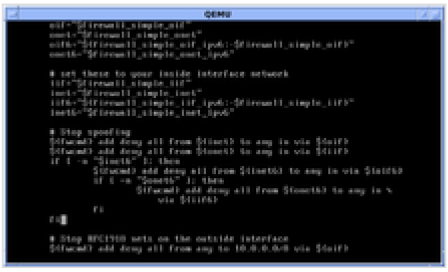
WIKIPEDIA
The Free Encyclopedia

Shell script

A **shell script** is a computer program designed to be run by a Unix shell, a command-line interpreter.[1] The various dialects of shell scripts are considered to be scripting languages. Typical operations performed by shell scripts include file manipulation, program execution, and printing text. A script which sets up the environment, runs the program, and does any necessary cleanup or logging, is called a **wrapper**.



```
oif="${firewall_simple_oif}"
onet="${firewall_simple_onet}"
oif6="${firewall_simple_oif_ipv6:-$firewall_simple_oif}"
onet6="${firewall_simple_onet_ipv6}"

# set these to your inside interface network
iif="${firewall_simple_iif}"
inet="${firewall_simple_inet}"
iif6="${firewall_simple_iif_ipv6:-$firewall_simple_iif}"
inet6="${firewall_simple_inet_ipv6}"

# Stop spoofing
${fwcmd} add deny all from ${inet} to any in via ${oif}
${fwcmd} add deny all from ${onet} to any in via ${iif}
if [ -n "$inet6" ]; then
        ${fwcmd} add deny all from ${inet6} to any in via ${oif6}
        if [ -n "$onet6" ]; then
                ${fwcmd} add deny all from ${onet6} to any in \
                    via ${iif6}
        fi
fi

# Stop RFC1918 nets on the outside interface
${fwcmd} add deny all from any to 10.0.0.0/8 via ${oif}
```

Editing a FreeBSD shell script for configuring ipfirewall

The term is also used more generally to mean the automated mode of running an operating system shell; each operating system uses a particular name for these functions including batch files (MSDos-Win95 stream, OS/2), command procedures (VMS), and shell scripts (Windows NT stream and third-party derivatives like 4NT—article is at cmd.exe), and mainframe operating systems are associated with a number of terms.

Shells commonly present in Unix and Unix-like systems include the Korn shell, the Bourne shell, and GNU Bash. While a Unix operating system may have a different default shell, such as Zsh on macOS, these shells are typically present for backwards compatibility.

Capabilities

Comments

Comments are ignored by the shell. They typically begin with the hash symbol (#), and continue until the end of the line.[2]

Configurable choice of scripting language

The shebang, or hash-bang, is a special kind of comment which the system uses to determine what interpreter to use to execute the file. The shebang must be the first line of the file, and start with "`#!`".[2] In Unix-like operating systems, the characters following the "`#!`" prefix are interpreted as a path to an executable program that will interpret the script.[3]

Shortcuts

A shell script can provide a convenient variation of a system command where special environment settings, command options, or post-processing apply automatically, but in a way that allows the new script to still act as a fully normal Unix command.

One example would be to create a version of ls, the command to list files, giving it a shorter command name of `l`, which would be normally saved in a user's `bin` directory as `/home/`*`username`*`/bin/l`, and a default set of command options pre-supplied.

```
#!/bin/sh
LC_COLLATE=C ls -FCas "$@"
```

Here, the first line uses a shebang to indicate which interpreter should execute the rest of the script, and the second line makes a listing with options for file format indicators, columns, all files (none omitted), and a size in blocks. The `LC_COLLATE=C` sets the default collation order to not fold upper and lower case together, not intermix dotfiles with normal filenames as a side effect of ignoring punctuation in the names (dotfiles are usually only shown if an option like `-a` is used), and the `"$@"` causes any parameters given to `l` to pass through as parameters to ls, so that all of the normal options and other syntax known to ls can still be used.

The user could then simply use `l` for the most commonly used short listing.

Another example of a shell script that could be used as a shortcut would be to print a list of all the files and directories within a given directory.

```
#!/bin/sh

clear
ls -al
```

In this case, the shell script would start with its normal starting line of `#!/bin/sh`. Following this, the script executes the command `clear` which clears the terminal of all text before going to the next line. The following line provides the main function of the script. The `ls -al` command lists the files and directories that are in the directory from which the script is being run. The `ls` command attributes could be changed to reflect the needs of the user.

