MORE FROM

NO STARCH PRESS

READ SAMPLE CHAPTERS FROM THESE NO STARCH BOOKS!

THE LINUX COMMAND LINE, 2ND EDITION

WILLIAM E. SHOTTS, JR.

•

BECOME A LINUX WIZARD! A ZINE BOXSET

JULIA EVANS

•

LINUX BASICS FOR HACKERS: GETTING STARTED WITH NETWORKING, SCRIPTING, AND SECURITY IN KALI

OCCUPYTHEWEB

•

ABSOLUTE FREEBSD, 3RD EDITION: THE COMPLETE GUIDE TO FREEBSD

MICHAEL W. LUCAS

•

AUTOTOOLS, 2ND EDITION

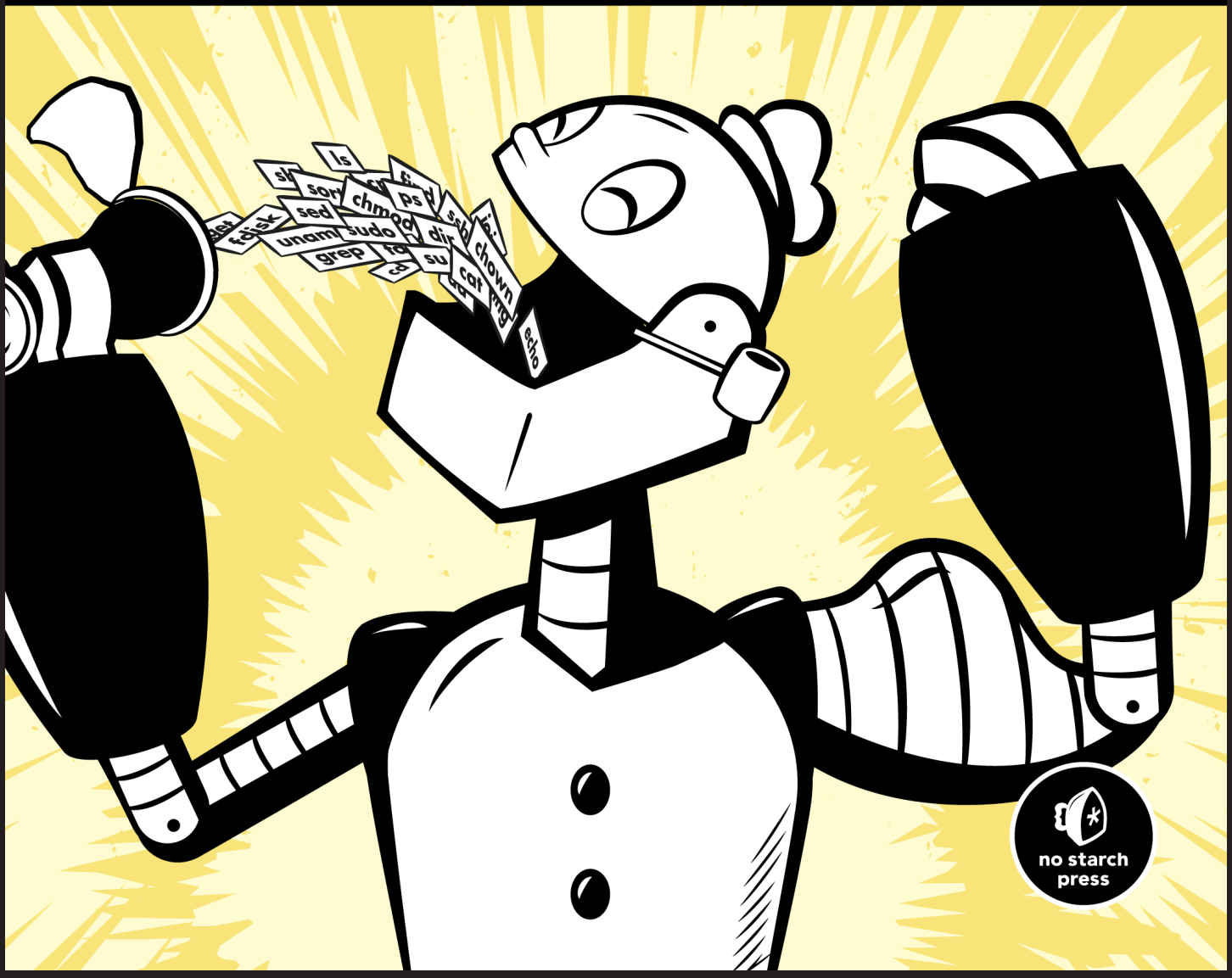
JOHN CALCOTE

2ND EDITION

THE LINUX COMMAND LINE

A *COMPLETE* INTRODUCTION

WILLIAM E. SHOTTS, JR.



7

SEEING THE WORLD AS THE SHELL SEES IT

In this chapter we are going to look at some of the "magic" that occurs on the command line when you press the ENTER key. While we will examine several interesting and complex features of the shell, we will do it with just one new command:

- `echo`—Display a line of text.

Expansion

Each time you type a command line and press the ENTER key, `bash` performs several processes upon the text before it carries out your command. We've seen a couple of cases of how a simple character sequence, for example *, can have a lot of meaning to the shell. The process that makes this happen is called *expansion*. With expansion, you enter something, and it is expanded into something else before the shell acts upon it. To demonstrate what we

mean by this, let's take a look at the echo command. echo is a shell builtin that performs a very simple task: It prints out its text arguments on standard output.

```
[me@linuxbox ~]$ echo this is a test
this is a test
```

That's pretty straightforward. Any argument passed to echo gets displayed. Let's try another example:

```
[me@linuxbox ~]$ echo *
Desktop Documents ls-output.txt Music Pictures Public Templates Videos
```

So what just happened? Why didn't echo print *? As you recall from our work with wildcards, the * character means "match any characters in a filename," but what we didn't see in our original discussion was how the shell does that. The simple answer is that the shell expands the * into something else (in this instance, the names of the files in the current working directory) before the echo command is executed. When the ENTER key is pressed, the shell automatically expands any qualifying characters on the command line before the command is carried out, so the echo command never saw the *, only its expanded result. Knowing this, we can see that echo behaved as expected.

