-level programs enables us to study buffer overflow, an important security vulnerability in many real-world systems. Finally, we examine

how machine-level programs implement cases where the amount of stack storage required by a function can vary from one execution to another.

3.10.1 Understanding Pointers

Pointers are a central feature of the C programming language. They serve as a uniform way to generate references to elements within different data structures. Pointers are a source of confusion for novice programmers, but the underlying concepts are fairly simple. Here we highlight some key principles of pointers and their mapping into machine code.

- *Every pointer has an associated type*. This type indicates what kind of object the pointer points to. Using the following pointer declarations as illustrations

  ```
  int *ip;
  char **cpp;
  ```

  variable `ip` is a pointer to an object of type `int`, while `cpp` is a pointer to an object that itself is a pointer to an object of type `char`. In general, if the object has type T, then the pointer has type $*T$. The special `void *` type represents a generic pointer. For example, the `malloc` function returns a generic pointer, which is converted to a typed pointer via either an explicit cast or by the implicit casting of the assignment operation. Pointer types are not part of machine code; they are an abstraction provided by C to help programmers avoid addressing errors.
- *Every pointer has a value*. This value is an address of some object of the designated type. The special `NULL` (0) value indicates that the pointer does not point anywhere.
- *Pointers are created with the '`&`' operator*. This operator can be applied to any C expression that is categorized as an *lvalue*, meaning an expression that can appear on the left side of an assignment. Examples include variables and the elements of structures, unions, and arrays. We have seen that the machine-code realization of the '`&`' operator often uses the `leaq` instruction to compute the expression value, since this instruction is designed to compute the address of a memory reference.
- *Pointers are dereferenced with the '`*`' operator*. The result is a value having the type associated with the pointer. Dereferencing is implemented by a memory reference, either storing to or retrieving from the specified address.
- *Arrays and pointers are closely related*. The name of an array can be referenced (but not updated) as if it were a pointer variable. Array referencing (e.g., `a[3]`) has the exact same effect as pointer arithmetic and dereferencing (e.g., `*(a+3)`). Both array referencing and pointer arithmetic require scaling the offsets by the object size. When we write an expression `p+i` for pointer `p` with value p, the resulting address is computed as $p + L \cdot i$, where L is the size of the data type associated with `p`.

- *Casting from one type of pointer to another changes its type but not its value.* One effect of casting is to change any scaling of pointer arithmetic. So, for example, if `p` is a pointer of type `char *` having value p, then the expression `(int *) p+7` computes $p + 28$, while `(int *) (p+7)` computes $p + 7$. (Recall that casting has higher precedence than addition.)
- *Pointers can also point to functions.* This provides a powerful capability for storing and passing references to code, which can be invoked in some other part of the program. For example, if we have a function defined by the prototype

```
int fun(int x, int *p);
```

then we can declare and assign a pointer `fp` to this function by the following code sequence:

```
int (*fp)(int, int *);
fp  = fun;
```

We can then invoke the function using this pointer:

```
int y = 1;
int result = fp(3, &y);
```

The value of a function pointer is the address of the first instruction in the machine-code representation of the function.

New to C? Function pointers

The syntax for declaring function pointers is especially difficult for novice programmers to understand. For a declaration such as

```
int (*f)(int*);
```

it helps to read it starting from the inside (starting with '`f`') and working outward. Thus, we see that `f` is a pointer, as indicated by `(*f)`. It is a pointer to a function that has a single `int *` as an argument, as indicated by `(*f)(int*)`. Finally, we see that it is a pointer to a function that takes an `int *` as an argument and returns `int`.

The parentheses around `*f` are required, because otherwise the declaration

```
int *f(int*);
```

would be read as

```
(int *) f(int*);
```

That is, it would be interpreted as a function prototype, declaring a function `f` that has an `int *` as its argument and returns an `int *`.

Kernighan and Ritchie [61, Sect. 5.12] present a helpful tutorial on reading C declarations.

3.10.2 Life in the Real World: Using the GDB Debugger

The GNU debugger GDB provides a number of useful features to support the run-time evaluation and analysis of machine-level programs. With the examples and exercises in this book, we attempt to infer the behavior of a program by just looking at the code. Using GDB, it becomes possible to study the behavior by watching the program in action while having considerable control over its execution.

