Optimiser son utilisation d'Unix

Bernard Tatin

2013/2017

Résumé — Ce document vient des tréfonds de l'espace temps. Il a débuté il y a bien plus de trois ans de cela, repris de manière plus systématique et se trouve fortement complété aujourd'hui. La première partie rappelle (rapidement) l'histoire et les concepts principaux des shells. La deuxième partie est très orientée sur la recherche de *qui a piraté ma machine* mais peut être d'une grande utilité pour les débutants. La troisième partie, quant à elle, se focalise sur les scripts. Une quatrième partie donnera des notions des outils indispensables pour utiliser correctement son système **Unix**.

Ce document
(en pleine réorganisation)

et ses sources en LaTeX sont disponibles sur GitHub.

Table des matières

Listings

L'HISTOIRE ET LES CONCEPTS

1 Une histoire d'Unix

Voici une (rapide) histoire d'**Unix**, choisie parmi d'autres, parmi celles qui évoluent avec le temps autant parce que des personnages hauts en couleur et ayant réussi à voler la vedette à de plus modestes collègues se font effacer eux-même par de plus brillants qu'eux, soit parce que, vieillissant ils se laissent aller à des confidences inattendues.

En nous basant sur Brève histoire d'**Unix**, on rappelle que *AT&T* travaillait à la fin des années 60, sur un système d'exploitation **Multics** qui devait révolutionner l'histoire de l'informatique. Si révolution il y eut, ce fut dans les esprits : de nombreux concepts de ce système ont influencés ses successeurs, dont **Unix**. Ken Thompson et Dennis Ritchie des fameux *Bell Labs* et qui travaillaient (sans grande conviction, semble-t-il) sur **Multics**, décidèrent de lancer leur propre projet d'OS :

> *baptisé initialement UNICS (UNiplexed Information and Computing Service) jeu de mot avec "eunuchs' (eunuque) pour "un* **Multics** *emasculé", par clin d'œil au projet* **Multics**, *qu'ils jugeaient beaucoup trop compliqué. Le nom fut ensuite modifié en* **Unix**.

cf. *l'article* **Multics** *de Wikipedia*

> *L'essor d'***Unix** *est très fortement lié à un langage de programmation,*

cf. *l'article Brève histoire d'***Unix**

> *le C. À l'origine, le premier* **Unix** *était écrit en assembleur, puis Ken Thompson crée un nouveau langage, le B. En 1971, Dennis Ritchie écrit à son tour un nouveau langage, fondé sur le B, le C. Dès 1973, presque tout* **Unix** *est réécrit en C. Ceci fait probablement d'***Unix** *le premier système au monde écrit dans un langage portable, c'est-à-dire autre chose que de l'assembleur.*

Ce que j'ai surtout retenu de tout cela, c'est qu'**Unix** a banalisé autant l'utilisation des stations de travail connectées en réseau que le concept de *shell*, des système de fichiers hiérarchisés, des périphériques considérés comme de simples fichiers, concepts repris (et certainement améliorés) à **Multics** comme à d'autres. Pour moi, la plus grande invention d'**Unix**, c'est le langage C qui permet l'écriture des systèmes d'exploitations et des logiciels d'une manière très portable. N'oublions pas qu'aujourd'hui encore, C (mais pas C++) est un des langages les plus portable, même s'il commence à être concurrencé par Java par exemple.

2 Les shells

Un shell est une *coquille*, pour reprendre la traduction littérale, autour du système d'exploitation. Voici un magnifique diagramme (d'après ce que l'on trouve sur le WEB comme dans d'anciens ouvrages) donnant une idée du concept :

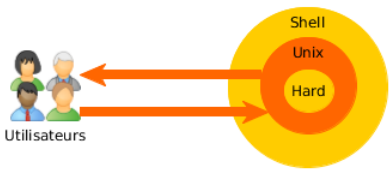

Figure 1 – *shell* **Unix**
Source: le WEB, ouvrages divers

Entre mes débuts dans le monde de l'informatique et aujourd'hui, le concept de shell a quelque peu évolué. Certains qualifient l'explorateur de Windows comme un shell. Ont-ils raison ? Certainement si l'on se réfère à l'image précédente : nos *commandes* (clique, clique et reclique) envoyée au *shell graphique* sont transmises au noyau qui nous renvoie, par l'intermédiaire du *shell graphique*, de belles images. Il faut dire que l'explorateur Windows est le premier contact que l'utilisateur a

avec sa machine. Et sur l'article interface système de Wikipedia, on trouve cette définition :

> *Une* interface système, shell *en anglais, est une couche logicielle qui fournit l'interface utilisateur d'un système d'exploitation. Il correspond à la couche la plus externe de ce dernier.*

Ce même article cite les :

> shells graphiques *fournissant une interface graphique pour l'utilisateur (GUI, pour Graphical User Interface)*

Dans le monde **Unix**, le concept de shell reste plus modeste, même si *Midnight Commander* (`mc`) est parfois considéré comme un shell :

Pour nous et dans tout ce qui suit, nous considérons comme shell :

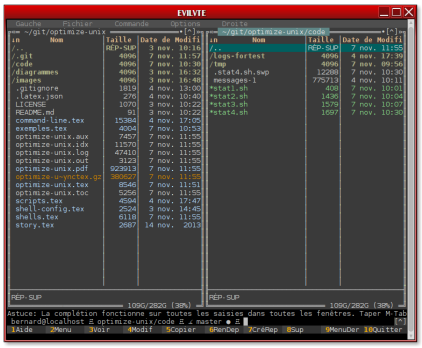

Figure 2 – *mc dans une session Cygwin*
Source: mon PC

> *un interpréteur de commandes destiné aux systèmes d'exploitation* **Unix** *et de type* **Unix** *qui permet d'accéder aux fonctionnalités internes du système d'exploitation. Il se présente sous la forme d'une interface en ligne de commande accessible depuis la console ou un terminal. L'utilisateur lance des commandes sous forme d'une entrée texte exécutée ensuite par le* shell. *Dans les différents systèmes* **Windows**, *le programme analogue est* `command.com` *ou* `cmd.exe`.

*cf.*shell **Unix** sur Wikipedia

2.1 Le fonctionnement Le fonctionnement général est assez simple, surtout si l'on ne tient pas compte de la gestion des erreurs comme dans le graphique suivant qui peut être appliqué à tout bon interpréteur. Seuls les détails de `process command` et `execute command` vont réellement changer.