Batch jobs

Shell scripts allow several commands that would be entered manually at a command-line interface to be executed automatically, and without having to wait for a user to trigger each stage of the sequence. For example, in a directory with three C source code files, rather than manually running the four commands required to build the final program from them, one could instead create a script for POSIX-compliant shells, here named `build` and kept in the directory with them, which would compile them automatically:

```
#!/bin/sh
printf 'compiling...\n'
cc -c foo.c
cc -c bar.c
cc -c qux.c
cc -o myprog foo.o bar.o qux.o
printf 'done.\n'
```

The script would allow a user to save the file being edited, pause the editor, and then just run `./build` to create the updated program, test it, and then return to the editor. Since the 1980s or so, however, scripts of this type have been replaced with utilities like make which are specialized for building programs.

Generalization

Simple batch jobs are not unusual for isolated tasks, but using shell loops, tests, and variables provides much more flexibility to users. A POSIX sh script to convert JPEG images to PNG images, where the image names are provided on the command-line—possibly via wildcards—instead of each being listed within the script, can be created with this file, typically saved in a file like `/home/`*`username`*`/bin/jpg2png`

```
#!/bin/sh
for jpg; do                                 # use $jpg in place of each filename given, in
turn
    png=${jpg%.jpg}.png                     # construct the PNG version of the filename by
replacing .jpg with .png
    printf 'converting "%s" ...\n' "$jpg"   # output status info to the user running the
script
    if convert "$jpg" jpg.to.png; then      # use convert (provided by ImageMagick) to
create the PNG in a temp file
        mv jpg.to.png "$png"                # if it worked, rename the temporary PNG image
to the correct name
    else                                    # ...otherwise complain and exit from the script
        printf >&2 'jpg2png: error: failed output saved in "jpg.to.png".\n'
        exit 1
    fi                                      # the end of the "if" test construct
done                                        # the end of the "for" loop
printf 'all conversions successful\n'       # tell the user the good news
```

The `jpg2png` command can then be run on an entire directory full of JPEG images with just `/home/`*`username`*`/bin/jpg2png *.jpg`

Programming

Many modern shells also supply various features usually found only in more sophisticated general-purpose programming languages, such as control-flow constructs, variables, comments, arrays, subroutines and so on. With these sorts of features available, it is possible to write reasonably sophisticated applications as shell scripts. However, they are still limited by the fact that most shell languages have little or no support for data typing systems, classes, threading, complex math, and other common full language features, and are also generally much slower than compiled code or interpreted languages written with speed as a performance goal.

The standard Unix tools sed and awk provide extra capabilities for shell programming; Perl can also be embedded in shell scripts as can other scripting languages like Tcl. Perl and Tcl come with graphics toolkits as well.

Typical POSIX scripting languages

Scripting languages commonly found on UNIX, Linux, and POSIX-compliant operating system installations include:

- KornShell (`ksh`) in several possible versions such as ksh88, Korn Shell '93 and others.
- The Bourne shell (`sh`), one of the oldest shells still common in use
- The C shell (`csh`)
- GNU Bash (`bash`)
- `tclsh`, a shell which is a main component of the Tcl/Tk programming language.
- The wish is a GUI-based Tcl/Tk shell.

The C and Tcl shells have syntax quite similar to that of said programming languages, and the Korn shells and Bash are developments of the Bourne shell, which is based on the ALGOL language with elements of a number of others added as well.[4] On the other hand, the various shells plus tools like awk, sed, grep, and BASIC, Lisp, C and so forth contributed to the Perl programming language.[5]

Other shells that may be available on a machine or for download and/or purchase include:

- Almquist shell (`ash`)
- PowerShell (`msh`)
- Z shell (`zsh`, a particularly common enhanced KornShell)
- The Tenex C Shell (`tcsh`).

Related programs such as shells based on Python, Ruby, C, Java, Perl, Pascal, Rexx etc. in various forms are also widely available. Another somewhat common shell is Old shell (`osh`), whose manual page states it "is an enhanced, backward-compatible port of the standard command interpreter from Sixth Edition UNIX."[6]

So called remote shells such as

- a Remote Shell (`rsh`)
- a Secure Shell (`ssh`)

are really just tools to run a more complex shell on a remote system and have no 'shell' like characteristics themselves.

Other scripting languages

Many powerful scripting languages have been introduced for tasks that are too large or complex to be comfortably handled with ordinary shell scripts, but for which the advantages of a script are desirable and the development overhead of a full-blown, compiled programming language would be disadvantageous. The specifics of what separates scripting languages from high-level programming languages is a frequent source of debate, but, generally speaking, a scripting language is one which requires an interpreter.

Life cycle

Shell scripts often serve as an initial stage in software development, and are often subject to conversion later to a different underlying implementation, most commonly being converted to Perl, Python, or C. The interpreter directive allows the implementation detail to be fully hidden inside the script, rather than being exposed as a filename extension, and provides for seamless reimplementation in different languages with no impact on end users.

