Programmation Bash pour la Cybersecurite

Automatisation, analyse de logs et reponse a incident

Introduction

Le Bash dans le quotidien d'un analyste securite

Le shell Bash (Bourne Again SHell) est l'interface en ligne de commande par defaut sur la plupart des systemes Linux et Unix.

Trois cas d'usage principaux en cybersecurite :

1. **Automatisation** : verification d'integrite, collecte de metriques, rotation des logs
2. **Analyse de logs** : extraction d'indicateurs de compromission (IoC)
3. **Reponse a incident** : isolation, collecte forensique, blocage d'IP

Mode interactif versus scripting

Mode interactif

L'operateur tape une commande, observe le resultat, puis decide de la suite. Adapte a l'exploration et au diagnostic ponctuel.

Mode scripting

Sequence de commandes ecrites dans un fichier texte execute comme un programme. Adapte aux operations repetitives et aux procedures standardisees.

En pratique : on explore en interactif, puis on formalise en script.

Environnement de travail

Distributions courantes en cybersecurite :

- **Kali Linux** : tests d'intrusion et forensique
- **Ubuntu Server** : environnements de production
- **CentOS/Rocky Linux** : serveurs d'entreprise

Prerequis :

- Acces terminal Linux
- Privileges administrateur (root) pour les fichiers systeme
- Commandes sensibles prefixees par `sudo`

Bonne pratique

> Travaillez toujours sur une machine virtuelle ou un environnement de test lorsque vous experimentez avec des commandes systeme. Ne jamais tester directement sur un systeme de production.

Commandes essentielles pour l'analyse

Arborescence Linux orientee securite

Repertoire	Description
`/var/log`	Journaux systeme (authentification, applications, activite)
`/etc/passwd`	Liste des comptes utilisateurs (lisible par tous)
`/etc/shadow`	Hachages des mots de passe (root uniquement)
`/tmp`	Repertoire temporaire (souvent utilise par les attaquants)
`/var/www`	Racine des serveurs web (cible frequente)
`/home`	Repertoires personnels des utilisateurs

La commande ls -la

```
ls -la /var/log
```

Sortie typique :

```
drwxr-xr-x  2 root root 4096 Jan 15 10:30 apt
-rw-r-----  1 root adm  8234 Jan 15 11:45 auth.log
```

Decodage de la premiere colonne :

- `d` = repertoire, `-` = fichier, `l` = lien symbolique
- `rwx` = permissions proprietaire (read, write, execute)
- `r-x` = permissions groupe
- `r-x` = permissions autres

Explication detaillee des permissions

```
-rw-r----- 1 root adm 8234 Jan 15 11:45 auth.log
```

Element	Signification
`-`	Type : fichier regulier
`rw-`	Proprietaire (root) : lecture et ecriture
`r--`	Groupe (adm) : lecture seule
`---`	Autres : aucun acces
`root`	Utilisateur proprietaire
`adm`	Groupe proprietaire
`8234`	Taille en octets

Commandes de lecture de fichiers

Commande	Description
`cat fichier`	Affiche l'integralite du fichier
`head -n 20 fichier`	Affiche les 20 premieres lignes
`tail -n 50 fichier`	Affiche les 50 dernieres lignes
`tail -f fichier`	Suit le fichier en temps reel
`less fichier`	Navigation interactive (recherche avec `/motif`)

Exemple : surveillance en temps reel

```
# Surveillance du log d'authentification
sudo tail -f /var/log/auth.log
```

Explication du code :

- `sudo` : execute la commande avec les privileges root (necessaire car auth.log n'est pas lisible par tous)
- `tail` : affiche la fin d'un fichier
- `-f` (follow) : reste actif et affiche les nouvelles lignes au fur et a mesure
- `/var/log/auth.log` : fichier journal des authentifications

Pour interrompre : Ctrl+C

Surveillance de plusieurs fichiers

```
# Surveillance simultanee de plusieurs logs
sudo tail -f /var/log/auth.log /var/log/syslog
```

Explication :

- `tail -f` accepte plusieurs fichiers en arguments
- Les nouvelles lignes sont prefixees par le nom du fichier source
- Utile pour correler des evenements entre differents journaux

La commande grep

grep (Global Regular Expression Print) recherche des motifs dans des fichiers ou des flux de donnees.

```
grep [options] 'motif' fichier
```

Options essentielles :

Option	Description
`-i`	Ignore la casse
`-v`	Inverse la selection (lignes qui NE correspondent PAS)
`-c`	Compte le nombre de lignes correspondantes
`-n`	Affiche les numeros de ligne
`-r`	Recherche recursive dans les sous-repertoires