L'exécution d'un programme suit l'algorithme :

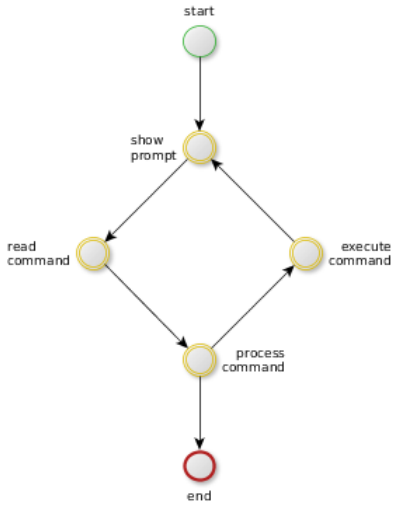

Figure 3 – shell *: fonctionnement général*
Source: créé avec *yEd*

À noter que la commande `exec` se comporte différemment : elle correspond à l'appel système `exec`.

2.2 Quelques shells célèbres

2.2.1 `sh` , le Bourne shell L'ancêtre, toujours vivant et avec lequel sont écrits une grande majorité des scripts actuels. Son intérêt essentiel est justement l'écriture de scripts. Pour l'interaction, il est absolument *nul* mais bien utile parfois pour dépanner.

2.2.2 `csh` , le C shell Il se voulait le remplaçant glorieux de l'*ancêtre* `sh` avec une syntaxe considérée plus lisible car proche du C. Il est de plus en plus abandonné y compris par ses admirateurs les plus fervents, vieillissants dans la solitude la plus complète. Essayez d'écrire un script en csh d'un peu d'envergure sans faire de copié/collé! Il n'y a en effet pas de possibilité de créer des fonctions et, ce qui gêne peut-être encore plus les administrateurs système, il n'y a pas de gestion d'exception. Cependant, il fût certainement

le premier à proposer l'historique des commandes.

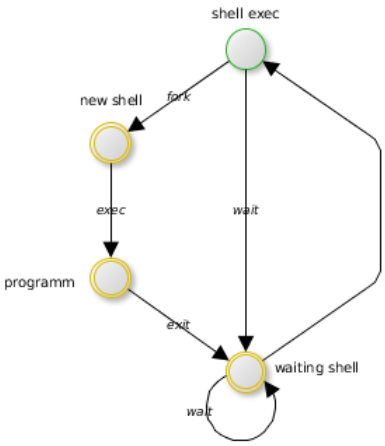

Figure 4 – shell *: exécution d'un programme*
Source: créé avec *yEd*

A noter qu'il fût crée par Bill Joy, l'un des fondateurs historiques de la société Sun Microsystems.

2.2.3 `tcsh` ou le `csh` interactif Le pendant interactif du précédent. Il lui reste des aficionados qui aiment bien sa gestion de l'historique et de la ligne de commande. Il est une *extension* de `csh` , *i.e.*tout ce qui peut-être fait par `csh` est fait par `tcsh` . Sur de nombreux systèmes (Mac OS X entre autre), ces deux shells pointent sur le même exécutable (avec un lien symbolique).

En séquence *nostalgie*, je me souviens que c'est ce shell interactif que j'utilisais sur mon premier **Unix**, en 87/88.

2.2.4 `ksh` , le Korn shell Initialement écrit pour **Unix** par David Korn au début des années 80, ce shell a été repris par Microsoft pour Windows. Compatible avec `sh` , il propose de nombreuses avancées comme beaucoup de fonctionnalités de `tcsh` , des fonctions, des exceptions, des manipulations très évoluées de chaînes de caractères…

2.2.5 `zsh` , le Z shell C'est mon préféré pour l'interactivité, la complétion et bien d'autres choses encore dont il est capable depuis sa création ou presque. Comme `ksh` , il est compilable en bytecode et propose des bibliothèques thématiques comme la couleur, les sockets, la gestion des dates…

2.2.6 `bash` , Bourne Again shell C'est le descendant le plus direct de `sh` . C'est certainement le shell le plus répandu dans le monde Linux aujourd'hui.

Lors de ma découverte de Linux, je l'ai vite abandonné car il était très en retard pour la complétion en ligne de commande par rapport à d'autres, y compris `tcsh` qui commençait pourtant à vieillir un peu. Il a fallu beaucoup d'années (pratiquement 10) pour qu'il en vienne à peu près au niveau de `zsh` .

Aujourd'hui, c'est le shell par défaut de nombreuses distributions Linux et il commence à devenir très utilisé comme shell de script par défaut.

CONFIGURATION ET LIGNE DE COMMANDE

3 La configuration

3.1 Le shell personnel La première des configuration est le choix de son shell par défaut sur son compte personnel. C'est très simple :

```
1 chsh
```

Listing 1 – changer de shell

Aidons-nous du manuel (sous **NetBSD**) :

3.2 Configurer le prompt Sur ma machine virtuelle **NetBSD**, j'obtiens quelque chose comme ceci :

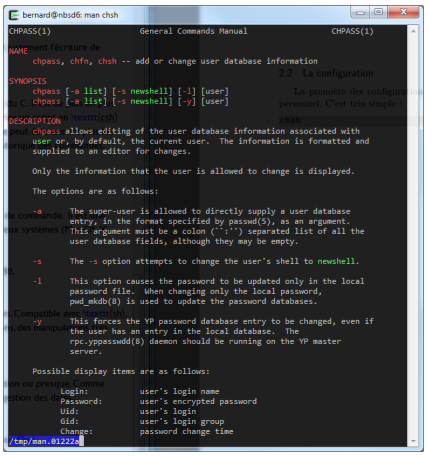

Figure 5 – `man chsh` *sous* **NetBSD**
Source: ma machine virtuelle

Le prompt, ce sont les caractères colorés que l'on voit en début de chaque lignes de commande. Ce prompt m'a aidé, voire sauvé plusieurs fois. Celui-ci m'affiche le nom de l'utilisateur courant en bleu, de la machine en blanc et du répertoire courant en blanc et gras. Lorsque j'ai des sessions sur plusieurs machines, je vois tout de suite où je me trouve avec son nom. Ensuite, lorsque je me déplace de répertoires en répertoires, je n'ai pas besoin de faire d'éternels `pwd` pour savoir où je me trouve. En plus, lorsque je trouve dans un dépôt SVN, j'ai un affichage me donnant les indications sur le répertoire de travail (on ne peut pas le faire sous **Cygwin**) :

