TP6: File descriptors, sandboxing et architecture sécurisée

Matthieu Lemerre Guillaume Girol Grégoire Menguy

IN201 - Cours de système d'exploitation

Le but de ce TP est de comprendre comment fonctionnent les file descriptors UNIX, et de comprendre comment mettre en place une architecture sécurisée dans ce système.

Les file descriptors Unix sont des entiers numérotant un flux de données entrant ou sortant, pouvant être un fichier, une connexion réseau, ou une connexion avec un autre processus.

L'ouverture de nouveaux fichiers avec `open`, la création de pipe avec `pipe` créent de nouveaux flux de données, et auxquels correspondent de nouveaux file descriptors.

Par défaut lorsqu'un nouveau processus s'exécute, il possède trois descriptors ouverts:

- 0 correspond à l'entrée standard (typiquement l'entrée au clavier)
- 1 correspond à la sortie standard (typiquement l'affichage sur un terminal)
- 2 correspond à la sortie d'erreur (typiquement également l'affichage sur un terminal)

La ligne de commande UNIX permet de changer facilement ces flux de données; par exemple

```
echo -e "Hello\nWorld" | programme
```

Permet de brancher la sortie standard de la commande "echo" vers l'entrée standard du programme "programme".

Question 1 Le but de cette première question est d'écrire un petit interpréteur, qui lit une commande sur son entrée standard, et écris la commande sur sa sortie standard.

Les commandes sont écrites sous la forme d'un caractère, suivi des arguments séparés par une virgule, la fin de la commande étant signalée par un retour à la ligne.

Les deux commandes à implémenter initialement sont '+', qui prend deux entiers et retourne leur addition; et '-', qui prend deux entiers et retourne leur soustraction.

L'exécution du programme doit ainsi donner:

```
echo "+3,2" | programme
# Affiche 7
echo "-8,2" | programme
# Affiche 6
```

Pour cela, utilisez les fonctions `read` et `write` (déclarés dans les header `<unistd.h>`), qui permettent de charger le contenu d'un descriptor dans un tampon de communication que vous aurez préalablement alloué; et les fonctions `sprintf` et `sscanf` (déclarés dans les header `<stdio.h>`), qui permettent de parser et formater les données contenues dans un tampon.

Question 2 Nous allons ajouter une nouvelle commande, 'e' suivi d'une chaîne de caractère. Cette commande exécute la chaîne de caractère suivant la commande (avec la fonction `system`)et retourne le code d'erreur de l'exécution.

```
echo "els ~" | programme
# Affiche le contenu du répertoire home de l'utilisateur.
```

Question 3 Supposons que l'entrée du programme soit contrôlable par un attaquant (par exemple, le programme prend ses entrées depuis un port réseau sur internet, ou depuis un fichier écrit par un attaquant). Comment un attaquant peut-il exploiter le programme? Écrivez par exemple un fichier de commande permettant de recopier le contenu du fichier `/etc/passwd` dans un fichier `~/attack_result`.

Question 4 Considérant que cet interpréteur de commande est utile mais souhaitant limiter les risques de son utilisation, vous allez encapsuler le processus.

Pour cela, incluez les headers `<linux/seccomp.h>`, `<sys/prctl.h>` et `<sys/syscall.h>`, et exécutez au début de votre programme:

```
prctl(PR_SET_SECCOMP, SECCOMP_MODE_STRICT);
```

Cette commande limite les actions qui peut faire votre processus aux appels systèmes `read`, `write` et `exit`: celui-ci ne peut donc plus que lire et écrire dans les file descripteurs déjà ouverts.

En quoi l'utilisation de cette commande apporte une protection pour votre système?

Note: vous devez maintenant quitter votre programme en appelant `syscall(SYS_exit, 0);` car la procédure standard qui clot un processus UNIX fait d'autres appels systèmes que `exit`.

Question 5 Vous souhaitez maintenant intégrer l'interpréteur de commande au sein d'un programme plus gros. Pour faire cela de manière sécurisée, vous allez isoler l'interpréteur dans un processus à part.

La manière de faire cela est la suivante:

1. Tout d'abord, votre programme doit créer deux canaux de communications, en appelant deux fois la fonction `pipe`. Chaque appel à `pipe` crée 2 file descriptors, l'un permettant d'envoyer des données, et l'autre de les recevoir.

2. Puis, votre programme doit se diviser avec la fonction `fork`. Celui des processus qui reçoit la valeur 0 en retour à cet appel est le fils, l'autre est le père.

3. Puis, le père et le fils doivent fermer chacun deux file descriptors de manière à ce que chaque processus lise dans un pipe et écrive dans l'autre pipe, et que les deux processus puissent ainsi communiquer.

4. Ensuite, le fils doit appeler `prctl(PR_SET_SECCOMP, SECCOMP_MODE_STRICT);` pour entrer dans le mode d'exécution encapsulé.

5. Une fois ceci mis en place, le père peut maintenant envoyer les commande au fils afin que celui-ci les exécute de manière sécurisée. Pour simplifier, vous pouvez faire envoyer par le père des commandes situées dans une chaîne de caractère constante dans la mémoire du père.