FICHE DE RÉVISION — LINUX AVANCÉ

TSSR — Droits étendus, ACL et scripting Bash

1. Définitions essentielles

Notion	Valeur	Définition
SUID (SetUID)	4	Quand un fichier exécutable a le SUID, il s'exécute avec l'UID (les droits) du **propriétaire du fichier**, et non ceux de l'utilisateur qui le lance. Ce n'est donc pas « seul root peut le manipuler », c'est « n'importe qui peut l'exécuter, mais l'exécution tourne avec les privilèges du propriétaire ».
SGID (SetGID)	2	Sur un **fichier exécutable** : même logique que le SUID mais avec le **GID du groupe propriétaire**. Sur un **répertoire** : tout fichier ou sous-dossier créé à l'intérieur hérite du **groupe du répertoire parent**.
Sticky bit	1	Sur un répertoire où plusieurs utilisateurs ont les droits d'écriture, il empêche un utilisateur de supprimer ou renommer les fichiers d'un **autre** utilisateur, même s'il a l'écriture sur le dossier.
ACL	—	**Access Control List** : permissions **fines, à plusieurs utilisateurs et plusieurs groupes distincts**, en plus du triplet classique User/Group/Other.

2. SUID (SetUID) — valeur 4

Exemple canonique : `/usr/bin/passwd`

```
-rwsr-xr-x 1 root root /usr/bin/passwd
```

`/etc/shadow` n'est modifiable que par root. Un utilisateur lambda qui lance `passwd` doit pourtant pouvoir écrire dedans pour changer son propre mot de passe. Le SUID sur `passwd` fait que le binaire s'exécute **temporairement avec les droits root**, le temps de la modification, puis rend la main.

Commande

```
chmod u+s fichier
chmod 4755 fichier
```

Notation

Le `x` du propriétaire est remplacé par un `s` : **minuscule si le bit `x` était déjà présent**, `S` **majuscule sinon**. Ce `S` majuscule est un **signal d'anomalie** — SUID posé sur un fichier non exécutable, ça ne sert à rien.

Un SUID root mal configuré, ou posé sur un binaire vulnérable, est un vecteur classique d'ESCALADE DE PRIVILÈGES.

3. SGID (SetGID) — valeur 2

Le comportement est **différent selon la cible** :

- **Sur un fichier exécutable** : même logique que le SUID, mais avec le **GID du groupe propriétaire**. Le processus hérite des droits du groupe, pas de l'utilisateur qui lance.
- **Sur un répertoire** : tout fichier ou sous-dossier créé à l'intérieur hérite automatiquement du **groupe du répertoire parent**, au lieu du groupe primaire de l'utilisateur créateur.

```
drwxrws--- 2 pierre SALARIES /TRAVAIL
```

Notation : `s` (ou `S` majuscule s'il n'y a pas de `x`) à la position du `x` **groupe**.

Commande

```
chmod g+s dossier
chmod 2775 dossier
```

4. Sticky bit — valeur 1

Historiquement : forçait le noyau à garder le binaire en mémoire (swap) après exécution, pour accélérer les lancements suivants. **Obsolète** sur les systèmes modernes, **ignoré sur un fichier** aujourd'hui.
Usage actuel (uniquement sur répertoire) : dans un dossier où plusieurs utilisateurs ont les droits d'écriture, le sticky bit empêche un utilisateur de supprimer ou renommer les fichiers d'un **autre** utilisateur, même s'il a l'écriture sur le dossier. Seuls le **propriétaire du fichier**, le **propriétaire du dossier** ou **root** peuvent le faire.
Exemple canonique : `/tmp`

```
drwxrwxrwt 15 root root /tmp
```

> *Tout le monde peut écrire dans /tmp, mais personne ne peut effacer les fichiers des autres.*

Notation : `t` (minuscule si le `x` **other** est présent, `T` majuscule sinon) en **dernière position**.

