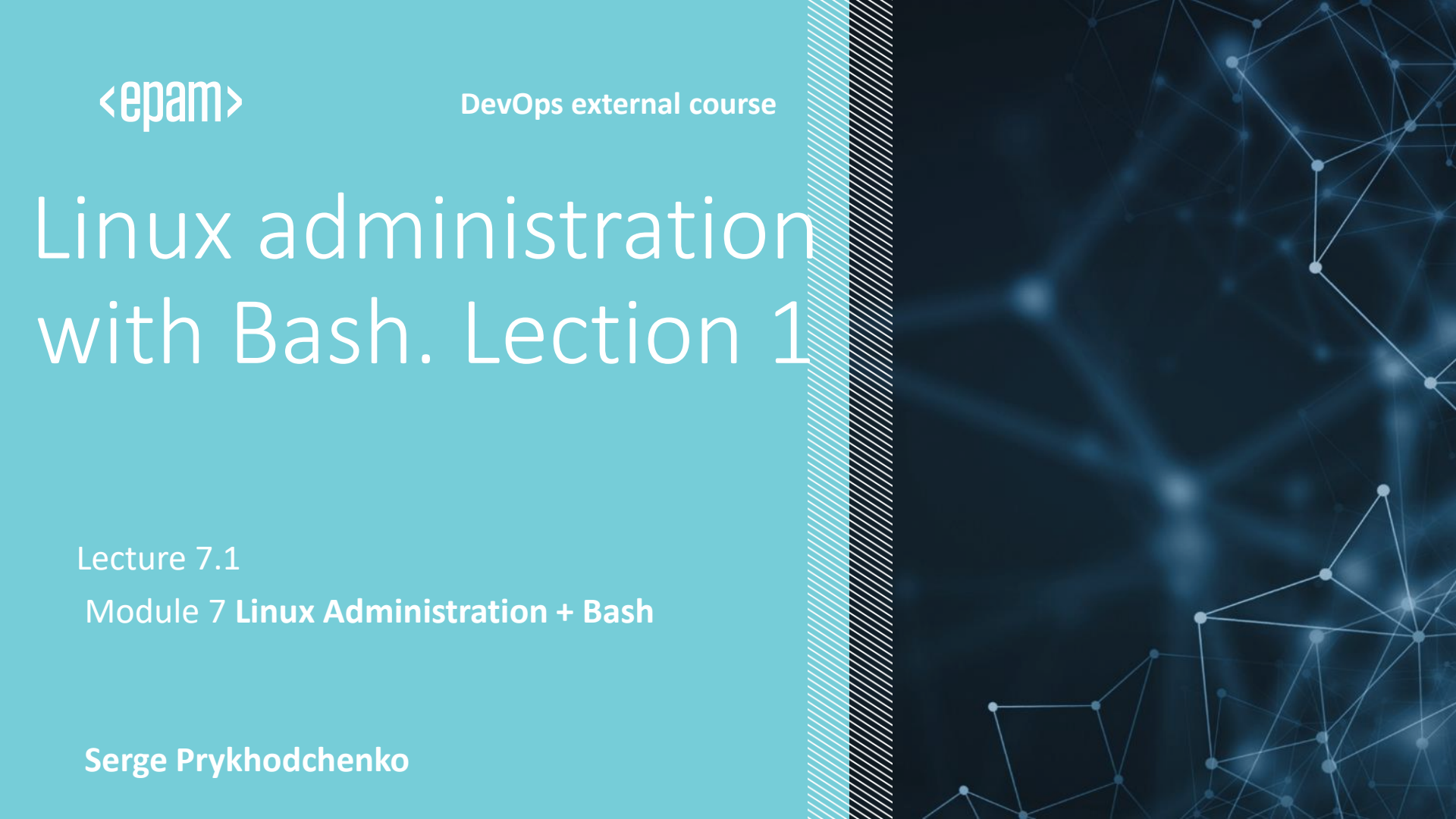

epam
DevOps external course
Linux administration with Bash. Lection 1
Lecture 7.1
Module 7 Linux Administration + Bash
Serge Prykhodchenko

Agenda

- Iptables – cont.
- Shell
- Scripting basics
- Q&A

iptables

Display a list of rules with line numbers.

