Bash

Command-line

The bash environment

Environment variables

```
env #lists default environment variables
echo $PATH
echo $USER
echo $PWD
echo $HOME

export b=10.11.1.220 #"export" makes b accessible to all subprocesses
ping -c 2 $b #OK

b=10.11.1.120 #b is only accessible to current shell process, not subprocesses
ping -c 2 $b  #not working

echo "$$" #process ID of current shell instance

var="What's up"
export var2="SUP"

echo $var  #OK
echo $var2 #OK

bash
echo "$$"  #different ID than previous
echo $var  #not working
echo $var2 #OK

exit
echo $var  #OK
```

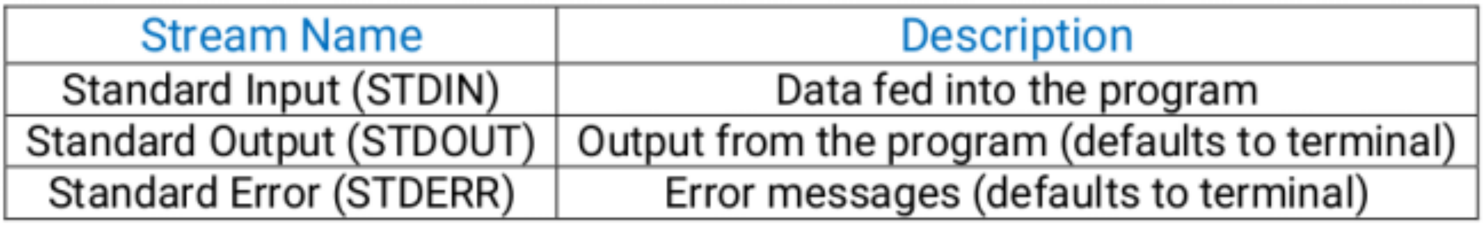
Bash history

```
history #commands history
1 cat /etc/lsb-release
2 clear
3 history

!1 #reissue the first command

sudo systemctl restart apache2

!! #reissue last command

cat /home/kali/.bash_history #history of commands
echo $HISTSIZE
echo $HSTFILESIZE #those two env variables control the size of .bash_history
                  #to change them permanently we can modify .bashrc
```

Search in the history: CTRL+R.
ENTER to execute the found command.

Redirection

```
wc -m < redirection_test.txt #redirect from the text file to count its characters
```



Stream Name	Description
Standard Input (STDIN)	Data fed into the program
Standard Output (STDOUT)	Output from the program (defaults to terminal)
Standard Error (STDERR)	Error messages (defaults to terminal)

STDIN=0
STDOUT=1
STDERR=2

```
ls ./test
#"no such file or directory"
ls ./test 2>error.txt
cat error.txt
#"no such file or directory"
```

Piping

```
cat error.txt | wc -m > count.txt
cat count.txt
```

Text searching/manipulation

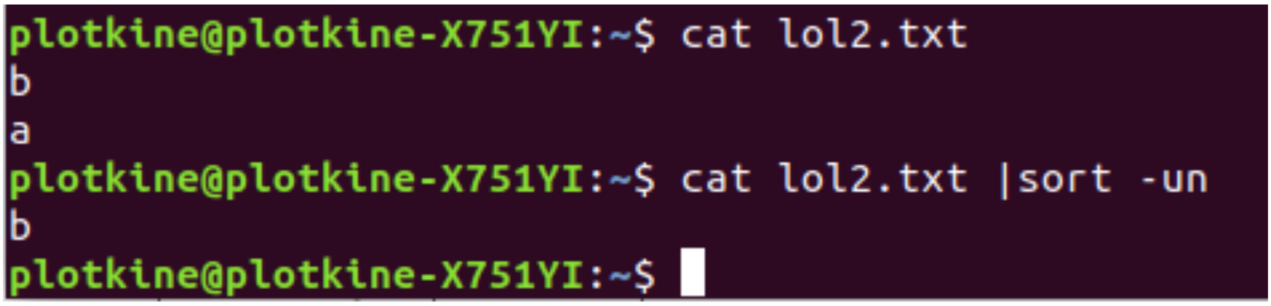
```
ls -la /usr/bin | grep -i zip #useful grep switches: -i to ignore text case, -r for recursive search
echo "I need to try hard" | sed 's/hard/harder/g' #replacing "hard" by "harder"
                                                  #"I need to try harder"
                                                  #/g is for global option
                                                  #/i would mean ignore case
echo "I hack bin, webapps, mobiles and all" | cut -f 2 -d "," #get second element with "," delimiter
                                                               #"webapps"
cut -d ":" -f 1 /etc/passwd #list users
echo "hello::there::friend" | awk -F "::" '{print $1, $3}' #"hello friend"

#extract the user and home directory fields for all users for which the shell is set to /bin/false
cat /etc/passwd |grep /bin/false |awk -F":" '{print "The user " $1 " home directory is " $6}'
```