Commande

```
chmod +t dossier
chmod 1777 dossier
```

5. Minuscule vs majuscule — la règle exacte

La casse ne dépend PAS du bit spécial lui-même, mais de l'état du x sous-jacent à cette position, avant que le bit spécial vienne l'écraser dans l'affichage.

Bit spécial	`x` déjà présent à cette position	`x` absent à cette position
SUID (User)	`s` minuscule	`S` majuscule
SGID (Group)	`s` minuscule	`S` majuscule
Sticky (Other)	`t` minuscule	`T` majuscule

6. Syntaxe générale de chmod

```
chmod [ugo][+-=][rwxXst] fichier
```

- **Cibles** : `u` (user/propriétaire), `g` (group), `o` (other)
- **Opérateurs** : `+` (ajoute), `-` (retire), `=` (définit exactement, écrase le reste)
- **Permissions** : `r`, `w`, `x`, plus `s` (SUID/SGID) et `t` (sticky)

> *Alphabétique ou numérique ? Si on veut cibler un ou tous les blocs, la méthode alphabétique suffit. En revanche, pour modifier de manière ciblée, il faut repasser en mode numérique.*

7. Les ACL sous Linux (Access Control List)

7.1 Pourquoi les ACL existent

Les permissions classiques (`rwx` pour User/Group/Other) sont limitées à **un seul propriétaire** et **un seul groupe** par fichier. Impossible nativement de dire « Pierre a le droit d'écrire, Carine seulement de lire, et le reste du groupe SALARIES rien du tout » sur le même fichier. Les ACL lèvent cette limite.

7.2 Prérequis

Le système de fichiers doit supporter les ACL (**ext4** et **XFS** le supportent nativement aujourd'hui). Paquet nécessaire :

```
apt install acl        # Debian/Ubuntu
dnf install acl        # Fedora/RHEL
```

7.3 Consulter les ACL — getfacl

`getfacl` = **Get File ACL**.

```
getfacl fichier
```

Exemple de sortie :

```
# file: fichier
# owner: root
# group: SALARIES
user::rw-
user:pierre:rwx
group::r--
group:STAGIAIRES:r--
mask::rwx
other::---
```

7.4 Poser une ACL — setfacl

```
setfacl -m [u|g]:nom:permissions fichier
```

- `-m` : **modify** (ajoute ou modifie une entrée)
- `u:nom:rwx` : ACL pour un **utilisateur** nommé
- `g:nom:rwx` : ACL pour un **groupe** nommé

Si un droit n'est pas stipulé clairement, c'est qu'il est ABSENT.

> *Pourquoi pas de o (other) ? Parce que « other » n'a aucun sens dans le contexte des ACL.*

Exemples :

```
setfacl -m u:pierre:rwx fichier      # Pierre a rwx sur ce fichier précis
setfacl -m g:STAGIAIRES:r-- fichier  # le groupe STAGIAIRES a lecture seule
setfacl -m u:carine:rw- fichier      # Carine a lecture/écriture
```

Retirer une entrée précise :

```
setfacl -x u:pierre fichier
```

Retirer toutes les ACL (retour aux permissions classiques UGO) :

```
setfacl -b fichier
```

7.5 Repérer un fichier avec ACL — le +

Un fichier possédant des ACL affiche un `+` après le triplet de permissions classique dans `ls -l` :

```
ls -l fichier
-rwxrwxr--+ 1 root SALARIES ... fichier
```

> *Ce + est le signal visuel qu'il faut lancer getfacl pour voir le détail complet : ls -l seul ne montre JAMAIS les entrées ACL nommées.*

7.6 Ordre d'évaluation des permissions

Quand un utilisateur accède à un fichier, le noyau évalue dans cet ordre :