 # iptables -n -L -v --line-numbers

Display the INPUT or OUTPUT chains of rules.

 # iptables -L INPUT -n -v

 # iptables -L OUTPUT -n -v --line-numbers

iptables

Stop, start, restart the firewall.
By the forces of the system itself:
 # service ufw stop
 # service ufw start
You can also use the iptables commands to stop the firewall and remove all rules:
 # iptables -F
 # iptables -X
 # iptables -t nat -F
 # iptables -t nat -X
 # iptables -t mangle -F
 # iptables -t mangle -X
 # iptables -P INPUT ACCEPT
 # iptables -P OUTPUT ACCEPT
 # iptables -P FORWARD ACCEPT

Where:
-F: Flush all rules.
-X: Remove the chain.
-t table_name: Select a table (nat or mangle) and remove all rules.
-P: Select default actions (such as DROP, REJECT, or ACCEPT).

iptables

Remove firewall rules.
To display the line number with existing rules:
 # iptables -L INPUT -n --line-numbers
 # iptables -L OUTPUT -n --line-numbers
 # iptables -L OUTPUT -n --line-numbers | less
 # iptables -L OUTPUT -n --line-numbers | grep 202.54.1.1
Let's get a list of IP addresses.
Just look at the number on the left and delete the corresponding line. For example, for number 3:
 # iptables -D INPUT 3
Or find the source IP address (202.54.1.1) and remove it from the rule:
 # iptables -D INPUT -s 202.54.1.1 -j DROP
Where:
-D: Remove one or more rules from the chain.

iptables

Add a rule to the firewall.

To add one or more rules to a chain, we first display a list using line numbers:

 # iptables -L INPUT -n --line-numbers

Approximate output:

Chain INPUT (policy DROP)

num target prot opt source destination

1 DROP all -- 202.54.1.1 0.0.0.0/0

2 ACCEPT all -- 0.0.0.0/0 0.0.0.0/0 state NEW,ESTABLISHED

To insert a rule between 1 and 2 lines:

 # iptables -I INPUT 2 -s 202.54.1.2 -j DROP

iptables

Let's check if the rule has been updated:
 # iptables -L INPUT -n --line-numbers

The output will look like this:

Chain INPUT (policy DROP)
num target prot opt source destination
1 DROP all -- 202.54.1.1 0.0.0.0/0
2 DROP all -- 202.54.1.2 0.0.0.0/0
3 ACCEPT all -- 0.0.0.0/0 0.0.0.0/0 state NEW,ESTABLISHED

iptables

Let’s save the firewall rules.
Via iptables-save:
 # iptables-save > /etc/iptables.rules

Restoring the rules.
Via iptables-restore
 # iptables-restore < /etc/iptables.rules

set the default policies.
To drop all traffic:
 # iptables -P INPUT DROP
 # iptables -P OUTPUT DROP
 # iptables -P FORWARD DROP
 # iptables -L -v -n

After the above commands, no packet will leave this host.
 # ping google.com

iptables

Block only incoming connections.
To drop all incoming packets not initiated by you, but allow outgoing traffic:

 # iptables -P INPUT DROP
 # iptables -P FORWARD DROP
 # iptables -P OUTPUT ACCEPT
 # iptables -A INPUT -m state --state NEW,ESTABLISHED -j ACCEPT
 # iptables -L -v -n

Outgoing packets and those that were remembered within the established sessions are allowed.