While files with the ".sh" file extension are usually a shell script of some kind, most shell scripts do not have any filename extension.[7][8][9][10]

Advantages and disadvantages

Perhaps the biggest advantage of writing a shell script is that the commands and syntax are exactly the same as those directly entered at the command-line. The programmer does not have to switch to a totally different syntax, as they would if the script were written in a different language, or if a compiled language were used.

Often, writing a shell script is much quicker than writing the equivalent code in other programming languages. The many advantages include easy program or file selection, quick start, and interactive debugging. A shell script can be used to provide a sequencing and decision-making linkage around existing programs, and for moderately sized scripts the absence of a compilation step is an advantage. Interpretive running makes it easy to write debugging code into a script and re-run it to detect and fix bugs. Non-expert users can use scripting to tailor the behavior of programs, and shell scripting provides some limited scope for multiprocessing.

On the other hand, shell scripting is prone to costly errors. Inadvertent typing errors such as `rm -rf * /` (instead of the intended `rm -rf */`) are folklore in the Unix community; a single extra space converts the command from one that deletes all subdirectories contained in the current directory, to one which deletes everything from the file system's root directory. Similar problems can transform `cp` and `mv` into dangerous weapons, and misuse of the `>` redirect can delete the contents of a file. This is made more problematic by the fact that many UNIX commands differ in name by only one letter: `cp`, `cd`, `dd`, `df`, etc.

Another significant disadvantage is the slow execution speed and the need to launch a new process for almost every shell command executed. When a script's job can be accomplished by setting up a pipeline in which efficient filter commands perform most of the work, the slowdown is mitigated, but a complex script is typically several orders of magnitude slower than a conventional compiled program that performs an equivalent task.

There are also compatibility problems between different platforms. Larry Wall, creator of Perl, famously wrote that "It's easier to port a shell than a shell script."[11]

Similarly, more complex scripts can run into the limitations of the shell scripting language itself; the limits make it difficult to write quality code, and extensions by various shells to ameliorate problems with the original shell language can make problems worse.[12]

Many disadvantages of using some script languages are caused by design flaws within the language syntax or implementation, and are not necessarily imposed by the use of a text-based command-line; there are a number of shells which use other shell programming languages or even full-fledged languages like Scsh (which uses Scheme).

Interoperability among scripting languages

Different scripting languages may share many common elements, largely due to being POSIX based, and some shells offer modes to emulate different shells. This allows a shell script written in one scripting language to be adapted into another.

One example of this is Bash, which offers the same grammar and syntax as the Bourne shell, and which also provides a POSIX-compliant mode.[13] As such, most shell scripts written for the Bourne shell can be run in BASH, but the reverse may not be true since BASH has extensions which are not present in the Bourne shell. As such, these features are known as bashisms.[14]

Shell scripting on other operating systems

Interoperability software such as Cygwin, the MKS Toolkit, Interix (which is available in the Microsoft Windows Services for UNIX), Hamilton C shell, UWIN (AT&T Unix for Windows) and others allow Unix shell programs to be run on machines running Windows NT and its successors, with some loss of functionality on the MS-DOS-Windows 95 branch, as well as earlier MKS Toolkit versions for OS/2. At least three DCL implementations for Windows type operating systems—in addition to XLNT, a multiple-use scripting language package which is used with the command shell, Windows Script Host and CGI programming—are available for these systems as well. Mac OS X and subsequent are Unix-like as well.[15]

In addition to the aforementioned tools, some POSIX and OS/2 functionality can be used with the corresponding environmental subsystems of the Windows NT operating system series up to Windows 2000 as well. A third, 16-bit subsystem often called the MS-DOS subsystem uses the Command.com provided with these operating systems to run the aforementioned MS-DOS batch files.[16]

The console alternatives 4DOS, 4OS2, FreeDOS, Peter Norton's NDOS and 4NT / Take Command which add functionality to the Windows NT-style cmd.exe, MS-DOS/Windows 95 batch files (run by Command.com), OS/2's cmd.exe, and 4NT respectively are similar to the shells that they enhance and are more integrated with the Windows Script Host, which comes with three pre-installed engines, VBScript, JScript, and VBA and to which numerous third-party engines can be added, with Rexx, Perl, Python, Ruby, and Tcl having pre-defined functions in 4NT and related programs. PC DOS is quite similar to MS-DOS, whilst DR DOS is more different. Earlier versions of Windows NT are able to run contemporary versions of 4OS2 by the OS/2 subsystem.