Pathname Expansion

The mechanism by which wildcards work is called *pathname expansion*. If we try some of the techniques that we employed in our earlier chapters, we will see that they are really expansions. Given a home directory that looks like this:

```
[me@linuxbox ~]$ ls
Desktop    ls-output.txt  Pictures  Templates
Documents  Music          Public    Videos
```

we could carry out the following expansions:

```
[me@linuxbox ~]$ echo D*
Desktop Documents
```

and

```
[me@linuxbox ~]$ echo *s
Documents Pictures Templates Videos
```

or even

```
[me@linuxbox ~]$ echo [[:upper:]]*
Desktop Documents Music Pictures Public Templates Videos
```

And looking beyond our home directory:

```
[me@linuxbox ~]$ echo /usr/*/share
/usr/kerberos/share /usr/local/share
```

PATHNAME EXPANSION OF HIDDEN FILES

As we know, filenames that begin with a period character are hidden. Pathname expansion also respects this behavior. An expansion such as

```
echo *
```

does not reveal hidden files.

It might appear at first glance that we could include hidden files in an expansion by starting the pattern with a leading period, like this:

```
echo .*
```

It almost works. However, if we examine the results closely, we will see that the names `.` and `..` will also appear in the results. Since these names refer to the current working directory and its parent directory, using this pattern will likely produce an incorrect result. We can see this if we try the command

```
ls -d .* | less
```

To correctly perform pathname expansion in this situation, we have to employ a more specific pattern. This will work correctly:

```
ls -d .[!.]?*
```

This pattern expands into every filename that begins with a period, does not include a second period, contains at least one additional character, and may be followed by any other characters.

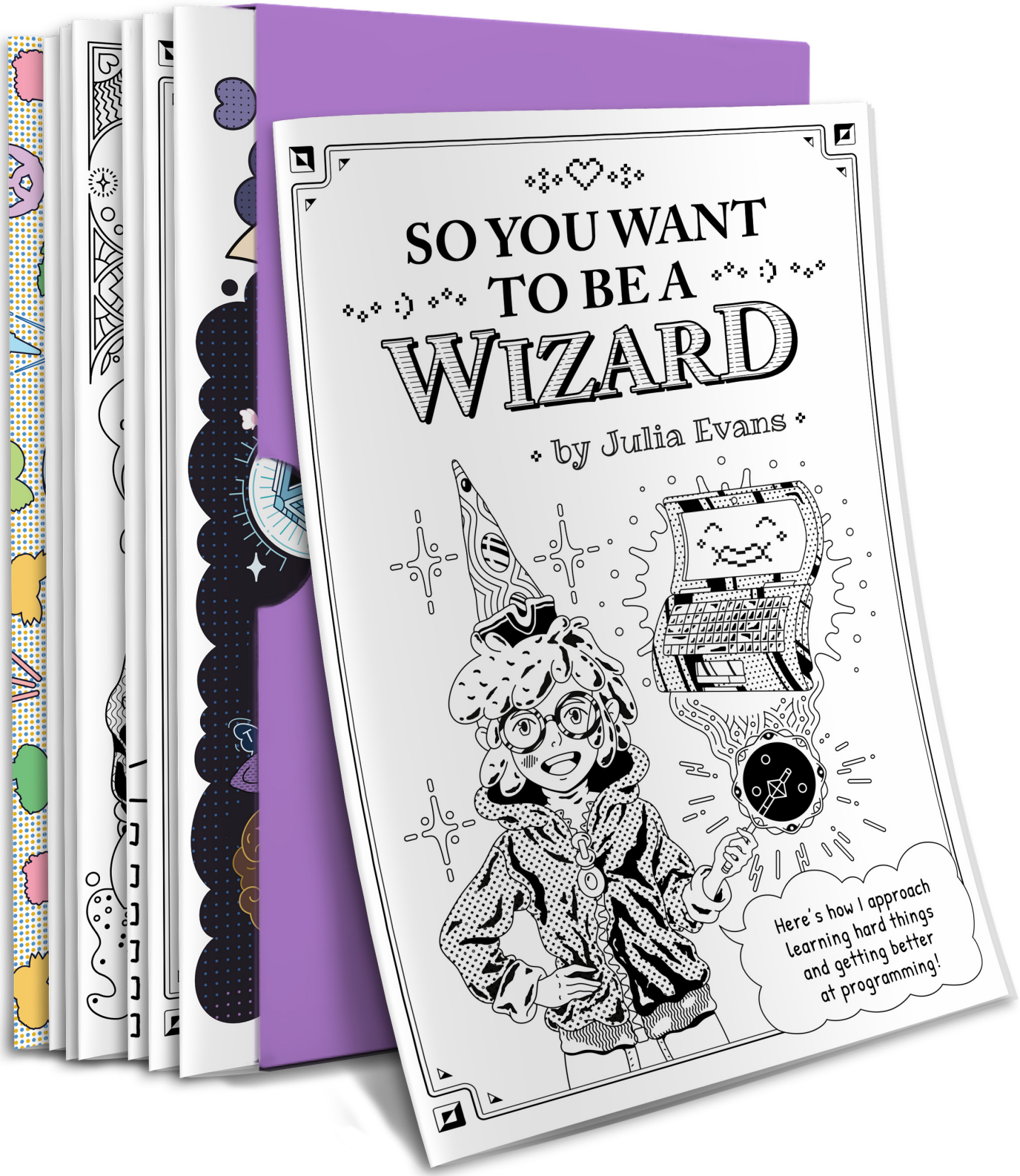
Tilde Expansion

As you may recall from our introduction to the `cd` command, the tilde character (~) has a special meaning. When used at the beginning of a word, it expands into the name of the home directory of the named user or, if no user is named, the home directory of the current user:

```
[me@linuxbox ~]$ echo ~
/home/me
```