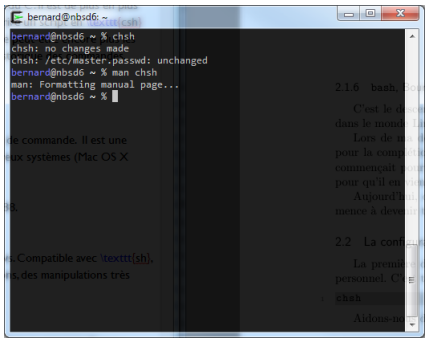

Figure 6 – *un prompt sous* **NetBSD**, *avec* `zsh`
Source: ma machine virtuelle

Pour finir, le nom de l'utilisateur change de couleur lorsque je suis en `root` :

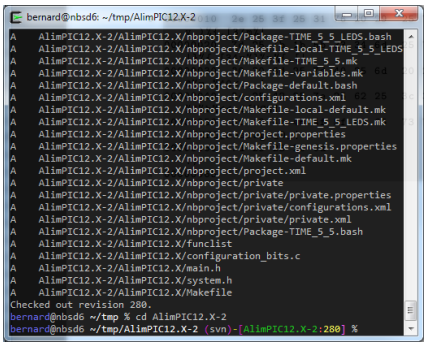

Figure 7 – *dans un répertoire de travail* **SVN**, *avec* `zsh`
Source: ma machine

Tous les shells interactifs de ma connaissance ont au moins un fichier de configuration exécuté au lancement : avec `zsh` , c'est `.zhrc`, avec `bash` , c'est `.bashrc` et avec `csh` , c'est `.cshrc`. Aussi loin que mes souvenirs remontent, on personnalise le prompt avec la variable `PS1` et ce, même pour le MS/-DOS.

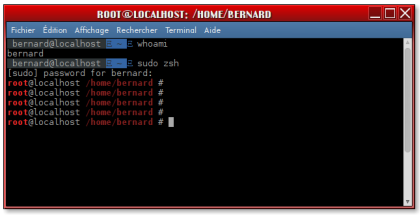

Figure 8 – *en* `root` *avec* `zsh`
Source: ma machine

Voici un hexdump de mon `PS1` :

```
00000000  25 7b 1b 5b 30 31 3b 33  31 6d 25 7d 25 28 3f 2e  |%{.[01;31m%}%(?.|
00000010  2e 25 3f 25 31 76 20 29  25 7b 1b 5b 33 37 6d 25  |.%?%1v )%{.[37m%|
00000020  7d 25 7b 1b 5b 33 34 6d  25 7d 25 6e 25 7b 1b 5b  |}%{.[34m%}%n%{.[|
00000030  30 30 6d 25 7d 40 25 6d  20 25 34 30 3c 2e 2e 2e  |00m%}@%m %40<...|
00000040  3c 25 42 25 7e 25 62 25  3c 3c 20 24 7b 56 43 53  |<%B%~%b%<< ${VCS|
00000050  5f 49 4e 46 4f 5f 6d 65  73 73 61 67 65 5f 30 5f  |_INFO_message_0_|
```

```
00000060  7d 25 23 20 0a                                       |}%
   # .|
00000065
```

Listing 2 – mon prompt

4 La ligne de commande

Pour de multiples raisons déjà plus ou moins évoquées plus haut, j'ai choisi de travailler avec `zsh` comme shell par défaut. C'est ce que nous allons faire ici, autant sous **FreeBSD** que sous **Linux**, tout simplement pour prouver que l'utilisation du shell est assez indépendante du système sous-jacent. Mais comme je sais que certains systèmes viennent avec `bash` ou `tcsh`, sans possibilité de modification, je les évoqueraient donc, en particulier `tcsh` qui est utilisé très souvent, avec `csh`, pour l'administration. Ce n'est que fortuitement que j'examinerais `ksh`, autant par manque d'habitude que parce que je ne l'ai jamais rencontré.

4.1 Les boucles Il y a `while` et `for`.

4.1.1 La boucle `for` On commence par celle-ci car elle en a dérouté plus d'un. Nous avons, avec `zsh` et `bash`, deux syntaxes essentielles. La première *parcourt* un ensemble de données :

```
for file_name in *.txt
do
    cat $file_name
done
```

Listing 3 – une boucle for

Il y a la boucle plus classique pour les spécialistes de Java :

```
for ((i=5; i< 8; i++))
do
```

```
    echo $i
done
```

Listing 4 – une boucle for comme en Java

Avec `tcsh` , nous aurons :

```
foreach file_name (*.txt)
    cat $file_name
end

foreach i (`seq 5 1 8`)
    echo $i
end
```

Listing 5 – parcours d'une liste avec tcsh

4.1.2 La boucle `while` Elle permet de boucles infinies comme celle-ci avec `zsh` , `bash` et `ksh` :

```
while true; do date ``+%T''; sleep 1; done
```

Listing 6 – une boucle while

Avec `tcsh` , nous écrirons en deux lignes :

```
while (1);  date ``+%T''; sleep 1;
end
```

Listing 7 – la même pour tcsh

4.2 Surprises avec `stat`, `find` et `xargs`

4.2.1 `stat` La commande `stat` permet de connaître bon nombre de détails à propos d'un fichier comme ici :

```
bernard@debian7 ~ % stat *
install:
device  70
inode   1837064
mode    16877
nlink   3
uid     1000
gid     1000
rdev    7337007
size    512
atime   1383862259
mtime   1383085660
ctime   1383085660
blksize 16384
blocks  4
link

userstart.tar.gz:
device  70
inode   1837156
mode    33188
nlink   1
uid     1000
gid     1000
rdev    7371584
size    3498854
atime   1383855566
mtime   1383855552
ctime   1383855552
blksize 16384
blocks  6880
link
```

Listing 8 – la commande stat depuis zsh

On obtient, sous `zsh` , un résultat totalement identique sous **NetBSD** et sous **Linux**. Si l'on fait un `which stat`, nous obtenons, sur les deux systèmes, le message `stat:`

`shell built-in command`. C'est ce qui me plait sous `zsh`, les commandes non standard comme `stat` sont remplacées par des fonctions dont le résultat ne réserve pas de surprise. Si je veux une sortie plus agréable et n'afficher que la date de dernière modification (*cf.* The zsh/stat module pour de plus amples explications) :

```
bernard@debian7 ~ % stat -F "%Y-%m-%d %T" +mtime -n *
install 2013-10-29 23:27:40
userstart.tar.gz 2013-11-07 21:19:12
bernard@debian7 ~ %
```

