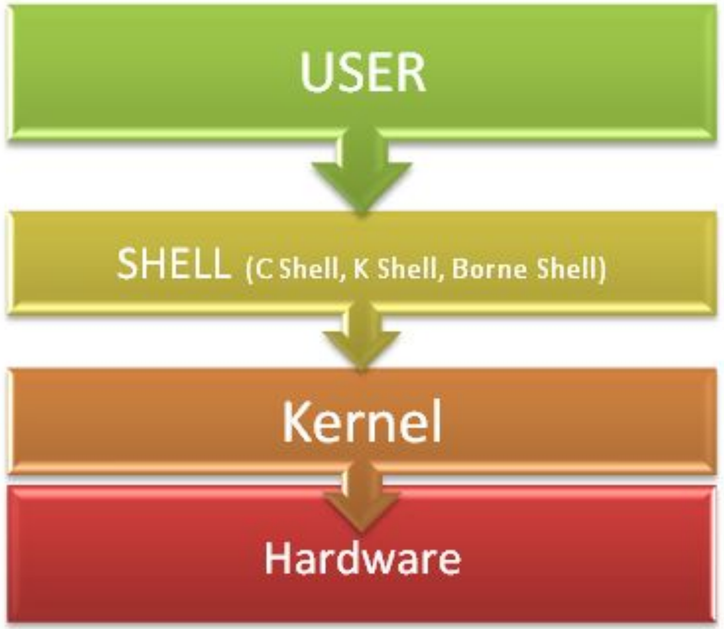
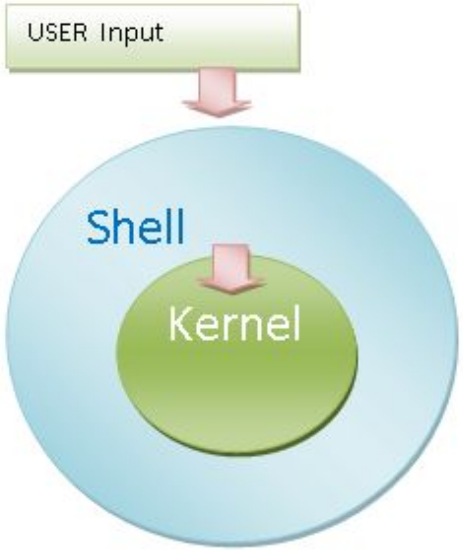
What is Shell Scripting?

A shell script is a list of commands in a computer program that is run by the Unix shell which is a command line interpreter. A shell script usually has comments that describe the steps. The different operations performed by shell scripts are program execution, file manipulation and text printing. A wrapper is also a kind of shell script that creates the program environment, runs the program etc.





Types Of Shell Scripting

.

no	Shell Type	Description
	Bash aka Bourne Again Shell	This is the most common shell available on all Linux and debian based systems. It is open source and freeware.
	CSH or C Shell	This Shell scripting program uses the C programming’s shell syntax and its almost similar to C.
	KSH or Korn Shell	Korn is a Unix based Shell scripting program, initially based on the Bash Shell Scripting. This shell is quite advanced and its a high level programming language.
	TCSH	There is no specific fullform of TCSH. It is as it is. TCSH is an advanced version of Berkeley Unix C shell. It again supports C style syntax

Benefits of Shell Script

1. Shell scripts don’t have to be boring:

Most shell scripts are uninteresting. They don’t show anything, nor have any coloring. What .shame... Lynis uses a color scheme, has an upgrade check, intercepts interruption (e.g. CTRL-C) and shows alerts if it was not dismissed properly last time. There is so much possible!

2. Reusing

Why recurring the same statements in your shell scripts every time? Develop an effective set of functions and consist of that in your existing and new shell scripts. Don’t use “echo” while you also can call your own function Display.

3. Always available:

Shell scripting can constantly be used, on each program you encounter. It makes your life simpler by automating repetitive steps. Just insert your preferred commands in a file, make it executable and run it happily. Simple to learn, and, well also quick to master.

