by Peter Collinson, **Hillside Systems**

Just what is that File?

Whenever I get a new computer system, I spend some time poking around the file system to see what is there. Of course, one of life's universal rules applies: looking is generally fine, touching needs to be discouraged. A long time ago, I learned that running commands you just happen to find lying around can cause huge amounts of grief. It's considerably more dangerous to run an unknown command as superuser, so don't ever do that. "Danger, Will Robinson!"

Incidentally, if you are a new user of a free UNIX system of whatever flavor, be sure to set up a user account for yourself that doesn't have superuser privilege. You should only log on as `root` in extremis; it's better to get into the habit of logging on as a normal user and using the `su` command to obtain unprotected privilege only when needed.

That being said, it seems to me there are an increasing number of commands and files on UNIX systems that are unexplained and mysterious. You'll find them scattered around the system and, occasionally, by looking at output from the `ps` command. It used to be that all (maybe only nearly all) commands had associated manual pages, so if you found a command called `doit`, there was a good chance that `man doit` would tell you what the command was. If the `man` command failed, the command was perhaps lurking quietly on an associated manual page and it was possible to use `grep` to search through the manual page sources with the hope of finding something sensible. Well, my Solaris system certainly seems to possess some "secret" commands, whose purposes are barely discernible. Just what does `/sbin/bpgetfile` do? I think the lack of documentation is a retrograde step.

On Linux, you are presented with at least three sources of basic information: manual pages, the GNU info system and the Linux `HOWTO` documentation. Having to look in three places is not helpful. Attempts are being made to create one single coherent set of documentation, or at least have a unified interface to the disparate document sources. The newly announced GNOME desktop has one GUI that accesses the various information sources. It's still not possible to search for some random text, and I find this annoying. The ability to `grep` through manual pages is a great boon and can save a great deal of time.

Looking at the File

OK, so you've found a mysterious file and want to know something about it. What do you do? Well, you may be able to deduce something about its contents from its file name. This is not guaranteed because UNIX itself cannot enforce any rules on how a file is named. Nevertheless, files are often named using standard suffixes. For example, source code for the C language will be found in a file that ends in `.c`. We are all kept in line following this convention because the C compilation command `cc` (or `gcc`) recognizes the standard suffixes and processes files differently based on their suffix. However, there's nothing to stop me from placing a shell script into a file called `script.c` and happily executing the script. So, the file name may not really tell you much about the file on a UNIX system.

We really need to look at the contents of our mysterious file to be sure what it contains. UNIX supplies the `file` command, which opens files and tries to tell us what they contain. Of course, you need to have appropriate permissions to read the mysterious file and, actually, most UNIX files have their permissions set to allow reading by all. In the example below, I ran the `file` command on my Solaris system in a directory where I have collected various files for demonstration purposes:

```
$ file *
core: ELF 32-bit MSB core file SPARC
       Version 1, from 'cat'
dir:  directory
ix:   English text
ls:   ELF 32-bit MSB executable SPARC
       Version 1, dynamically linked, stripped
pdficon.gif:    GIF file, v87
typescript:     ascii text
```

I've folded some of the lines for printing purposes. The `file` command examines each of the files you give it on the command line. Each file is subjected to a sequence of tests and a message is printed guessing the contents of the file.

If the file program cannot locate a magic number it knows, then it uses a set of heuristic tests to check on the file.

The first set of tests uses standard UNIX system calls to see what can be determined about the file. UNIX provides the `stat` system call that returns a file's *metadata*, the information the system holds about each file. The metadata contains the file's owner, its access permissions, modification time information and size. The file type is stored as part of the file permissions. UNIX doesn't have many different file types; the types stored are only those that are used internally by the kernel to distinguish a plain file from a directory, or a symbolic link from a file that's used to access a peripheral and so on. In the above example, `file` tells us that `dir` is a directory by looking at the file's metadata.

The second set of tests in the `file` command looks for magic numbers in the file. Many file formats, both text and binary, contain a magic value at their start that is used in the normal course of processing to check that it does contain correct information. For example, a file holding a document in Adobe Systems Inc.'s Portable Document Format (PDF) starts with the string `%PDF`, which allows the Acrobat reader to refuse to play when supplied with any other file format. Actually, `%PDF` is followed by a hyphen and a PDF version number, which enables Adobe to alter the way the PDF protocol works as time progresses. The Acrobat program can look at the version number and say: "OK, this version did *this* in such and such a way." The Acrobat reader is also able to announce its own demise: "Sorry, the file contains a format I don't understand, please get a new version of Acrobat."