Listing 9 – stat depuis zsh sous Debian

```
bernard@NBSD-64bits ~ % stat -F "%Y-%m-%d %T" +mtime -n *
install 2013-11-14 09:52:56
userstart.tar.gz 2013-11-08 00:54:06
bernard@NBSD-64bits ~ %
```

Listing 10 – stat depuis zsh sous NetBSD

4.2.2 Réfléchissons un peu Grâce à `zsh`, nous avons une méthode extrêmement portable entre Unix pour afficher des données détaillées des fichiers. Pour l'exemple, prenons le `stat` d'origine :

```
bernard@debian7 ~ % /usr/bin/stat --printf="%n %z\n" *
install 2013-10-29 23:27:40.000000000 +0100
userstart.tar.gz 2013-11-07 21:19:12.000000000 +0100
bernard@debian7 ~ %
```

Listing 11 – binaire stat depuis Debian

```
bernard@NBSD-64bits ~ % /usr/bin/stat -t "%Y-%m-%d %T" -f "%Sc
    %N" *
2013-11-14 09:52:56 install
2013-11-08 00:54:06 userstart.tar.gz
bernard@NBSD-64bits ~ %
```

Listing 12 – binaire stat depuis NetBSD

4.2.3 Tous les fichiers du monde Si je veux faire la même chose que précédemment, mais sur tous les fichiers de la machine, on peut tomber sur ce message d'erreur :

```
bernard@debian7 ~ % stat -F "%Y-%m-%d %T" +ctime -n $(find / -
    name "*")
zsh: liste d'arguments trop longue: stat
bernard@debian7 ~ %
```

C'est là que `xargs` entre en jeu, mais avec un nouveau problème :

```
bernard@debian7 ~ % find / -name "*" | xargs stat -F "%Y-%m-%d %
    T" +ctime -n
stat: option non valide -- F
Saisissez `` stat --help `` pour plus d'informations.
...
stat: option non valide -- F
Saisissez `` stat --help `` pour plus d'informations.
123 bernard@debian7 ~ %
```

Listing 13 – stat rien ne va plus

La commande `xargs` va chercher non pas la fonction de `zsh` mais le binaire qui se trouve sur le `PATH` de la machine. On doit donc faire :

```
bernard@debian7 ~ % find . -name "*" | xargs stat --printf="%n %
    z\n"
...
./.w3m 2013-11-07 23:07:03.000000000 +0100
./.w3m/configuration 2013-11-07 23:03:48.000000000 +0100
./.w3m/history 2013-11-07 23:06:29.000000000 +0100
./.w3m/cookie 2013-11-07 23:06:29.000000000 +0100
./.viminfo 2013-11-07 23:07:03.000000000 +0100
bernard@debian7 ~ %
```

Listing 14 – stat tout rentre dans l'ordre sous Debian

Sous **NetBSD** :

```
bernard@NBSD-64bits ~ % find . -name "*" | xargs stat -t "%Y-%m
    -%d %T" -f "%N %Sc"
...
./.lesshst 2013-11-08 01:02:35
./install 2013-11-14 09:52:56
./.zshrc.private~ 2013-11-08 01:13:05
./.viminfo 2013-11-08 01:14:38
./.xinitrc 2013-11-08 01:14:44
./.Xauthority 2013-11-08 01:16:25
bernard@NBSD-64bits ~
```

Listing 15 – stat tout rentre dans l'ordre sous NetBSD

4.2.4 Application pratique Sur le serveur, qui est sous **Linux**, faisons la même chose ou presque, on place la date en premier et c'est la surprise du jour :

```
[bigserver] (688) ~ % find /etc -name  "*" | xargs stat --printf
    ="%z %n\n" | sort
find: "/etc/ssl/private": Permission non accordee
stat: option invalide -- 'o'
Pour en savoir davantage, faites:  stat --help .
[bigserver] (689) ~ %
```

Listing 16 – problèmes de droits sous Linux?

En rajoutant l'option `-printo` à find, l'option `-o` à `xargs`, nous obtenons le bon résultat :

```
[bigserver] (689) ~ % find /etc -name  "*" -printo | xargs -o
    stat --printf="%z %n\n" | sort
...
2013-11-12 10:51:42.041176453 +0100 /etc/php5/conf.d/ldap.ini
2013-11-12 10:55:05.017425662 +0100 /etc/php5/cgi
2013-11-12 10:55:05.017425662 +0100 /etc/php5/cgi/php.ini
2013-11-13 14:55:28.001191244 +0100 /etc/apache2/sites-
    available/aenercom.preprod.conf
```

```
2013-11-13 14:56:35.601352228 +0100 /etc/apache2/sites-
    available/device.sigrenea.conf
2013-11-13 14:56:35.601352228 +0100 /etc/apache2/sites-enabled
2013-11-13 17:16:04.377217037 +0100 /etc/apache2/sites-
    available
2013-11-13 17:16:04.377217037 +0100 /etc/phpmyadmin
2013-11-14 01:03:38.589252417 +0100 /etc/php5/conf.d/mysqli.ini
2013-11-14 01:05:14.997350491 +0100 /etc/php5/conf.d
2013-11-14 01:05:14.997350491 +0100 /etc/php5/conf.d/mcrypt.ini
```

Listing 17 – problèmes résolus!

En fait, les noms de fichier sous **Unix** peuvent contenir beaucoup de caractères étranges en dehors de `/`. `xargs` prend le caractère `LF` comme fin d'enregistrement de la part de son entrée standard. Si jamais un fichier contient ce caractère, plus rien ne va. Les nouvelles options permettent à `find` d'utiliser `0x00` comme séparateur d'enregistrement et à `xargs` de bien l'interpréter.

Il y a aussi une autre explication, depuis bien longtemps les outils **GNU** fonctionnent comme ceci et ce n'est que très récemment que le couple `find`/`xargs` fonctionne ainsi.

Après toutes ces considérations, on constate que le 14 Novembre 2013, un peu après 1 heure du matin, quelqu'un a modifié les fichiers `/etc/php5/conf.d/mysqli.ini` et `/etc/php5/conf.d/mcrypt.ini`, tout simplement pour remplacer les commentaires de type shell par des commentaires de type fichier ini.

