You will start by exploring the nature of the security mechanisms behind Linux and SELinux, and as you complete the chapters, you will integrate and enable SE for Android into a System on Chip (SoC), a process that, prior to this book, has never before been documented in its entirety! Discover Android’s unique user space, from its use of the common UID and GID model to promote its security goals to its custom binder IPC mechanism. Explore the interface between the kernel and user space with respect to SELinux and investigate contexts and labels and their application to system objects.

This book will help you develop the necessary skills to evaluate and engineer secured products with the Android platform, whether you are new to world of Security Enhanced Linux (SELinux) or experienced in secure system deployment.

Who this book is written for

This book is intended for developers and engineers with some familiarity of operating system concepts as implemented by Linux. A basic background in C code would be helpful. Their positions range from hobbyists wanting to secure their Android powered creations to OEM engineers building handsets to engineers of emerging areas where Android is seeing growth.

What you will learn from this book

- Experiment with Linux and SELinux access controls
- Build custom Android kernels
- Backport SE for Android patches to different Android versions
- Explore binder and property services, what they are, and how and why SELinux integrates them
- Work with Android core internal systems like init and zygote
- Learn how to keep pace with and navigate the details of fast moving open source projects
- Overcome obstacles in policy development through directed experimentation

Discover Security Enhancements (SE) for Android to build your own protected Android-based systems

Foreword by Kenny Root
In this package, you will find:

- The authors biography
- A preview chapter from the book, Chapter 1 "Linux Access Controls"
- A synopsis of the book’s content
- More information on Exploring SE for Android

About the Authors

William Confer has been engineering embedded and mobile systems since 1997. He has worked for Samsung Mobile as a managing staff engineer and currently teaches computer science at SUNY Polytechnic Institute. He holds a patent in low-cost character recognition for extremely resource-limited devices and has multiple other patents pending for mobile technologies.

My wife, Ása, sacrificed endlessly to help give me the space and time needed for this work, and I owe her more than I can say. My three daughters also ensured I couldn't always be working on this book and distracted me in the best possible ways. I couldn't rest if I didn't thank all my fall 2014 students from SUNY Polytechnic Institute who put up with me when I was sidetracked by this book. Finally, and most importantly, my greatest thanks goes to my coauthor (and friend, student, and teacher), William Roberts, without whom I would have to have found another.
William Roberts is a software engineer who is focused on OS-level security and platform enhancements. He is one of the engineers who founded the Samsung KNOX product and an early adopter of SE for Android. He has made contributions to several open source projects, such as SE for Android, the Android Open Source Project, the Linux Kernel, CyanogenMod, and OpenSC. His recent interests have taken him to Smart Card technologies and the virtualization of smart cards. In his spare time, he works with Dr. Confer on the Miniat project (http://www.miniat.org), a virtual, embedded architecture simulator.

I would like to thank Dr. William Confer, the coauthor, for helping me write this book; his contributions were invaluable. Also, I would like to thank my wife for supporting me and giving me the time to do this, even though we were renovating the house. Also, I would like to thank my family and friends for their encouragement along the way.
Exploring SE for Android

This book introduces the Security Enhancements (SE) for Android open source project and walks you through the process of securing new embedded systems with SE for Android. To our knowledge, this book is the first source to document such a process in its entirety so that students, DIY hobbyists, and engineers can create custom systems secured by SE for Android. Generally, only original equipment manufacturers (OEMs) do this, and quite commonly, the target device is a phone or tablet. We truly hope our book will change that, engaging a wide audience in development so they can use and understand these modern security tools.

We worked very hard to ensure this text is not just a step-by-step technology book. Specifically, we've chosen a model that directs you to fail your way to success. You will first gain appropriate theoretical understanding of how security is gained and enforced. Then we will introduce a system that has never been secured that way (not even by us, prior to writing this book). Next, we'll guide you through all our intelligent guesswork, embracing unexpected failures for the newly found idiosyncrasies they expose, and eventually enforcing our custom security policies. It requires you to learn to resolve differences between major open source projects such as SELinux, SE for Android, and Google Android, each of which has independent goals and deployment schedules. This prepares you to secure other devices, the process for which is always different, but hopefully, will now be more accessible.

