## Building a Bash web admin script

# Gaining Control

For some it is just the command line, but for others it is a full-fledged programming language. Bash can be as simple or as sophisticated as you want it to be. We'll show you a great Bash script for checking up on your web server, and you'll meet with many useful tips along the way.

BY PEER HEINLEIN

A shell is simply an interface between a user and the operating system kernel. The shell performs simple tasks and derives its name from the fact that it surrounds the kernel just like a seashell protects its inhabitant. But it wasn't long until shells started to get smart, and now most shells have powerful features that almost put them on a par with full-fledged programming languages. Bash, the Bourne Again Shell, is the most popular of Unix shells at present, and has easily outpaced its predecessor, the Bourne Shell, and derivatives such as the C shell, the TC shell or the Korn shell.

Shell scripts cannot be compiled, and this can affect their speed of execution. On the other hand, experienced admins can put together a shell script in no time at all; shell scripts are easily readable, and if the programmer uses the options the shell provides, a script really is capable of solving complex problems.

A shell script is basically a series of commands that a user could enter at the command line prompt. In other words, any commands you enter manually could be scripted with just a few lines of shell script. With a minimum of effort, admins can write scripts to automate recurring jobs, using cron to run the script or running it automatically at login time. These scripts could manage backups, statistical analyses, or even availability checks.

The trick with shell programming is to put small building blocks together in as efficient a way as possible. As this article will show, you can build very sophisti-

### Keep It Simple

When you are trying to achieve a specific result with shell programming, the lack of options or functions is not really the issue. This said, it is often difficult to identify a simple approach. Many administrators are used to performing recurring tasks manually in the shell, but they refuse to script the exact same steps and instead go through all kinds of contortions to find a solution.

It is easy to check the number of user accounts with user ID *0* in */etc/passwd*: just open */etc/passwd*, search for zeros, and count. But what if you want to script that?

In fact, there is no need to change anything: just *grep :0: /etc/passwd* to filter any accounts with user or group ID 0 from the file, and use *wc -l* to count them. A simple task, but even experienced admins can end up with complicated approaches. To avoid this, it makes sense to divide any administrative work you want to script into smaller steps. This makes it easier to identify the right tools for the job.

Don't start off by launching your favorite editor to enter the script; instead try out all the commands one by one in a terminal window, and check the output. If these manual steps lead to the solution you need, you have a script module that is worth "saving for posterity" in a file.

Of course you are not restricted to the built in bash commands. On the contrary, the real strength of the shell is the amazing variety of popular public domain tools.

cated and useful Bash scripts from small blocks of code dedicated to specific tasks.

## A Web Server Administration Script

Suppose you perform a series of web server checks on a regular basis and you would like to automate the tasks using a Bash script. You would, perhaps, come up with a script similar to the script shown in Listing 1. As you can see, the script consists of several different blocks of code dedicated to different web server administration tasks. You'll learn about the steps of the server check in the following sections.

The script in Listing 1 begins with the definition of the interpreter. If this definition is missing, the default shell of the user executing the script would attempt to run the script commands. Today's Linux distros typically use */bin/bash* as the default shell, but an explicit definition at this point doesn't hurt, and it can help avoid potential errors.

You may have also noticed something that *isn't* referenced at the beginning of the script: a config file. For simple scripts, it is not worth creating a script config file. If the parameters were more complex, however, it might make sense to create a special file below */etc*; the main script could then parse . */etc/server check.conf* – note that the dot and the

### Variables – the $ Sign in a Different Light

It is probably safe to assume that most readers know how to use the bash shell interactively. But commands that hardly ever occur in interactive mode – variables, loops, and control structures – may be a different thing altogether.

Variables should be capitalized for the sake of readability; they do not need to be typified (integer, string or the like). To assign a value to a variable, leave out the dollar sign:

```
USER="Tux"
```

But if you want bash to replace a variable with its current value, you need to use the dollar sign prefix:

```
echo "Hallo $USER"
```

Variables are only visible within the current shell. If you launch a sub-shell, the variables you have been using will not be visible, unless you have exported them explicitly to create global variables:

```
USER="Tux"
export USER
```

You can combine these steps:

```
export USER=Tux
```

Variables also support simple math. You need to place the mathematical operation in brackets *$[ ... ]*. Make sure you insert space characters to separate the variable from the brackets:

```
NEW = $[ $OLD + OLD * 2 ]
```

Variables can occur in a text string. For example, the following call

```
echo "Hallo $USER-User"
```

would output "Hallo Tux-User" in our example. Variables that start with the same letters, such as *$USER* and *$USE* could lead to some confusion; it makes sense to put the variable names in curly brackets to avoid this:

```
echo "Hallo ${USER}-User"
```


from only £49.95 / month

"The power of Dedicated Hosting, at the price of Shared."