Figure 3.39 shows examples of some GDB commands that help when working with machine-level x86-64 programs. It is very helpful to first run OBJDUMP to get a disassembled version of the program. Our examples are based on running GDB on the file `prog`, described and disassembled on page 211. We start GDB with the following command line:

```
linux> gdb prog
```

The general scheme is to set breakpoints near points of interest in the program. These can be set to just after the entry of a function or at a program address. When one of the breakpoints is hit during program execution, the program will halt and return control to the user. From a breakpoint, we can examine different registers and memory locations in various formats. We can also single-step the program, running just a few instructions at a time, or we can proceed to the next breakpoint.

As our examples suggest, GDB has an obscure command syntax, but the online help information (invoked within GDB with the `help` command) overcomes this shortcoming. Rather than using the command-line interface to GDB, many programmers prefer using DDD, an extension to GDB that provides a graphical user interface.

3.10.3 Out-of-Bounds Memory References and Buffer Overflow

We have seen that C does not perform any bounds checking for array references, and that local variables are stored on the stack along with state information such as saved register values and return addresses. This combination can lead to serious program errors, where the state stored on the stack gets corrupted by a write to an out-of-bounds array element. When the program then tries to reload the register or execute a `ret` instruction with this corrupted state, things can go seriously wrong.

A particularly common source of state corruption is known as *buffer overflow*. Typically, some character array is allocated on the stack to hold a string, but the size of the string exceeds the space allocated for the array. This is demonstrated by the following program example:

```
/* Implementation of library function gets() */
char *gets(char *s)
{
    int c;
    char *dest = s;
```

Command	Effect
Starting and stopping	
`quit`	Exit GDB
`run`	Run your program (give command-line arguments here)
`kill`	Stop your program
Breakpoints	
`break multstore`	Set breakpoint at entry to function `multstore`
`break *0x400540`	Set breakpoint at address `0x400540`
`delete 1`	Delete breakpoint 1
`delete`	Delete all breakpoints
Execution	
`stepi`	Execute one instruction
`stepi 4`	Execute four instructions
`nexti`	Like `stepi`, but proceed through function calls
`continue`	Resume execution
`finish`	Run until current function returns
Examining code	
`disas`	Disassemble current function
`disas multstore`	Disassemble function `multstore`
`disas 0x400544`	Disassemble function around address `0x400544`
`disas 0x400540, 0x40054d`	Disassemble code within specified address range
`print /x $rip`	Print program counter in hex
Examining data	
`print $rax`	Print contents of `%rax` in decimal
`print /x $rax`	Print contents of `%rax` in hex
`print /t $rax`	Print contents of `%rax` in binary
`print 0x100`	Print decimal representation of `0x100`
`print /x 555`	Print hex representation of 555
`print /x ($rsp+8)`	Print contents of `%rsp` plus 8 in hex
`print *(long *) 0x7fffffffe818`	Print long integer at address `0x7fffffffe818`
`print *(long *) ($rsp+8)`	Print long integer at address `%rsp` + 8
`x/2g 0x7fffffffe818`	Examine two (8-byte) words starting at address `0x7fffffffe818`
`x/20b multstore`	Examine first 20 bytes of function `multstore`
Useful information	
`info frame`	Information about current stack frame
`info registers`	Values of all the registers
`help`	Get information about GDB

Figure 3.39 Example GDB commands. These examples illustrate some of the ways GDB supports debugging of machine-level programs.

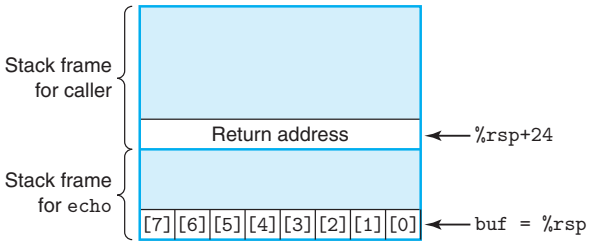


Figure 3.40
Stack organization for `echo` function. Character array `buf` is just part of the saved state. An out-of-bounds write to `buf` can corrupt the program state.

```
    while ((c = getchar()) != '\n' && c != EOF)
        *dest++ = c;
    if (c == EOF && dest == s)
        /* No characters read */
        return NULL;
    *dest++ = '\0'; /* Terminate string */
    return s;
}

/* Read input line and write it back */
void echo()
{
    char buf[8];  /* Way too small! */
    gets(buf);
    puts(buf);
}
```

The preceding code shows an implementation of the library function `gets` to demonstrate a serious problem with this function. It reads a line from the standard input, stopping when either a terminating newline character or some error condition is encountered. It copies this string to the location designated by argument `s` and terminates the string with a null character. We show the use of `gets` in the function `echo`, which simply reads a line from standard input and echos it back to standard output.