Options avancees de grep

Option	Description
`-E`	Active les expressions regulieres etendues
`-o`	Affiche uniquement la partie correspondante
`-A n`	Affiche n lignes apres chaque correspondance
`-B n`	Affiche n lignes avant chaque correspondance
`-C n`	Affiche n lignes avant ET apres

Exemples pratiques avec grep

```
# Rechercher les tentatives de connexion echouees
grep 'Failed password' /var/log/auth.log
```

Explication :

- `grep` recherche le motif exact 'Failed password'
- Chaque ligne contenant ce motif est affichee
- Les guillemets simples protegent le motif des interpretations du shell

Compter les occurrences

```
# Compter le nombre d'echecs de connexion
grep -c 'Failed password' /var/log/auth.log
```

Explication :

- L'option `-c` (count) ne renvoie que le nombre de lignes correspondantes
- Resultat : un entier (ex: 47)
- Plus efficace que `grep ... | wc -l` pour un simple comptage

Combiner plusieurs filtres

```
# Rechercher les connexions root reussies
grep 'Accepted' /var/log/auth.log | grep 'root'
```

Explication :

- Premier `grep` : selectionne les lignes contenant 'Accepted'
- Le pipe `|` envoie le resultat au second `grep`
- Second `grep` : filtre pour ne garder que les lignes contenant 'root'
- Resultat : connexions reussies de l'utilisateur root

Afficher le contexte

```
# Afficher 2 lignes avant et apres chaque erreur
grep -B2 -A2 'error' /var/log/syslog
```

Explication :

- `-B2` (Before) : affiche 2 lignes avant chaque correspondance
- `-A2` (After) : affiche 2 lignes apres chaque correspondance
- Utile pour comprendre le contexte d'un evenement
- Alternative : `-C2` pour 2 lignes avant ET apres

Expressions regulieres simples

Metacaractere	Signification
`.`	N'importe quel caractere unique
`*`	Zero ou plusieurs occurrences du caractere precedent
`+`	Une ou plusieurs occurrences (avec `-E`)
`^`	Debut de ligne
`$`	Fin de ligne
`[abc]`	Un caractere parmi a, b ou c
`[0-9]`	Un chiffre

Extraction d'adresses IP

```
# Extraire les adresses IP du log
grep -oE '[0-9]+\.[0-9]+\.[0-9]+\.[0-9]+' /var/log/auth.log
```

Explication :

- `-o` : affiche uniquement la partie correspondante, pas la ligne entiere
- `-E` : active les expressions regulieres etendues (pour utiliser `+`)
- `[0-9]+` : un ou plusieurs chiffres
- `\.` : un point litteral (le backslash echappe le point)
- Le motif complet : 4 groupes de chiffres separes par des points

Attention

Ce motif simplifie peut correspondre a des chaines invalides (comme 999.999.999.999). Pour une validation stricte des adresses IP, utilisez un motif plus precis ou validez les resultats avec un traitement supplementaire.

La commande find

find localise des fichiers selon de nombreux criteres.

```
find chemin [criteres] [actions]
```

Criteres principaux :

Critere	Description
`-name 'motif'`	Recherche par nom (sensible a la casse)
`-type f`	Fichiers reguliers uniquement
`-type d`	Repertoires uniquement
`-mtime -1`	Modifie dans les dernieres 24 heures
`-user nom`	Appartenant a l'utilisateur specifie

Criteres de find pour la securite

Critere	Description
`-mmin -60`	Modifie dans les 60 dernieres minutes
`-perm 777`	Permissions exactement 777
`-perm -002`	Accessible en ecriture par tous
`-perm -4000`	Fichier SUID (Set User ID)
`-size +10M`	Taille superieure a 10 Mo
`-nouser`	Fichier sans proprietaire valide

Exemples de find pour l'analyse

```
# Fichiers modifies dans les dernieres 24 heures
find /var -type f -mtime -1
```

Explication :

- `/var` : repertoire de depart de la recherche
- `-type f` : uniquement les fichiers (pas les repertoires)
- `-mtime -1` : modification time inferieur a 1 jour
- Le signe `-` signifie "moins de" (sans signe = exactement, `+` = plus de)

Detection de fichiers suspects

```
# Fichiers executables dans /tmp (potentiellement suspect)
find /tmp -type f -perm -111
```

Explication :

- `/tmp` : repertoire temporaire accessible a tous
- `-perm -111` : fichiers avec le bit d'execution pour tous
- Le `-` devant 111 signifie "au moins ces permissions"
- Un executable dans /tmp peut indiquer une compromission

Detection de fichiers SUID

```
# Fichiers SUID (risque d'elevation de privileges)
find / -type f -perm -4000 2>/dev/null
```

Explication :