Difference between cut and awk: cut can only accept one character as delimiter.

Practical example

We want to analyze a zipped log file.

```
gunzip access_log.txt.gz
mv access_log.txt access.log
head access.log
wc -l access.log
cat access.log | cut -d " " -f 1 | sort -u #sort -u for alphabetical order, unique occurences
                                                             #we wanted to list the IP addresses that
appear first in each line
cat access.log | cut -d " " -f 1 | sort | uniq -c | sort -urn #sort for alphabetical order
                                                                                        #we first sort
because uniq will group only adjacent same occurences
                                                                                        #uniq for
unique adjacent occurences, -c for counting
                                                                                        #sort -urn for
biggest occurences first
                                                                                        #-n = sort by
string numerical value
                                                                                        #-r = reverse
order
                                                                                        #-u = unique
```



```
plotkine@plotkine-X751YI:~$ cat lol2.txt
b
a
plotkine@plotkine-X751YI:~$ cat lol2.txt |sort -un
b
plotkine@plotkine-X751YI:~$ 
```

I don't understand what happens

Comparing files

```
comm scan-a.txt scan-b.txt #compare the two text files
                           #column 1 = lines unique to the first file
                           #column 2 = lines unique to the second file
                           #column 3 = lines shared by both files
comm -12 scan-a.txt scan-b.txt #display only lines shared by both files (we delete columns 1 & 2)

diff -c scan-a.txt scan-b.txt #display differences in context format
diff -u scan-a.txt scan-b.txt #display differences in unified format
# The output uses the “-” indicator to show that the line appears in the first file, but not in the
second. Conversely, the “+” indicator shows that the line appears in the second file, but not in the
first.

vimdiff scan-a.txt scan-b.txt #visual difference between files
                              #exit as in vim
```

Process managing

difference between job and process:

```
cat test.txt | wc -l #here we have two processes but a single job
```

backgrounding a process is useful when we launch wireshark or firefox from the terminal, to free it:

```
ping -c 400 localhost > ping_results.txt & #& to background the job right after it starts
                                           #(the shell is now free to execute another command)

ping -c 400 localhost > ping_results.txt
#now press CTRL+Z to stop the process (resume processes adds up to a stack)
bg #resume the latest stopped process (top of stack)

ping -c 400 localhost > ping_results.txt
#now press CTRL+Z to pause the process
find / -name sbd.exe
#now press CTRL+Z to pause the process
jobs #show the paused processes stack
fg %1 #resume process labelled "1" in the stack
fg    #if only one process has been paused, the job number is not necessary
```

In the ping example, the echo reply may come back but if the process is suspended when the packet comes in, the process may miss it.

⇒ Always consider the context of what the commands you are running are doing.

```
ps -ef #-e for all processes, -f for full format listing (UID, PID, PPID,...)
       #quite the same as "ps aux"
ps -fC leafpas #-f for full format, -C to filter command by name
```

File displaying

```
type example.txt #equivalent of cat on Windows

head example.log #first 10 lines of example.log

head -n25 example.log #first 25 lines of example.log

tail -f example.log     #last 10 lines of example.log
                        #-f to display new lines as they are being added to the file
tail -n25 example.log #last 25 lines of example.log

watch -n 5 w #run "w" (show who is logged on and what they are doing)
             #every 5 seconds

#use a combination of watch and ps to monitor the most CPU-intensive processes
watch -n 1 "ps -e --sort=-pcpu |head -n10"
```