Let's face it – if you're building and testing your own web-based applications, designing & hosting your clients' websites, or just own a very busy site – what you really want is your own server. But sometimes the cost of running a dedicated environment is just too prohibitive.

So wouldn't it be great if you could share the cost of the server but not share the resources?

In other words, the full power of your own server - in a shared environment.

Hostway's new VPS solution has a proprietary partitioning system that allows multiple server set-ups on the same box, with the guaranteed CPU, Disk Space and Network resources that you wouldn't get in a normal Shared Environment.

Together with full root or control panel access, choice of open source software at install, IP- table firewall, 24hr Roll-Back backup option and Hostway's steadfast commitment to customer care, it adds up to a pretty unique solution.

**Hostway** – Hosting the way **you** want it.

call us now on **0808 180 1880** • sales@hostway.co.uk • www.hostway.co.uk/virtual

HOSTWAY THE HOSTING COMPANY

Domain Registration • Web Hosting • Resellers • E-commerce • VPS • Dedicated Servers | Australia • Canada • Germany • Korea • Netherlands • United Kingdom • United States

path are separated by space characters. This would even allow you to support "includes" just like the ones used by other programming languages, providing some degree of modularity for more complex programs. A file referenced in this way would be inserted into the execution path and executed.

## The Heading

Before we start recording information on the status of the server, it is a good idea to print a heading giving the host name and the date of the check. This heading information is generated using the following lines:

```
DATUM=`date +'%d %e'`
echo Subject: Server-Status
 `date +"%b %e"` $HOSTNAME
```

This first line uses the technique of command substitution, executing the whole of the expression within the quotes, and inserting the output instead of the command – a kind of cut & paste. There are three different kinds of quotes, but the shell will only accept one of them as the command substitution character. What we need is a so-called back tick.

The command uses *date* to obtain the current date. The *date* manpage is unclear in parts, and you need some bash programming experience to read it correctly. The Synopsis area of *man date* shows a call to *date +FORMAT*, and the manpage later goes on to explain the values that *FORMAT* can have. It is easy to overlook the fact that you need to keep the + parameter, as you are only substituting *FORMAT*. In other words, you need the following syntax to substitute a date: *date +%Y%M%D*.

The second problem is the blank in the date output format, as in *AUG 25*, which is often seen in logfiles. *date +%d %e* will cause an error, as will *date +%d +%e*. You also need to enclose the format block in quotes or ticks. This indicates that the space character does not mark the point where the next *date* parameter starts: *date +"%d %e"* or *date +'%d %e'* – but of course you cannot use back ticks here, as they are reserved for command substitution.

## Is Your Server Running?

The first check the script will perform is to determine if the web server is running. The code that performs this check is as follows:

```
01 # Check if web server is
   running
02 # Customized for Apache2, use
   "/etc/init.d/apache"
03 # and "/usr/sbin/httpd -T"
   otherwise.
04 if [ $CHECK_WEB = "yes" ] ;
```