- `/` : recherche depuis la racine (tout le systeme)
- `-perm -4000` : bit SUID active (execute avec les droits du proprietaire)
- `2>/dev/null` : redirige les erreurs vers /dev/null
  - `2` = descripteur de fichier stderr (erreurs)
  - `>` = redirection
  - `/dev/null` = "poubelle" qui ignore tout ce qu'on lui envoie

Fichiers orphelins

```
# Fichiers sans proprietaire valide
find / -nouser -o -nogroup 2>/dev/null
```

Explication :

- `-nouser` : fichiers dont l'UID ne correspond a aucun utilisateur
- `-o` : operateur OU logique
- `-nogroup` : fichiers dont le GID ne correspond a aucun groupe
- Ces fichiers peuvent indiquer un compte supprime ou une anomalie

Enchainement avec les pipes

Le pipe `|` connecte la sortie d'une commande a l'entree de la suivante.

```
# Pipeline : lire -> filtrer -> compter
cat /var/log/auth.log | grep 'Failed' | wc -l
```

Explication :

1. `cat` affiche le contenu du fichier
2. `|` envoie ce contenu a `grep`
3. `grep 'Failed'` filtre les lignes contenant "Failed"
4. `|` envoie les lignes filtrees a `wc`
5. `wc -l` compte le nombre de lignes

Pipeline optimise

```
# Version plus efficace
grep -c 'Failed' /var/log/auth.log
```

Explication :

- `grep` peut lire directement un fichier (pas besoin de `cat`)
- L'option `-c` compte directement les correspondances
- Moins de processus = meilleure performance
- Resultat identique, execution plus rapide

Pipeline complet pour l'analyse

```
# Extraire les IP, trier, compter les occurrences
grep 'Failed password' /var/log/auth.log \
  | grep -oE '[0-9]+\.[0-9]+\.[0-9]+\.[0-9]+' \
  | sort \
  | uniq -c \
  | sort -rn
```

Explication detaillee sur les slides suivantes...

Analyse du pipeline (1/2)

```
grep 'Failed password' /var/log/auth.log
```

Selectionne les lignes d'echec de connexion.

```
| grep -oE '[0-9]+\.[0-9]+\.[0-9]+\.[0-9]+'
```

Extrait uniquement les adresses IP de ces lignes.

```
| sort
```

Trie les IP par ordre alphabetique (necessaire pour uniq).

Analyse du pipeline (2/2)

```
| uniq -c
```

Supprime les doublons consecutifs et compte les occurrences.
Resultat : `15 192.168.1.100`

```
| sort -rn
```

- `-r` : ordre inverse (decroissant)
- `-n` : tri numerique (pas alphabetique)

Resultat final : liste des IP triees par nombre de tentatives.

Exercice 1

Sur le fichier auth.log fourni :

1. Identifier le nombre total de tentatives de connexion echouees
2. Lister les adresses IP sources uniques
3. Trouver l'IP avec le plus de tentatives

Premier script d'audit

Structure d'un script Bash

```
#!/bin/bash
# Description : Mon premier script
# Auteur : Analyste SOC
# Date : 2025-01-16

# Le code commence ici
echo "Hello, Security!"
```

Explication :

- `#!/bin/bash` : shebang, indique l'interpreteur a utiliser
- Les lignes commencant par `#` sont des commentaires
- `echo` affiche du texte sur la sortie standard

Creer et executer un script

```
# Creer le fichier
nano mon_script.sh

# Rendre executable
chmod +x mon_script.sh

# Executer
./mon_script.sh
```

Explication :

- `chmod +x` : ajoute la permission d'execution
- `./` : execute le fichier dans le repertoire courant
- Sans `./`, le shell cherche dans les repertoires du PATH

Variables

```bash
#!/bin/bash

# Declaration de variable (sans espace autour du =)
nom="auth.log"
chemin="/var/log"

# Utilisation avec $
echo "Fichier : $chemin/$nom"

# Accolades pour delimiter le nom
echo "Fichier : ${chemin}/${nom}"
```

Explication :

- Pas d'espace autour de `=` (sinon erreur de syntaxe)
- `$variable` ou `${variable}` pour acceder a la valeur
- Les accolades evitent les ambiguites : `${var}suite`

Variables speciales

Variable	Description
`$0`	Nom du script
`$1, $2, ...`	Arguments positionnels
`$#`	Nombre d'arguments
`$@`	Tous les arguments (separes)
`$?`	Code de retour de la derniere commande
`$$`	PID du processus courant

Arguments positionnels

```
#!/bin/bash
# script : analyse.sh
# Usage : ./analyse.sh fichier.log 10

fichier=$1     # Premier argument
seuil=$2       # Deuxieme argument

echo "Analyse de $fichier avec seuil $seuil"
```