Ordre	Entrée évaluée	Comportement
1	`user::` — user propriétaire	si l'utilisateur est le propriétaire, cette entrée s'applique et **rien d'autre n'est consulté**
2	`user:nom:` — ACL utilisateur nommé	si une entrée existe pour cet utilisateur précis
3	`group::` et `group:nom:`	groupe propriétaire et ACL de groupes nommés, **filtrés par le** `mask`
4	`other::`	en dernier recours

7.7 Résumé des commandes ACL

Commande	Effet
`getfacl fichier`	affiche les ACL
`setfacl -m u:nom:rwx fichier`	ajoute / modifie une ACL utilisateur
`setfacl -m g:nom:rwx fichier`	ajoute / modifie une ACL groupe
`setfacl -x u:nom fichier`	retire une entrée ACL précise
`setfacl -b fichier`	retire toutes les ACL
`setfacl -d -m g:nom:rwx dossier`	ACL par défaut (héritage sur les nouveaux fichiers)
`ls -l` **(repérer le** `+`**)**	signale la présence d'ACL sur le fichier

8. Scripting sous Linux — les variables

> *Convention Linux : tous les scripts portent l'extension .sh. Cela permet de savoir immédiatement qu'on est face à un script.*

8.1 Déclarer, appeler, effacer une variable

```
nom="pepito"       # déclaration
echo $nom          # appel de la variable
unset nom          # suppression de la variable
```

Si la variable n'existe pas, l'OS ne renvoie PAS d'erreur : il retourne simplement un vide.

8.2 Stocker le résultat d'une commande

```
ip=$(ip a)
```

- `$` symbolise que c'est le **résultat** de la commande que l'on veut
- la commande est entourée de **parenthèses** : `$(commande)`

8.3 La variable $? — code de retour

```
echo $?
```

Résultat	Signification
`0`	aucune erreur dans l'exécution de la commande

Résultat	Signification
différent de `0`	erreur dans l'exécution de la commande

> *La seule réponse commune à toutes les commandes est le 0 (exécution correcte). Tout autre retour est lié à la commande elle-même : c'est le code erreur documenté dans son aide avancée.*

8.4 Les arguments

Une commande a des options : ce qu'on appelle des **arguments**. Le script vérifie s'il a des arguments, et combien.

```
$1     # 1er argument
$2     # 2nd argument
```

9. Le chaînage des commandes

Opérateur	Rôle	Condition d'exécution de la commande 2
`;`	séquence	toujours
`&&`	ET logique	si commande1 **réussit** (code 0)
`||`	OU logique	si commande1 **échoue** (code ≠ 0)
`|`	pipe	pas de condition, transfert de flux stdout → stdin
`&`	background	s'applique à une seule commande, pas à un chaînage entre deux

9.1 ; — chaîneur inconditionnel

La commande d'après s'exécutera **sans condition** de résultat de la précédente.

9.2 && — ET logique (exécution conditionnelle sur succès)

La commande 2 ne s'exécute **que si** commande1 retourne un code de sortie **0** (succès).

```
ls >/dev/null && ls -al
```

- `ls` : liste le contenu du répertoire courant
- `>/dev/null` : redirige le flux **stdout** de `ls` vers `/dev/null`, le device spécial qui absorbe et détruit tout ce qu'on lui envoie (le « trou noir » de Linux). La sortie de ce premier `ls` n'apparaît nulle part, elle est jetée
- `&&` : la commande suivante ne s'exécute que si la précédente a retourné le code de sortie 0
- `ls -al` : liste à nouveau le répertoire, avec `-a` (fichiers cachés, commençant par un point) et `-l` (format long : permissions, propriétaire, groupe, taille, date de modification)

9.3 || — OU logique (exécution conditionnelle sur échec)

La commande 2 ne s'exécute **que si** commande1 retourne un code de sortie **différent de 0**. Autrement dit, elle ne s'exécute que si la commande de gauche **échoue**.

9.4 | — le pipe

Le pipe transfère le **résultat** de la commande précédente à la commande suivante.

```
ls /etc | grep host*
```

Le résultat de `ls /etc` est transféré à `grep`. On peut aussi **enchaîner plusieurs pipes**.