The problem with `gets` is that it has no way to determine whether sufficient space has been allocated to hold the entire string. In our `echo` example, we have purposely made the buffer very small—just eight characters long. Any string longer than seven characters will cause an out-of-bounds write.

By examining the assembly code generated by GCC for `echo`, we can infer how the stack is organized:

```
     void echo()
1    echo:
2      subq    $24, %rsp          Allocate 24 bytes on stack
3      movq    %rsp, %rdi         Compute buf as %rsp
4      call    gets               Call gets
5      movq    %rsp, %rdi         Compute buf as %rsp
```

```
6       call    puts                Call puts
7       addq    $24, %rsp           Deallocate stack space
8       ret                         Return
```

Figure 3.40 illustrates the stack organization during the execution of `echo`. The program allocates 24 bytes on the stack by subtracting 24 from the stack pointer (line 2). Character `buf` is positioned at the top of the stack, as can be seen by the fact that `%rsp` is copied to `%rdi` to be used as the argument to the calls to both `gets` and `puts`. The 16 bytes between `buf` and the stored return pointer are not used. As long as the user types at most seven characters, the string returned by `gets` (including the terminating null) will fit within the space allocated for `buf`. A longer string, however, will cause `gets` to overwrite some of the information stored on the stack. As the string gets longer, the following information will get corrupted:

Characters typed	Additional corrupted state
0–7	None
9–23	Unused stack space
24–31	Return address
32+	Saved state in caller

No serious consequence occurs for strings of up to 23 characters, but beyond that, the value of the return pointer, and possibly additional saved state, will be corrupted. If the stored value of the return address is corrupted, then the `ret` instruction (line 8) will cause the program to jump to a totally unexpected location. None of these behaviors would seem possible based on the C code. The impact of out-of-bounds writing to memory by functions such as `gets` can only be understood by studying the program at the machine-code level.

Our code for `echo` is simple but sloppy. A better version involves using the function `fgets`, which includes as an argument a count on the maximum number of bytes to read. Problem 3.71 asks you to write an echo function that can handle an input string of arbitrary length. In general, using `gets` or any function that can overflow storage is considered a bad programming practice. Unfortunately, a number of commonly used library functions, including `strcpy`, `strcat`, and `sprintf`, have the property that they can generate a byte sequence without being given any indication of the size of the destination buffer [97]. Such conditions can lead to vulnerabilities to buffer overflow.

Practice Problem 3.46 (solution page 382)

Figure 3.41 shows a (low-quality) implementation of a function that reads a line from standard input, copies the string to newly allocated storage, and returns a pointer to the result.

Consider the following scenario. Procedure `get_line` is called with the return address equal to `0x400776` and register `%rbx` equal to `0x0123456789ABCDEF`. You type in the string

```
0123456789012345678901234
```

(a) C code

```
/* This is very low-quality code.
   It is intended to illustrate bad programming practices.
   See Practice Problem 3.46. */
char *get_line()
{
    char buf[4];
    char *result;
    gets(buf);
    result = malloc(strlen(buf));
    strcpy(result, buf);
    return result;
}
```

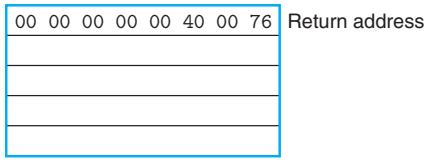
(b) Disassembly up through call to `gets`

```
      char *get_line()
1   0000000000400720 <get_line>:
2     400720:  53                      push   %rbx
3     400721:  48 83 ec 10             sub    $0x10,%rsp
      Diagram stack at this point
4     400725:  48 89 e7                mov    %rsp,%rdi
5     400728:  e8 73 ff ff ff          callq  4006a0 <gets>
      Modify diagram to show stack contents at this point
```

Figure 3.41 C and disassembled code for Practice Problem 3.46.

The program terminates with a segmentation fault. You run GDB and determine that the error occurs during the execution of the `ret` instruction of `get_line`.