What This Book Covers

Chapter 1, Linux Access Controls, discusses the basics of Discretionary Access Control (DAC), how some Android exploits leverage DAC problems, and demonstrate the need for more robust solutions.

Chapter 2, Mandatory Access Controls and SELinux, examines Mandatory Access Control (MAC) and its manifestation in SELinux. This chapter also explores tangible policy to control SELinux object interaction.

Chapter 3, Android Is Weird, introduces the Android security model and investigates binder, zygote, and the property service.

Chapter 4, Installation on the UDOO, walks through building and deploying Android from source to the UDOO-embedded board and turns on SELinux support.

Chapter 5, Booting the System, follows the boot process from the policy loading perspective and corrects issues to get SELinux to a usable state on the UDOO.

Chapter 6, Exploring SELinuxFS, examines the SELinuxFS filesystem and how it provides the kernel-to-userspace interface for higher-level idioms.
Chapter 7, Utilizing Audit Logs, investigates the audit subsystem, revealing how to interpret SELinux audit logs for the benefit of policy writing.

Chapter 8, Applying Contexts to Files, teaches you how filesystems and filesystem objects get their labels and contexts, demonstrating techniques to change them, including dynamic type transitions.

Chapter 9, Adding Services to Domains, emphasizes process labeling, notably the Android services run and managed by init.

Chapter 10, Placing Applications in Domains, shows you how to properly label the private data directories of applications, as well as application runtime contexts via configuration files and SELinux policy.

Chapter 11, Labeling Properties, demonstrates how to create and label new and existing properties, and some of the anomalies that occur when doing so.

Chapter 12, Mastering the Tool Chain, covers how the various components that control policy on the device are actually built and created. This chapter reviews the Android.mk components, detailing how the heart of the build and configuration management works.

Chapter 13, Getting to Enforcing Mode, utilizes all the skills you learned in the earlier chapters to respond to audit logs from CTS and get the UDOO in enforcing mode.

Appendix, The Development Environment, walks you through the necessary steps of setting up a Linux environment suitable for you to follow all the activities in this book.
Linux Access Controls

Android is an operating system composed of two distinct components. The first component is a forked mainline Linux kernel and shares almost everything in common with Linux. The second component, which will be discussed later, is the user space portion, which is very custom and Android specific. Since the Linux kernel underpins this system and is responsible for the majority of access control decisions, it is the logical place to begin a detailed look at Android.

In this chapter we will:

• Examine the basics of Discretionary Access Control
• Introduce Linux permissions flags and capabilities
• Trace syscalls as we validate access policies
• Make the case for more robust access control technology
• Discuss Android exploits that leverage problems with Discretionary Access Control

Linux’s default and familiar access control mechanism is called Discretionary Access Control (DAC). This is just a term that means permissions regarding access to an object are at the discretion of its creator/owner.

In Linux, when a process invokes most system calls, a permission check is performed. As an example, a process wishing to open a file would invoke the open() syscall. When this syscall is invoked, a context switch is performed, and the operating system code is executed. The OS has the ability to determine whether a file descriptor should be returned to the requesting process or not. During this decision-making process, the OS checks the access permissions of both the requesting process and the target file it wishes to obtain the file descriptor to. Either the file descriptor or EPERM is returned, dependent on whether the permission checks pass or fail respectively.
Linux Access Controls

Linux maintains data structures in the kernel for managing these permission fields, which are accessible from user space, and ones that should be familiar to Linux and *NIX users alike. The first set of access control metadata belongs to the process, and forms a portion of its credential set. The common credentials are user and group. In general, we use the term group to mean both primary group and possible secondary group(s). You can view these permissions by running the `ps` command:

```
$ ps -eo pid,comm,user,group,supgrp
PID COMMAND         USER     GROUP    SUPGRP
 1 init            root     root     -
... 
 2993 system-service- root     root     root
 3276 chromium-browse bookuser sudo fuse bookuser 
... 
```

As you can see, we have processes running as the users `root` and `bookuser`. You can also see that their primary group is only one part of the equation. Processes also have a secondary set of groups called supplementary groups. This set might be empty, indicated by the dash in the `SUPGRP` field.