### Listing 1: servercheck

```
001 #!/bin/bash
002 #
003 # Sample script for
    monitoring a httpd server
004 #
005 # Heinlein Professional Linux
    Support GmbH, 8/04
006 #
    http://www.heinlein-support.d
    e
007 #
008
009 # Generate secure tmp file
    for output
010 umask 077
011 TEMPFILE=`mktemp /tmp/serverc
    heck.tmp.XXXXXXXX`
012 # Minimum space for "/" in
    percent
013 HDMINFREE="90"
014
015 # To-do list
016 CHECK_WEB=yes
017 CHECK_ACCOUNTS=yes
018 CHECK_HDMINFREE=yes
019 CHECK_LOGINS=yes
020 CHECK_STATUS=yes
021
022 #
023 # No config changes from this
    point on.
024 #
025
026 DATUM=`date +'%d %e'`
027
028 echo Subject: Server-Status
    `date +"%b %e"` $HOSTNAME
029 echo
030 echo
031
032 # Check if web server is
    running
033 # Customized for Apache2, use
    "/etc/init.d/apache"
034 # and "/usr/sbin/httpd -T"
    otherwise.
035 if [ $CHECK_WEB = "yes" ] ;
    then
036 if /etc/init.d/apache2 status
    &> /dev/null && wget
    --delete-after
    http://www.domain.local/check
    file.txt &>/dev/null ; then
037         # Web server is
    running! Try reload, perform
    syntax check before doing so
038         if /usr/sbin/httpd2
    -t &>/dev/null; then
039                 /etc/init.d/
    apache2 reload &>/dev/null &&
    echo "    Web server running
    and config reload okay."
040         else
041                 echo "###
    WARNUNG: Reload failed due to
    CFG error! "
042         fi
043 else
044         # Web server not
    running! Try to launch!
045         if /usr/sbin/httpd2
    -t &>/dev/null && /etc/init.
    d/apache2 start &>/dev/null ;
    then
046                 echo "###
    WARNING: Web server was down,
    restart successful."
047         else
048                 echo "###
    WARNING: Web server was down,
    restart FAILED! "
049         fi
050 fi
051 fi
052
053 # Check /etc/passwd for
    hidden root users
```

```
    then
05 if /etc/init.d/apache2 status
    &> /dev/null && wget --delete-
    after http://www.domain.
    local/ checkfile.txt &>/dev/
    null ; then    # Web server
    is running! Try reload,
    perform syntax check before
    doing so
06         if /usr/sbin/httpd2 -t
           &>/dev/null; then
07         /etc/init.d/apache2
           reload &>/dev/null
           && echo "  Web server
           running and config
           reload okay."
08         else
09         echo "### WARNUNG:
           Reload failed due to
           CFG error! "
10         fi
11 else
12         # Web server not
           running! Try to launch!
13         if /usr/sbin/httpd2 -t
           &>/dev/null && /etc/
           init.d/apache2 start
           &>/dev/null ;then
14         echo "### WARNING:
           Web server was down,
           restart successful."
15         else
16         echo "### WARNING:Web
           server was down,
           restart FAILED! "
17           fi
18 fi
19 fi
```

The second *if* condition is as an example of the elegant programming style the shell supports. The start/stop script */etc/init.d/apache* has an option for checking if the Apache daemon is running. Instead of interpreting the text output from this command, we'll use the return value of the program. The return code or *errorlevel* will be *0* for a program that completes correctly; this is equivalent to *TRUE* in the shell's binary logic, whereas any error will return a code with a value other than zero, and thus the binary equivalent of *FALSE*.

As the *if* condition simply checks for true/false statements, you can substitute the command here: *if /etc/init.d/apache status ; then....* Exactly the same approach is used just two lines down. The *httpd*, also known as Apache, has a *-T* option for checking the configuration – again the return value can be true (if the configuration is okay) or false (if a syntax error occurs). This test is particularly useful as it avoids killing a running web server by attempting to reload after inadvertently adding a fatal error to the web server configuration.