Execution :

```
./analyse.sh auth.log 10
# Affiche : Analyse de auth.log avec seuil 10
```

Codes de retour

```bash
#!/bin/bash

# Chaque commande retourne un code
grep "pattern" fichier.log
code=$?

echo "Code de retour : $code"
# 0 = succes (pattern trouve)
# 1 = echec (pattern non trouve)
# 2 = erreur (fichier inexistant)
```

Convention :

- `0` = succes
- Autre valeur = echec ou erreur

Tester le code de retour

```bash
#!/bin/bash

if grep -q "Failed" /var/log/auth.log; then
    echo "Des echecs de connexion ont ete detectes"
else
    echo "Aucun echec detecte"
fi
```

Explication :

- `-q` (quiet) : grep ne produit pas de sortie, seulement un code retour
- `if commande; then` : execute le bloc si le code retour est 0
- `else` : execute si le code retour est different de 0
- `fi` : ferme le bloc if (if a l'envers)

La commande test

```
# Syntaxe avec test
if test -f "/etc/shadow"; then
    echo "Le fichier existe"
fi

# Syntaxe equivalente avec crochets
if [ -f "/etc/shadow" ]; then
    echo "Le fichier existe"
fi
```

Important : Espaces obligatoires apres `[` et avant `]`

Tests sur les fichiers

Test	Description
`-f fichier`	Vrai si le fichier existe et est un fichier regulier
`-d repertoire`	Vrai si le repertoire existe
`-r fichier`	Vrai si le fichier est lisible
`-w fichier`	Vrai si le fichier est modifiable
`-x fichier`	Vrai si le fichier est executable
`-s fichier`	Vrai si le fichier n'est pas vide
`-e fichier`	Vrai si le fichier existe (tout type)

Exemple de test de fichier

```
#!/bin/bash

fichier="/etc/shadow"

if [ -f "$fichier" ]; then
    echo "$fichier existe"

    if [ -r "$fichier" ]; then
        echo "  -> Lisible (attention : probleme de securite potentiel)"
    else
        echo "  -> Non lisible (normal pour un utilisateur standard)"
    fi
else
    echo "$fichier n'existe pas"
fi
```

Tests sur les chaines

Test	Description
`-z "$var"`	Vrai si la chaine est vide
`-n "$var"`	Vrai si la chaine n'est pas vide
`"$a" = "$b"`	Vrai si les chaines sont egales
`"$a" != "$b"`	Vrai si les chaines sont differentes

Important : Toujours mettre les variables entre guillemets pour eviter les erreurs avec les valeurs vides ou contenant des espaces.

Tests numeriques

Test	Description
`$a -eq $b`	Egal (equal)
`$a -ne $b`	Different (not equal)
`$a -lt $b`	Inferieur (less than)
`$a -le $b`	Inferieur ou egal (less or equal)
`$a -gt $b`	Superieur (greater than)
`$a -ge $b`	Superieur ou egal (greater or equal)

Exemple de test numerique

```bash
#!/bin/bash

seuil=10
tentatives=$(grep -c 'Failed password' /var/log/auth.log)

if [ "$tentatives" -gt "$seuil" ]; then
    echo "ALERTE : $tentatives tentatives (seuil : $seuil)"
else
    echo "OK : $tentatives tentatives"
fi
```

Explication :

- `$(commande)` : substitution de commande (execute et capture la sortie)
- `-gt` : greater than (superieur a)

Operateurs logiques

```
# ET logique
if [ -f "$fichier" ] && [ -r "$fichier" ]; then
    echo "Fichier existant et lisible"
fi

# OU logique
if [ -z "$1" ] || [ -z "$2" ]; then
    echo "Usage : $0 fichier seuil"
    exit 1
fi
```

Explication :

- `&&` : ET logique (les deux conditions doivent etre vraies)
- `||` : OU logique (au moins une condition doit etre vraie)
- `exit 1` : termine le script avec le code d'erreur 1

La boucle for

```bash
#!/bin/bash

# Iteration sur une liste explicite
for fichier in auth.log syslog kern.log; do
    echo "Analyse de $fichier"
done

# Iteration sur le resultat d'une commande
for ip in $(cat ip_suspectes.txt); do
    echo "Verification de $ip"
done
```

Explication :

- `for variable in liste; do ... done` : structure de boucle
- La variable prend successivement chaque valeur de la liste
- `$(cat fichier)` : substitution, chaque ligne devient un element

Boucle for avec glob

```
#!/bin/bash

# Iteration sur les fichiers .log
for fichier in /var/log/*.log; do
    if [ -f "$fichier" ]; then
        lignes=$(wc -l < "$fichier")
        echo "$fichier : $lignes lignes"
    fi
done
```

Explication :