Anyway, there are a great number of these telltale magic numbers used by the `file` program to deduce the basic contents of the file. In the original versions of `file`, the magic values were compiled into the binary and could not be altered. These days, you'll find them in a text file called `/etc/magic` (the actual position of the file may vary from system to system; it's in `/etc` on Solaris). Solaris has a manual page that describes the format of the file, so you can add in the knowledge of your local files should you wish to.

If a magic value is found in the target file, then the `file` program may continue to obtain information about the file using some code that knows about specific file contents. For example, most versions of `file` will tell you the name of the command that dumped core and created a `core` file. Incidentally, I induced the `cat` command to generate the `core` file in the example above. The `cat` program didn't crash and burn.

Executable programs use several magic numbers to indicate their type, which are understood by the kernel when the program is launched. The `file` program will decode the binary header and tell you not just the type of program, but will supply other information as well. Perhaps the program is "dynamically" linked, meaning it pulls some portions of its code from a shared library at the start of its run. The `file` command will tell you whether the file containing a program has been "stripped," which means its symbol table has been removed, making the file considerably smaller. The symbol table is useful when debugging the program, it allows debuggers to find the value of named variables in the program. The information is not really relevant when the command is "in production use."

If the `file` program cannot locate a magic number it knows, then it uses a set of heuristic tests to check on the file. These tests are used to tell us that the `ix` file in my example is "English text" and the `typescript` file is "ASCII text." Actually, the `ix` file contains HTML in English, and the `typescript` file is the output from the `script` program, logging a terminal session. It contains several nonprinting characters, like the backspaces that I typed when I deleted some text on the input line.

The `file` program does a fairly good job. At least, it can stop you loading binary rubbish down to the screen. Random binary data that is sent to a screen can have bad effects because the data can contain sequences the terminal (or terminal emulator) interprets as programming information. I find my `xterm` windows hang occasionally because I've managed to `cat` a binary file at the screen.

The strings Command

Well, we've looked at our mysterious file and discovered it's a program or a text file that contains what looks like configuration data. What next? Well, something must be using the file, and that something could be a compiled program. On

Linux, or other source-full systems, you can search the source with `grep` looking for the file name; assuming, of course, that the file name is significantly distinctive, allowing the string to stand out from any normal file contents.

However, on a binary-only system, like Solaris, things are a little more complicated, and we need to employ a different strategy. We can use the `strings` command to scan files looking for C-language text strings. When programming in C, any string (such as a file name) will be compiled into the binary and stored in contiguous bytes terminated by a byte containing a null character (a character of value zero). The `strings` command makes use of this knowledge by scanning any file that is supplied on its command line looking for a minimum of four contiguous ASCII characters terminated by a null byte or a newline indicator. The four-byte minimum is imposed because sometimes machine code instructions can appear to be a legal string, and the printing of these sequences will get in the way of any real strings the program may contain. An option to the `strings` program allows you to control the number count.

Printing the output from a `strings` command for publication is likely to fill loads of column inches and tell you little. I suggest you run the command and see the full gory details for yourself. Sometimes, the raw output will tell you things about the mysterious file because the file will contain help strings or error messages. When looking for something specific, I usually run the output from `strings` through `grep`. Here, I am looking at the `sendmail` binary to see where it expects to find its control file, `sendmail.cf`:

```
$ strings /usr/lib/sendmail | grep sendmail.cf
sendmail.cf
/etc/mail/sendmail.cf
```

This `sendmail` binary contains `/etc/mail/sendmail.cf`, so there's a good chance that's where it will look for its configuration file.

I used the `strings` command a few years back to look at Year 2000 compliance on several machines. The standard C routine that translates time in seconds from January 1, 1970, into a structure that holds the day, month, year and time of day, delivers only the last two digits of the year. When UNIX was invented, many programmers printed the year from programs using a constant string in a print statement:

`"19%d"` *or*
`"19%2d"` *or*
`"19%02d"`

which made the printing code output "19" followed by the number of the year (`%d`), or the number of the year as two digits (`%2d`) or, more properly, the number of the year as two digits ensuring that any leading zero digit is printed as a zero and not a space (`%02d`). Well, as we approach 2000, we've hopefully learned to add "1900" to the year value to obtain the correct date, and the print statements have changed to print the number as four digits:

`"%d"` *or*
`"%4d"`

Several years back, when I scanned binaries of a couple of systems looking for strings that contained `19%`, I found only a couple of commands that contained old code. They are fixed now in recent releases, but it was illuminating to find how few commands did have this problem.