4. Readability

With a shell script, the possibility of creating things really unreadable is much lower. Sure you can use unique features of the shell, which others don’t comprehend.

5. Shell scripting is powerful

It is convenient, no compilation required and helps almost every single UNIX based system there is. Merge it with the commonly accessible tools like awk, grep and sed and you have a great basis.

What is Bash Scripting?

Bash (AKA **B**ourne **A**gain **Sh**ell) is a **type of interpreter** that processes shell commands. A **shell interpreter** takes commands in plain text format and calls **Operating System services** to do something. For example, `ls` command lists the files and folders in a directory.

A **shell scripting** is writing a program for the shell to execute and a **shell script** is a file or program that shell will execute.

Bash (`bash`) is one of many available (yet the most commonly used) Unix shells. and is a replacement/improvement of the original Bourne shell (`sh`).

Shell scripting is scripting in *any* shell, whereas Bash scripting is scripting specifically for Bash. In practice, however, "shell script" and "bash script" are often used interchangeably, unless the shell in question is not Bash.

Basic Linux Commands Part 1

We need to know basic Linux/Unix Commands before jumping into shell scripting

pwd - Get the full path of the current working directory.

cd - Navigate to the last directory you were working in.

cd ~ or just cd Navigate to the current user's home directory.

cd .. Go to the parent directory of current directory
(mind the space between cd and ..)

ls -l - List the files and directories in the current directory in long (table) format (It is recommended to use -l with ls for better readability).

ls -ld dir-name - List information about the directory dir-name instead of its contents.

ls -a - List all the files including the hidden ones (File names starting with a . are hidden files in Linux).

ls -F - Appends a symbol at the end of a file name to indicate its type (* means executable, / means directory, @ means symbolic link, = means socket, | means named pipe, > means door)

ls -lt - List the files sorted by last modified time with most recently modified files showing at the top (remember -l option provides the long format which has better readability)

ls -lh - List File Sizes in Human Readable Format

ls -lR - Shows all subdirectories recursively.

tree - Will generate a tree representation of the file system starting from the current directory.

cp -p source destination - Will copy the file from source to destination. -p stands for preservation. It preserves the original attributes of file while copying like file owner, timestamp, group, permissions etc.

cp -R source_directory destination_directory - Will copy source directory to specified destination recursively.

mv file1 file2 - In Linux there is no rename command as such. Hence mv moves/renames the file1 to file2

rm -i filename - Asks you before every file removal for confirmation. IF YOU ARE A NEW USER TO LINUX COMMAND LINE, YOU SHOULD ALWAYS USE rm -i. You can specify multiple files.

rm -R dir-name - Will remove the directory dir-name recursively.

rm -rf dir-name - Will remove the directory dir recursively, ignoring non-existent files and will never prompt for anything. BE CAREFUL USING THIS COMMAND! You can specify multiple directorie

rmdir dir-name - Will remove the directory dir-name, if it's empty. This command can only remove empty directories.

mkdir dir-name - Create a directory dir-name.

mkdir -p dir-name/dir-name - Create a directory hierarchy. Create parent directories as needed, if they don't exist. You can specify multiple directories.

touch filename - Create a file filename, if it doesn't exist, otherwise change the timestamp of the file to current time

Basic Linux Commands Part 2

File/directory permissions and groups

chmod <specification> filename - Change the file permissions. Specifications = u user, g group, o other, + add permission, - remove, r read, w write,x execute.

chmod -R <specification> dir-name - Change the permissions of a directory recursively. To change permission of a directory and everything within that directory, use this command.

chmod go=+r myfile - Add read permission for the owner and the group.