The script does not entirely rely on */etc/init.d/apache status* to check the

## Listing 1: servercheck

```
054 if [ $CHECK_ACCOUNTS = "yes"
    ] ; then
055 cat /etc/passwd | grep ":0:"
    | grep -v ^root: > $TEMPFILE
056 ACCOUNTS=`cat $TEMPFILE | wc
    -l`
057 if [ $ACCOUNTS -ge 1 ] ; then
058         echo "### WARNING:
    Discovered $ACCOUNTS
    additional accounts with root
    privileges!"
059         cat $TEMPFILE
060 else
061         echo "    No
    additional root accounts."
062 fi
063 fi
064
065 # Check for HDMINFREE
066 if [ $CHECK_HDMINFREE = "yes"
    ] ; then
067 KBISFREE=`df | grep /$ | cut
    -b 52-54`
068 INODEISFREE=`df -i | grep /$
    | cut -b 47-49`
069 if [ $KBISFREE -ge $HDMINFREE
    -o $INODEISFREE -ge
    $HDMINFREE ] ; then
070         echo "### WARNING:
    $KDISFREE% / INODEISFREE% of
    root partition occupied! "
071 else
072         echo "    Hard disk space
    for root partition okay."
073 fi
074 fi
075
076 # Check for failed logins
077 # Modify grep pattern to
    reflect your own log format
    if needed.
078 if [ $CHECK_LOGINS = "yes" ]
    ; then
079 cat /var/log/messages | grep
    "$DATUM" | grep -i "FAILED "
    | grep -i "root" | grep -v
    "tty" |> $TEMPFILE
080 FAILLOGINCOUNT=`cat $TEMPFILE
    | wc -l`
081 if [ $FAILLOGINCOUNT -ge 15 ]
    ; then
082         echo "### WARNING:
    More than 15 failed root
    logins! "
083 elif [ $FAILLOGINCOUNT -ge 1
    ] ; then
084         echo "    Discovered
    following failed root
    logins:"
085 fi
086 cat $TEMPFILE
087 fi
088
089 # Generate status:
090 if [ $CHECK_STATUS = "yes" ]
    ; then
091 echo
092 echo "General server status:"
093 echo
    "------------------------"
094 echo
095 echo "Current load in last 15
    minutes: `cat /proc/loadavg |
    sed "s/.* .* \(.*\) .*
    .*/\1/"`"
096 echo
097 echo "Memory usage:"
098 cat /proc/meminfo | head -n 3
099 echo
100 echo "The following users are
    currently logged on:"
101 who
102 fi
103 echo
104 echo
105 rm $TEMPFILE
```

## Listing 2

```
01 # Check /etc/passwd for hidden
   root users
02 if [ $CHECK_ACCOUNTS = "yes" ]
   ; then
03 cat /etc/passwd | grep ":0:" |
   grep -v ^root: > $TEMPFILE
04 ACCOUNTS=`cat $TEMPFILE | wc
   -l`
05 if [ $ACCOUNTS -ge 1 ] ; then
06      echo "### WARNING:
   Discovered $ACCOUNTS
   additional accounts with root
   privileges!"
07      cat $TEMPFILE
08 else
09      echo "   No additional
   root accounts."
10 fi
11 fi
```

Apache status. Instead, it uses *wget* to check if a file download works out; admittedly this is really just our way of demonstrating the use of the tool. Again *wget* returns a true/false code which leads us to the *if true AND true ; then...* expression. If one of these tests does not check out, the script moves on to the else branch.

The text output from the commands in the *if* conditions here is redirected to */dev/null*. The idea behind this script is to give us a status report; output from the programs themselves would only get in the way of that.

## Checking /etc/passwd for Hidden Root Users

Since you are already checking to see if the web server is running, you might as well also check to see if any hidden root users are currently logged in to the system. The code in Listing 2, which is an excerpt of Listing 1, checks /etc/passwd for hidden root users.

As you can see, the code in Listing 2 uses a *grep* statement to search the contents of the /etc/passwd file to see if any users with root status are currently using the system. Results of successful searches for root users are accumulated in ACCOUNTS, and, if ACCOUNTS reveals one or more entries indicating that a root user has gotten onto the system, the script outputs a warning. Otherwise the script states that no root users were found.

## Checking Hard Disks for Minimum Free Space

The next check looks to see whether the server hard disk has sufficient free space? This code block in Listing 1, which begins with the line:

```
# Check for HDMINFREE
```

uses the HDMINFREE variable, which is defined earlier in the script:

```
# Minimum space for "/"↩
 in percent
HDMINFREE="90"
```

HDMINFREE defines the minimum free disk space threshold that will trigger a warning. Placing this value in a variable allows you to change the setting more easily than if the value were hard coded into the logic of the Bash script.

How do you discover the current level of disk usage? There are two tools for this job: *du* gives you the hard disk usage, and is not very useful within the confines of our script. *df* shows you the free space on one or multiple partitions, but in a fairly verbose way. The best approach is to do what any human would do: launch *df*, locate the lines for the *root* partition, and read the numbers in front of the percentage signs.

Instead of checking for the partition name (*hda3*), it is a lot easier to look for a slash character / at the end of a line. This makes it easier to apply the script to other configurations. In other words, we just grep the output from *df* for */$*, where *$* is the symbol for end of line (EOL).