A. Fill in the diagram that follows, indicating as much as you can about the stack just after executing the instruction at line 3 in the disassembly. Label the quantities stored on the stack (e.g., "Return address") on the right, and their hexadecimal values (if known) within the box. Each box represents 8 bytes. Indicate the position of `%rsp`. Recall that the ASCII codes for characters 0–9 are 0x30–0x39.



B. Modify your diagram to show the effect of the call to `gets` (line 5).

C. To what address does the program attempt to return?

D. What register(s) have corrupted value(s) when `get_line` returns?

E. Besides the potential for buffer overflow, what two other things are wrong with the code for `get_line`?

A more pernicious use of buffer overflow is to get a program to perform a function that it would otherwise be unwilling to do. This is one of the most common methods to attack the security of a system over a computer network. Typically, the program is fed with a string that contains the byte encoding of some executable code, called the *exploit code*, plus some extra bytes that overwrite the return address with a pointer to the exploit code. The effect of executing the `ret` instruction is then to jump to the exploit code.

In one form of attack, the exploit code then uses a system call to start up a shell program, providing the attacker with a range of operating system functions. In another form, the exploit code performs some otherwise unauthorized task, repairs the damage to the stack, and then executes `ret` a second time, causing an (apparently) normal return to the caller.

As an example, the famous Internet worm of November 1988 used four different ways to gain access to many of the computers across the Internet. One was a buffer overflow attack on the finger daemon `fingerd`, which serves requests by the FINGER command. By invoking FINGER with an appropriate string, the worm could make the daemon at a remote site have a buffer overflow and execute code that gave the worm access to the remote system. Once the worm gained access to a system, it would replicate itself and consume virtually all of the machine's computing resources. As a consequence, hundreds of machines were effectively paralyzed until security experts could determine how to eliminate the worm. The author of the worm was caught and prosecuted. He was sentenced to 3 years probation, 400 hours of community service, and a $10,500 fine. Even to this day, however, people continue to find security leaks in systems that leave them vulnerable to buffer overflow attacks. This highlights the need for careful programming. Any interface to the external environment should be made "bulletproof" so that no behavior by an external agent can cause the system to misbehave.

3.10.4 Thwarting Buffer Overflow Attacks

Buffer overflow attacks have become so pervasive and have caused so many problems with computer systems that modern compilers and operating systems have implemented mechanisms to make it more difficult to mount these attacks and to limit the ways by which an intruder can seize control of a system via a buffer overflow attack. In this section, we will present mechanisms that are provided by recent versions of GCC for Linux.

Stack Randomization

In order to insert exploit code into a system, the attacker needs to inject both the code as well as a pointer to this code as part of the attack string. Generating

Aside Worms and viruses

Both worms and viruses are pieces of code that attempt to spread themselves among computers. As described by Spafford [105], a *worm* is a program that can run by itself and can propagate a fully working version of itself to other machines. A *virus* is a piece of code that adds itself to other programs, including operating systems. It cannot run independently. In the popular press, the term "virus" is used to refer to a variety of different strategies for spreading attacking code among systems, and so you will hear people saying "virus" for what more properly should be called a "worm."

this pointer requires knowing the stack address where the string will be located. Historically, the stack addresses for a program were highly predictable. For all systems running the same combination of program and operating system version, the stack locations were fairly stable across many machines. So, for example, if an attacker could determine the stack addresses used by a common Web server, it could devise an attack that would work on many machines. Using infectious disease as an analogy, many systems were vulnerable to the exact same strain of a virus, a phenomenon often referred to as a *security monoculture* [96].

The idea of *stack randomization* is to make the position of the stack vary from one run of a program to another. Thus, even if many machines are running identical code, they would all be using different stack addresses. This is implemented by allocating a random amount of space between 0 and n bytes on the stack at the start of a program, for example, by using the allocation function `alloca`, which allocates space for a specified number of bytes on the stack. This allocated space is not used by the program, but it causes all subsequent stack locations to vary from one execution of a program to another. The allocation range n needs to be large enough to get sufficient variations in the stack addresses, yet small enough that it does not waste too much space in the program.

The following code shows a simple way to determine a "typical" stack address:

```
int main() {
    long local;
    printf("local at %p\n", &local);
    return 0;
}
```

This code simply prints the address of a local variable in the `main` function. Running the code 10,000 times on a Linux machine in 32-bit mode, the addresses ranged from `0xff7fc59c` to `0xffffd09c`, a range of around 2^{23}. Running in 64-bit mode on the newer machine, the addresses ranged from `0x7fff0001b698` to `0x7ffffffaa4a8`, a range of nearly 2^{32}.