 # ping google.com

iptables

Blocking a specific IP address.
To block the address of the cracker 1.2.3.4:
 # iptables -A INPUT -s 1.2.3.4 -j DROP
 # iptables -A INPUT -s 192.168.0.0/24 -j DROP

Block incoming port requests.
To block all incoming requests on port 80:
 # iptables -A INPUT -p tcp --dport 80 -j DROP
 # iptables -A INPUT -i eth1 -p tcp --dport 80 -j DROP

To block a request for port 80 from address 1.2.3.4:
 # iptables -A INPUT -p tcp -s 1.2.3.4 --dport 80 -j DROP
 # iptables -A INPUT -i eth1 -p tcp -s 192.168.1.0/24 --dport 80 -j DROP

iptables

Block requests to the outgoing IP address.
To block a specific domain, find out its address:
 # host -t a facebook.com
So, facebook.com has address 69.171.228.40
Let’s find CIDR for 69.171.228.40:
 # whois 69.171.228.40 | grep CIDR
output:
CIDR: 69.171.224.0/19
Block access to 69.171.224.0/19:
 # iptables -A OUTPUT -p tcp -d 69.171.224.0/19 -j DROP
You can also use a domain to block:
 # iptables -A OUTPUT -p tcp -d www.facebook.com -j DROP
 # iptables -A OUTPUT -p tcp -d facebook.com -j DROP

iptables

Record event and reset.
To log the movement of packets before dropping, add a rule:
 # iptables -A INPUT -i eth1 -s 10.0.0.0/8 -j LOG --log-prefix "IP_SPOOF A: "
 # iptables -A INPUT -i eth1 -s 10.0.0.0/8 -j DROP
Let's check the log (by default / var / log / messages):
 # tail -f /var/log/messages
 # grep -i --color 'IP SPOOF' /var/log/messages
Record the event and reset (with a limit on the number of records).
In order not to fill the section with a bloated log, we will limit the number of entries using -m.
For example, to write a maximum of 7 lines every 5 minutes:
 # iptables -A INPUT -i eth1 -s 10.0.0.0/8 -m limit --limit 5/m --limit-burst 7 -j LOG --log-prefix "IP_SPOOF A: "
 # iptables -A INPUT -i eth1 -s 10.0.0.0/8 -j DROP

iptables

Drop or allow traffic from specific MAC addresses.

 # iptables -A INPUT -m mac --mac-source 00:0F:EA:91:04:08 -j DROP

* allow only for TCP port # 8080 from mac address 00: 0F: EA: 91: 04: 07 *

iptables -A INPUT -p tcp --destination-port 22 -m mac --mac-source 00:0F:EA:91:04:07 -j ACCEPT

Allow or deny ICMP Ping requests.

To disable ping:

 # iptables -A INPUT -p icmp --icmp-type echo-request -j DROP

 # iptables -A INPUT -i eth1 -p icmp --icmp-type echo-request -j DROP

Allow for specific networks / hosts:

 # iptables -A INPUT -s 192.168.1.0/24 -p icmp --icmp-type echo-request -j ACCEPT

Allow only part of ICMP requests:

** assumes default inbound policies are set to DROP **

 # iptables -A INPUT -p icmp --icmp-type echo-reply -j ACCEPT

 # iptables -A INPUT -p icmp --icmp-type destination-unreachable -j ACCEPT

 # iptables -A INPUT -p icmp --icmp-type time-exceeded -j ACCEPT

** let us reply to the request **

 # iptables -A INPUT -p icmp --icmp-type echo-request -j ACCEPT

iptables

Open port range.

 # iptables -A INPUT -m state --state NEW -m tcp -p tcp --dport 7000:7010 -j ACCEPT

Open address range.

allow connection to port 80 (Apache) if the address is in the range from 192.168.1.100 to 192.168.1.200

 # iptables -A INPUT -p tcp --destination-port 80 -m iprange --src-range 192.168.1.100-192.168.1.200 -j ACCEPT

example for nat

 # iptables -t nat -A POSTROUTING -j SNAT --to-source 192.168.1.20-192.168.1.25

iptables

Close or open standard ports.