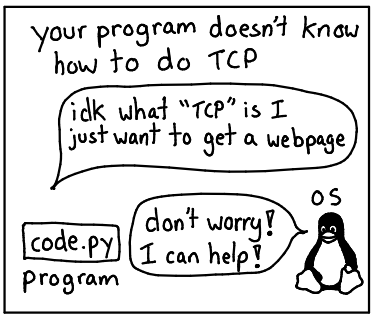
If user *foo* has an account, then

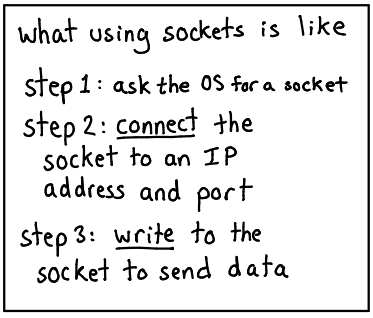
```
[me@linuxbox ~]$ echo ~foo
/home/foo
```

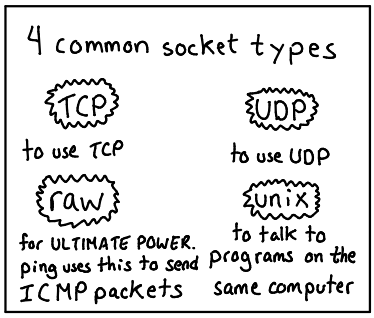

SO YOU WANT TO BE A
WIZARD
by Julia Evans
Here's how I approach learning hard things and getting better at programming!

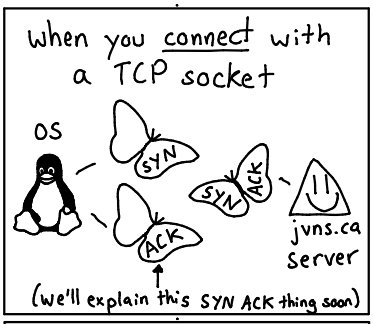
sockets

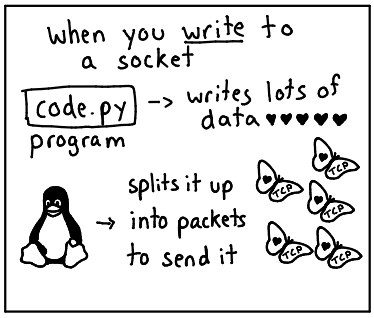
Step ②: now that we have an IP address, the next step is to open a socket! Let's learn what that is.

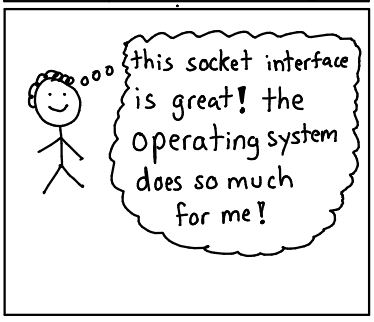










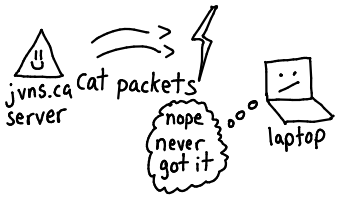


TCP: how to reliably get a cat

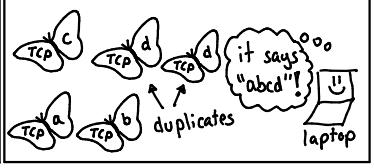
Step ③ in our plan is "open a TCP connection!"

Let's learn what this "TCP" thing even is

When you send a packet on the internet sometimes it gets lost.



TCP lets you send a stream of data reliably even if packets get lost or sent in the wrong order.



how does TCP work, you ask? WELL!

how to know what order the packets should go in:

Every packet says what range of bytes it has

Like this:

once upon a ti ← bytes 0-13
agical oyster ← bytes 30-42
me there was a m ← bytes 14-29

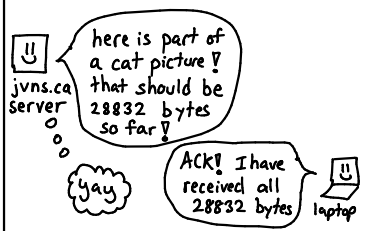
Then the client can assemble all the pieces into:

"once upon a time there was a magical oyster"

The position of the first byte (0, 14, 30 in our example) is called the "sequence number"

how to deal with lost packets:

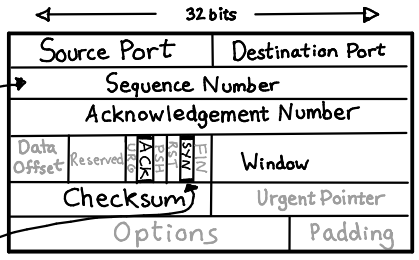
When you get TCP data, you have to acknowledge it: (ACK)



If the server doesn't get an ACKnowledgement, it will retry sending the data.