Download files

```
wget -O report.pdf https://www.offensive-security.com/reports/penetration-testing-sample-
report-2013.pdf #download the pdf as “report_wget.pdf”

curl -o report.pdf https://www.offensive-security.com/reports/penetration-testing-sample-
report-2013.pdf #same

axel -o report.pdf -n 20 -a https://www.offensive-security.com/reports/penetration-testing-sample-
report-2013.pdf #axel is a download accelerator (useful for large downloads)
                       #same but using 20 simultanate connections (faster download)
                       #-a for concise output
                       #if 403 forbidden change the User-Agent
```

Customize bash environment

```
#By default, duplicate commands and commands beginning with space are removed from the bash history
export HISTCONTROL=ignoredups #remove only duplicate commands (not the ones beginning with space)
export HISTIGNORE ="&:ls:[bf]g:exit:history" #filtering out &,ls,bf,bg,exit and history commands
export HISTTIMEFORMAT='%F %T ' #controls date/timestamps in the output of the "history" command
                               #%F=year-month-day; %T=24-hour time
                               #"man strftime" for more formats

alias #list current aliases
alias lsa='ls -la'
unalias lsa #deleted the "lsa" alias

cat ~/.bashrc #insert here an alias for persistence
              #this file is executed whenever the user logs in
```

Bash scripting

Intro

First line of bash scripts:

```
#!/bin/bash
#!/bin/bash -x
```

-x = debug output (print all script commands and their output)

```
chmod +x hello-world.sh #make script executable
```

Variables

Single vs double quotes

```
lol=Good #no need for quotes if no spaces
lol=Hello World   #error
lol='Hello World' #OK
lol="Hello World" #OK
```

With single quotes, bash interprets every char literally
With double quotes, bash interprets every char literally except "$", "`", and "\":

```
plotkine@plotkine-X751YI:~$ lol='$$'
plotkine@plotkine-X751YI:~$ echo $lol
$$
plotkine@plotkine-X751YI:~$ lol="$$"  #$ isn't interpreted literally
                                      #=> bash recognizes the special char $$
                                      #= PID for current shell process
plotkine@plotkine-X751YI:~$ echo $lol
6891
```

Result of command in variable

```
user=$(whoami) #place result of whoami command in variable user
user=`whoami`  #same but don't use this old syntax
```

Variables scope

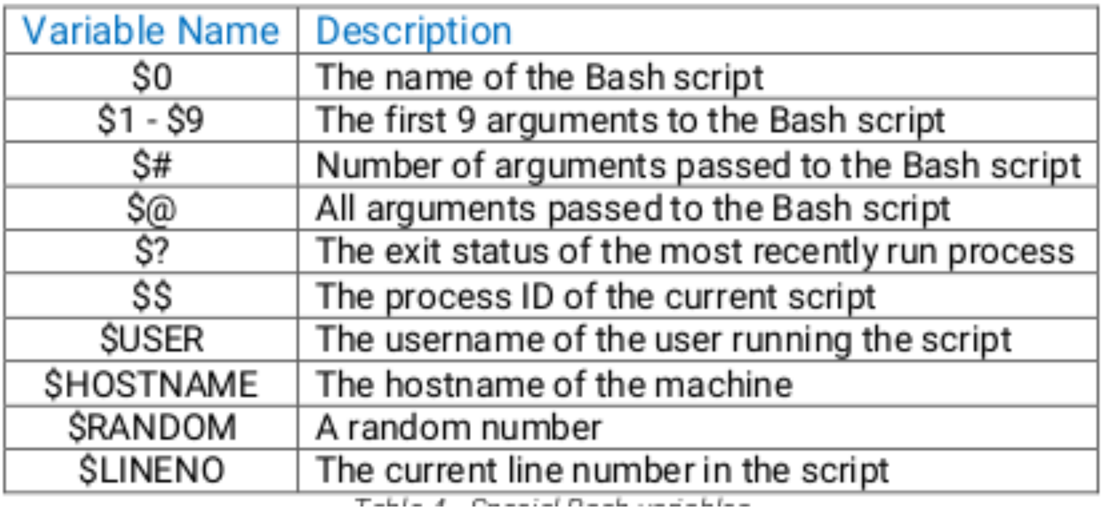
Changes to variables in the subshell will not alter variables from the master process:

