# Speaking UNIX: The best-kept secrets of UNIX power users

Skill Level: Intermediate

Martin Streicher
Software Developer
Pixel, Byte, and Comma

25 May 2010

You don't have to break into a Watergate office to uncover the secrets of UNIX® power users. There's an informant, and this month he's spilling the beans.

If you're wondering why I'm wearing dark sunglasses, a fake moustache, and a baseball cap (featuring the logo of professional curling team, The Floating Stones), I'm on the lam. I'm dodging black remote-controlled helicopters, pasty-white systems administrators, and the combined forces of many daemons to bring you some of the best-kept secrets of UNIX® power users. Don your aluminum foil hat and read on.

## Save the environment variables

Most UNIX users amass settings in shell startup files, such as .bashrc (for the Bash shell) and .zshrc (for the Z shell), to recreate a preferred shell environment time and again. Startup files can create aliases, set shell options, create functions, and set environment variables. Essential environment variables include HOME (which points to your home directory), PATH (which enumerates directories in which to search for applications), and MANPATH (which lists directories in which to search for man pages). To see which environment variables are set in your shell, type `printenv`. Consult your shell's man page for a complete list of available environment variables.

Like the shell, you can customize many other UNIX applications through environment variables. For example, the Java™ subsystem requires that you define JAVA_HOME to point to the root of a Java run time. Similarly, the suite of Amazon Web Services (AWS) utilities mandates the use of AWS_CREDENTIAL_FILE to point to a file with valid private key credentials. Individual applications provide environment variables, too. The trick is discovering the variables. Luckily, the work need not involve breaking and entering; instead, simply consult the man page of the utility at hand and look for a separate section titled, "Environment Variables."

For example, the pager utility `less` defines a number of useful environment variables:

- The environment variable LESS stores command-line options, reducing what you type each time you invoke the pager. For instance, if you read a good number of log files, put the following in a shell startup file:

```
export LESS='--RAW-CONTROL-CHARACTERS
--squeeze-lines --ignore-case'
```

  The options interpret control characters (usually syntax coloring), reduce runs of blank lines to a single line, and ignore case in string matches, respectively. If you work with code, try these options:

```
export LESS='--LINE-NUMBERS
--quit-if-one-screen --quit-on-intr'
```

- The environment setting named LESSKEY points to a file of key bindings. You can use key bindings to customize the behavior of `less`, say, to match the behavior of another pager or editor.
- Like the shell, `less` can maintain history between invocations. Set LESSHISTFILE and LESSHISTSIZE to point to a file of persist commands and to set the maximum number of commands to record, respectively.

Another good application of environment variable can be found in the GNU Compiler Collection (GCC). GCC defines a variety of environment variables to customize its operation. LIBRARY_PATH, as its name implies, is a list of directories to search for libraries to link to; COMPILER_PATH works much like the shell's PATH but is used internally by GCC to find subprograms used during the compilation process.

If you write code and build binaries for a single platform, you may never use these environment variables, However, if you cross-compile the same code for a number of platforms, such variables are essential for accessing the varied headers and libraries for each platform. You might set variables to one collection of values for one kind of machine and set the values to another collection for a system of an alternate flavor.

In fact, you can take a cue from GCC: Maintain many sets of environment variables per application and switch from one pool to another depending on the work at-hand.

One approach is to keep an environment initialization file in each project directory and `source` it as needed. For instance, many Ruby developers use such a solution to switch between Ruby versions, changing environment variables PATH, GEM_HOME, and GEM_PATH as needed to hop from one version to another.

## Dotting the landscape

Much like environment variables, many Linux® and UNIX applications provide a *dot* file—a small file whose name begins with a period—for customizations. However, unlike environment variables that capture a handful of flags or a relatively tiny amount of information, a dot file can be much more extensive, even complex, with its own peculiar syntax rules and even its own programming language. A dot file is a convenient place to keep options and settings, because (per UNIX heritage) file names that begin with a dot do not appear in a standard directory listing. (Use `ls -a` to see these so-called *hidden files*.) Except for its special name, a dot file is a plain text file.