Replace ACCEPT with DROP to block the port.

```
## ssh tcp port 22 ##
iptables -A INPUT -m state --state NEW -m tcp -p tcp --dport 22 -j ACCEPT
iptables -A INPUT -s 192.168.1.0/24 -m state --state NEW -p tcp --dport 22 -j ACCEPT
## cups (printing service) udp/tcp port 631 для локальной сети ##
iptables -A INPUT -s 192.168.1.0/24 -p udp -m udp --dport 631 -j ACCEPT
iptables -A INPUT -s 192.168.1.0/24 -p tcp -m tcp --dport 631 -j ACCEPT
## time sync via NTP для локальной сети (udp port 123) ##
iptables -A INPUT -s 192.168.1.0/24 -m state --state NEW -p udp --dport 123 -j ACCEPT
## tcp port 25 (smtp) ##
iptables -A INPUT -m state --state NEW -p tcp --dport 25 -j ACCEPT
# dns server ports ##
iptables -A INPUT -m state --state NEW -p udp --dport 53 -j ACCEPT
iptables -A INPUT -m state --state NEW -p tcp --dport 53 -j ACCEPT
```

iptables

Limit the number of parallel connections to the server for one address.
The connlimit module is used for restrictions. To allow only 3 ssh connections per client:
 # iptables -A INPUT -p tcp --syn --dport 22 -m connlimit --connlimit-above 3 -j REJECT

Set the number of HTTP requests to 20:
 # iptables -p tcp --syn --dport 80 -m connlimit --connlimit-above 20 --connlimit-mask 24 -j DROP

Where are:
--connlimit-above 3: Specifies that the rule only takes effect if the number of connections exceeds 3.
--connlimit-mask 24: Specifies the netmask.

iptables

The standard set of rules for a web server on port 80 and SSH:

```
# Default policies
iptables -P INPUT DROP
iptables -P FORWARD DROP
iptables -P OUTPUT ACCEPT
# Allow incoming connections from localhost
iptables -A INPUT -i lo -j ACCEPT
# Allow established incoming connections
iptables -A INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT
# Allow HTTP traffic
iptables -A INPUT -p TCP --dport 80 -j ACCEPT
# Allow SSH traffic
iptables -A INPUT -p TCP --dport 22 -j ACCEPT
```

https://wiki.dieg.info/iptables

nftables

nftables is a subsystem of the Linux kernel providing filtering and classification of network packets/datagrams/frames. It has been available since Linux kernel 3.13 released on 19 January 2014.
nftables replaces the legacy iptables portions of Netfilter. Among the advantages of nftables over iptables is less code duplication and easier extension to new protocols. nftables is configured via the user-space utility *nft*, while legacy tools are configured via the
utilities *iptables*, *ip6tables*, *arptables* and *ebtables* frameworks.
nftables utilizes the building blocks of the Netfilter infrastructure, such as the existing hooks into the networking stack, connection tracking system, userspace queueing component, and logging subsystem.

https://en.wikipedia.org/wiki/Nftables

nftables

/etc/nftables

To add config from file do

```
nft -f [filename].conf
```

To drop invalid traffic and allow out

```
ct state invalid counter drop comment "INVALID PACKET DROP"
ct state { established, related } counter accept comment "RELATED CONNECTION ACCEPT"
```

Close bruteforce ssh

```
tcp dport ssh ct state new limit rate 15/minute accept
```

To include

/etc/sysconfig/nftables.conf

https://en.wikipedia.org/wiki/Nftables

https://voxlink.ru/kb/linux/configuring-nftables-on-centos-8-for-ip-telephony-security/