- `*.log` : glob pattern, correspond a tous les fichiers .log
- `wc -l < "$fichier"` : compte les lignes (l'entree vient du fichier)
- L'utilisation de `<` evite d'afficher le nom du fichier dans la sortie

Script d'audit : structure

```
#!/bin/bash
# Script d'audit de securite basique
# Usage : ./audit.sh

echo "=== AUDIT DE SECURITE ==="
echo "Date : $(date)"
echo ""

# Section 1 : Fichiers sensibles
echo "--- Verification des fichiers sensibles ---"

# Section 2 : Utilisateurs
echo "--- Analyse des utilisateurs ---"

# Section 3 : Reseau
echo "--- Ports en ecoute ---"
```

Verification des fichiers sensibles

```
# Verifier si /etc/shadow est lisible par tous
if [ -r /etc/shadow ]; then
    echo "[ALERTE] /etc/shadow est lisible !"
    ls -l /etc/shadow
else
    echo "[OK] /etc/shadow n'est pas lisible"
fi

# Verifier les permissions de /etc/passwd
perms=$(stat -c %a /etc/passwd)
if [ "$perms" != "644" ]; then
    echo "[ALERTE] /etc/passwd permissions anormales : $perms"
else
    echo "[OK] /etc/passwd permissions correctes"
fi
```

Explication : stat -c %a

```
stat -c %a /etc/passwd
```

Explication :

- `stat` : affiche les informations detaillees d'un fichier
- `-c` : format personnalise
- `%a` : permissions en octal (ex: 644)

Autres formats utiles :

Format	Description
`%U`	Nom du proprietaire
`%G`	Nom du groupe
`%s`	Taille en octets
`%y`	Date de modification

Liste des utilisateurs avec shell actif

```
# Utilisateurs avec /bin/bash comme shell
echo "Utilisateurs avec shell bash :"
grep '/bin/bash$' /etc/passwd | cut -d: -f1

# Explication du pipeline :
# 1. grep '/bin/bash$' : lignes finissant par /bin/bash
# 2. cut -d: -f1 : extrait le premier champ (separateur :)
```

Structure de /etc/passwd :

```
nom:x:uid:gid:description:home:shell
root:x:0:0:root:/root:/bin/bash
```

La commande cut

```
# Syntaxe
cut -d'delimiteur' -f champs fichier
```

Options :

Option	Description
`-d':'`	Delimiteur (ici le deux-points)
`-f1`	Premier champ
`-f1,3`	Champs 1 et 3
`-f1-3`	Champs 1 a 3

Exemple :

```
echo "root:x:0:0:root:/root:/bin/bash" | cut -d: -f1,7
# Resultat : root:/bin/bash
```

Ports en ecoute avec ss

```
# Afficher les ports TCP en ecoute
ss -tuln
```

Options :

Option	Description
`-t`	Connexions TCP
`-u`	Connexions UDP
`-l`	Sockets en ecoute (listen)
`-n`	Affichage numerique (pas de resolution DNS)

Exemple de sortie ss

```
Netid  State   Recv-Q  Send-Q  Local Address:Port   Peer Address:Port
tcp    LISTEN  0       128     0.0.0.0:22           0.0.0.0:*
tcp    LISTEN  0       511     0.0.0.0:80           0.0.0.0:*
tcp    LISTEN  0       128     127.0.0.1:3306       0.0.0.0:*
```

Lecture :

- Port 22 : SSH accessible depuis toutes les interfaces
- Port 80 : HTTP accessible depuis toutes les interfaces
- Port 3306 : MySQL accessible uniquement en local (127.0.0.1)

Extraire les ports en ecoute

```
# Lister uniquement les numeros de port TCP en ecoute
ss -tln | awk 'NR>1 {print $4}' | cut -d: -f2 | sort -n | uniq
```

Explication :

1. `ss -tln` : liste les sockets TCP en ecoute
2. `awk 'NR>1 {print $4}'` : saute l'entete, affiche la 4e colonne
3. `cut -d: -f2` : extrait le port (apres le :)
4. `sort -n` : tri numerique
5. `uniq` : supprime les doublons

Exercice 2

Ecrire un script `audit_basique.sh` qui :

1. Verifie si `/etc/shadow` est lisible par tous
2. Liste les utilisateurs ayant `/bin/bash` comme shell
3. Affiche les ports TCP en ecoute

Le script doit afficher des messages clairs avec [OK] ou [ALERTE].