A dot file is usually found in your home directory, but some utilities look for a dot file in the present working directory, too. If an application supports more than one dot file, the program typically applies precedence rules to favor one file over another. In general, a "local" dot file—one found in the current working directory—has highest precedent, followed by one in your home directory and then by a system-wide configuration. None, one, or all of these files can exist, and it's up to the application to treat the files as mutually exclusive or incremental. In the former case, the first dot file found in the precedent chain is definitive. In the latter case, the configuration might cascade or be reconciled into a final result.

An example of a simple dot file is `less`'s key bindings file, located in $HOME/.lesskey. Each line in the file is a pair (a keystroke and a command) resembling something like the snippet below:

```
\r        forw-line
\n        forw-line
e         forw-line
j         forw-line
^E        forw-line
^N        forw-line
k         back-line
y         back-line
^Y        back-line
```

At the other extreme, consider `fetchmail`. The utility picks up e-mail from multiple remote sources and delivers the messages locally. The operation of the utility is controlled solely through $HOME/.fetchmailrc. (See the man page for its many options.) `cron`, `git`, `vi`, and many other commands recognize dot files, too. Again, read the man page for the application to learn what you configure in a dot file. Some dot files are rich enough to warrant a separate man page, such as `crontab`.

## Shhh . . . secrets about SSH

Secure Shell (SSH) is a rich subsystem used to log in to remote systems, copy files, and tunnel through firewalls—securely. Since SSH is a subsystem, it offers plenty of options to customize and streamline its operation. In fact, SSH provides an entire "dot directory", named $HOME/.ssh, to contain all its data. (Your .ssh directory must be mode 600 to preclude access by others. A mode other than 600 interferes with proper operation.) Specifically, the file $HOME/.ssh/config can define lots of shortcuts, including aliases for machine names, per-host access controls, and more.

Here is a typical block found in $HOME/.ssh/config to customize SSH for a specific host:

```
Host worker
HostName worker.example.com
IdentityFile ~/.ssh/id_rsa_worker
User joeuser
```

Each block in ~/.ssh/config configures one or more hosts. Separate individual blocks with a blank line. This block uses four options: `Host`, `HostName`, `IdentityFile`, and `User`. `Host` establishes a nickname for the machine specified by `HostName`. A nickname allows you to type `ssh worker` instead of `ssh worker.example.com`. Moreover, the `IdentityFile` and `User` options dictate how to log in to `worker`. The former option points to a private key to use with the host; the latter option provides the login ID. Thus, this block is the equivalent of the command:

```
ssh joeuser@worker.example.com -i ~/.ssh/id_rsa_worker
```

A powerful but little-known option is `ControlMaster`. If set, multiple SSH sessions to the same host share a single connection. Once the first connection is established, credentials are not required for subsequent connections, eliminating the drudgery of typing a password each and every time you connect to the same machine. `ControlMaster` is so handy, you'll likely want to enable it for every machine. That's accomplished easily enough with the host wildcard, `*`:

```
Host *
ControlMaster auto
ControlPath ~/.ssh/master-%r@%h:%p
```

As you might guess, a block tagged `Host *` applies to every host, even those not explicitly named in the config file. `ControlMaster auto` tries to reuse an existing connection but will create a new connection if a shared connection cannot be found. `ControlPath` points to a file to persist a control socket for sharing. `%r` is replaced by the remote login user name, `%h` is replaced by the target host name, and `%p` stands in for the port used for the connection. (You can also use `%l`; it is replaced with the local host name.) The specification above creates control sockets with file names akin to:

```
master-joeuser@worker.example.com:22
```

Each control socket is removed when all connections to the remote host are severed. If you want to know which machines you are connected to at any time, simply type `ls ~/.ssh` and look at the host name portion of the control socket (%h).