The TCP Handshake

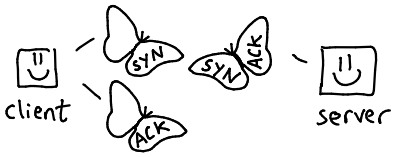
This is what a TCP header looks like:



the "sequence number" lets you assemble packets in the right order ☺

this is the SYN bit

Every TCP connection starts with a "handshake". This makes sure both sides of the connection can communicate with each other.



But what do "SYN" and "ACK" mean? Well! TCP headers have 6 bit flags (SYN, ACK, RST, FIN, PSH, URG) that you can set (you can see them in the diagram.) A SYN packet is a packet with the SYN flag set to 1.

When you see "connection refused" or "connection timeout" errors, that means the TCP handshake didn't finish!

I ran `sudo tcpdump host jvns.ca` in one and `curl jvns.ca` in another. This is some of the output:

```
localhost:51104 > 104.28.6.94:80   Flags [S]
104.28.6.94:80  > localhost:51104  Flags [S.]
localhost:51104 > 104.28.6.94:80   Flags [.]
```

TCP handshake!

jvns.ca IP address

S is for SYN
. is for ACK

LINUX BASICS FOR HACKERS

GETTING STARTED WITH *NETWORKING*, *SCRIPTING*, AND *SECURITY* IN KALI

OCCUPYTHEWEB

no starch press

8

BASH SCRIPTING

Any self-respecting hacker must be able to write scripts. For that matter, any self-respecting Linux administrator must be able to script. Hackers often need to automate commands, sometimes from multiple tools, and this is most efficiently done through short programs they write themselves.

In this chapter, we build a few simple BASH shell scripts to start you off with scripting, adding capabilities and features as we progress, eventually building a script capable of finding potential attack targets over a range of IP addresses.

To become an *elite* hacker, you also need the ability to script in one of the widely used scripting languages, such as Ruby (Metasploit exploits are written in Ruby), Python (many hacking tools are Python scripts), or Perl (Perl is the best text-manipulation scripting language). I give you a brief introduction to Python scripting in Chapter 17.

A Crash Course in BASH

A *shell* is an interface between the user and the operating system that enables you to manipulate files and run commands, utilities, programs, and much more. The advantage of a shell is that you perform these tasks immediately from the computer and not through an abstraction, like a GUI, allowing you to customize your task to your needs. A number of different shells are available for Linux, including the Korn shell, the Z shell, the C shell, and the *B*ourne-*a*gain *sh*ell, more widely known as BASH.

Because the BASH shell is available on nearly all Linux and UNIX distributions (including MacOS and Kali), we'll be using the BASH shell here, exclusively.

The BASH shell can run any system commands, utilities, or applications your usual command line can run, but it also includes some of its *own* built-in commands. Table 8-1 later in the chapter gives you a reference to some useful the commands that reside within the BASH shell.

In earlier chapters, you used the `cd`, `pwd`, `set`, `umask`, `alias`, and `unalias` commands. In this section, you will be adding the `echo` command, which displays messages to the screen, and the `read` command, which reads in data and stores it somewhere else, to your repertoire. Just learning these two tools alone will enable you to build a simple but powerful tool.

You'll need a text editor to create shell scripts. You can use whichever Linux text editor you like best, including vi, vim, emacs, gedit, kate, and so on. I'll be using Leafpad (Figure 8-1) in these tutorials, as I have in previous chapters. Using a different editor should *not* make any difference in your script or its functionality.

Your First Script: "Hello, Hackers-Arise!"

For your first script, we will start with a simple program that returns a message to the screen that says "Hello, Hackers-Arise!" Open your text editor, and let's go.

To start, you need to tell your operating system which interpreter you want to use for the script. To do this, you enter a *shebang*, which is a combination of a hash mark and an exclamation mark, like so:

```
#!
```

You then follow the shebang (`#!`) with `/bin/bash` to indicate that you want the operating system to use the BASH shell interpreter. As you'll see in later chapters, you could also use the shebang to use other interpreters, such as Perl or Python. Here, you want to use the BASH interpreter, so enter the following:

```
#! /bin/bash
```

Next, we enter the `echo` command, which tells the system to simply repeat (or *echo*) back to your monitor whatever follows the command.

In this case, we want the system to echo back to us `"Hello, Hackers-Arise!"`, as done in Listing 8-1. Note that the text or message we want to echo back must be in double-quotation marks.

```
#! /bin/bash

# This is my first bash script. Wish me luck

echo "Hello, Hackers-Arise!"
```

Listing 8-1: Your "Hello, Hackers-Arise!" script

Here, you also see a line that's preceded by a hash mark (`#`). This is a *comment*, which is simply a note you leave to yourself or anyone else reading the code to explain what you're trying to do in the script. Programmers use comments in every coding language. These comments are not read or executed by the interpreter, so you don't need to worry about messing up your code. They are simply text visible only to humans. The BASH shell knows a line is a comment if it starts with the `#` character.

Now, save this file as *HelloHackersArise* with no extension and exit from your text editor.