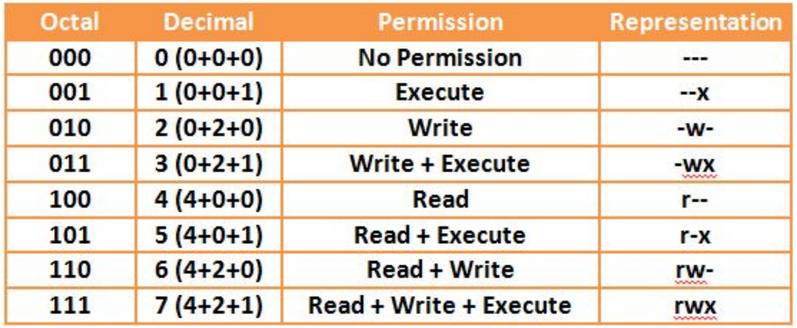
chmod a +rwx myfile - Allow all users to read, write or execute myfile.

chmod go -r myfile - Remove read permission from the group and others.

chown owner1 filename - Change ownership of a file to user owner1.

chgrp grp_owner filename - Change primary group ownership of file filename to group grp_owner.

chgrp -R grp_owner dir-name - Change primary group ownership of directory dir-name to group grp_owner
recursively. To change group ownership of a directory and everything within that directory, use this command.



Octal	Decimal	Permission	Representation
000	0 (0+0+0)	No Permission	---
001	1 (0+0+1)	Execute	--x
010	2 (0+2+0)	Write	-w-
011	3 (0+2+1)	Write + Execute	-wx
100	4 (4+0+0)	Read	r--
101	5 (4+0+1)	Read + Execute	r-x
110	6 (4+2+0)	Read + Write	rw-
111	7 (4+2+1)	Read + Write + Execute	rwx

In more simple way:
0 = 0 = nothing
1 = 1 = execute
2=2= write
3 = 2 + 1 = w+x
4 = 4 = read
5 = 4+1 = r+x
6 = 4 + 2 = r+w
7 = 4 + 2 + 1 = r+w+x

Basic Bash Scripting

Variables

```
NAME="John"
echo $NAME
echo "$NAME"
echo "${NAME}!"
```

String quotes

```
NAME="John"
echo "Hi $NAME"   #=> Hi John
echo 'Hi $NAME'   #=> Hi $NAME
```

Shell execution

```
echo "I'm in $(pwd)"
echo "I'm in `pwd`"
```

Conditional execution

```
git commit && git push
git commit || echo
```

Functions

```
get_name() {
  echo "John"
}

echo "You are $(get_name)"
```

Conditionals

```
if [[ -z "$string" ]];
then
  echo "String is
empty"
elif [[ -n "$string"
]]; then
  echo "String is not
empty"
  fi
```

Brace expansion

```
echo {A,B}.js
```

`{A,B}` - Same as `A B`
`{A,B}.js` -Same as `A.js B.js`
`{1..5}`-Same as `1 2 3 4 5`

Strict mode

```
set -euo pipefail
IFS=$'\n\t'
```

Parameter expansions

Basics

```
name="John"
echo ${name}
echo ${name/J/j}
#=> "john" (substitution)
echo ${name:0:2}
#=> "Jo" (slicing)
echo ${name::2}
#=> "Jo" (slicing)
echo ${name::-1}
#=> "Joh" (slicing)
echo ${name:(-1)}
#=> "n" (slicing from right)
echo ${name:(-2):1}
#=> "h" (slicing from right)
echo ${food:-Cake}
#=> $food or "Cake"
length=2
echo ${name:0:length}  #=> "Jo"
```

Length

`${#FOO}` - Length of `$FOO`

```
STR="/path/to/foo.cpp"
echo ${STR%.cpp}    # /path/to/foo
echo ${STR%.cpp}.o  # /path/to/foo.o
echo ${STR%/*}      # /path/to
echo ${STR##*.}     # cpp (extension)
echo ${STR##*/}     # foo.cpp (basepath)
echo ${STR#*/}      # path/to/foo.cpp
echo ${STR##*/}     # foo.cpp
echo ${STR/foo/bar} # /path/to/bar.cpp
```

Substitution

`${FOO%suffix}` - Remove suffix
`${FOO#prefix}` - Remove prefix
`${FOO%%suffix}` - Remove long suffix
`${FOO##prefix}` - Remove long prefix
`${FOO/from/to}` - Replace first match
`${FOO//from/to}` - Replace all
`${FOO/%from/to}` - Replace suffix
`${FOO/#from/to}` - Replace prefix