The SSH configuration file is so expansive, it too has its own man page. Type `man ssh_config` to see all possible options. And here's a clever SSH trick: You can tunnel from a local system to a remote one via SSH. The command line to use looks something like this:

```
$ ssh example.com -L 5000:localhost:3306
```

This command says, "Connect via example.com and establish a tunnel between port 5000 on the local machine and port 3306 [the MySQL server port] on the machine named 'localhost.'" Because `localhost` is interpreted on example.com as the tunnel is established, `localhost` is example.com. With the outbound tunnel—formally called a *local forward*—established, local clients can connect to port 5000 and talk to the MySQL server running on example.com.

This is the general form of tunneling:

```
$ ssh proxyhost localport:targethost:targetport
```

Here, *`proxyhost`* is a machine you can access via SSH and one that has a network connection (not via SSH) to *`targethost`*. *`localport`* is a non-privileged port (any unused port above 1024) on your local system, and *`targetport`* is the port of the service you want to connect to.

The previous command tunnels *out* from your machine to the outside world. You can also use SSH to tunnel *in,* or connect to your local system from the outside world. This is the general form of an inbound tunnel:

```
$ ssh user@proxyhost -R proxyport:targethosttargetport
```

When establishing an inbound tunnel—formally called a *remote forward*—the roles of *`proxyhost`* and *`targethost`* are reversed: The target is your local machine, and the proxy is the remote machine. *`user`* is your login on the proxy. This command provides a concrete example:

```
$ ssh joe@example.com -R 8080:localhost:80
```

The command reads, "Connect to example.com as joe, and connect the remote port 8080 to local port 80." This command gives users on example.com a tunnel to Joe's machine. A remote user can connect to 8080 to hit the Web server on Joe's machine.

In addition to `-L` and `-R` for local and remote forwards, respectively, SSH offers `-D` to create an HTTP proxy on a remote machine. See the SSH man page for the proper syntax.

## Rewriting with history

If you spend a lot of time at the shell prompt, recording shell history can save time and typing. But there are a few annoyances with history, if left unmodified: History records duplicate commands, and multiple shell instances can clobber each other's history. Both complications are easily overcome. Add two lines to your .bashrc:

```
export HISTCONTROL=ignoreboth
shopt -s histappend
```

The first line removes consecutive duplicate commands from your shell history. If you want to remove all duplicates independent of sequence, change `ignoreboth` to `erasedups`. The second line appends a shell's history to your history file when the shell exits. By default, the Bash history file is named (yes, a dot file) *~/~/.bash_history.* You can change its location by setting (yes, an environment variable) HISTFILE. If you want to save a shell's most recent 10,000 commands in a history file with 100,000 entries, add `export HISTSIZE=10000 HISTFILESIZE=100000` to your shell startup file. To see a shell's history, type `history` at any prompt.

Saving a history of commands is of little use if you cannot recall it. That's the purpose of the shell `!`, or bang, operator:

- `!!` ("bang bang") repeats the last command in its entirety.
- `!:0` is the name of the previous command.
- `!^` is the first argument of the previous command. `!:2`, `!:3`, and so on, ending with `!$` are the second, third, and eventually the last argument of the previous command.

- `!*` is all the arguments of the last command, except the command name.
- `!`*n* repeats the command numbered *n* in history.
- `!`*`handle`* repeats the last command that begins with the string of characters in *`handle`*. For example, `!ca` would repeat the last command that began with the characters `ca`, such as `cat README`.
- `!?`*`handle`* repeats the last command that *contains* the string of characters in *`handle`*. For example, `!?READ` would also match `cat README`.
- `^`*`original`*`^`*`substitution`* replaces the *first* occurrence of *`original`* with *`substitution`*. For example, if the previous command was `cat README`, the command `^README^license.txt` would yield a new command `cat license.txt`.
- `!:gs/`*`original`*`/`*`substitution`* replaces *all* occurrences of *`original`* with *`substitution`*. (`!:gs` means "global substitution.")
- `!-2` is the penultimate command, `!-3` is third most recent command, and so on.

You can even combine history expressions to yield sigil soup such as `!-2:0 -R !^ !-3:2`, which would expand to the command name of the penultimate command, followed by `-R`, the first argument of the previous command, and the second argument of the third most recent command. To make such cryptic commands more readable, you can expand history references as you type. Type the command `bind Space:magic-space` at any prompt, or add it to a startup file to bind the Space key to the function magic-space, which expands history substitutions inline.