Après une attaque du serveur, il est intéressant de faire le même exercice sur les répertoires vitaux comme `/bin`. Pour éviter des listings trop important, on limite la sortie à l'année 2013 et on fait une jolie boucle :

```
[bigserver] (694) ~ % for d in /bin /sbin /lib /lib32 /usr/bin
    /usr/sbin /usr/lib /usr/lib32; do
find $d -name  "*" -print0 | xargs -0  stat --printf="%z %n\n" |
    egrep "^2013"
done | sort
```

Listing 18 – recherche d'une attaque part I

Nous obtenons un listing fort long, correspondant aux mises à jour faites le 8 et le 12 Novembre. Maintenant que nous savons que l'attaque a eu lieu avant le 8 Novembre, on sélectionne plus sévèrement :

```
[bigserver] (695) ~ % for d in /bin /sbin /lib /lib32 /usr/bin
   /usr/sbin /usr/lib /usr/lib32; do
find $d -name  "*" -print0 | xargs -0  stat --printf="%z %n\n" |
    egrep "^2013-11-0[1-7]"
done | sort
[bigserver] (696) ~ %
```

Listing 19 – recherche d'une attaque part II

Cependant, rien ne prouve que nous n'avons pas eu de désordres un peu avant ou un peu pendant. Comme le gros des fichiers est dans `/usr/lib`, éliminons le de la liste :

```
[bigserver] (695) ~ % for d in /bin /sbin /lib /lib32 /usr/bin
   /usr/sbin; do
find $d -name  "*" -print0 | xargs -0  stat --printf="%z %n\n" |
    egrep "^2013"
done | sort
...
[bigserver] (696) ~ %
```

Listing 20 – recherche d'une attaque part III

Nous n'avons des modifications qu'entre le 8 et le 12 Novembre.

Plus fort encore, afficher les fichiers modifiés ce jour :

```
find / -name  "*" -print0 | xargs -0  stat --printf="%z %n\n" |
   egrep "^$(date '+%Y-%m-%d')" | sort
```

Listing 21 – recherche d'une attaque part IV

On est débordé par l'affichage des fichiers système de Linux. Pour palier à cet inconvénient, on demande à `find` d'abandonner les répertoire `/sys` et `/proc` :

```
find / \( -path /proc -o -path /sys \) -prune -o -name  "*" -
    print0 | xargs -0  stat --printf="%z %n\n" | egrep "^$(date
    '+%Y-%m-%d')" | sort
```

Listing 22 – recherche d'une attaque part V et fin

4.3 Les surprises de sudo Reprenons l'exemple précédent en redirigeant la sortie standard vers `/dev/null` :

```
find / \( -path /proc -o -path /sys \) -prune -o -name  "*" -
    print0 | xargs -0  stat --printf="%z %n\n" | egrep "^$(date
    '+%Y-%m-%d')" > /dev/null
```

Listing 23 – avec redirection

On aura une sortie comme celle-ci :

```
find: "/var/lib/postgresql/9.1/main": Permission non accordee
find: "/var/lib/sudo": Permission non accordee
find: "/var/cache/ldconfig": Permission non accordee
find: "/var/log/exim4": Permission non accordee
find: "/var/log/apache2": Permission non accordee
...
```

Listing 24 – problèmes de droits part I

Pour éliminer les `find: ... Permission non accordee`, on utilise `sudo` :

```
sudo find / \( -path /proc -o -path /sys \) -prune -o -name  "*"
     -print0 | xargs -0  stat --printf="%z %n\n" | egrep "^$(
    date '+%Y-%m-%d')" > /dev/null
```

Listing 25 – problèmes de droits part I en pire

C'est pire :

```
...
stat: impossible d'evaluer ' /root/.aptitude ': Permission non
    accordee
```

```
stat: impossible d'evaluer ' /root/.aptitude/cache ':
   Permission non accordee
stat: impossible d'evaluer ' /root/.aptitude/config ':
   Permission non accordee
stat: impossible d'evaluer ' /root/.viminfo ': Permission non
   accordee
stat: impossible d'evaluer ' /root/.bash_history ': Permission
   non accordee
...
```

Listing 26 – problèmes de droits part I en pire certifié

Nous avons demandé à `sudo` de traité `find` et avec le *pipe*, nous demandons à `xargs` de traiter les lignes de sorties avec `stat`. Ce dernier récupère un nom de fichier et le traite comme tel mais comme il n'est pas lancé avec `sudo`, nous avons ces erreurs. Essayons ceci :

```
sudo find / \( -path /proc -o -path /sys \) -prune -o -name  "*"
    -print0 | xargs -0  sudo stat --printf="%z %n\n" | egrep "^
   $(date '+%Y-%m-%d')" > /dev/null
```

Listing 27 – problèmes de droits part II

C'est pas mieux, autant sous **Linux** que sous **NetBSD** :

```
sudo: unable to execute /usr/bin/stat: Argument list too long
sudo: unable to execute /usr/bin/stat: Argument list too long
sudo: unable to execute /usr/bin/stat: Argument list too long
sudo: unable to execute /usr/bin/stat: Argument list too long
sudo: unable to execute /usr/bin/stat: Argument list too long
sudo: unable to execute /usr/bin/stat: Argument list too long
```

Listing 28 – problèmes de droits part II avec un pir

Il faut bien l'avouer, je ne sais pas quoi dire de plus ici - sinon noter un *TODO : comprendre ce qui ce passe*. Ce qui est bien avec **Unix**, c'est qu'il y a toujours un moyen de s'en sortir. On remarquera quelques différences entre les mondes **Linux** et **BSD**, en particulier

avec `man sh`, où le premier nous renvoie sur `bash` alors que le second traite bien directement de `sh` . Dans tous les cas, `sudo sh -c "..."` est notre amie et nous obtenons avec **NetBSD** :

```
bernard@NBSD-64bits ~ % sudo sh -c "find / \( -path /proc -o -
    path /sys \) -prune -o -name '*' -type f | xargs stat -t '%Y
    -%m-%d %T' -f '%Sc %N' | egrep '^$(date "+%Y-%m-%d")' | sort
    "
2013-11-15 08:55:19 /var/run/dmesg.boot
2013-11-15 08:55:22 /var/log/messages
4 2013-11-15 08:55:22 /var/run/ntpd.pid
2013-11-15 08:55:22 /var/run/powerd.pid
6 2013-11-15 08:55:22 /var/run/sshd.pid
```