Stack randomization has become standard practice in Linux systems. It is one of a larger class of techniques known as *address-space layout randomization*, or ASLR [99]. With ASLR, different parts of the program, including program code, library code, stack, global variables, and heap data, are loaded into different

regions of memory each time a program is run. That means that a program running on one machine will have very different address mappings than the same program running on other machines. This can thwart some forms of attack.

Overall, however, a persistent attacker can overcome randomization by brute force, repeatedly attempting attacks with different addresses. A common trick is to include a long sequence of `nop` (pronounced "no op," short for "no operation") instructions before the actual exploit code. Executing this instruction has no effect, other than incrementing the program counter to the next instruction. As long as the attacker can guess an address somewhere within this sequence, the program will run through the sequence and then hit the exploit code. The common term for this sequence is a "nop sled" [97], expressing the idea that the program "slides" through the sequence. If we set up a 256-byte nop sled, then the randomization over $n = 2^{23}$ can be cracked by enumerating $2^{15} = 32{,}768$ starting addresses, which is entirely feasible for a determined attacker. For the 64-bit case, trying to enumerate $2^{24} = 16{,}777{,}216$ is a bit more daunting. We can see that stack randomization and other aspects of ASLR can increase the effort required to successfully attack a system, and therefore greatly reduce the rate at which a virus or worm can spread, but it cannot provide a complete safeguard.

Practice Problem 3.47 **(solution page 383)**

Running our stack-checking code 10,000 times on a system running Linux version 2.6.16, we obtained addresses ranging from a minimum of `0xffffb754` to a maximum of `0xffffd754`.

A. What is the approximate range of addresses?

B. If we attempted a buffer overrun with a 128-byte nop sled, about how many attempts would it take to test all starting addresses?

Stack Corruption Detection

A second line of defense is to be able to detect when a stack has been corrupted. We saw in the example of the `echo` function (Figure 3.40) that the corruption typically occurs when the program overruns the bounds of a local buffer. In C, there is no reliable way to prevent writing beyond the bounds of an array. Instead, the program can attempt to detect when such a write has occurred before it can have any harmful effects.

Recent versions of GCC incorporate a mechanism known as a *stack protector* into the generated code to detect buffer overruns. The idea is to store a special *canary* value[4] in the stack frame between any local buffer and the rest of the stack state, as illustrated in Figure 3.42 [26, 97]. This canary value, also referred to as a *guard value*, is generated randomly each time the program runs, and so there is no

4. The term "canary" refers to the historic use of these birds to detect the presence of dangerous gases in coal mines.

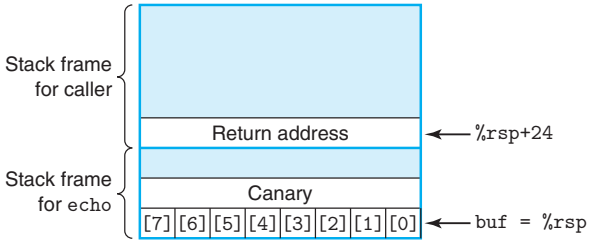


Figure 3.42 Stack organization for `echo` function with stack protector enabled. A special "canary" value is positioned between array `buf` and the saved state. The code checks the canary value to determine whether or not the stack state has been corrupted.

easy way for an attacker to determine what it is. Before restoring the register state and returning from the function, the program checks if the canary has been altered by some operation of this function or one that it has called. If so, the program aborts with an error.

Recent versions of GCC try to determine whether a function is vulnerable to a stack overflow and insert this type of overflow detection automatically. In fact, for our earlier demonstration of stack overflow, we had to give the command-line option `-fno-stack-protector` to prevent GCC from inserting this code. Compiling the function `echo` without this option, and hence with the stack protector enabled, gives the following assembly code:

```
    void echo()
1   echo:
2     subq    $24, %rsp              Allocate 24 bytes on stack
3     movq    %fs:40, %rax           Retrieve canary
4     movq    %rax, 8(%rsp)          Store on stack
5     xorl    %eax, %eax             Zero out register
6     movq    %rsp, %rdi             Compute buf as %rsp
7     call    gets                   Call gets
8     movq    %rsp, %rdi             Compute buf as %rsp
9     call    puts                   Call puts
10    movq    8(%rsp), %rax          Retrieve canary
11    xorq    %fs:40, %rax           Compare to stored value
12    je      .L9                    If =, goto ok
13    call    __stack_chk_fail       Stack corrupted!
14  .L9:                           ok:
15    addq    $24, %rsp              Deallocate stack space
16    ret
```