nftables

https://wiki.nftables.org/wiki-nftables/index.php/Main_Page

https://en.wikipedia.org/wiki/Nftables

quagga

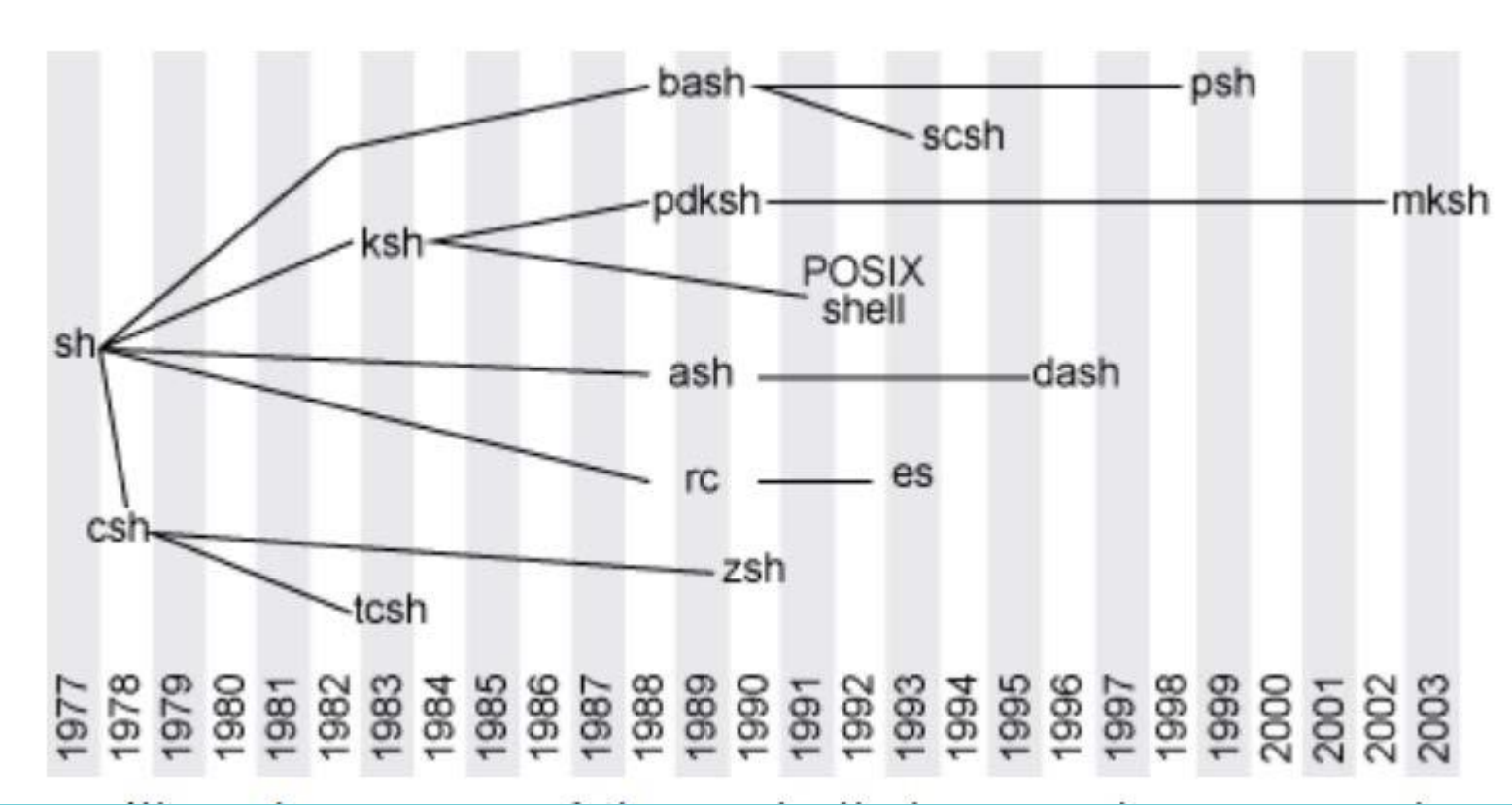
```
sudo apt-get install quagga && sudo mkdir -p /var/log/quagga && sudo chown quagga:quagga /var/log/quagga
```

Create the configuration files:
sudo nano /etc/quagga/bgpd.conf
sudo nano /etc/quagga/isisd.conf
sudo nano /etc/quagga/ospf6d.conf
sudo nano /etc/quagga/ospfd.conf
sudo nano /etc/quagga/pimd.conf
sudo nano /etc/quagga/ripd.conf
sudo nano /etc/quagga/ripngd.conf
sudo nano /etc/quagga/vtysh.conf
sudo nano /etc/quagga/zebra.conf
sudo /etc/init.d/quagga restart

https://wiki.ubuntu.com/JonathanFerguson/Quagga
https://github.com/krobus00/Tubes-JARKOM/wiki/Install-Quagga-di-Ubuntu-20.04