Listing 29 – problèmes de droits part II et fin du pire pour NetBSD

Et sous **Linux** :

```
bernard@debian7 ~ % sudo sh -c "find / \( -path /proc -o -path /
    sys \) -prune -o -name  '*' -print0 | xargs -0  stat --
    printf='%z %n\n' | egrep '^$(date "+%Y-%m-%d")'"
2 2013-11-15 09:54:42.000000000 +0100 /var/lib/misc/statd.status
2013-11-15 09:54:41.000000000 +0100 /var/lib/urandom/random-
    seed
4 2013-11-15 09:55:09.000000000 +0100 /var/lib/dhcp/dhclient.em0.
    leases
2013-11-15 09:54:49.000000000 +0100 /var/lib/exim4
6 2013-11-15 09:54:49.000000000 +0100 /var/lib/exim4/config.
    autogenerated
2013-11-15 09:54:45.000000000 +0100 /var/lib/postgresql/9.1/
    main
8 2013-11-15 09:54:46.000000000 +0100 /var/lib/postgresql/9.1/
    main/global
...
```

Listing 30 – problèmes de droits part II et fin du pire pour Debian

4.4 POSIX et GNU Profitons d'un moment de calme pour remarquer que de nombreuses commandes se comportent de manière très standard entre différents systèmes, y compris parfois, sous **MS/DOS**. Tout cela vient de **POSIX** ou de **GNU**.

Les guerres de religions qui opposent parfois violemment les mondes **BSD** et **Linux**, les supporters de `Vi` et `Emacs`, ... finissent par être absorbées avec le temps et seuls quelques irréductibles les raniment, souvent plus pour s'exposer aux yeux (blasés maintenant) du petit monde concerné. Seule reste l'opposition farouche entre tenants du libre et leurs opposants.

Ici, nous avons utilisé `find` de la même manière sous **Linux** et sous **NetBSD** ce qui n'a pas été toujours le cas, de même, `sh` se comporte de manière identique à quelques octets près sur les deux systèmes, ce qui n'était pas forcément vrai il y a quelques années. Pour revenir à `find`, nous avions un paquet de compatibilité **GNU** disponible sur plusieurs **BSD** qui reprenait le `find` que nous connaissons maintenant et l'on pouvait différencier `gfind` de `bsdfind`[1].

1. A vérifier dans les détails.

LES SCRIPTS ET LES EXEMPLES

5 Les scripts shell

La magie des shellsest infinie, ils nous permettent en effet de créer des programmes complets, complexes… parfois aux limites du lisible. On les appelle *scripts* pour les opposer aux applications généralement créées à partir de langages compilés mais cela ne devrait rien changer au fait qu'ils doivent être conçus avec un soin égal à celui apporté aux autres langages comme *C/C++*, *Java*…

Dans tout ce qui suit, il ne faut pas perdre de vue que le shellest une *coquille* entourant le noyau d'**Unix**. Certains aspects des shellsne font que recouvrir des appels systèmes.

5.1 Structure des scripts Ce qui est décrit ici est valable autant pour des langages interprétés comme l'horrible *Perl*[2], le sublime *Scheme*[3], le célèbre *Python* que pour n'importe quel *shell*.

La première ligne d'un script est le *shebang*. Cette ligne est très importante car elle indique de manière sûre quel interpréteur il doit utiliser pour exécuter le corps du script.

2. je ne suis pas objectif, mais quand même…
3. là, je me sens plus objectif… ou presque.

Voici quelques exemples :

sh : `#!/bin/sh`

bash : `#!/bin/sh`

Perl : `#!/usr/bin/env perl`

Python 2.7 : `#!/usr/bin/env python2.7`

Python : `#!/usr/bin/env python`

awk : `#!/bin/awk -f`

Les deux caractères `#!` sont considérés comme un nombre magique par le système d'exploitation qui comprend immédiatement qu'il doit utiliser le script dont le nom et les arguments suivent les deux caractères.

Dans un shellinteractif, l'exécution d'un script suit l'algorithme suivant :

```
fork ();
if (child) {
    open(script);
    switch(magic_number) {
        case 0x7f'ELF':
            exec_binaire();
            break;
        case '\#!':
            load_shell(first_line);
            exec_binaire(shellname, args);
            break;
        ...
    }
} else {
    wait(child);
}
```

5.2 Choisir son shell Par tradition autant que par prudence, on écrit ses scripts shellpour le shelld'origine, soit `sh` . Par prudence car on est certain qu'il sera présent sur la machine même si elle démarre en mode dégradé. Cependant, beaucoup de scripts sont *applicatifs* et ne pourront pas fonctionner en mode dégradé. Autant se servir d'un shellplus complet comme `bash` .

5.3 Les paramètres des scripts Les paramètres, leur nombre et leur taille n'ont de limites que de l'ordre de la dizaine de Ko. Il faut donc pouvoir y accéder. Le paramètre `$0` est le nom du script parfois avec le nom du répertoire. Les neufs suivants sont nommés `$1`, ..., `$9`. Pour accéder aux autres il faut ruser un peu avec l'instruction `shift`.

5.4 Tests et boucles Les tests se font avec `if` de cette manière :

```
if condition
then
    ...
else
    ...
fi
```

Dans le même ordre d'idée, nous avons le `while` :

```
while condition
do
done
```

La construction des `condition` est tout un art, d'autant plus qu'en lieu et place du `if` nous pouvons écrire :

```
condition && condition_true && ...
```

ou bien :

```
condition || condition_false
```

Nous avons aussi une boucle `for` :

```
for index in ensemble
do
    ...
done
```

La détermination de `ensemble` est assez naturelle comme par exemple avec `$(ls *.java)`. Mais il faut être prudent : selon les shellsles résultats peuvent différer.