The scanning script for such an exercise is easy to construct. If we change into the `bin` directory, we can type

```
$ cd /usr/bin
$ ls | while read name
do
    echo $name
    strings $name | grep '19%'
done
```

Here, we use the `ls` command to generate a list of the file names and this is piped into a loop. Incidentally, you can't pipe program results into loops in `csh`, which is one very compelling reason to use a Bourne shell variant when programming. The loop reads the names one at a time into the name variable. For each name, we first `echo` it so that we can see what we are processing. We then run `strings` on the file and pass the output into `grep` looking for the appropriate string. We end up with a list of file names on the standard output. If we find anything in a file, the matched string will appear after the file name.

This is a quick hack, and is fine if there are only a small number of files being scanned. It's about the level of complexity where the code is easy to type into the shell and to get a result. Ideally, we'd like to have output only for the files that contain the string we are looking for. The script becomes slightly more complicated, and perhaps pushes the code to a point where you might think about putting it into a command file. Here's the more selective version:

```
$ cd /bin
$ ls | while read name
do
   $op = $(strings $name | grep '19%')
   if [ $op != "" ]
   then
       echo "$name"
       echo "$op"
   fi
done
```

The inner loop runs the `strings` ... `grep` command sequence as before, but captures any output into a shell variable using the POSIX `$(...)` syntax that replaced the previous (unnestable) backquote mechanism. We now test whether we've found anything by looking at the result and only proceeding further if the result is non-null. I've quoted the arguments to the echo commands to preserve any newlinesor spaces in the values.

We can extend this command sequence to search for all binary programs on a file system, by replacing the ls command in the scripts above with

```
find / -type f -exec file {} \; |
    grep ELF |
    awk -F: '{print $1}' |
    while read name
```
The commands as above

The find command is one of the delights of UNIX, and it is worth getting to grips with its somewhat arcane syntax. It scans a file system tree applying a set of tests to all the files it finds. In the above example, it says: start at the root of the file system (/) and look for all regular files (-type f). When one is found, the command between the -exec and the semicolon is run. The backslash immediately before the semicolon is used to tell the shell to pass the semicolon into the find command; otherwise, the shell would see it and think it was an end-of-statement separator. The curly braces in the argument to the -exec option are replaced by the name of the file. The effect is to run the file command on every regular file in the file system.

The output we obtain from the find command is the same as in my example at the beginning of the article. We now select only the executable files by using grep to choose the output lines that contain the word ELF. Having found the files we want, we need to reduce each output line so that it contains just the file name.

There are several different ways to reduce the line. I've chosen to use awk because the file command adds a colon after each file name and we can use this as a column-separator character. So, I tell awk to split its input into columns using the colon character (-F:). I know then that the first column will contain the file name I want. The rest of the awk command prints the first column '{print $1}', and we now have a stream of file names that can be processed by the scanning code.

The scanning code I've outlined has many uses. I once used the technique to find the program that was writing a mysterious message to a log file. The message wasn't an obvious system error, and there was a good chance of finding the culprit and sorting out its problems by looking for the string in the binary.

Follow-Up

I'd like to finish this month with some observations sent to me by several readers via email in reaction to my article entitled "Automation" (*S/W Expert*, April 1999, Page 26, http://sw.expert.com/C2/SE.C2.APR.99.pdf). One of the things I discussed in that article was the shell's PATH variable, and I showed you how you could create a series of directories to be searched when a command is typed into the shell. My correspondents complained that I had put the current working directory into the example path. I still do this, so I can create commands in the current directory and execute them without having to explicitly say $ ./command.

However, putting the current working directory in your search path can be dangerous in some environments, because it allows an unscrupulous person to place their own version of a common command in a directory and fool you into running it. A favorite student trick when I was running academic systems was to send mail saying, "Look at this neat command fggxu in my directory." You would use cd to move into the directory, type fggxu and get a "Command not found" message. You would then type ls without thinking, and find that the command was fgxu. It was probably neat, too. However, what you didn't spot was the private version of the ls command that was lurking in the directory and, of course, you'd run that command when you typed ls. The private ls was run as you, with your permissions at the time, and could do a variety of things ranging from funny to nasty.

The potential for running a fake or Trojan command is the reason why you may not want to place the current working directory into your PATH variable. Doing so adds a security hole to your environment, meaning that any standard command you are trying to run may not be the command you think it is.

One way to avoid the problem is to never allow the shell to execute a command from the current working directory by ensuring that your PATH variable doesn't contain it. Another way is to place the current working directory at the end of the chain of directories, so the common commands will be found and used before any fake ones.

Whether this is likely to happen to you is dependent on your environment. You need to take a view on this, as lawyers say.

***Peter Collinson** runs his own UNIX consultancy, dedicated to earning enough money to allow him to pursue his own interests: doing whatever, whenever, wherever... He writes, teaches, consults and programs using Solaris running on a SPARCstation 2. Email:* pc@cpg.com.