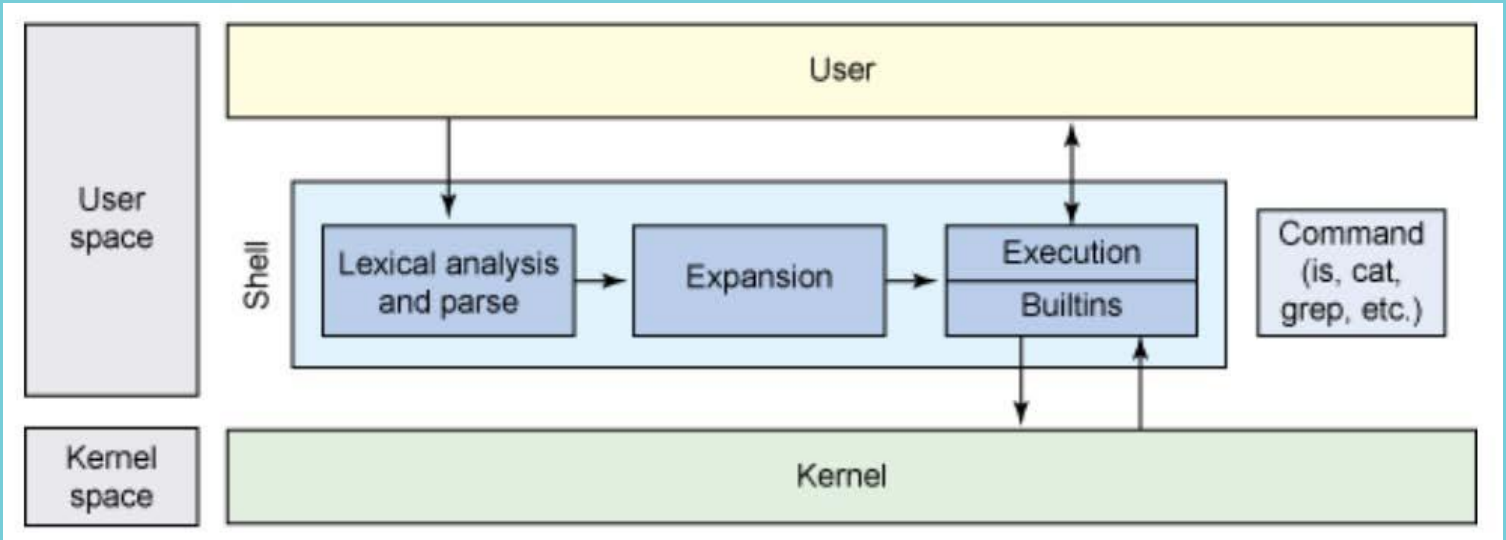
SHELL

Shell Evolution



Basic shell architecture

The fundamental architecture of a hypothetical shell is simple (as evidenced by Bourne's shell). As you can see below, the basic architecture looks similar to a pipeline, where input is analyzed and parsed, symbols are expanded (using a variety of methods such as brace, tilde, variable and parameter expansion and substitution, and file name generation), and finally commands are executed (using shell built-in commands, or external commands)



Bash. Scripting. Summary (1)

When **not** to use shell scripts:
- Resource-intensive tasks, especially where speed is a factor (sorting, hashing, recursion, etc.)
- Procedures involving heavy-duty math operations, especially floating point arithmetic, arbitrary precision calculations, or complex numbers (use C++ or FORTRAN instead)
- Cross-platform portability required (use C or Java instead)
- Complex applications, where structured programming is a necessity (type-checking of variables, function prototypes, etc.)
- Mission-critical applications upon which you are betting the future of the company
- Situations where security is important, where you need to guarantee the integrity of your system and protect against intrusion, cracking, and vandalism
- Project consists of subcomponents with interlocking dependencies
- Extensive file operations required (Bash is limited to serial file access)

Bash. Scripting. Summary (2)

When **not** to use shell scripts:
- Need native support for multi-dimensional arrays
- Need data structures, such as linked lists or trees
- Need to generate / manipulate graphics or GUIs
- Need direct access to system hardware or external peripherals
- Need port or socket I/O
- Need to use libraries or interface with legacy code
- Proprietary, closed-source applications (Shell scripts put the source code right out in the open for all the world to see.)