5.5 Conditions, valeurs de retour des programmes Tout les programmes sous **Unix** s'achèvent par un `return EXIT_CODE` ou bien un `exit(EXIT_CODE)` bien senti. La valeur `EXIT_CODE` est renvoyée au programme appelant, notre shell. On peut le récupérer depuis la variable `$#?` puis étudier le cas :

```
myprogram arg1 arg2 ...
case \$\#? in
    0)
        its-okayyy
        ;;
    1)
        bad_parameterzzz
        ;;
    2|3|4)
        cant-open-filezzz
        ;;
    *)
        unknow-error
        ;;
esac
```

Unix considère que la valeur de retour 0 est signe que tout va bien et que tout autre valeur exprime une condition d'erreur. On peut donc utiliser cette propriété ainsi :

```
myprogram arg1 arg2 ... || onerror ``Error code \$\#?''
```

5.6 Redirections et tubes (ou pipes) Le premier piège dans l'utilisation des tubes dans un scripts est simple : pour chaque tube, on crée un nouveau processus. Ainsi le script suivant ne renvoie pas le résultat escompté :

```
#!/bin/sh

compteur=0

ls -l /bin | while read line; do
    compteur=$(( compteur + 1 ))
done

printf "Il y a %d fichiers dans /bin\n" ${compteur}
```

code/bad1.sh

```
$ ./bad1.sh
Il y a 0 fichiers dans /bin
```

La variable compteur fait partie de l'environnement de `bad1.sh`. Lorsque la boucle `while` se lance, elle est dans un nouveau contexte et sa modification se perd à la fin de la boucle. On corrige de cette manière :

```
#!/bin/sh

```

```
compteur=0

ls -l /bin > /tmp/ls-l.tmp
while read line; do
    compteur=$(( compteur + 1 ))
done < /tmp/ls-l.tmp

printf "Il y a %d fichiers dans /bin\n" ${compteur}
```

code/not-so-bad1.sh

```
$ ./not-so-bad1.sh
Il y a 173 fichiers dans /bin
```

6 Exemples de manipulation de texte

6.1 Des stats Voici un extrait d'un fichier `/var/log/messages` :

```
Nov  3 10:16:19 localhost org.gnome.zeitgeist.SimpleIndexer
   [2637]: ** \ldots
Nov  3 10:16:34 localhost org.freedesktop.FileManager1[2637]:
   Initializing \ldots
Nov  3 10:16:34 localhost nautilus: [N-A] Nautilus-Actions Menu
    Extender 3.2\ldots
Nov  3 10:16:34 localhost org.freedesktop.FileManager1[2637]:
   Initializing naut\ldots
Nov  3 10:16:34 localhost nautilus: [N-A] Nautilus-Actions
   Tracker 3.2.3 initializing\ldots
```

Nous voulons déterminer les moments les plus actifs de ce fichier avec une granularité de une heure. La manipulation est simple :

afficher le fichier `cat file-name`,

découper le ficher `cut -d ':' -f 1`,

trier le fichier `sort`,

compter les occurences `uniq -c`,

trier en décroissant `sort -n`.

Ce qui nous donne la commande :

```
cat $file-name |
    cut -d ':' -f 1 |
        sort |
            uniq -c |
                sort -n
```

On obtient rapidement un script (`stat1.sh`) à partir de cette ligne de commande :

```
#!/bin/sh

scriptname="$(basename $0)"

dohelp() {
    cat << DOHELP
${scriptname} [-h|--help] : this text
${scriptname} file file\ldots : stats
DOHELP
    exit 0
}

[ $# -eq 0 ] && dohelp
case $1 in
    -h | --help)
        dohelp
        ;;
    *)
        cat "$@" | \
```

```
            cut -d ':' -f 1 | \
                sort | \
                    uniq -c | \
                        sort -n
        ;;
esac
```

code/stat1.sh

On peut tester :

```
$ ./stat1.sh messages-1 /var/log/messages /var/log/messages.1
...
  152 Nov  4 10
  155 Oct 27 11
  156 Oct 28 09
  164 Oct 28 17
  186 Oct 25 14
  213 Oct 28 11
  216 Nov  3 10
  260 Oct 28 10
  636 Oct 26 15
  774 Oct 28 07
 1770 Nov  3 09
 3844 Oct 28 14
26201 Oct 28 15
```

6.2 Peut-on faire mieux ? Bien sûr ! On peut avoir d'autres options que la simple aide, on peut aussi gérer correctement les erreurs, les *signaux* **Unix**…

6.2.1 Les options Depuis longtemps il existe une norme **POSIX** permettant de gérer les options de la ligne de commande. Malheureusement, il fut une époque où la norme avait beaucoup de variantes ce qui m'a poussé à faire ma propre gestion de ces paramètres.

Voici ma méthode, facile à mémoriser mais pas parfaite et un peu lourde :

l'aide créer une fonction `dohelp` comme dans l'exemple précédent ; le nom `dohelp` permet d'éviter un clash avec une éventuelle commande `help`.

s'assurer de l'existence de paramètres il suffit de faire le test `[ $# -eq 0 ]` et exécuter le code nécessaire.

vider la liste des paramètres une boucle `while [ $# -ne 0 ]` fait l'affaire.

Voici un exemple plus parlant (script `stat2.sh`) :

```
#!/bin/sh

scriptname="$(basename $0)"

dohelp() {
    cat << DOHELP
${scriptname} [-h|--help] : this text
${scriptname} [options] file file... : stats
options:
    -s|--size N : number of most important hours, default 5
    -b|--byhour : for each hours
DOHELP
    exit 0
}

size=5
byhour="cut -d ':' -f 1"
is_byhour=0
after=""

set_size() {
    [ "$1" -lt 1 ] && onerror 3 "size must be > 1"
    size=$1
}
set_byhour() {
    byhour="${byhour} | tr -s ' ' | cut -d ' ' -f 3"
```

```
    after=" | sort -k 2"
    is_byhour=1
    set_size 24
}
doit() {
    end=1
    cmd="cat $@ | ${byhour} | sort | uniq -c | sort -nr | head -n ${size} ${after}"
    case ${is_byhour} in
        0)
            printf "%-7.7s %-6.6s %-2.2s\n" "occurences" "date" "hour"
            printf "%-17.17s\n" "-----------------"
            ;;
        1)
            printf "%-7.7s %-2.2s\n" "occurences" "hour"
            printf "%-10.10s\n" "----------"
            ;;
    esac
    eval "${cmd}"
}

[ $# -eq 0 ] && dohelp
end=0

while [ $end -eq 0 ]
do
    case $1 in
        -h | --help)
            dohelp
            ;;
        -s | --size)
            shift
            [ $# -eq 0 ] && onerror 2 "$1 needs a parameter"
            set_size "$1"
            shift
```

```
            ;;
        -b | --byhour)
            shift
            set_byhour
            ;;
        *)
            doit "$@"
            ;;
    esac
done
```

code/stat2.sh

Et maintenant avec le `getopts`[4] :

```
#!/bin/sh

scriptname=$(basename $0)

dohelp() {
    cat << DOHELP
${scriptname} [-h] : this text
${scriptname} [options] file file\ldots : stats
options:
    -s N : number of most important hours, default 5
    -b : for each hours
DOHELP
    exit 0
}
onerror() {
    local exit_code=$1
    shift
    local error_msg="$@"

    echo "ERROR: $error_msg" 1>&2
    exit "$exit_code"
}
```