Analyse de logs et detection

Introduction a awk

awk est un langage de traitement de texte oriente colonnes.

```
awk 'condition { action }' fichier
```

Structure de base :

- Chaque ligne est decoupee en champs (`$1`, `$2`, ...)
- `$0` represente la ligne entiere
- Par defaut, le separateur est l'espace ou la tabulation

Exemples simples avec awk

```
# Afficher la premiere colonne
awk '{print $1}' fichier.txt

# Afficher les colonnes 1 et 3
awk '{print $1, $3}' fichier.txt

# Avec un separateur personnalise (deux-points)
awk -F: '{print $1, $7}' /etc/passwd
```

Explication :

- `{print $1}` : action appliquee a chaque ligne
- `-F:` : definit le separateur de champs (field separator)

Conditions avec awk

```
# Lignes ou le 3e champ est superieur a 100
awk '$3 > 100 {print $0}' fichier.txt

# Lignes contenant "Failed"
awk '/Failed/ {print $0}' auth.log

# Combinaison
awk '/Failed/ && $9 > 5 {print $1, $9}' fichier.txt
```

Explication :

- `$3 > 100` : condition numerique sur le 3e champ
- `/Failed/` : condition de correspondance (regex)
- `&&` : ET logique

Variables speciales awk

Variable	Description
`NR`	Numero de ligne (Number of Record)
`NF`	Nombre de champs (Number of Fields)
`FS`	Separateur de champs (Field Separator)
`$NF`	Dernier champ
`$(NF-1)`	Avant-dernier champ

Exemples avec NR et NF

```
# Afficher le numero de ligne et le contenu
awk '{print NR, $0}' fichier.txt

# Sauter les 5 premieres lignes
awk 'NR > 5 {print $0}' fichier.txt

# Afficher le dernier champ de chaque ligne
awk '{print $NF}' fichier.txt

# Lignes avec plus de 5 champs
awk 'NF > 5 {print $0}' fichier.txt
```

Comptage et agregation avec awk

```
# Compter les occurrences par IP
awk '{count[$1]++} END {for (ip in count) print ip, count[ip]}' fichier.txt
```

Explication detaillee :

- `{count[$1]++}` : pour chaque ligne, incremente le compteur pour la valeur de $1
- `count[$1]` : tableau associatif (cle = valeur du champ 1)
- `END {...}` : bloc execute apres la lecture de toutes les lignes
- `for (ip in count)` : parcourt toutes les cles du tableau

Exemple concret : analyse de logs

```
# Extraire et compter les IP des echecs de connexion
grep 'Failed password' /var/log/auth.log \
  | awk '{print $(NF-3)}' \
  | sort | uniq -c | sort -rn
```

Format typique d'une ligne auth.log :

```
Jan 15 10:30:45 server sshd[1234]: Failed password for user from 192.168.1.100 port 52341 ssh2
```

L'IP est a la position `$(NF-3)` (4e champ en partant de la fin).

sort et uniq

```
# Trier par ordre alphabetique
sort fichier.txt

# Trier par ordre numerique
sort -n fichier.txt

# Trier en ordre inverse
sort -r fichier.txt

# Trier par la 2e colonne
sort -k2 fichier.txt
```

uniq et ses options

```
# Supprimer les doublons (fichier doit etre trie)
sort fichier.txt | uniq

# Compter les occurrences
sort fichier.txt | uniq -c

# Afficher uniquement les lignes uniques
sort fichier.txt | uniq -u

# Afficher uniquement les lignes dupliquees
sort fichier.txt | uniq -d
```

Important : `uniq` ne detecte que les doublons consecutifs, d'ou la necessite de trier d'abord.

Redirection vers fichiers

```
# Rediriger la sortie vers un fichier (ecrase)
commande > fichier.txt

# Rediriger en ajout (append)
commande >> fichier.txt

# Rediriger les erreurs
commande 2> erreurs.txt

# Rediriger sortie et erreurs
commande > sortie.txt 2>&1

# Ou avec la syntaxe moderne
commande &> tout.txt
```

Exemple : generation de rapport

```
#!/bin/bash

rapport="rapport_$(date +%Y%m%d_%H%M%S).txt"

{
    echo "=== RAPPORT D'ANALYSE ==="
    echo "Date : $(date)"
    echo ""
    echo "--- Statistiques ---"
    echo "Nombre d'echecs : $(grep -c 'Failed' auth.log)"
    echo ""
    echo "--- Top 10 IP ---"
    grep 'Failed' auth.log | awk '{print $(NF-3)}' \
      | sort | uniq -c | sort -rn | head -10
} > "$rapport"

echo "Rapport genere : $rapport"
```

Explication : bloc de redirection

```
{
    commande1
    commande2
    commande3
} > fichier.txt
```

Explication :

- Les accolades `{ }` groupent les commandes
- La redirection `>` s'applique a l'ensemble du bloc
- Toutes les sorties sont combinees dans le meme fichier
- Plus propre que de rediriger chaque commande individuellement

Cas pratique : detection de bruteforce

Objectif : Identifier les adresses IP effectuant plus de N tentatives de connexion echouees.