The file we wish to open, referred to as the target object, target, or object also maintains a set of permissions. The object maintains `USER` and `GROUP`, as well as a set of permission bits. In the context of the target object, `USER` can be referred to as `owner` or `creator`.

```
$ ls -la
total 296
  d rwxr-xr-x 38 bookuser bookuser 4096 Aug 23 11:08 .
  d rwxr-xr-x  3 root     root      4096 Jun  8 18:50 ..
-rw-rw-r--  1 bookuser bookuser   116 Jul 22 13:13 .c
  d rwxrwxr-x  4 bookuser bookuser  4096 Aug  4 16:20 .android
-rw-rw-r--  1 bookuser bookuser  130 Jun 19 17:51 .apport-ignore.xml
```
If we look at the preceding command's output, we can see that `hello.txt` has a **USER** of `bookuser` and **GROUP** as `bookuser`. We can also see the permission bits or flags on the left-hand side of the output. There are seven fields to consider as well. Each empty field is denoted with a dash. When printed with `ls`, the first fields can get convoluted by semantics. For this reason, let's use `stat` to investigate the file permissions:

```
$ stat hello.txt
  File: `hello.txt'
    Size: 365         Blocks: 8          IO Block: 4096   regular file
    Device: 801h/2049d  Inode: 1587858     Links: 1
    Access: (0664/-rw-rw-r--)  Uid: ( 1000/bookuser)   Gid: ( 1000/bookuser)
    Birth: -
```

The first access line is the most compelling. It contains all the important information for the access controls. The second line is just a timestamp letting us know when the file was last accessed. As we can see, **USER** or **UID** of the object is `bookuser`, and **GROUP** is `bookuser` as well. The permission flags, `(0664/-rw-rw-r--)`, identify the two ways that permission flags are represented. The first, the octal form `0664`, condenses each three-flag field into one of the three base-8 (octal) digits. The second is the friendly form, `-rw-rw-r--`, equivalent to the octal form but easier to interpret visually. In either case, we can see the leftmost field is 0, and the rest of our discussions will ignore it. That field is for **setuid** and **setgid** capabilities, which is not important for this discussion. If we convert the remaining octal digits, `664`, to binary, we get `110 110 100`. This binary representation directly relates to the friendly form. Each triple maps to read, write, and execute permissions. Often you will see this permission triple represented as `RWX`. The first triple are the permissions given to **USER**, the second are the permissions given to **GROUP**, and the third is what is given to **OTHERS**. Translating to conventional English would yield, "The user, `bookuser`, has permission to read from and write to `hello.txt`. The group, `bookuser`, has permission to read from and write to `hello.txt`, and everyone else has permission only to read from `hello.txt." Let's test this with some real-world examples.
Changing permission bits

Let's test the access controls in the example running processes as user bookuser. Most processes run in the context of the user that invoked them (excluding setuid and getuid programs), so any command we invoke should inherit our user's permissions. We can view it by issuing:

```
$ groups bookuser
bookuser : bookuser sudo fuse
```

My user, bookuser, is USER bookuser, GROUP bookuser and SUPGRP sudo and fuse.

To test for read access, we can use the `cat` command, which opens the file and prints its content to `stdout`:

```
$ cat hello.txt
Hello, "Exploring SE for Android"
Here is a simple text file for
your enjoyment.
...
```

We can introspect the syscalls executed by running the `strace` command and viewing the output:

```
$ strace cat hello.txt
...
open("hello.txt", O_RDONLY)                   = 3
...
read(3, "Hello, \"Exploring SE for Android\"\n..., 32768) = 365
...
```

The output can be quite verbose, so I am only showing the relevant parts. We can see that `cat` invoked the `open` syscall and obtained the file descriptor 3. We can use that descriptor to find other accesses via other syscalls. Later we will see a read occurring on file descriptor 3, which returns 365, the number of bytes read. If we didn't have permission to read from `hello.txt`, the open would fail, and we would never have a valid file descriptor for the file. We would additionally see the failure in the `strace` output.

Now that read permission is verified, let's try write. One simple way to do this is to write a simple program that writes something to the existing file. In this case, we will write the line `my new text\n` (refer to `write.c`).

[10]
Compile the program using the following command:

$ gcc -o mywrite write.c

Now run using the newly compiled program:

$ strace ./mywrite hello.txt

On verification, you will see:

... 
open("hello.txt", O_WRONLY) = 3
write(3, "my new text\n", 12) = 12
...

As you can see, the write succeeded and returned 12, the number of bytes written to hello.txt. No errors were reported, so the permissions seem in check so far.