Setting Execute Permissions

By default, a newly created BASH script is not executable even by you, the owner. Let's look at the permissions on our new file in the command line by using `cd` to move into the directory and then entering `ls -l`. It should look something like this:

```
kali >ls -l
--snip--
-rw-r--r-- 1 root root 42 Oct 22 14:32 HelloHackersArise
--snip--
```

As you can see, our new file has `rw-r--r--` (644) permissions. As you learned in Chapter 4, this means the owner of this file only has read (`r`) and write (`w`) permissions, but no execute (`x`) permissions. The group and all other users have only read permission. We need to modify it to give us execute permissions in order to run this script. We change the permissions with the `chmod` command, as you saw in Chapter 4. To give the owner, the group, and all others execute permissions, you enter the following:

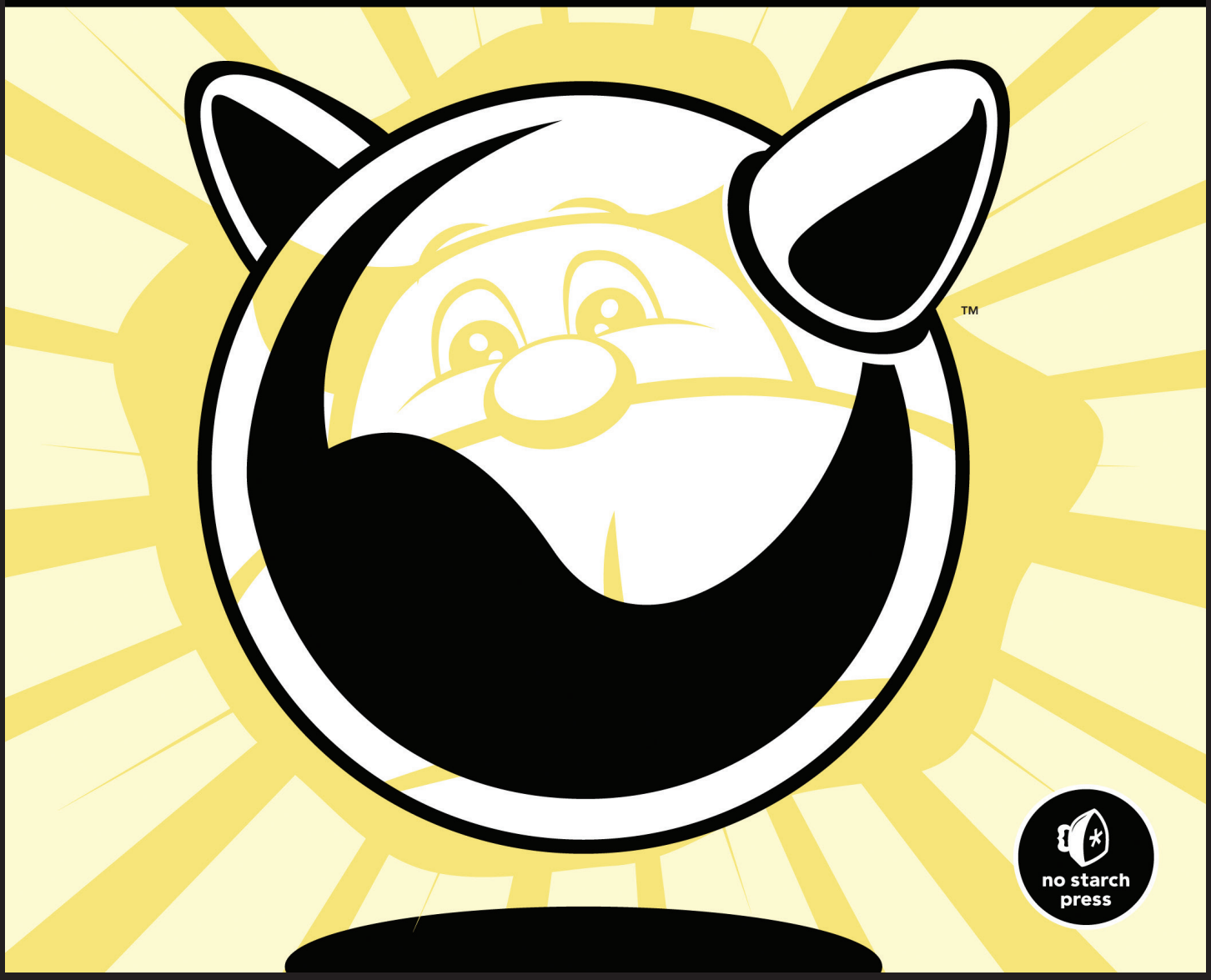
```
kali >chmod 755 HelloHackersArise
```

3RD EDITION

ABSOLUTE FREEBSD®

THE *COMPLETE* GUIDE TO *FREEBSD*

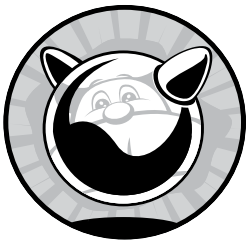
MICHAEL W. LUCAS



no starch press

5

READ THIS BEFORE YOU BREAK SOMETHING ELSE! (BACKUP AND RECOVERY)



The most common cause of system failure is those pesky humans, but hardware and operating systems also fail. Hackers learn new ways to disrupt networks and penetrate applications, and you'll inevitably need to upgrade and patch your system on a regular basis. (Whether or not you *will* upgrade and patch is an entirely separate question.) Any time you touch a system, there's a chance you'll make a mistake, misconfigure a vital service, or otherwise totally ruin your system. Just think of how many times you've patched a computer running any OS and found something behaving oddly afterward! Even small system changes can damage data. You should, therefore, always assume that the worst is about to happen. In our case, this means that if either the hardware or a human being destroys the data on your hard drive, you must be able to restore that data.

We'll start with system backups and managing tape drives using tar(1) and then review recording system behavior with script(1). Finally, should you suffer a partial or near-total disaster, we'll consider recovering and rebuilding with single-user mode and the install media.

System Backups

You need a system backup only if you care about your data. That isn't as inane as it sounds. The real question is, "How much would it cost to replace my data?" A low-end tape backup system can run a few hundred dollars. How much is your time worth, and how long will it take to restore your system from the install media? If the most important data on your hard disk is your web browser's bookmarks file, a backup system probably isn't worth the investment. But if your server is your company's backbone, you'll want to take this investment very seriously.