Substrings

`${FOO:0:3}` - Substring (position, length)
`${FOO:(-3):3}` - Substring from the right

Manipulation

```
STR="HELLO WORLD!"
echo ${STR,}
#=> "hELLO WORLD!" (lowercase 1st
letter)
echo ${STR,,}
#=> "hello world!" (all
lowercase)

STR="hello world!"
echo ${STR^}
#=> "Hello world!" (uppercase 1st
letter)
echo ${STR^^}
#=> "HELLO WORLD!" (all
uppercase)
```

Loops

Basic for loop

```
for i in /etc/rc.*; do
  echo $i
  done
```

C-like for loop

```
for ((i = 0 ; i < 100 ;
i++)); do
  echo $i
  done
```

Ranges

```
for i in {1..5}; do
    echo "Welcome $i"
  done
```

Reading lines

```
cat file.txt | while
read line; do
  echo $line
  done
```

Forever

```
while true; do
  ...
  done
```

Ranges

```
for i in {1..5}; do
    echo "Welcome $i"
done
```

With step size

```
for i in {5..50..5}; do
    echo "Welcome $i"
  done
```

Functions

Defining & Calling functions

```
myfunc() {
    echo "hello $1"
}

# Same as above (alternate syntax)

function myfunc() {
    echo "hello $1"
}


myfunc "John"
```

Returning values

```
myfunc() {
    local myresult='some value'
    echo $myresult
}

result="$(myfunc)"
```

Raising Condition based errors

```
myfunc() {
  return 1
}

if myfunc; then
  echo "success"
else
  echo "failure"
fi
```

Arguments

```
$# -> Number of arguments
$* -> All positional arguments (as a
single word)
$@ -> All positional arguments (as
separate strings)
$1 -> First argument
$_ -> Last argument of the previous
command

Note: $@ and $* must be quoted in
order to perform as described.
Otherwise, they do exactly the same
thing (arguments as separate
strings).

See Special parameters.
```

Conditionals

Conditions

```
[[ -z STRING ]]     - Empty
string
[[ -n STRING ]]     - Not empty
string
[[ STRING == STRING ]]     -
Equal
[[ STRING != STRING ]]     - Not
Equal
[[ NUM -eq NUM ]]   - Equal
[[ NUM -ne NUM ]]   - Not equal
[[ NUM -lt NUM ]]   - Less than
[[ NUM -le NUM ]]   - Less than
or equal
[[ NUM -gt NUM ]]   - Greater
than
[[ NUM -ge NUM ]]   - Greater
than or equal
[[ STRING =~ STRING ]]     -
Regexp
(( NUM < NUM ))     - Numeric
conditions
[[ -o noclobber ]]  - If
OPTIONNAME is enabled
[[ ! EXPR ]] - Not
[[ X && Y ]] - And
[[ X || Y ]] - Or
```

File conditions

```
[[ -e FILE ]] - Exists
[[ -r FILE ]] - Readable
[[ -h FILE ]] - Symlink
[[ -d FILE ]] - Directory
[[ -w FILE ]] - Writable
[[ -s FILE ]] - Size is > 0
bytes
[[ -f FILE ]] - File
[[ -x FILE ]] - Executable
[[ FILE1 -nt FILE2 ]] - 1 is
more recent than 2
[[ FILE1 -ot FILE2 ]] - 2 is
more recent than 1
[[ FILE1 -ef FILE2 ]] - Same
files
```

Examples

```
# String
if [[ -z "$string" ]]; then
  echo "String is empty"
elif [[ -n "$string" ]]; then
  echo "String is not empty"
else
  echo "This never happens"
Fi

# Combinations
if [[ X && Y ]]; then
  ...
Fi

# Equal
if [[ "$A" == "$B" ]]

# Regex
if [[ "A" =~ . ]]

if (( $a < $b )); then
  echo "$a is smaller than $b"
fi

if [[ -e "file.txt" ]]; then
  echo "file exists"
fi
```