P.S. If any of the above applies, consider a more powerful scripting language -- perhaps Perl, Python, Ruby - or possibly a compiled language such as C, C++, or Java.
Even then, prototyping the application as a shell script might still be a useful development step.

Bash. Scripting. Summary (2)

Shells like **bash** have support for programming constructs that can be saved as scripts.

These scripts in turn then become more shell commands. Many Linux commands are scripts.

User profile scripts are run when a user logs on and init scripts are run when a daemon is stopped or started.

This means that system administrators also need basic knowledge of scripting to understand how their servers and their applications are started, updated, upgraded, patched, maintained, configured and removed, and also to understand how a user environment is built.

The goal of this module is to give enough information to be able to read and understand scripts. And to become a writer of simple scripts.

Bash. Scripting. Hello World

Just like in every programming course, we start with a simple hello_world script. The following script will output Hello World.

> echo Hello World

After creating this simple script in vi or with echo, you'll have to chmod +x hello_world to make it executable. And unless you add the scripts directory to your path, you'll have to type the path to the script for the shell to be able to find it.

[student@localhost ~]$ echo echo Hello World > hello_world
[student@localhost ~]$ chmod +x hello_world
[student@localhost ~]$ ./hello_world
Hello World
[student@localhost ~]$

Bash. Scripting. She-bang

Let's expand our example a little further by putting #!/bin/bash on the first line of the script. The **#!** is called a she-bang (sometimes called sha-bang), where the she-bang is the first two characters of the script.

```
#!/bin/bash
echo Hello World
```

You can never be sure which shell a user is running. A script that works flawlessly in bash might not work in ksh, csh, or dash. To instruct a shell to run your script in a certain shell, you can start your script with a she-bang followed by the shell it is supposed to run in. This script will run in a bash shell.

```
#!/bin/bash
echo -n hello
echo A bash subshell `echo -n hello
```

Bash. Scripting. Comments. Variables

Let's expand our example a little further by adding comment lines.

```
#!/bin/bash
#
# Hello World Script
#
echo Hello World
```

Here is a simple example of a variable inside a script.

```
#!/bin/bash
#
# simple variable in script
#
var1=3
echo var1 = $var1
```

Bash. Scripting.Variables. Sourcing a script

Scripts can contain variables, but since scripts are run in their own shell, the variables do not survive the end of the script.

[student@localhost ~]$ ./simple_variable_in_script
var1 = 3
[student@localhost ~]$ echo $var1
[student@localhost ~]$

But we can force a script to run in the same shell, this is called ***sourcing*** a script (2 ways).

```
[student@localhost ~]$ source ./simple_variable_in_script
var1 = 3
[student@localhost ~]$ echo $var1
3
[student@localhost ~]$
```

```
[student@localhost ~]$ . ./simple_variable_in_script
var1 = 3
[student@localhost ~]$ echo $var1
3
[student@localhost ~]$
```

Bash. Scripting. Conditions and loops. test[]

The ***test*** command can test whether something is true or false. Let's start by testing whether 10 is greater than 55.

$ test 10 -gt 55 ; echo $?

1

The test command returns ***1*** if the ***test*** fails. And as you see in the next screenshot, ***test*** returns 0 when a test succeeds.

$ test 56 -gt 55 ; echo $?

0

If you prefer true and false, then write the test like this.

$ test 56 -gt 55 && echo true || echo false

true

$ test 6 -gt 55 && echo true || echo false

false

The test command can also be written as square brackets, the screenshot below is identical to the one above.

$ [56 -gt 55] && echo true || echo false

true

$ [6 -gt 55] && echo true || echo false

false

Bash. Scripting. Conditions and loops. test[]

Below are some example tests. Take a look at ***man test*** to see more options for tests.