Etapes :

1. Extraire les lignes d'echec de connexion
2. Extraire les adresses IP
3. Compter les occurrences par IP
4. Filtrer celles depassant le seuil
5. Generer un rapport

Script de detection (1/3)

```
#!/bin/bash
# detect_bruteforce.sh - Detection de tentatives de bruteforce
# Usage : ./detect_bruteforce.sh [fichier_log] [seuil]

# Parametres avec valeurs par defaut
LOG_FILE="${1:-/var/log/auth.log}"
SEUIL="${2:-10}"

# Verification des parametres
if [ ! -f "$LOG_FILE" ]; then
    echo "Erreur : fichier $LOG_FILE introuvable"
    exit 1
fi

echo "Analyse de $LOG_FILE (seuil : $SEUIL tentatives)"
```

Explication : valeurs par defaut

```
LOG_FILE="${1:-/var/log/auth.log}"
```

Syntaxe `${variable:-valeur_defaut}` :

- Si `$1` est defini et non vide : utilise `$1`
- Sinon : utilise `/var/log/auth.log`

Autres variantes :

Syntaxe	Description
`${var:-default}`	Valeur par defaut si non definie ou vide
`${var:=default}`	Assigne la valeur par defaut
`${var:?message}`	Erreur si non definie

Script de detection (2/3)

```
# Fichier temporaire pour les resultats
TEMP_FILE=$(mktemp)

# Extraction et comptage des IP
grep 'Failed password' "$LOG_FILE" \
  | grep -oE '[0-9]+\.[0-9]+\.[0-9]+\.[0-9]+' \
  | sort \
  | uniq -c \
  | sort -rn > "$TEMP_FILE"

# Comptage des IP suspectes
nb_suspectes=$(awk -v seuil="$SEUIL" '$1 >= seuil {count++} END {print count+0}' "$TEMP_FILE")

echo "IP suspectes detectees : $nb_suspectes"
```

Explication : mktemp et variables awk

```
TEMP_FILE=$(mktemp)
```

`mktemp` cree un fichier temporaire unique dans /tmp et retourne son chemin.

```
awk -v seuil="$SEUIL" '$1 >= seuil ...'
```

- `-v seuil="$SEUIL"` : passe la variable shell a awk
- `$1 >= seuil` : compare le premier champ a la valeur seuil
- `count+0` : force l'affichage de 0 si count n'est pas defini

Script de detection (3/3)

```
# Generation du rapport
RAPPORT="rapport_incident_$(date +%Y%m%d_%H%M%S).txt"

{
    echo "============================================="
    echo "     RAPPORT D'INCIDENT - BRUTEFORCE SSH"
    echo "============================================="
    echo ""
    echo "Date d'analyse    : $(date)"
    echo "Fichier analyse   : $LOG_FILE"
    echo "Seuil de detection: $SEUIL tentatives"
    echo ""
    echo "--- IP SUSPECTES ---"
    awk -v seuil="$SEUIL" '$1 >= seuil {printf "%-20s %d tentatives\n", $2, $1}' "$TEMP_FILE"
    echo ""
    echo "=== FIN DU RAPPORT ==="
} > "$RAPPORT"

# Nettoyage
rm -f "$TEMP_FILE"

echo "Rapport genere : $RAPPORT"
```

Explication : printf dans awk

```
printf "%-20s %d tentatives\n", $2, $1
```

Format printf :

Code	Description
`%s`	Chaine de caracteres
`%d`	Entier decimal
`%-20s`	Chaine alignee a gauche sur 20 caracteres
`\n`	Saut de ligne

Resultat :

```
192.168.1.100        47 tentatives
10.0.0.55            23 tentatives
```

Enrichissement : horodatage

```
# Extraire la premiere et derniere tentative pour une IP
ip="192.168.1.100"

grep "Failed password" auth.log | grep "$ip" | head -1
# -> Premier evenement

grep "Failed password" auth.log | grep "$ip" | tail -1
# -> Dernier evenement
```

Script enrichi : periode d'attaque

```
# Pour chaque IP suspecte, afficher la periode
while read count ip; do
    if [ "$count" -ge "$SEUIL" ]; then
        premier=$(grep "Failed.*$ip" "$LOG_FILE" | head -1 | awk '{print $1, $2, $3}')
        dernier=$(grep "Failed.*$ip" "$LOG_FILE" | tail -1 | awk '{print $1, $2, $3}')
        echo "IP: $ip"
        echo "   Tentatives: $count"
        echo "   Debut: $premier"
        echo "   Fin: $dernier"
        echo ""
    fi
done < "$TEMP_FILE"
```