We see that this version of the function retrieves a value from memory (line 3) and stores it on the stack at offset 8 from `%rsp`, just beyond the region allocated for `buf`. The instruction argument `%fs:40` is an indication that the canary value is read from memory using *segmented addressing*, an addressing mechanism that dates

back to the 80286 and is seldom found in programs running on modern systems. By storing the canary in a special segment, it can be marked as "read only," so that an attacker cannot overwrite the stored canary value. Before restoring the register state and returning, the function compares the value stored at the stack location with the canary value (via the `xorq` instruction on line 11). If the two are identical, the `xorq` instruction will yield zero, and the function will complete in the normal fashion. A nonzero value indicates that the canary on the stack has been modified, and so the code will call an error routine.

Stack protection does a good job of preventing a buffer overflow attack from corrupting state stored on the program stack. It incurs only a small performance penalty, especially because GCC only inserts it when there is a local buffer of type `char` in the function. Of course, there are other ways to corrupt the state of an executing program, but reducing the vulnerability of the stack thwarts many common attack strategies.

Practice Problem 3.48 (solution page 383)

The functions `intlen`, `len`, and `iptoa` provide a very convoluted way to compute the number of decimal digits required to represent an integer. We will use this as a way to study some aspects of the GCC stack protector facility.

```
int len(char *s) {
    return strlen(s);
}

void iptoa(char *s, long *p) {
    long val = *p;
    sprintf(s, "%ld", val);
}

int intlen(long x) {
    long v;
    char buf[12];
    v = x;
    iptoa(buf, &v);
    return len(buf);
}
```

The following show portions of the code for `intlen`, compiled both with and without stack protector:

(a) Without protector

```
     int intlen(long x)
     x in %rdi
1    intlen:
2      subq    $40, %rsp
3      movq    %rdi, 24(%rsp)
```

```
4     leaq    24(%rsp), %rsi
5     movq    %rsp, %rdi
6     call    iptoa
```

(b) With protector

```
      int intlen(long x)
      x in %rdi
1     intlen:
2       subq    $56, %rsp
3       movq    %fs:40, %rax
4       movq    %rax, 40(%rsp)
5       xorl    %eax, %eax
6       movq    %rdi, 8(%rsp)
7       leaq    8(%rsp), %rsi
8       leaq    16(%rsp), %rdi
9       call    iptoa
```

A. For both versions: What are the positions in the stack frame for `buf`, `v`, and (when present) the canary value?

B. How does the rearranged ordering of the local variables in the protected code provide greater security against a buffer overrun attack?

Limiting Executable Code Regions

A final step is to eliminate the ability of an attacker to insert executable code into a system. One method is to limit which memory regions hold executable code. In typical programs, only the portion of memory holding the code generated by the compiler need be executable. The other portions can be restricted to allow just reading and writing. As we will see in Chapter 9, the virtual memory space is logically divided into *pages*, typically with 2,048 or 4,096 bytes per page. The hardware supports different forms of *memory protection*, indicating the forms of access allowed by both user programs and the operating system kernel. Many systems allow control over three forms of access: read (reading data from memory), write (storing data into memory), and execute (treating the memory contents as machine-level code). Historically, the x86 architecture merged the read and execute access controls into a single 1-bit flag, so that any page marked as readable was also executable. The stack had to be kept both readable and writable, and therefore the bytes on the stack were also executable. Various schemes were implemented to be able to limit some pages to being readable but not executable, but these generally introduced significant inefficiencies.

More recently, AMD introduced an NX (for "no-execute") bit into the memory protection for its 64-bit processors, separating the read and execute access modes, and Intel followed suit. With this feature, the stack can be marked as being readable and writable, but not executable, and the checking of whether a page is executable is performed in hardware, with no penalty in efficiency.

Some types of programs require the ability to dynamically generate and execute code. For example, "just-in-time" compilation techniques dynamically generate code for programs written in interpreted languages, such as Java, to improve execution performance. Whether or not the run-time system can restrict the executable code to just that part generated by the compiler in creating the original program depends on the language and the operating system.