9.5 Décorticage d'un enchaînement de pipes

```
ip a | grep "inet " | tr -s ' ' ':' | cut -d":" -f3
```

Étape 0 — `ip a` : affiche toutes les interfaces réseau avec leurs détails complets. Ici deux interfaces : `lo` (loopback, 127.0.0.1/8, adresse locale de la machine elle-même) et `eth0` (interface physique/virtuelle, IPv4 `10.50.0.14/16`).
Étape 1 — `grep "inet "` : le motif `"inet "` (avec **l'espace final**, entre guillemets pour que le shell ne coupe pas la chaîne) filtre uniquement les lignes contenant exactement « inet » suivi d'un espace. Ça exclut volontairement `inet6` et ne garde que les lignes IPv4 :

```
inet 127.0.0.1/8 scope host lo
inet 10.50.0.14/16 brd 10.50.255.255 scope global dynamic eth0
```

Étape 2 — `tr -s ' ' ':'` : `tr` fait ici deux choses combinées. `-s` (squeeze) compresse les répétitions consécutives du caractère cible en une seule occurrence (utile car les lignes de `ip a` contiennent plusieurs espaces d'indentation), et `' ' ':'` remplace chaque espace par un `:`.

```
:inet:127.0.0.1/8:scope:host:lo
:inet:10.50.0.14/16:brd:10.50.255.255:scope:global:dynamic:eth0
```

> *Le : en tout début de ligne vient de l'espace d'indentation initial, converti lui aussi.*

Étape 3 — `cut -d":" -f3` : `cut` découpe chaque ligne selon le délimiteur `-d":"` et extrait uniquement le **3e champ** (`-f3`).

Champ	Contenu	Remarque
1	(vide)	avant le premier `:`
2	`inet`	le mot-clé filtré par `grep`
3	`127.0.0.1/8`	**c'est celui qu'on extrait**

Résultat final :

```
127.0.0.1/8
10.50.0.14/16
```

Commande / option	Rôle
`tr`	remplace ou supprime des caractères **un par un** dans un flux (ex : espace → `:`)
`cut`	découpe chaque ligne en **champs** selon un délimiteur et n'en garde que certains
`-f` (option de `cut`)	précise quel(s) numéro(s) de champ extraire (ex : `-f3` = 3e champ)

Lorsqu'on fait du chaînage conditionnel, attention : les chaînages se basent sur le résultat de la DERNIÈRE commande qui a fonctionné, que ce résultat soit positif ou en échec.

10. Pièges et points à retenir pour l'ECF

SUID = 4, SGID = 2, Sticky = 1. C'est le 4e chiffre placé devant le triplet classique : chmod 4755, 2775, 1777.

- `s` / `t` **minuscule** = le `x` était déjà là. `S` / `T` **majuscule** = pas de `x` → sur un SUID, c'est un **signal d'anomalie**.
- Le sticky bit sur un **fichier** ne sert plus à rien aujourd'hui : usage **répertoire uniquement**.
- SGID sur répertoire = héritage du **groupe du parent**, pas du groupe primaire du créateur.
- Un `+` dans `ls -l` = présence d'ACL → lancer `getfacl`. `ls -l` seul ne montre jamais les entrées nommées.
- Pas de `o` (other) dans les ACL : ça n'a aucun sens dans ce contexte.
- Ordre d'évaluation : propriétaire d'abord, et si l'utilisateur est le propriétaire, **rien d'autre n'est consulté**.
- `echo $?` renvoie **0** = succès. Tout autre code est **spécifique à la commande**.
- Une variable inexistante ne provoque **aucune erreur** : elle retourne un vide.

- `&&` = la suite s'exécute si ça **réussit** ; `||` = la suite s'exécute si ça **échoue**.

Un SUID root mal configuré = escalade de privilèges. C'est LE vecteur d'attaque classique sur les droits étendus.