Online backups can easily be damaged or destroyed by whatever ruins the live server. Proper backups are stored safely offline. Tools like rsync(1), and even ZFS replication, don't create actual backups; they create convenient online copies.

A complete backup and restore operation requires a tape drive and media. You can also back up to files, across the network, or to removable media, such as CDs or DVDs. Many people use removable multiterabyte hard drives connected via USB 3 for backups. Despite our best efforts, tape is still an important medium for many environments.

Backup Tapes

FreeBSD supports SCSI and USB tape drives. SCSI drives are the fastest and most reliable. USB tape drives are not always standards-compliant and hence not always compatible with FreeBSD. Definitely check the release notes or the FreeBSD mailing list archives to confirm that your tape drive is compatible with FreeBSD.

Once you've physically installed your tape drive, you need to confirm that FreeBSD recognizes it. The simplest way is to check the */var/run/dmesg.boot* file for *sa* devices (see Chapter 4). For example, the following three lines from *dmesg.boot* describe the SCSI tape device in this machine:

```
❶ sa0 at mps0 bus 0 ❷ scbus0 ❸ target 3 lun 0
sa0: ❹ <QUANTUM ULTRIUM 5 3210> Removable Sequential Access SPC-4 SCSI device
sa0: Serial Number HU1313V6JA
sa0: ❺ 600.000MB/s transfers
sa0: Command Queueing enabled
```

Of all the information we have on this tape drive, the most important is that your FreeBSD system knows this device as sa0 ❶. We also see that it's attached to the SCSI card mps0 ❷ at SCSI ID 3 ❸, and we see the drive's model number ❹ as well as the fact that it can run at 600MB per second ❺.

Tape Drive Device Nodes, Rewinding, and Ejecting

Tape is a linear storage medium. Each section of tape holds a particular piece of data. If you back up multiple chunks of data to tape, avoid rewinding after each backup operation. Imagine that you wrote a backup of one system to

tape, rewound the tape, and backed up another system. The second backup would overwrite the first because it used the same chunk of tape. When you run multiple backups on a single tape, use the appropriate device node to ensure you don't rewind the tape between tasks.

As with many Unix devices with decades worth of history, the way you access a tape drive controls how it behaves. Tape drives have several different device nodes, and each one makes the tape drive behave differently. The most basic tape-control mechanism is the device node used to access it. Normal tape drives have three nodes: */dev/esa0*, */dev/nsa0*, and */dev/sa0*.

Tapes are sequential access devices, meaning that data is stored on the tape linearly. A particular section of tape contains certain data, and to access that data, you must roll the tape to expose that section. To rewind or not to rewind is an important question.

NOTE *The behavior of different tape device nodes varies between operating systems. Different versions of Unix, with different tape management software, handle tapes differently. Do not make assumptions with your backup tapes!*

If you use the node name that matches the device name, the tape drive will automatically rewind when your command finishes. Our sample SCSI tape drive has a device name of sa0, so if you run a command using */dev/sa0* as the device node, the tape will rewind when the command finishes.

If you don't want the tape to automatically rewind when the command completes, stop it from rewinding by using the node name that starts with *n*. Perhaps you need to append a second backup from a different machine onto the tape or you want to catalog the tape before rewinding and ejecting. In our example, use */dev/nsa0* to run your command without rewinding.

To automatically eject a tape when a command finishes, use the node that begins with *e*. For example, if you're running a full system backup, you probably want the tape to eject when the command finishes so the operator can put the tape in a case to ship offsite or place in storage. Our example uses the */dev/esa0* device name to eject the tape when the command finishes. Some tape drives might not support automatic ejection; they'll require you to push the physical button to work the lever that winches the tape out of the drive. The easiest way to identify such a drive is to try to eject it via the device node and see what happens.

The $TAPE Variable

Many programs assume that your tape drive is */dev/sa0*, but that isn't always correct. Even if you have only one tape drive, you might want to eject it automatically (*/dev/esa0*) or not to rewind it upon completion (*/dev/nsa0*).

Many (but not all) backup-related programs use the environment variable `$TAPE` to control which device node they use by default. You can always override `$TAPE` on the command line, but setting it to your most commonly used choice can save you some annoyances later.

2ND EDITION

AUTOTOOLS

A PRACTITIONER'S GUIDE TO GNU AUTOCONF, AUTOMAKE, AND LIBTOOL

JOHN CALCOTE

no starch press

1

AN END USER'S PERSPECTIVE ON THE GNU AUTOTOOLS

I am not afraid of storms, for I
am learning how to sail my ship.
—Louisa May Alcott, Little Women

If you're not a software developer, either by trade or by hobby, you may still have a need or desire at some point to build open source software to be installed onto your computer. Perhaps you're a graphic artist who wishes to use the latest version of GIMP, or maybe you're a video enthusiast and you need to build a late version of FFmpeg. This chapter, therefore, may be the only one you read in this book. I hope this is not the case, for even a power user can gain so much more by striving to understand what goes on under the covers. Nevertheless, this chapter is designed for you. Here we'll discuss what to do with that so-called *tarball* you downloaded from that project web site. We'll use the Autoconf package to illustrate how you might do this yourself for any package you download from a project web site.

If you are a software developer, there's a good chance the material in this chapter is too basic for you; I'd recommend skipping right to the next chapter where we'll jump into a more developer-centric discussion of the Autotools.