Now let's attempt to execute hello.txt and see what happens. We are expecting to see an error. Let's execute it like a normal command:

$ ./hello.txt
bash: ./hello.txt: Permission denied

This is exactly what we expected, but let's invoke it with strace to gain a deeper understanding of what failed:

$ strace ./hello.txt

... 
execve("./hello.txt", ["./hello.txt"], [/* 39 vars */]) = -1 EACCESS
(Permission denied)
...

The execve system call, which launches processes, failed with EACCESS. This is just the sort of thing one would hope for when no execute permission is given. The Linux access controls worked as expected!

Let's test the access controls in the context of another user. First, we'll create a new user called testuser using the adduser command:

$ sudo adduser testuser
[sudo] password for bookuser:
Adding user `testuser' ...
Adding new group `testuser' (1001) ...
Adding new user `testuser' (1001) with group `testuser' ...

Creating home directory `/home/testuser' ...
...

Verify the USER, GROUP, and SUPGRP of testuser:

$ groups testuser
testuser : testuser

Since the USER and GROUP do not match any of the permissions on a .S, all accesses will be subject to the OTHERS permissions checks, which if you recall, is read only (0664).

Start by temporarily working as testuser:

$ su testuser
Password:
testuser@ubuntu:/home/bookuser$

As you can see, we are still in bookuser's home directory, but the current user has been changed to testuser.

We will start by testing read with the cat command:

$ strace cat hello.txt
...
open("hello.txt", O_RDONLY) = 3
...
read(3, "my new text\n", 32768) = 12
...

Similar to the earlier example, testuser can read the data just fine, as expected.

Now let's move on to write. The expectation is that this will fail without appropriate access:

$ strace ./mywrite hello.txt
...
open("hello.txt", O_WRONLY) = -1 EACCES (Permission denied)
...

As expected, the syscall operation failed. When we attempt to execute hello.txt as testuser, this should fail as well:

$ strace ./hello.txt
...
execve("./hello.txt", ["./hello.txt"], /* 40 vars */) = -1 EACCES
(Permission denied)
...

Now we need to test the group access permissions. We can do this by adding a supplementary group to testuser. To do this, we need to exit to bookuser, who has permissions to execute the sudo command:

$ exit
exit
$ sudo usermod -G bookuser testuser

Now let’s check the groups of testuser:

$ groups testuser
testuser : testuser bookuser

As a result of the previous usermod command testuser now belongs to two groups: testuser and bookuser. That means when testuser accesses a file or other object (such as a socket) with the group bookuser, the GROUP permissions, rather than OTHERS, will apply to it. In the context of hello.txt, testuser can now read from and write to the file, but not execute it.

Switch to testuser by executing the following command:

$ su testuser

Test read by executing the following command:

$ strace cat ./hello.txt
...
open("./hello.txt", O_RDONLY)      = 3
...
read(3, "my new text\n", 32768)    = 12
...

As before, testuser is able to read the file. The only difference is that it can now read the file through the access permissions of OTHERS and GROUP.

Test write by executing the following command:

$ strace ./mywrite hello.txt
...
open("hello.txt", O_WRONLY)        = 3
write(3, "my new text\n", 12)       = 12
...
This time, testuser was able to write the file as well, instead of failing with the EACCESS permission error shown before.

Attempting to execute the file should still fail:

$ strace ./hello.txt
execve("./hello.txt", ["./hello.txt"], [/* 40 vars */]) = -1 EACCES
(Permission denied)
...

These concepts are the foundation of Linux access control permission bits, users and groups.

### Changing owners and groups

Using hello.txt for exploratory work in the previous sections, we have shown how the owner of an object can allow various forms of access by managing the permission bits of the object. Changing the permissions is accomplished using the chmod syscall. Changing the user and/or group is done with the chown syscall. In this section, we will investigate the details of these operations in action.

Let's start by granting read and write permissions only to the owner of hello.txt file, bookuser.

$ chmod 0600 hello.txt
$ stat hello.txt
   File: `hello.txt'
   Size: 12          Blocks: 8          IO Block: 4096   regular file
   Device: 801h/2049d  Inode: 1587858     Links: 1
   Access: (0600/-rw-------)  Uid: ( 1000/bookuser)   Gid: ( 1000/bookuser)
   Birth: -

As we can see, the file permissions are now set to only allow read and write access for bookuser. A thorough reader could execute the commands from earlier sections in this chapter to verify that permissions work as expected.