[-d foo] Does the directory foo exist ?
[-e bar] Does the file bar exist ?
['/etc' = $PWD] Is the string /etc equal to the variable $PWD ?
[$1 != 'secret'] Is the first parameter different from secret ?
[55 -lt $bar] Is 55 less than the value of $bar ?
[$foo -ge 1000] Is the value of $foo greater or equal to 1000 ?
["abc" < $bar] Does abc sort before the value of $bar ?
[-f foo] Is foo a regular file ?
[-r bar] Is bar a readable file ?
[foo -nt bar] Is file foo newer than file bar ?
[-o nounset] Is the shell option nounset set ?

Tests can be combined with logical AND and OR.

```
$ [ 66 -gt 55 -a 66 -lt 500 ] && echo true || echo false
true
$ [ 66 -gt 55 -a 660 -lt 500 ] && echo true || echo false
false
$ [ 66 -gt 55 -o 660 -lt 500 ] && echo true || echo false
true
```

Bash. Scripting. Conditions *If then else*

The ***if then else*** construction is about choice. If a certain condition is met, then execute something, else execute something else. The example below tests whether a file exists, and if the file exists then a proper message is echoed.

```
#!/bin/bash
if [ -f isit.txt ]
        then echo isit.txt exists!
else echo isit.txt not found!
fi
```

If we name the above script 'choice', then it executes like this.

```
$ ./choice
isit.txt not found!
$ touch isit.txt
$ ./choice isit.txt exists!
$
```

Bash. Scripting. Conditions *If then elif*

You can nest a new ***if*** inside an ***else*** with ***elif***. This is a simple example.

```
#!/bin/bash
count=42
if [ $count -eq 42 ]
then
echo "42 is correct."
elif [ $count -gt 42 ]
then
echo "Too much."
else
echo "Not enough."
fi
```

Bash. Scripting. Loops. *for loop*

The example below shows the syntax of a classical ***for loop*** in bash:

```
for i in 1 2 4
do
echo $i
done
```

An example of a for loop combined with an embedded shell:

```
#!/bin/bash
for counter in `seq 1 20`
do
echo counting from 1 to 20, now at $counter
sleep 1
done
```

Bash. Scripting. Loops. *for loop*

The same example as above can be written without the embedded shell using the bash {from..to} shorthand.

```
#!/bin/bash
for counter in {1..20}
do
echo counting from 1 to 20, now at $counter
sleep 1
done
```

This for loop uses file globbing (from the shell expansion). Putting the instruction on the command line has identical functionality.

```
$ ls
count.ksh go.ksh
$ for file in *.ksh ; do cp $file $file.backup ; done
$ ls
count.ksh count.ksh.backup go.ksh go.ksh.backup
```

Bash. Scripting. Loops. *while loop*

Below a simple example of a ***while loop***

```
i=100;
while [ $i -ge 0 ] ;
do
        echo Counting down, from 100 to 0, now at $i;
        let i--;
done
```

Endless loops can be made with
while true or ***while :*** ,
where the ***colon*** is the equivalent of
no operation in the ***bash*** shell.

```
#!/bin/bash
# endless loop while :
do
        echo hello
        sleep 1
done
```

Below a simple example of an ***until loop***

```
let i=100;
until [ $i -le 0 ] ;
do
        echo Counting down, from 100 to 1, now at $i;
        let i--;
done
```

Bash. Scripting. Loops. *for loop*

Write a script that counts the number of files ending in ***.txt*** in the current directory

```
#!/bin/bash
let i=0
for file in *.txt
do
        let i++
done
echo "There are $i files ending in .txt"
```

Wrap an ***if*** statement around the script so it is also correct when there are zero files ending in ***.txt***

```
#!/bin/bash
ls *.txt > /dev/null 2>&1
if [ $? -ne 0 ]
then echo “Directory contains 0 *.txt files"
else
    let i=0
    for file in *.txt
    do
        let i++
    done
    echo " Directory contains $i *.txt files "
fi
```

QUESTIONS & ANSWERS


THANK YOU!