Arrays

Defining arrays

```
Fruits=('Apple' 'Banana' 'Orange')

Fruits[0]="Apple"

Fruits[1]="Banana"

Fruits[2]="Orange"
```

Working with arrays

```
echo ${Fruits[0]}              # Element #0
echo ${Fruits[-1]}             # Last element
echo ${Fruits[@]}              # All elements, space-separated
echo ${#Fruits[@]}             # Number of elements
echo ${#Fruits}                # String length of the 1st element
echo ${#Fruits[3]}             # String length of the Nth element
echo ${Fruits[@]:3:2}          # Range (from position 3, length 2)
echo ${!Fruits[@]}             # Keys of all elements, space-separated
```

Operations

```
Fruits=("${Fruits[@]}" "Watermelon")    # Push
Fruits+=('Watermelon')                  # Also Push
Fruits=( ${Fruits[@]/Ap*/} )            # Remove by regex match
unset Fruits[2]                         # Remove one item
Fruits=("${Fruits[@]}")                 # Duplicate
Fruits=("${Fruits[@]}" "${Veggies[@]}") # Concatenate
lines=(`cat "logfile"`)                 # Read from file
```

Iteration

```
for i in "${arrayName[@]}"; do
  echo $i
done
```

Dictionaries

Defining

```
declare -A sounds
```

```
sounds[dog]="bark"

sounds[cow]="moo"

sounds[bird]="tweet"

sounds[wolf]="howl"
```

Declares sound as a Dictionary object (aka associative array).

Working with dictionaries

```
echo ${sounds[dog]} # Dog's sound
echo ${sounds[@]}   # All values
echo ${!sounds[@]}  # All keys
echo ${#sounds[@]}  # Number of elements
unset sounds[dog]   # Delete dog
```

Iteration

Iterate over values

```
for val in "${sounds[@]}"; do
  echo $val
done
```

Iterate over keys

```
for key in "${!sounds[@]}"; do
  echo $key
done
```

Options

Options

```
  set -o noclobber  # Avoid overlay files (echo "hi" > foo)
  set -o errexit    # Used to exit upon error, avoiding cascading errors
  set -o pipefail   # Unveils hidden failures
  set -o nounset    # Exposes unset variables
```

Glob options

```
shopt -s nullglob    # Non-matching globs are removed  ('*.foo' => '')
shopt -s failglob    # Non-matching globs throw errors
shopt -s nocaseglob  # Case insensitive globs
shopt -s dotglob     # Wildcards match dotfiles ("*.sh" => ".foo.sh")
shopt -s globstar    # Allow ** for recursive matches ('lib/**/*.rb' =>
'lib/a/b/c.rb')
```

Set `GLOBIGNORE` as a colon-separated list of patterns to be removed from glob matches.

Miscellaneous

Numeric calculations

```
$((a + 200))      # Add 200 to $a

$(($RANDOM%200))  # Random number 0..199
```

Trap errors

```
trap 'echo Error at about $LINENO' ERR
```

or

```
traperr() {
  echo "ERROR: ${BASH_SOURCE[1]} at
about ${BASH_LINENO[0]}"
}

set -o errtrace
trap traperr ERR
```

Source relative

```
source "${0%/*}/../share/foo.sh"
```

Inspecting commands

```
command -V cd
#=> "cd is a function/alias/whatever"
```

Directory of script

```
DIR="${0%/*}"
```

Case/switch

```
case "$1" in
  start | up)
    vagrant up
    ;;

  *)
    echo "Usage: $0
{start|stop|ssh}"
    ;;
esac
```

Subshells

```
(cd somedir; echo "I'm now in
$PWD")
pwd # still in first directory
```

printf

```
printf "Hello %s, I'm %s" Sven Olga
#=> "Hello Sven, I'm Olga

printf "1 + 1 = %d" 2
#=> "1 + 1 = 2"

printf "This is how you print a float: %f" 2
  #=> "This is how you print a float:

  2.000000"
```

Reading input

```
echo -n "Proceed? [y/n]: "
read ans
echo $ans
read -n 1 ans    # Just one character
```