The techniques we have outlined—randomization, stack protection, and limiting which portions of memory can hold executable code—are three of the most common mechanisms used to minimize the vulnerability of programs to buffer overflow attacks. They all have the properties that they require no special effort on the part of the programmer and incur very little or no performance penalty. Each separately reduces the level of vulnerability, and in combination they become even more effective. Unfortunately, there are still ways to attack computers [85, 97], and so worms and viruses continue to compromise the integrity of many machines.

3.10.5 Supporting Variable-Size Stack Frames

We have examined the machine-level code for a variety of functions so far, but they all have the property that the compiler can determine in advance the amount of space that must be allocated for their stack frames. Some functions, however, require a variable amount of local storage. This can occur, for example, when the function calls `alloca`, a standard library function that can allocate an arbitrary number of bytes of storage on the stack. It can also occur when the code declares a local array of variable size.

Although the information presented in this section should rightfully be considered an aspect of how procedures are implemented, we have deferred the presentation to this point, since it requires an understanding of arrays and alignment.

The code of Figure 3.43(a) gives an example of a function containing a variable-size array. The function declares local array `p` of n pointers, where n is given by the first argument. This requires allocating $8n$ bytes on the stack, where the value of n may vary from one call of the function to another. The compiler therefore cannot determine how much space it must allocate for the function's stack frame. In addition, the program generates a reference to the address of local variable `i`, and so this variable must also be stored on the stack. During execution, the program must be able to access both local variable `i` and the elements of array `p`. On returning, the function must deallocate the stack frame and set the stack pointer to the position of the stored return address.

To manage a variable-size stack frame, x86-64 code uses register `%rbp` to serve as a *frame pointer* (sometimes referred to as a *base pointer*, and hence the letters `bp` in `%rbp`). When using a frame pointer, the stack frame is organized as shown for the case of function `vframe` in Figure 3.44. We see that the code must save the previous version of `%rbp` on the stack, since it is a callee-saved register. It then keeps `%rbp` pointing to this position throughout the execution of the function, and it references fixed-length local variables, such as `i`, at offsets relative to `%rbp`.

(a) C code

```
long vframe(long n, long idx, long *q)  {
    long i;
    long *p[n];
    p[0] = &i;
    for (i = 1; i < n; i++)
        p[i] = q;
    return *p[idx];
}
```

(b) Portions of generated assembly code

```
     long vframe(long n, long idx, long *q)
     n in %rdi, idx in %rsi, q in %rdx
     Only portions of code shown
 1   vframe:
 2     pushq   %rbp                    Save old %rbp
 3     movq    %rsp, %rbp              Set frame pointer
 4     subq    $16, %rsp               Allocate space for i (%rsp = s1)
 5     leaq    22(,%rdi,8), %rax
 6     andq    $-16, %rax
 7     subq    %rax, %rsp              Allocate space for array p (%rsp = s2)
 8     leaq    7(%rsp), %rax
 9     shrq    $3, %rax
10     leaq    0(,%rax,8), %r8         Set %r8 to &p[0]
11     movq    %r8, %rcx               Set %rcx to &p[0] (%rcx = p)
          . . .
     Code for initialization loop
     i in %rax and on stack, n in %rdi, p in %rcx, q in %rdx
12   .L3:                              loop:
13     movq    %rdx, (%rcx,%rax,8)     Set p[i] to q
14     addq    $1, %rax                Increment i
15     movq    %rax, -8(%rbp)          Store on stack
16   .L2:
17     movq    -8(%rbp), %rax          Retrieve i from stack
18     cmpq    %rdi, %rax              Compare i:n
19     jl      .L3                     If <, goto loop
          . . .
     Code for function exit
20     leave                            Restore %rbp and %rsp
21     ret                              Return
```

Figure 3.43 Function requiring the use of a frame pointer. The variable-size array implies that the size of the stack frame cannot be determined at compile time.