Take this script "subshell.sh" for example (-x is for debug output):

```
#!/bin/bash -x
#-x for debug

var1=value1
echo $var1

var2=value2
echo $var2

$(var1=newvar1)
echo $var1 #"var1"

`var2=newvar2`
echo $var2 #"var2"
```

```
kali@kali:~$ ./subshell.sh
+ var1=value1
+ echo value1
value1
+ var2=value2
+ echo value2
value2
++ var1=newvar1
+ echo value1
value1
++ var2=newvar2
+ echo value2
value2
```

Special variables



Variable Name	Description
$0	The name of the Bash script
$1 - $9	The first 9 arguments to the Bash script
$#	Number of arguments passed to the Bash script
$@	All arguments passed to the Bash script
$?	The exit status of the most recently run process
$$	The process ID of the current script
$USER	The username of the user running the script
$HOSTNAME	The hostname of the machine
$RANDOM	A random number
$LINENO	The current line number in the script

Table 1 - Special Bash variables

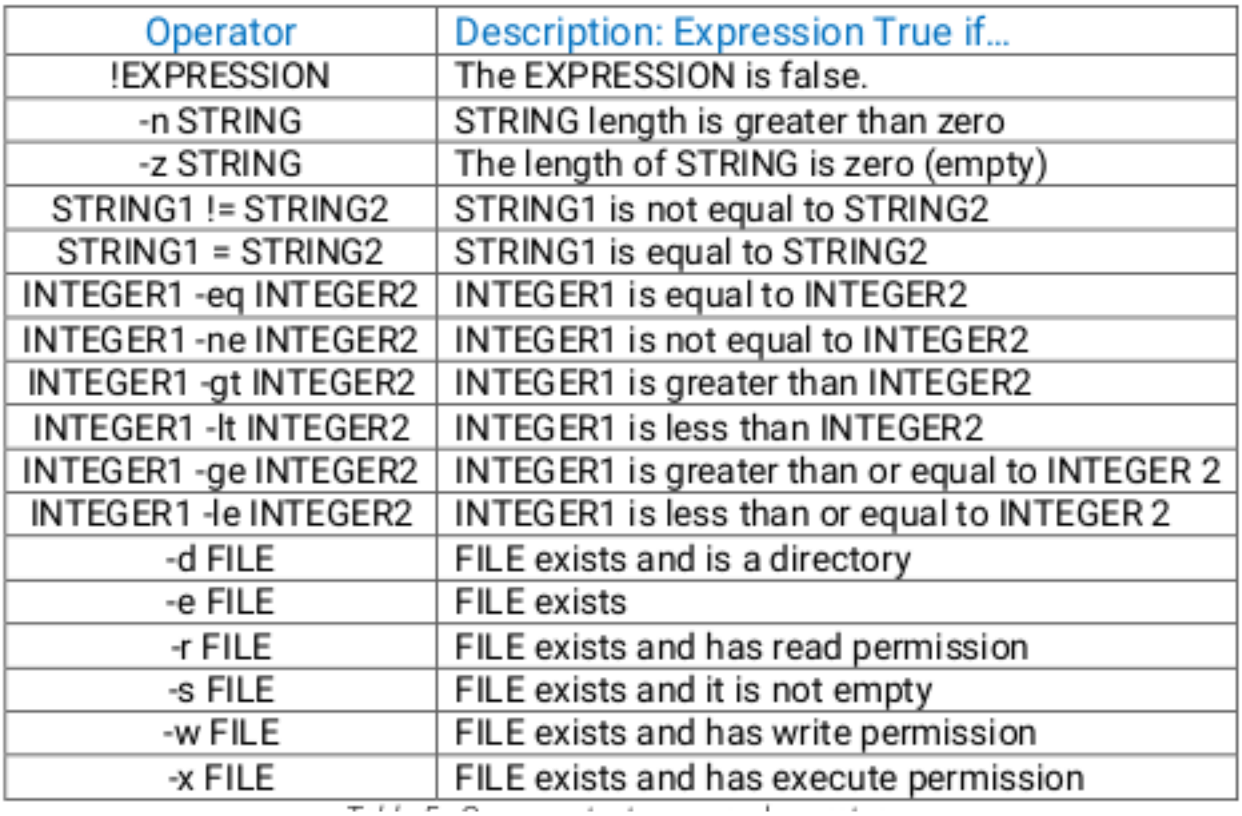
$LINENO is useful for debugging.