Software Source Archives

Open source software is distributed as single-file source archives containing the source and build files necessary to build the software on your system. Linux distributions remove much of the pain for end users by pre-building these source archives and packaging the built binaries into installation packages ending in extensions like *.rpm* (Redhat based systems) and *.deb* (Debian/Ubuntu based systems). Installing software using your system package manager is easy, but sometimes you need the latest feature set of some software and it hasn't yet been packaged for your distro. When this happens, you need to download the source archive from the project web site and then build and install it yourself:

```
$ wget https://ftp.gnu.org/gnu/autoconf/autoconf-2.68.tar.gz
```

Source archive names generally follow a defacto-standard format supported by the Autotools. Unless the project maintainer has gone out of his way to modify this format, the Autotools will automatically generate a source archive file named according to the following template: *pkgname-version.format*. Here, *pkgname* is the short name of the software, *version* is the version of the software and *format* represents the archive format, or file extensions. The *format* portion may contain more than one period, depending on the way the archive was built. For instance, *.tar.gz* represents two encodings in the format: a tar archive that's been compressed with the gzip utility, as is the case with the Autoconf source archive:

```
$ ls -1
autoconf-2.69.tar.gz
automake-1.15.1.tar.gz
gettext-0.19.8.1.tar.gz
libtool-2.4.6.tar.gz
pkg-config-0.29.2.tar.gz
```

Unpacking a Source Archive

By convention, source archives contain a single root directory as the top-level entry. You should feel safe unpacking a source archive to find only a single new directory in the current directory, named the same as the archive file, minus the *format* portion. Source archives packaged using Autotools build systems never unpack the contents of the original top-level directory into the current directory.

Nevertheless, occasionally, you'll download an archive and unpack it to find dozens of new files in the current directory. It's therefore prudent to unpack a source archive of unknown origin into a new empty sub-directory. You can always move it up a level if you need to.

Source archives can take many shapes, each ending in a unique file extension: *.zip*, *.tar*, *.tar.gz* (or *.tgz*), *.tar.bz2*, *.tar.xz*, *tar.Z*, and so on. The files

contained in these source archives are the source code and build files used to build the software. The most common of these formats are *.zip*, *.tar.gz* (or *.tgz*) and *.tar.bz2*.

Zip files use compression techniques developed decades ago by Phil Katz on Microsoft DOS systems as a proprietary multi-file compressed archive format. The Zip format was released into the public domain and versions have been written for Microsoft Windows and Linux as well as other Unix-like operating systems. Late versions of Windows can unpack a *.zip* file merely by right clicking on it in Windows Explorer and selecting an **Extract** menu option. The same is true of the Nemo file browser on Linux Gnome desktops.

Zip files can be unpacked at the Linux command line using the more or less ubiquitous unzip program[1]:

```
$ unzip some-package.zip
```

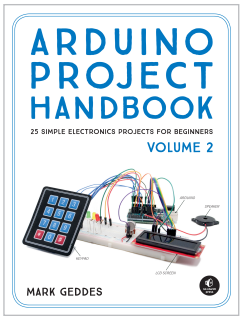
Zip files are most often intended by project maintainers to be used on Microsoft Windows systems. A much more common format used on Linux platforms is the compressed tar file. The name *tar* comes from *tape archive*. The tar utility was originally designed to stream the contents of on-line storage media, such as hard disk drives, to more archival storage formats, such as magnetic tape. Because it's not a random-access format, magnetic tape doesn't have a hierarchical file system. Rather, data is written to tape in one long string of bits, with these archive files appended end to end. To find a particular file on tape, you have to read from the beginning of the tape through to the file you're interested in. Hence, it's better to store fewer files on tape to reduce search time.

The tar utility was designed to convert a set of files in a hierarchical file system into just such a long string of bits--an archive. The tar utility was specifically *not* designed to compress this data in a manner that would reduce the amount of space it takes up, as are other utilities to do that sort of thing, and a founding Unix principle is that of a single-responsibility per tool. In fact, a *.tar* file is usually slightly larger than the sum of the sizes of the files it contains because of the overhead of storing the hierarchy, names and attributes of the archived files.

Occasionally, you'll find a source archive that ends only in a *.tar* extension. This implies that the file is an uncompressed tar archive. More often, however, we see extensions such as *.tar.gz*, *.tgz* or *.tar.bz2*. These are compressed tar archives. An archive is created from the contents of a directory tree using the tar utility, and then the archive is compressed using the gzip or bzip2 utilities. A file with an extension of *.tar.gz* or *.tgz* is a tar archive that

1. Reading the man page, you might get the impression that the gunzip utility can handle *.zip* files as well as *.gz* files, but this feature is intended only to convert *.tar.zip* files into *.tar.gz* files. Essentially, the gunzip utility cannot understand compressed archives that contain more than one file. If you do have a *.tar.zip* file, you can uncompress it to a *.tar* file using a command like, `gunzip < file.tar.zip`. It doesn't recognize the *.zip* extension, so piping it from stdin is the only way to get it to work.

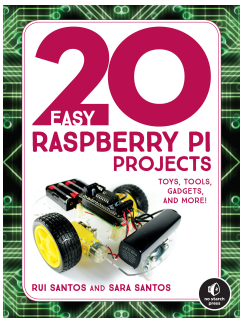
COMING SOON FROM NO STARCH PRESS!



ARDUINO PROJECT HANDBOOK, VOLUME 2

25 SIMPLE ELECTRONICS PROJECTS FOR BEGINNERS

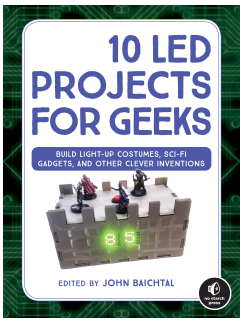
by Mark Geddes
August 2017, 272 pp, $24.95
ISBN-13: 978-1-59327-818-2



20 EASY RASPBERRY PI PROJECTS

TOYS, TOOLS, GADGETS, AND MORE!