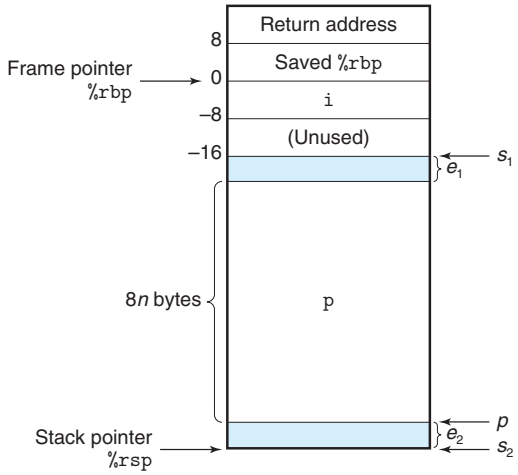


Figure 3.44
Stack frame structure for function `vframe`. The function uses register `%rbp` as a frame pointer. The annotations along the right-hand side are in reference to Practice Problem 3.49.

Figure 3.43(b) shows portions of the code GCC generates for function `vframe`. At the beginning of the function, we see code that sets up the stack frame and allocates space for array `p`. The code starts by pushing the current value of `%rbp` onto the stack and setting `%rbp` to point to this stack position (lines 2–3). Next, it allocates 16 bytes on the stack, the first 8 of which are used to store local variable `i`, and the second 8 of which are unused. Then it allocates space for array `p` (lines 5–11). The details of how much space it allocates and where it positions `p` within this space are explored in Practice Problem 3.49. Suffice it to say that by the time the program reaches line 11, it has (1) allocated at least $8n$ bytes on the stack and (2) positioned array `p` within the allocated region such that at least $8n$ bytes are available for its use.

The code for the initialization loop shows examples of how local variables `i` and `p` are referenced. Line 13 shows array element `p[i]` being set to `q`. This instruction uses the value in register `%rcx` as the address for the start of `p`. We can see instances where local variable `i` is updated (line 15) and read (line 17). The address of `i` is given by reference `-8(%rbp)`—that is, at offset -8 relative to the frame pointer.

At the end of the function, the frame pointer is restored to its previous value using the `leave` instruction (line 20). This instruction takes no arguments. It is equivalent to executing the following two instructions:

```
movq %rbp, %rsp     Set stack pointer to beginning of frame
popq %rbp           Restore saved %rbp and set stack ptr
                      to end of caller's frame
```

That is, the stack pointer is first set to the position of the saved value of `%rbp`, and then this value is popped from the stack into `%rbp`. This instruction combination has the effect of deallocating the entire stack frame.

In earlier versions of x86 code, the frame pointer was used with every function call. With x86-64 code, it is used only in cases where the stack frame may be of variable size, as is the case for function `vframe`. Historically, most compilers used frame pointers when generating IA32 code. Recent versions of GCC have dropped this convention. Observe that it is acceptable to mix code that uses frame pointers with code that does not, as long as all functions treat `%rbp` as a callee-saved register.

Practice Problem 3.49 (solution page 383)

In this problem, we will explore the logic behind the code in lines 5–11 of Figure 3.43(b), where space is allocated for variable-size array `p`. As the annotations of the code indicate, let us let s_1 denote the address of the stack pointer after executing the `subq` instruction of line 4. This instruction allocates the space for local variable `i`. Let s_2 denote the value of the stack pointer after executing the `subq` instruction of line 7. This instruction allocates the storage for local array `p`. Finally, let p denote the value assigned to registers `%r8` and `%rcx` in the instructions of lines 10–11. Both of these registers are used to reference array `p`.

The right-hand side of Figure 3.44 diagrams the positions of the locations indicated by s_1, s_2, and p. It also shows that there may be an offset of e_2 bytes between the values of s_1 and p. This space will not be used. There may also be an offset of e_1 bytes between the end of array `p` and the position indicated by s_1.

A. Explain, in mathematical terms, the logic in the computation of s_2 on lines 5–7. *Hint:* Think about the bit-level representation of -16 and its effect in the `andq` instruction of line 6.

B. Explain, in mathematical terms, the logic in the computation of p on lines 8–10. *Hint:* You may want to refer to the discussion on division by powers of 2 in Section 2.3.7.

C. For the following values of n and s_1, trace the execution of the code to determine what the resulting values would be for s_2, p, e_1, and e_2.

n	s_1	s_2	p	e_1	e_2
5	2,065	_____	_____	_____	_____
6	2,064	_____	_____	_____	_____

D. What alignment properties does this code guarantee for the values of s_2 and p?

3.11 Floating-Point Code

The *floating-point architecture* for a processor consists of the different aspects that affect how programs operating on floating-point data are mapped onto the machine, including

- How floating-point values are stored and accessed. This is typically via some form of registers.