4. soyez prudents avec les (très) anciennes versions de *Red Hat*

```
size=5
byhour="cut -d ':' -f 1"
is_byhour=0
after=""

set_size() {
    [ "$1" -lt 1 ] && onerror 3 "size must be > 1"
    size=$1
}
set_byhour() {
    byhour="${byhour} | tr -s ' ' | cut -d ' ' -f 3"
    after=" | sort -k 2"
    is_byhour=1
    set_size 24
}
doit() {
    cmd="cat $@ | ${byhour} | sort | uniq -c | sort -nr | head
   -n ${size} ${after}"
    case ${is_byhour} in
        0)
            printf "%-7.7s %-6.6s %-2.2s\n" "occurences" "date"
    "hour"
            printf "%-17.17s\n" "
   ---------------------------------"
            ;;
        1)
            printf "%-7.7s %-2.2s\n" "occurences" "hour"
            printf "%-10.10s\n" "
   ---------------------------------"
            ;;
    esac
    eval "${cmd}"
}

# [ $# -eq 0 ] && dohelp

[ $# -eq 0 ] && echo "you need arguments" && dohelp
```

```
58 while getopts "s:bh" opt
   do
60     case $opt in
           h)
62             dohelp
               ;;
64         s)
               set_size "$OPTARG"
66             ;;
           b)
68             set_byhour
               ;;
70         :)
               onerror 2 "$OPTARG needs a parameter"
72             ;;
           \?)
74             onerror 7 "option $OPTARG is unknown"
               ;;
76     esac
   done
78
   shift $((OPTIND-1))
80 doit "$@"
```

code/stat3.sh

En fait `getopts` ne sait traiter que les *options courtes* et classiques d'**Unix**. Les *options longues* à la mode **Linux** ne sont pas supportées. L'avantage de `getopts` est son mode de fonctionnement assez simple. Son inconvénient principal est d'être très spécifique à `bash` même si **POSIX** le soutient, ce qui fait qu'il n'est pas forcément disponible partout.

Pour avoir les *options longues*, il faut utiliser l'outil **GNU** `getopt` (sans le `s` de fin)[5]. Je reste donc sur ma méthode qui n'est finalement ni meilleure ni pire.

5. voir cette discussion sur StackOverflow

6.2.2 Avec bash On peut profiter des avantages de `bash` (boucles, tableaux, ...) comme dans ce script (qui ne fonctionne **pas** avec `sh`) :

```
#!/usr/bin/env bash

scriptname="$(basename $0)"
# set -e

dohelp() {
    cat << DOHELP
${scriptname} [-h|--help] : this text
${scriptname} [options] file file... : stats
options:
    -s|--size N : number of most important hours, default 5
    -b|--byhour : for each hours
DOHELP
    exit 0
}

size=5
byhour="cut -d ':' -f 1"
is_byhour=0
after=""
hours=

set_size() {
    [ "$1" -lt 1 ] && onerror 3 "size must be > 1"
    size=$1
}
set_byhour() {
    local i
    byhour="${byhour} | tr -s ' ' | cut -d ' ' -f 3"
    after=" | sort -k 2"
    is_byhour=1
    set_size 24
    for ((i=0; i<24; i++))
    do
        hours[i]=0
```

```
    done
}

doit() {
    end=1
    cmd="cat $@ | LC_ALL=C tr -cd '\t\n[:print:]' | ${byhour} |
    sort | uniq -c | sort -nr | head -n ${size} ${after}"
    case ${is_byhour} in
        0)
            printf "%-7.7s %-6.6s %-2.2s\n" "occurences" "date"
    "hour"
            printf "%-17.17s\n" "
   ----------------------------------"
            eval "$cmd"
            ;;
        1)
            printf "%-7.7s %-2.2s\n" "occurences" "hour"
            printf "%-10.10s\n" "
   ----------------------------------"
            while read count index reste; do
                local h=${index#0}
                hours[h]=${count}
            done <<<"$(eval ${cmd})"
            for ((i=0; i<24; i++))
            do
                printf "%7d %02d\n" ${hours[i]} $i
            done
            ;;
    esac
}

[ $# -eq 0 ] && dohelp
end=0

while [ $end -eq 0 ]
do
    case $1 in
        -h|--help)
```

```
            dohelp
            ;;
        -s|--size)
            shift
            [ $# -eq 0 ] && onerror 2 "$1 needs a parameter"
            set_size "$1"
            shift
            ;;
        -b|--byhour)
            shift
            set_byhour
            ;;
        *)
            doit "$@"
            ;;
    esac
done
```

code/stat4.sh

Conrairement aux précédents, si une tranche horaire n'est pas représentée dans les fichiers logs passés en paramètres, elle sera tout de même affichée avec la valeur 0.

COMMANDES UTILES

7 les noms de fichiers

Les commandes les plus utiles sont `dirname` et `basename`. La première renvoie le repertoire du nom fichier et la seconde renvoie simplement le nom de base comme ici :

```
$ which firefox
/usr/local/bin/firefox
$ basename $(which firefox)
firefox
$ dirname $(which firefox)
/usr/local/bin
$
```

On en profite pour présenter l'indispensable `which` qui donne le nom complet d'une application se trouvant dans le `PATH`.

8 cherche et remplace

Les outils de base sont `egrep`, `sed` et `tr`[6]. Ces trois outils utilisent les expressions régulières. Ces expressions régulières ressemblent fortement à ce que l'on trouve dans Perl, Python, Java et les autres. La différence fondamentale à ne pas oublier : les expressions régulières des outils **GNU** sont rapides, efficaces, les autres… beaucoup moins[7].

Les expressions régulières méritent une formation complète car elles ne sont pas vraiment intuitives. De plus, les variations qui existent entre **POSIX**, **GNU**, **BSD**, les shells qui en rajoutent parfois, sans compter que certains outils, certaines distributions n'ont pas leurs outils vraiment à jour (*cf.* **RedHat**).

8.1 cherche Pour la recherche, nous avons `grep` et `egrep`. En fait, la plupart du temps, `egrep` équivaut à `grep -E` permettant l'utilisation des expressions régulières dites étendues ou **POSIX** dont la documentation se trouve dans `man 7 regex` sous Debian.

6. L'outil `awk`, est, à mon humble avis, à reléguer dans les musées.
7. L'introduction du *backtracking* peut détruire complètement les performances

Index