Changing the group can be done in a similar fashion with chown. Let's change the group to testuser:

$ chown bookuser:testuser hello.txt
chown: changing ownership of `hello.txt': Operation not permitted
This did not work as we intended, but what is the issue? In Linux, only privileged processes can change the USER and GROUP fields of objects. The initial USER and GROUP fields are set during object creation from the effective USER and GROUP, which are checked when attempting to execute that process. Only processes create objects. Privileged processes come in two forms: those running as the almighty root and those that have their capabilities set. We will dive into the details of capabilities later. For now, let's focus on the root.

Let's change the user to root to ensure executing the chown command will change the group of that object:

```
$ sudo su
# chown bookuser:testuser hello.txt
```

Now, we can verify the change occurred successfully:

```
# stat hello.txt
   File: `hello.txt'
   Size: 12          Blocks: 8           IO Block: 4096   regular file
  Device: 801h/2049d  Inode: 1587858     Links: 1
 Access: (0600/-rw-------)  Uid: ( 1000/bookuser)   Gid: ( 1001/testuser)
Change: 2014-08-23 13:08:46.059058649 -0700
```

The case for more

You can see the GROUP (GID) is now testuser, and things seem reasonably secure because in order to change the user and group of an object, you need to be privileged. You can only change the permission bits on an object if you own it, with the exception of the root user. This means that if you're running as root, you can do whatever you like to the system, even without permission. This absolute authority is why a successful attack or an error on a root running process can cause grave damage to the system. Also, a successful attack on a non-root process could also cause damage by inadvertently changing the permissions bits. For example, suppose there is an unintended chmod 0666 command on your SSH private key. This would expose your secret key to all users on the system, which is almost certainly something you would never want to happen. The root limitation is partially addressed by the capabilities model.
Capabilities model

For many operations on Linux, the object permission model doesn't quite fit. For instance, changing UID and GID requires some magical USER known as root. Suppose you have a long running service that needs to utilize some of these capabilities. Perhaps this service listens to kernel events and creates the device nodes for you? Such a service exists, and it's called ueventd or user event daemon. This daemon traditionally runs as root, which means if it is compromised, it could potentially read your private keys from your home directory and send them back to the attacker. This might be an extraordinary example, but it's meant to showcase that running processes as root can be dangerous. Suppose you could start a service as the root user and have the process change its UID and GID to something not privileged, but retain some smaller set of privileged capabilities to do its job? This is exactly what the capabilities model in Linux is.

The capabilities model in Linux is an attempt to break down the set of permissions that root has into smaller subsets. This way, processes can be confined to the set of minimum privileges they need to perform their intended function. This is known as least privilege, a key ideology when securing systems that minimizes the amount of damage a successful attack can do. In some instances, it can even prevent a successful attack from occurring by blocking an otherwise open attack vector.

There are many capabilities. The man page for capabilities is the de facto documentation. Let's take a look at the CAP_SYS_BOOT capability:

$ man capabilities
...
CAP_SYS_BOOT
   Use reboot(2) and kexec_load(2).

This means a process running with this capability can reboot the system. However, that process can't arbitrarily change USERS and GROUP as it could if it was running as root or with CAP_DAC_READ_SEARCH. This limits what an attacker can do:

<FROM MAN PAGE>

CAP_DAC_READ_SEARCH
   Bypass file read permission checks and directory read and execute permission checks.
Now suppose the case where our restart process runs with CAP_CHOWN. Let's say it uses this capability to ensure that when a restart request is received, it backs up a file from each user's home directory to a server before restarting. Let's say this file is 
backup, the permissions are 0600, and USER and GROUP are the respective user of that home directory. In this case, we have minimized the permissions as best we can, but the process could still access the users SSH keys and upload those either by error or attack. Another approach to this would be to set the group to backup and run the process with GROUP backup. However, this has limitations. Suppose you want to share this file with another user. That user would require a supplementary group of backup, but now the user can read all of the backup files, not just the ones intended. An astute reader might think about the bind mounts, however the process doing the bind mounts and file permissions also runs with some capability, and thus suffers from this granularity problem as well.