Scripting languages are, by definition, able to be extended; for example, a MS-DOS/Windows 95/98 and Windows NT type systems allows for shell/batch programs to call tools like KiXtart, QBasic, various BASIC, Rexx, Perl, and Python implementations, the Windows Script Host and its installed engines. On Unix and other POSIX-compliant systems, awk and sed are used to extend the string and numeric processing ability of shell scripts. Tcl, Perl, Rexx, and Python have graphics toolkits and can be used to code functions and procedures for shell scripts which pose a speed bottleneck (C, Fortran, assembly language &c are much faster still) and to add functionality not available in the shell language such as sockets and other connectivity functions, heavy-duty text processing, working with numbers if the calling script does not have those abilities, self-writing and self-modifying code, techniques like recursion, direct memory access, various types of sorting and more, which are difficult or impossible in the main script, and so on. Visual Basic for Applications and VBScript can be used to control and communicate with such things as spreadsheets, databases, scriptable programs of all types, telecommunications software, development tools, graphics tools and other software which can be accessed through the Component Object Model.

See also

- Glue code
- Interpreter directive
- Shebang symbol (#!)
- Unix shells
- PowerShell
- Windows Script Host

References

1. Kernighan, Brian W.; Pike, Rob (1984), "3. Using the Shell", *The UNIX Programming Environment*, Prentice Hall, Inc., p. 94, ISBN 0-13-937699-2, "The shell is actually a programming language: it has variables, loops, decision-making, and so on."
2. Johnson, Chris (2009). *Pro Bash Programming: Scripting the Linux Shell* (https://books.google.com/books?id=NJmhi6T0nGMC&q=comment). Apress. ISBN 9781430219989. Retrieved September 27, 2019.

3. "exec(3p) – POSIX Programmer's Manual" (https://linux.die.net/man/3/execve). Retrieved 2020-07-24.
4. Unix Shells By Example, pp 7-10,
5. Programming Perl, 5th Edition, preface
6. "osh - manned.org" (https://manned.org/osh/f30afb07). *manned.org*. Retrieved 2019-01-16.
7. Robbins, Arnold; Hannah, Elbert; Lamb, Linda (2008). *Learning the vi and Vim Editors* (https://books.google.com/books?id=J5nKVVg5YHAC). "O'Reilly Media, Inc.". p. 205. ISBN 9781449313258.
8. Easttom, Chuck (2012). *Essential Linux Administration:: A Comprehensive Guide for Beginners* (https://books.google.com/books?id=zZYLAAAAQBAJ). Course Technology/Cengage Learning. p. 228. ISBN 978-1435459571.
9. Kumari, Sinny (November 23, 2015). *Linux Shell Scripting Essentials* (https://books.google.com/books?id=9vCoCwAAQBAJ&q=shell+script+file+extension&pg=PA2). Packt Publishing Ltd. ISBN 9781783552375. Retrieved May 7, 2017. "Rather than using a file extension for shell scripts, it's preferred to keep a filename without extension and let an interpreter identify the type by looking into shebang(#!)."
10. Taylor, Dave; Perry, Brandon (December 16, 2016). *Wicked Cool Shell Scripts, 2nd Edition: 101 Scripts for Linux, OS X and UNIX Systems* (https://books.google.com/books?id=Mpi7DQAAQBAJ&q=shell+script+file+extension&pg=PA5). No Starch Press. ISBN 9781593276027. Retrieved May 7, 2017. "Shell scripts don't need a special file extension, so leave the extension blank (or you can add the extension .sh if you prefer, but this isn't required."
11. Larry Wall (January 4, 1991). "Finding the last arg" (https://www.tuhs.org/Usenet/comp.unix.shell/1991-January/002464.html). Newsgroup: comp.unix.shell (news:comp.unix.shell). Retrieved January 5, 2023.
12. Christiansen, Tom. "Csh Programming Considered Harmful" (https://www.ooblick.com/text/CshProgrammingConsideredHarmful.html).
13. "Major Differences From The Bourne Shell" (https://www.gnu.org/software/bash/manual/html_node/Major-Differences-From-The-Bourne-Shell.html).
14. "24 Bashism To Avoid for POSIX-Compliant Shell Scripts" (https://betterprogramming.pub/24-bashism-to-avoid-for-posix-compliant-shell-scripts-8e7c09e0f49a). 18 May 2022.
15. MSDN
16. Windows NT 4 Workstation Resource Kit

External links

- *An Introduction To Shell Programming* by Greg Goebel (http://www.faqs.org/docs/air/tsshell.html)
- *UNIX / Linux shell scripting tutorial* by Steve Parker (https://www.shellscript.sh)
- *Shell Scripting Primer* (Apple) (https://developer.apple.com/mac/library/documentation/OpenSource/Conceptual/ShellScripting/)
- *What to watch out for when writing portable shell scripts* by Peter Seebach (https://www.linux.com/articles/34658)
- Free Unix Shell scripting books (http://freebookcentre.net/UnixCategory/Free-Unix-Shell-Programming-Books-Download.html)
- Beginners/BashScripting (https://help.ubuntu.com/community/Beginners/BashScripting), Ubuntu Linux