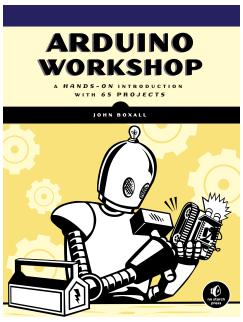
by Rui Santos and Sara Santos
April 2018, 288 pp, $24.95
ISBN-13: 978-1-59327-843-4



10 LED PROJECTS FOR GEEKS

BUILD LIGHT-UP COSTUMES, SCI-FI GADGETS, AND OTHER CLEVER INVENTIONS

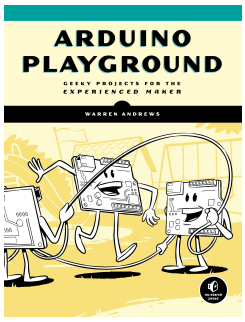
by John Baichtal
July 2018, 240 pp, $24.95
ISBN-13: 978-1-59327-825-0



ARDUINO WORKSHOP

A HANDS-ON INTRODUCTION WITH 65 PROJECTS

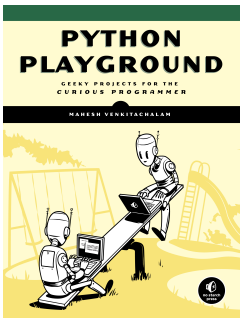
by John Boxall
May 2013, 392 pp, $29.95
ISBN-13: 978-1-59327-448-1



ARDUINO PLAYGROUND

GEEKY PROJECTS FOR THE EXPERIENCED MAKER

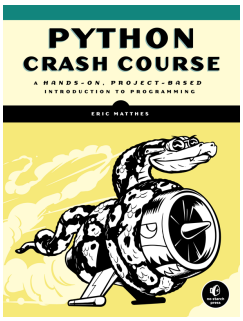
by Warren Andrews
March 2017, 344 pp, $29.95
ISBN-13: 978-1-59327-744-4



PYTHON PLAYGROUND

GEEKY PROJECTS FOR THE CURIOUS PROGRAMMER

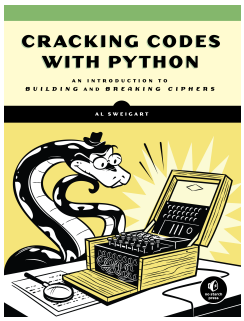
by Mahesh Venkitachalam
October 2015, 352 pp, $29.95
ISBN-13: 978-1-59327-604-1



PYTHON CRASH COURSE

A HANDS-ON, PROJECT-BASED INTRODUCTION TO PROGRAMMING

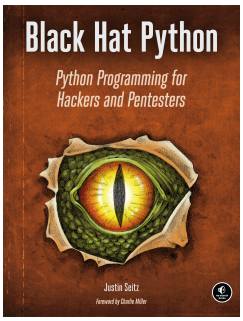
by Eric Matthes
November 2015, 560 pp, $39.95
ISBN-13: 978-1-59327-603-4



CRACKING CODES WITH PYTHON

AN INTRODUCTION TO BUILDING AND BREAKING CIPHERS

by Al Sweigart
January 2018, 416 pp, $29.95
ISBN-13: 978-1-59327-822-9

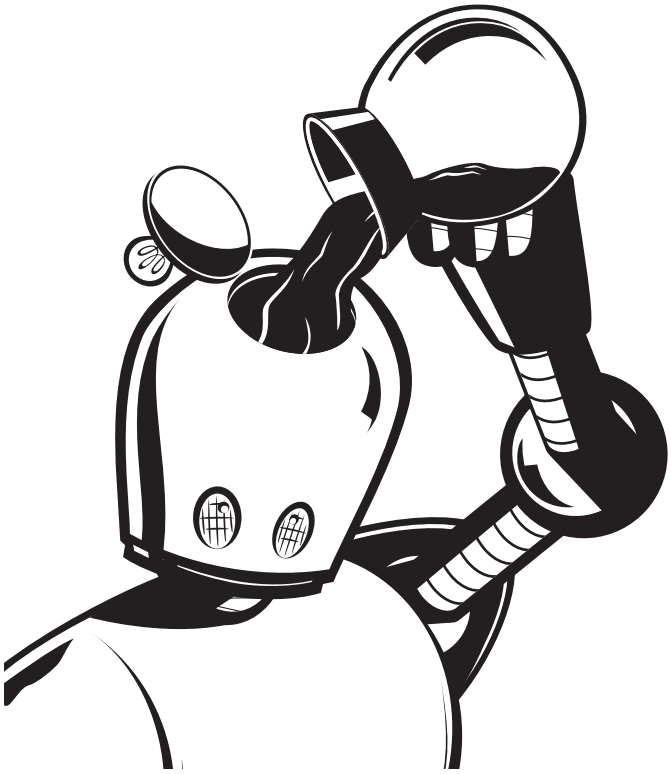


BLACK HAT PYTHON

PYTHON PROGRAMMING FOR HACKERS AND PENTESTERS

by Justin Seitz
December 2014, 192 pp, $34.95
ISBN-13: 978-1-59327-590-7

no starch press



Founded in 1994, No Starch Press is one of the few remaining independent technical book publishers. We publish the finest in geek entertainment—unique books on technology, with a focus on open source, security, hacking, programming, alternative operating systems, and LEGO. Our titles have personality, our authors are passionate, and our books tackle topics that people care about.

VISIT WWW.NOSTARCH.COM
FOR A COMPLETE CATALOG

no starch press