## Expand-o-Matic

With so much code available on the Internet, you're likely to download dozens of files every day. And chances are, all those files are packaged differently—a ZIP file here, a RAR file there, and tarballs galore, albeit each one compressed with a different utility. Remembering how to decompress and expand each package format can be taxing. So, why not capture all those tasks in a single command? This function is widely available in many sample dot files:

```
ex () {
  if [ -f $1 ] ; then
    case $1 in
      *.tar.bz2)   tar xjf $1        ;;
      *.tar.gz)    tar xzf $1     ;;
      *.bz2)       bunzip2 $1       ;;
      *.rar)       rar x $1     ;;
      *.gz)        gunzip $1     ;;
```

```
       *.tar)       tar xf $1        ;;
       *.tbz2)      tar xjf $1      ;;
       *.tgz)       tar xzf $1       ;;
       *.zip)       unzip $1     ;;
       *.Z)         uncompress $1  ;;
       *.7z)        7z x $1    ;;
       *)           echo "'$1' cannot be extracted via
extract()" ;;
    esac
  else
    echo "'$1' is not a valid file"
  fi
}
```

This function, `ex`, expands 11 file formats and can be extended if you deal with some other package type. Once defined—say, in a shell startup file—you can simply type `ex` *`somefile`*, where *`somefile`* ends with one of the named extensions:

```
$ ls
source
$ tar czf source.tgz source
$ ls -1
source
source.tgz
$ rm -rf source
$ ex source.tgz
$ ls -1
source
source.tgz
```

By the way, if you ever misplace something you downloaded today, run `find` to discover it:

```
$ find ~ -type f -mtime 0
```

The command `-type f` looks for plain files, and `-mtime 0` looks for files created since midnight of the current day.

## Many more secrets

There are a lot more expert secrets to be discovered. Search the Web for "shell auto-complete" to learn more about automatic completion, a feature that provides context-sensitive expansions as you type a command. Also, search for "shell prompts" to learn how to customize your shell prompt. You can make it colorful; you can show your current working directory or Git branch; you can also show the history number—a convenient reference if you recall history a good deal. For working examples, search Github for "dot files." Many experts post their shell configurations on Github.

Now, if you'll excuse me, I have to find my wig and bronzer. It's not easy to hide when you resemble Groucho Marx.

## Resources

### Learn

- Speaking UNIX: Check out other parts in this series.
- dotfiles.org: You can find a large collection of dot files online.
- UNIX shells: Learn more about UNIX shells.
- GNU Bash shell: Find documentation and source for the GNU Bash Shell on its project page.
- AIX and UNIX developerWorks zone: The AIX and UNIX zone provides a wealth of information relating to all aspects of AIX systems administration and expanding your UNIX skills.
- New to AIX and UNIX? Visit the New to AIX and UNIX page to learn more.
- Technology bookstore: Browse the technology bookstore for books on this and other technical topics.

### Get products and technologies

- Github: Github houses thousands of Git repositories, including the personal dot files of many expert users. Search for "dot files" to find examples. (You can learn more about Git online and in the Github Guides.)

### Discuss

- developerWorks blogs: Check out our blogs and get involved in the developerWorks community.
- Follow developerWorks on Twitter.
- Get involved in the My developerWorks community.
- Participate in the AIX and UNIX forums:
    - AIX Forum
    - AIX Forum for developers
    - Cluster Systems Management
    - IBM Support Assistant Forum
    - Performance Tools Forum
    - Virtualization Forum
    - More AIX and UNIX Forums

## About the author

Martin Streicher

Martin Streicher is a freelance Ruby on Rails developer and the former Editor-in-Chief of *Linux Magazine.* Martin holds a Masters of Science degree in computer science from Purdue University and has programmed UNIX-like systems since 1986. He collects art and toys. You can reach Martin at martin.streicher@gmail.com.