User input

```
read answer                        #variable "answer" <- user input
read -p 'Username: ' username      #variable "username" <- user input
                                   #-p = specify a prompt
read -sp 'Password: ' password #-s = silent input (useful for password asking)
```

If/elif/else

```
if [ $variable -lt 16 ]    #-lt = strictly less than
                           #brackets are a reference to the "test" command
elif [ $variable -ge 16 ] #-ge = greater or equal
then
...
else
...
fi

#This is strictly equivalent:
if test $variable -lt 16   #-lt = strictly less than
                           #brackets are a reference to the "test" command
elif test $variable -ge 16 #-ge = greater or equal
then
...
else
...
fi
```



Operator	Description: Expression True if...
!EXPRESSION	The EXPRESSION is false.
-n STRING	STRING length is greater than zero
-z STRING	The length of STRING is zero (empty)
STRING1 != STRING2	STRING1 is not equal to STRING2
STRING1 = STRING2	STRING1 is equal to STRING2
INTEGER1 -eq INTEGER2	INTEGER1 is equal to INTEGER2
INTEGER1 -ne INTEGER2	INTEGER1 is not equal to INTEGER2
INTEGER1 -gt INTEGER2	INTEGER1 is greater than INTEGER2
INTEGER1 -lt INTEGER2	INTEGER1 is less than INTEGER2
INTEGER1 -ge INTEGER2	INTEGER1 is greater than or equal to INTEGER 2
INTEGER1 -le INTEGER2	INTEGER1 is less than or equal to INTEGER 2
-d FILE	FILE exists and is a directory
-e FILE	FILE exists
-r FILE	FILE exists and has read permission
-s FILE	FILE exists and it is not empty
-w FILE	FILE exists and has write permission
-x FILE	FILE exists and has execute permission

Boolean logical operators

In the terminal

| (PIPE)

Passes the output of the first command to the input of the second.

&& (AND)

Executes a command iff the previous command succeeds (returns True or 0).

```
user2=kali
grep $user2 /etc/passwd && echo "$user2 found!"
```

|| (OR)

Executes a command only if the previous command fails (returns False or non-0).

Usually we use this syntax:

```
#we usually use this syntax to run command2 or command3 depending on success of command1
command1 && command2 || command3
```

In a script

&& (AND)

```
#/bin/bash
if [ $USER == 'kali' ] && [ $HOSTNAME == 'kali' ]
then
  echo "Multiple statements are true!"
else
  echo "Not much to see here..."
fi
```

|| (OR)

```
#!/bin/bash
# or example
if [ $USER == 'kali' ] || [ $HOSTNAME == 'pwn' ]
then
  echo "One condition is true, this line is printed"
else
  echo "You are out of luck!"
fi
```

For loops

In the terminal

```
for ip in $(seq 1 10); do echo 10.11.1.$ip; done #";" must be there only for loops in one-liner, not
in scripts
for i in {1..10}; do echo 10.11.1.$i; done
```

In a script

```
#!/bin/bash
for ip in $(seq 1 10)
do
  echo 10.11.1.$ip
done
```

While loops

```
#!/bin/bash
counter=1
while [ $counter -lt 10 ]
do
  echo "10.11.1.$counter"
((counter++)) #double parentheses to perform arithmetic expansion and evaluation at the same time
done
```

Functions

Arguments

```
#!/bin/bash
foo() {
echo "Today's random number is: $1"
}
foo $RANDOM
```

Return value

```
#!/bin/bash
foo() {
echo "Hello"
return $RANDOM
}
foo
echo "The previous function returned $?"
```

If no return value is specified and no error has been encountered the function returns "0" by default.