One alternative might have been to use *df /* , but *df* would have given us output with column headers, and that would have meant using *tail -n1* to filter the line. In other words, we would not have made things any easier.

*cut -b 52-54* and *cut -b s47-49* surgically remove the percentage, which we can then store in a variable for later comparison in the *if* condition. Watch out for the pitfalls here: if you cut *53-54*, the script would just take the two zeros from 100 percent, and that would mean the script giving you unreliable results when you are most in the need of accuracy – that is when the disk is completely full.

The *if* condition in the square brackets simply contains the abbreviation of a *test* command. If we expand this, we get *if test $HDISFREE -ge $HDMINFREE ; then* – these notations are equivalent. *test* can check use variables or files to check for true/false conditions. In our case *-ge* stands for “greater than or equal”; in other words, we are checking to see if the disk space usage for the system is

## Listing 3: Evaluating Parameters

```
01 while [ "$#" -gt 0 ] ; do
02    case $1 in
03    --web)
04         CHECK_WEB=yes
05         shift
06         ;;
07    --accounts)
08         CHECK_ACCOUNTS=yes
09         shift
10         ;;
11    --hdminfree)
12         CHECK_HDMINFREE=yes
13         shift
14         ;;
15    --status)
16         CHECK_STATUS=yes
17         shift
18         ;;
19     --logins)
20         CHECK_LOGINS=yes
21         shift
22         ;;
23    --all)
24         CHECK_WEB=yes
25         CHECK_ACCOUNTS=yes
26         CHECK_HDMINFREE=yes
27         CHECK_LOGINS=yes
28         CHECK_STATUS=yes
29         shift
30         ;;
31    *)
32         echo "Usage: $0
   [--web] [--accounts]
   [--hdminfree] [--logins]
   [--status] [--all]"
33         exit
34         ;;
35     esac
36 done
```

equal to, or greater than, the threshold value.

Again, this is just the true/false return code of the programs. If the statement is true, such as in *test 5 -ge 3*, *test* returns a value of true. If the statement is false, as in *test 3 -ge 5*, *test* returns false. The *if* condition here actually has nothing to do with the numbers, it simply looks for a true or false value between *if* and *; then*.

## Check for Failed Root Logins

In the next block of Listing 1, the script searches for repeated root login attempts with the wrong password. This code follows a familiar pattern. We can grep for a combination of words to filter password errors out of local and SSH-based login attempts. Of course it would have been possible to add a call to | *wc -l* to count the invalid login attempts, but the file-based approach allows us to add the suspicious login attempts to the status report. After all, the idea behind this script is to give the administrator all the information he or she needs in a consolidated report.

Typos in password entries can and do happen every day – we do not want the script to alert the admin to a failed login that was caused by a user simply typing the wrong password. In this case. failed login attempts start to become suspicious when a flood of failed logins occurs within a short period of time. This kind of behavior is indicative of a brute force attack on the server. If the server is the subject of a brute force attack, the admin needs to know about it in a hurry. 15 failed login attempts is a useful threshold. A brute force attack could involve thousands of failed login attempts over a period of several days. It is not much help if you discover a bunch of failed login entries in your logfiles

### Control Structures

Bash has typical command and loop structures just like any other programming language. They do not play an important role within an interactive context. *if* or *Select Case* conditions do not make much sense if a user is sitting in front of the keyboard, and can check his or her options before making a decision. The following *if* condition:

```
if bla1 ; then
cmd1
elif bla2 ; then
cmd2
else
cmd3
fi
```

demonstrates the classical format, as used by more or less any high level programming language. Note the semicolon in front of *then*, and the option of combining *else if* to a more elegant *elif*.

The test expression, *bla1*, will return *TRUE* or *FALSE*. But it can reflect the value assigned to a variable (*if $check ; then...*), or the return value of a program that you have called (*if mkdir /tmp/test ; then.*) A program that exits cleanly will return a value of *TRUE*, or *FALSE* in case of error. Most programming languages also have a *select* condition; *select* has the following syntax in bash:

```
01 case "$VAR" in
02 TEST1)
03 cmd1
04 ;;
05 TEST2
06 cmd2
07 ;;
08 *)
09 cmd3
10 ;
11 esac
```

Note the double semicolons at the end of each case block. **)* is a special expression that matches any conditions not already met. Select structures are often used to collect parameters, or to output help for parameters in some cases.