Explication : while read

```
while read count ip; do
    # commandes utilisant $count et $ip
done < fichier.txt
```

Explication :

- `while read` : lit le fichier ligne par ligne
- `count ip` : les deux premiers champs sont assignes a ces variables
- `< fichier.txt` : redirection de l'entree depuis le fichier
- A chaque iteration, une nouvelle ligne est lue

Exercice 3

Completer le script `detect_bruteforce.sh` pour qu'il :

1. Accepte le seuil en parametre (defaut : 10)
2. Analyse le fichier auth.log fourni
3. Identifie les IP avec plus de N tentatives
4. Genere un fichier `rapport_incident.txt` contenant :
    - La date du rapport
    - La liste des IP suspectes avec le nombre de tentatives
    - La periode de l'attaque (premier et dernier evenement)

Conclusion et ressources

Recapitulatif des commandes cles

Commande	Usage
`grep`	Recherche de motifs dans les fichiers
`find`	Localisation de fichiers par criteres
`awk`	Traitement de texte oriente colonnes
`sort`	Tri de lignes
`uniq`	Suppression de doublons et comptage
`cut`	Extraction de champs
`tail -f`	Surveillance en temps reel
`ss -tuln`	Ports en ecoute

Structures Bash essentielles

```
# Variable
variable="valeur"
echo "$variable"

# Condition
if [ condition ]; then
    commandes
fi

# Boucle
for element in liste; do
    commandes
done

# Substitution de commande
resultat=$(commande)
```

Pistes d'approfondissement

Entrainement pratique :

- **OverTheWire Bandit** : https://overthewire.org/wargames/bandit/
  - Challenges progressifs bases sur le shell Linux
  - Ideal pour pratiquer les commandes apprises

Scripts de reference :

- **CIS Benchmarks** : scripts de hardening standardises
- **Lynis** : outil d'audit de securite open source

Integration avec cron

```
# Editer la crontab
crontab -e

# Syntaxe : minute heure jour mois jour_semaine commande

# Execution quotidienne a 6h00
0 6 * * * /home/analyst/scripts/audit.sh

# Execution toutes les heures
0 * * * * /home/analyst/scripts/check_bruteforce.sh

# Execution toutes les 5 minutes
*/5 * * * * /home/analyst/scripts/surveillance.sh
```

Ressources complementaires

Documentation :

- `man commande` : manuel de la commande
- `commande --help` : aide rapide
- https://www.gnu.org/software/bash/manual/

Sites de reference :

- https://explainshell.com : explication de commandes
- https://www.shellcheck.net : verification de scripts
- https://linuxcommand.org : tutoriels

Points cles a retenir

1. **Bash est un outil essentiel** pour l'analyste securite
2. **Les pipes permettent** de combiner des commandes simples en traitements complexes
3. **Les scripts automatisent** les taches repetitives et garantissent la reproductibilite
4. **grep, awk, sort, uniq** forment le coeur de l'analyse de logs
5. **Toujours tester** dans un environnement isole avant la production

Merci pour votre attention

Questions / Reponses

Annexe : Aide-memoire

Navigation

```
ls -la          # Liste detaillee avec fichiers caches
cd /chemin      # Changer de repertoire
pwd             # Afficher le repertoire courant
cat fichier     # Afficher le contenu
less fichier    # Navigation interactive
```

Annexe : Recherche

```
# grep
grep 'motif' fichier          # Recherche basique
grep -i 'motif' fichier       # Ignore la casse
grep -c 'motif' fichier       # Compte les occurrences
grep -v 'motif' fichier       # Inverse la selection
grep -E 'regex' fichier       # Expressions regulieres etendues

# find
find /chemin -name "*.log"    # Par nom
find /chemin -mtime -1        # Modifie < 24h
find /chemin -perm -4000      # Fichiers SUID
```

Annexe : Traitement de texte

```
# awk
awk '{print $1}' fichier            # Premiere colonne
awk -F: '{print $1}' fichier        # Separateur :
awk '$1 > 10 {print}' fichier       # Condition

# sort et uniq
sort fichier                        # Tri alphabetique
sort -n fichier                     # Tri numerique
sort -rn fichier                    # Tri numerique decroissant
uniq -c                             # Compte les doublons
```

Annexe : Scripts

```bash
#!/bin/bash

# Variables
var="valeur"
var=$(commande)

# Arguments
$0    # Nom du script
$1    # Premier argument
$#    # Nombre d'arguments
$?    # Code retour

# Tests
[ -f fichier ]    # Fichier existe
[ -d rep ]        # Repertoire existe
[ -r fichier ]    # Fichier lisible
[ "$a" -gt "$b" ] # a > b (numerique)
```