The major issue, and the case for another access control system can be summarized by one word, granularity. The DAC model doesn't have the granularity required to safely handle complex access control models or to minimize the amount of damage a process can do. This is particularly important on Android, where the entire isolation system is dependent on this control, and a rogue root process can compromise the whole system.

**Android's use of DAC**

In the Android sandbox model, every application runs as its own UID. This means that each app can separate its stored data from one another. The user and group are set to the UID and GID of that application, so no app can access the private files of an application without the application explicitly performing chmod on its objects. Also, applications in Android cannot have capabilities, so we don't have to worry about capabilities such as CAP_SYS_PTRACE, which is the ability to debug another application. In Android, in a perfect world, only system components run with privileges, and applications don't accidentally chmod private files for all to read. This issue was not corrected by the current AOSP SELinux policy due to app compatibility, but could be closed with SELinux. The proper way to share data between applications on Android is via binder, and sharing file descriptors. For smaller amounts of data, the provider model suffices.

**Glancing at Android vulnerabilities**

With our newly found understanding of the DAC permission model and some of its limitations, let's look at some Android exploits against it. We will cover only a few exploits to understand how the DAC model failed.
Linux Access Controls

Skype vulnerability
CVE-2011-1717 was released in 2011. In this exploit, the Skype application left a SQLite3 database world readable (something analogous to 0666 permissions). This database contained usernames and chat logs, and personal data such as name and e-mail. An application called Skypwned was able to demonstrate this capability. This is an example of how being able to change the permissions on your objects could be bad, especially when the case opens READ to OTHERS.

GingerBreak
CVE-2011-1823 showcases a root attack on Android. The volume management daemon (vold) on Android is responsible for the mounting and unmounting of the external SD card. The daemon listens for messages over a NETLINK socket. The daemon never checked where the messages were sourced from, and any application could open and create a NETLINK socket to send messages to vold. Once the attacker opened the NETLINK socket, they sent a very carefully crafted message to bypass a sanity check. The check tested a signed integer for a maximum bound, but never checked it for negativity. It was then used to index an array. This negative access would lead to memory corruption and, with a proper message, could result in the execution of arbitrary code. The GingerBreak implementation resulted in an arbitrary user gaining root privileges, a textbook privilege execution attack. Once rooted, the device's sandboxes were no longer valid.

Rage against the cage
CVE-2010-EASY is a setuid exhaustion via fork bomb attack. It successfully attacks the adb daemon on Android, which starts life as root and downgrades its permissions if root is not needed. This attack keeps adb as root and returns a root shell to the user. In Linux kernel 2.6, the setuid system call returns an error when the number of running processes RLIMIT_NPROC is met. The adb daemon code does not check the return of setuid, which leaves a small race window open for the attacker. The attacker needs to fork enough processes to reach RLIMIT_NPROC and then kill the daemon. The adb daemon downgrades to shell UID and the attacker runs the program as shell USER, thus the kill will work. At this point, the adb service is respawned, and if RLIMIT_NPROC is maxed out, setuid will fail and adb will stay running as root. Then, running adb shell from a host returns a nice root shell to the user.
MotoChopper
CVE-2013-2596 is a vulnerability in the mmap functionality of a Qualcomm video driver. Access to the GPU is provided by apps to do advanced graphics rendering such as in the case of OpenGL calls. The vulnerability in mmap allows the attacker to mmap kernel address space, at which point the attacker is able to directly change their kernel credential structure. This exploit is an example where the DAC model was not at fault. In reality, outside of patching the code or removing direct graphics access, nothing but programming checks of the mmap bounds could have prevented this attack.

Summary
The DAC model is extremely powerful, but its lack of fine granularity and use of an extraordinarily powerful root user leaves something to be desired. With the increasing sensitivity of mobile handset use, the case to increase the security of the system is well-founded. Thankfully, Android is built on Linux and thus benefits from a large ecosystem of engineers and researchers. Since the Linux Kernel 2.6, a new access control model called Mandatory Access Controls (MAC) was added. This is a framework by which modules can be loaded into the kernel to provide a new form of access control model. The very first module was called SELinux. It is used by Red Hat and others to secure sensitive government systems. Thus, a solution was found to enable such access controls for Android.
Where to buy this book

You can buy Exploring SE for Android from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.