This for loop

```
for A in n1 n2 n3 n4 ... nn ; do
cmd $A
done
```

iterates against the elements *n1* through *nn*. *$A* accepts the current value of the element line by line. The list could also be a variable containing multiple space separated values:

```
LIST="Harry Sally ➘
Martin Yvonne"
for A in $LIST ; do
echo "Hallo $A"
done
```

It is interesting to note that bash can only handle a limited number of parameters; this means that *rm ** will fail in directories with many thousands of files. In contrast to this, *for A in * ; do rm $A ; done* will work perfectly, expanding * as a file wildcard.

```
while condition ; do
cmd
done
```

will iterate through the loop while *condition* returns a value of *TRUE*. So we need a *test* command. The following example counts from 0 to 1000.

```
i=0
while test $i -le 1000 ; do
echo $1
i = $ [ $i + 1 ]
done
```

or if we use *test* shorthand:

```
i=0
while [ $i -le 1000 ] ; do
echo $1
i = $ [ $i + 1 ]
done
```

Whereas a *while* loop will keep going as long as the condition is fulfilled, an *until* loop stops iterating as soon as the condition is fulfilled:

```
until condition ; do
cmd
done
```

Theoretically we could write all of these syntactical examples in a single line, using a semicolon (;) to separate them: *if bla1 ; then cmd1 ; cmd2 ; else cmd3 ; fi*. Or using a *for* loop: *for A in $LISTE ; do echo "Hallo $A" ; done*.

Newcomers will be happy to hear that bash has a debugging option; the debugger is a big help if you are trying to locate logical errors or typos (see Bash Debugging.)

The script file itself does not need a special name, although you might like to keep to the convention of adding an *.sh* extension. However, you will need to set the file permissions to make your script executable, that is, you must set the x bit for users who need to run the script.

Although you can create fairly complex programs with shell scripts – including a GUI if needed , quick (but permanent!) hacks are a more common use. Shell scripts are thus an invaluable tool for any admin, especially for admins who do not have the time or inclination to learn a higher level scripting language like Perl or Python just to achieve the same results with more or less the same amount of code.

while investigating a break in.

## Status Information

The final block of code in Listing 1, which begins with the line:

```
# Generate status:
```

gives the admin more useful information. The */proc* directory proves especially useful; the innumerable virtual files below */proc* have a wealth of system information that can be parsed and processed using *cat*, *grep*, *cut*, *head*, and *tail*, without needing access to operating system internals, and without using a high level programming language. */proc/loadavg* lends itself to all kinds of hacks to grep information on the CPU load and memory usage from the *top* output.

## Automated cron Job

After running the script manually to check that everything is working to your satisfaction, you can add a *crontab* entry to run the script automatically with root privileges. The mailer will automatically create a *Subject: …* line for the email message generated by the script:

```
57  23  *  *  *
/usr/local/bin
/servercheck
 | sendmail
admin@bla.local
```

Of course, we could have added a section to create the email message to the script proper, but we didn't want to lose the ability to run the script manually in a terminal window as a quick manual system check.

## There's Always Room for Improvement

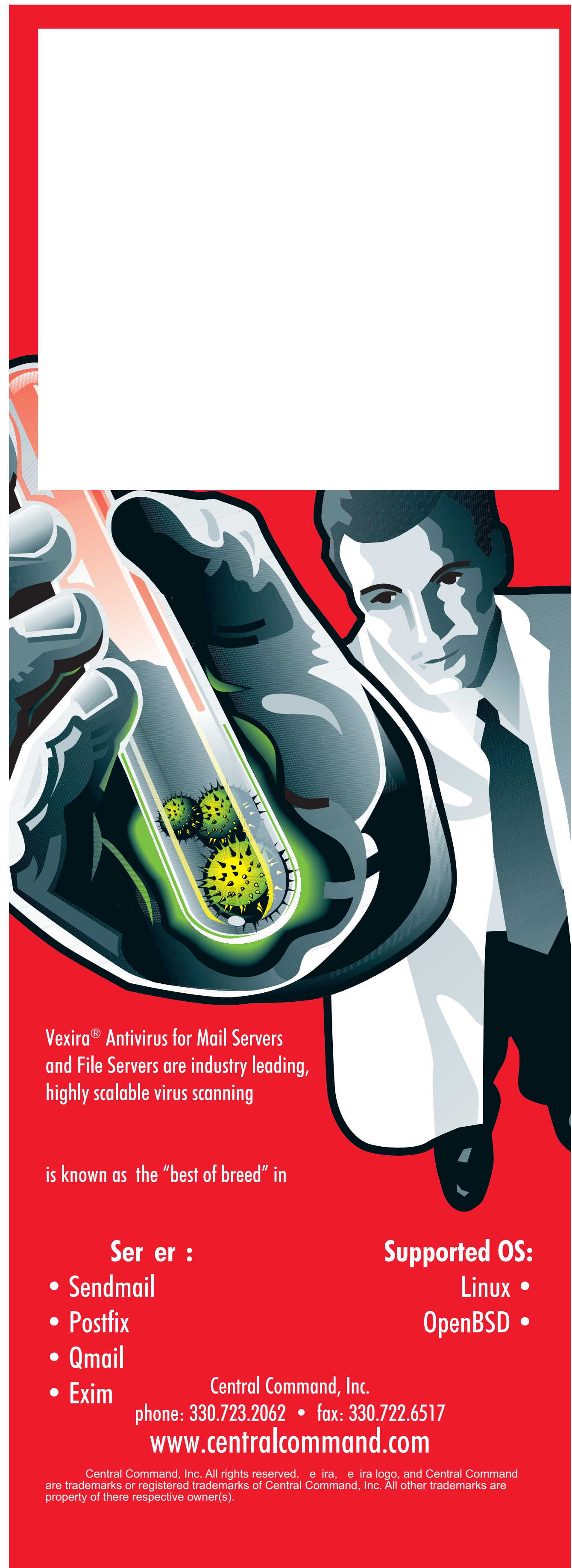
A script is more flexible if it supports command line parameters. If we need to disable a module in our script (such as the hard disk or root login check), we need to launch an editor and modify the *CHECK_* variables. There is a more elegant way of doing this.

The first nine command line parameters are stored in special variables called *$1*, *$2*, *$3* … *$9*. *$0* stores the command itself, and *$#* gives us the number of parameters.

The *shift* command can be useful here: it deletes the first parameter and moves all the other parameters one position to the "left." In other words, what used to be *$2* now becomes *$1*. We can use a *while* loop to set the *CHECK_ variables* depending on the

### Listing 4: Regexp-based Search

```
01 if $CHECK_HDMINFREE ; then
02 GETPERCENTAGE='s/.* \([0-9]\{1,3\}\)%.*/\1/'
03 KBISFREE=`df | grep /$ | sed -e "$GETPERCENTAGE"`
04 INODEISFREE=`df -i | grep /$ | sed -e
   "$GETPERCENTAGE"`
05 if [ $KBISFREE -ge $HDMINFREE -o $INODEISFREE -ge
   $HDMINFREE ] ; then
06         echo "### WARNING: $KDISFREE% /
   INODEISFREE% of root partition occupied! "
07 else
08         echo "    Hard disk space okay."
09 fi
10 fi
```



parameters (see Listing 3, "Evaluating Parameters.").

To parametrize the script, all we need to do is insert this piece of code instead of the variable declarations in lines 15 through 20 of Listing 1.

## Best Behavior

Well-behaved scripts clean up when they're done, removing temporary files. Under normal circumstances, *rm $TEMPFILE* at the end of the script could handle this, but if the script is interrupted, it would not reach this step.

Although admins could probably live with the disk space loss, as the temporary file is quite small, the temporary file might contain sensitive information that should not be left up for grabs in the */tmp* directory.

*trap* is a special function that can listen for specific signals and perform a predefined task. In this example, we only need to monitor the three most important signals: *SIGKILL* (*kill -9* pid), *SIGTERM* (*kill -15* pid), and *SIGINT*, which occurs when a user presses Ctrl-C (alias signal 2). If you need more information on signals, try *man 7 signal*.

If one of these signals occurs, we need to delete the temporary file – if it exists – and quit the script:

```
trap "test -e $TMPILE && rm ⮐
$TMPFILE ; exit" 2 9 15
```

Some sections of the sample script could be more compact and shorter – but this would impact the script's readability, especially for inexperienced users. For example, we could set the variables at the start of the script, *CHECK_WEB*, *CHECK_ACCOUNTS*, and so on, to *true* rather than *yes* to support shorter *if* conditions:

```
CHECK_WEB=true
if $CHECK_WEB ; then
        cmd
fi
```

Bash would expand the variables and launch the *true* program, which is located in */bin/true* and returns a binary *true*. Its counterpart is */bin/false*.

Continuing in the same vein, we could slim down the call to *grep*: *cat /etc/passwd | grep ":0:"* ... (line 55) could be abbreviated to *grep ":0:" /etc/passwd*, removing *cat* altogether.

Additionally, we could use either AWK or regular expressions to revamp the hard disk space check and give us more elegant code. This said, regular expressions are not everyone's idea of fun. And we would have needed to search for the percent character to extract the preceding values, as shown in Listing 4.

Regular expressions might have been preferable to the grep escapades in line 79. If you like, feel free to define a regexp pattern to match the logfile lines. On the other hand, our sample script just goes to prove that simple steps will take you to your goal, possibly in a more roundabout way.

Individual circumstances may necessitate other checks, but considering the fact that thousands of leased root servers in data centers are not checked at all, and have only survived so far by pure luck, the sample script is a definite improvement. This said, the script is not recommended for monitoring server farms, or in scenarios where higher security levels apply.

The script does its job without any fuss. No more, and no less – and that makes it a useful example of shell scripting within the context of the administrator's daily routine.

## Tools for Bash

Besides the builtin shell commands, a number of useful external programs are also used with Bash.

AWK is a powerful pattern-action programming language that is useful for editing data in tables. (You'll learn more about AWK in the article titled "Regular Wizardry" later in this issue.)

*cut* cuts single lines from a given position in the text. *head* outputs the first 10 lines of a file by default, and its counterpart *tail* outputs the last 10 lines.

*join* merges the lines in two files with a common index field. *split* splits a file into multiple individual files.

*nl* enumberates the lines of a file. This is useful if you need to create numbered listings of records.

The ubiquitous *cat* tool outputs files to standard output or or redirects standard output to files. *tac* is like *cat* only it outputs a file backwards, starting with the last line.

*tee* redirects input to a file, and *sort* sorts files. The list goes on. ■

### Tutorials and How-tos

**Bash Tutorials:**

- *http://www.linuxfibel.de/bash.htm*
- *http://www.tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html*
- *http://www.tldp.org/HOWTO/Bash-Prompt-HOWTO/index.html*
- *http://www.tldp.org/LDP/abs/html/*

**Special Shell Scripting Tutorials:**

- *http://www.linuxfibel.de/bashprog.htm*
- *http://www.freeos.com/guides/lsst/ch03sec03.html*
- *http://quong.best.vwh.net/shellin20/#LtohTOCentry-11*

**Good Bash Fan Pages and Script Collections:**

- *http://www.shelldorado.de*

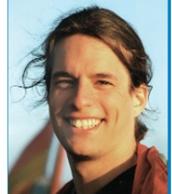
### Bash Debugging

It is a little known fact that bash has a debugging feature. You can use *set* at the beginning of a script to set Xtrace mode, for example:

- ***set -x*** tells the shell to output the expanded form of each line before executing the commands in the line. In other words, the shell will first insert filenames for any wildcards and also replace variables with their current values. * ***set -v*** enables verbose output of any commands executed by the script. * ***set -n*** performs a syntax check, but without running the script (no-exec mode).

A well-placed *echo $VARIABLE*, or simply *echo* xyz, wherever appropriate in the script or loop, is also an invaluable troubleshooting tool. It tells you exactly what a program is doing, and breakpoints can tell you how often a script has iterated through a specific loop.


### THE AUTHOR



*Peer Heinlein has been an Internet Service Provider since 1992. Besides his book on Postfix, Peer has published two other books on "LPIC-1" and the "Snort" Intrusion Detection System with Open Source Press. Peer's company, www.heinlein-support.de, educates and trains administrators, and provides consulting and support services all over Europe.*