PNL - 4I402

TP 09 – Appels systèmes

Redha Gouicem, Maxime Lorrillere et Julien Sopena

avril 2019

Environnement : ce TP reprend l'environnement de programmation mis en place lors du troisième TP. Il suppose entre autre l'utilisation dans qemu de l'image *pnl-tp.img*, ainsi que l'existence d'un fichier personnel *myHome.img* correspondant au `/root`. Il suppose aussi que vous ayez un fichier de configuration pour le noyau Linux 4.19.3 et que vous ayez dans `/tmp` un exemplaire des sources.

Exercice 1 : Mon premier appel système

Les appels systèmes sont des interfaces de communication fournies par le noyau pour permettre aux applications d'accéder à certaines de ses ressources. Un appel système est défini par son *numéro* et par ses paramètres. Le noyau possède une *table des appels systèmes* contenant les fonctions à exécuter : le numéro d'un appel système correspondant à un indice de cette table, et donc à sa fonction.

La plupart du temps, les appels systèmes sont exécutés par les applications par l'intermédiaire d'APIs, telles que POSIX ou la bibliothèque standard C. Lorsqu'aucune API ne fournit de fonction pour accéder à un appel système, la fonction `syscall` peut être utilisée pour accéder à l'appel système correspondant. Cette fonction exécute l'appel système, et s'il échoue (valeur de retour inférieure à 0), place le code d'erreur correspondant dans la variable `errno` et renvoie `-1`. Sinon elle renvoie la valeur retournée par l'appel système.

Les appels systèmes sont très dépendants de l'architecture utilisée. Dans l'architecture x86 64 bits (que vous utilisez en TP), la table des appels système est définie dans le fichier `arch/x86/entry/syscalls/syscall_64.tbl`. Cette table définit pour chaque appel système son numéro, son ABI (`common` pour nous), son nom et le nom de la fonction correspondante.

L'implémentation d'un appel système passe par l'utilisation de macros fournie par le noyau de la forme suivante :

```
#define SYSCALL_DEFINEx(nom, type1, nom1, type2, nom2, ...)
```

Ces macros définissent une fonction, dont le nom est `sys_nom` et prenant x paramètres de types donnés. Le type de retour de la fonction ainsi définie sera `long`. Par convention, une valeur inférieure à 0 est un code d'erreur négatif.

Par exemple, un appel système ne prenant pas de paramètre, tel que l'appel système `sync`, peut être sera défini de la façon suivante :

```
SYSCALL_DEFINE0(sync)
{
    /* ... */
    return 0;
}
```

Un appel système prenant 2 paramètres, tel que l'appel système `chmod`, peut être défini de la façon suivante :

```
SYSCALL_DEFINE2(chmod, char __user *, filename, umode_t, mode)
{
    /* ... */
    return 0;
}
```

Dans ce cas, les paramètres de type pointeur doivent être manipulés avec précautions, en utilisant par exemple les fonctions `copy_to_user` et `copy_from_user`.

Question 1

Ouvrez la page de manuel de la fonction `syscall`. Quel est le rôle de cette fonction ? Est-ce un appel système ? Quels sont ses paramètres et sa valeur de retour ?

Question 2

Réalisez un petit programme C qui exécute l'appel système `kill`, dont le numéro est `__NR_kill`, en utilisant la fonction `syscall`.

Question 3

Ajoutez un nouvel appel système `hello` à votre noyau et testez le avec un programme `test_hello.c`. Lorsqu'il est exécuté, cet appel affiche le message *Hello world !"* dans le syslog.

Question 4

Lors de l'exécution de votre appel système, affichez le PID du processus courant dans le syslog. Que constatez vous ?

Question 5

On souhaite maintenant modifier dynamiquement l'affichage généré par l'appel système en y ajoutant un paramètre `who`. Par exemple, passer la valeur *"Redha"* comme paramètre génèrera l'affichage *Hello redha !"* dans le syslog. Modifiez votre appel système en conséquence et testez le.

Question 6

Un grand pouvoir implique de grandes responsabilités. Votre appel système a peut-être introduit une faille de sécurité. Pour le vérifier essayez de passer en paramètre l'adresse obtenu en chargeant le module `one_addr.c` dont le code est fourni avec le sujet. Comment peut-on résoudre ce problème.

Question 7

Plutôt que d'afficher le résultat dans le syslog, on souhaite renvoyer la chaîne générée dans un tampon fourni par l'utilisateur. Modifiez votre appel système en conséquence, celui-ci devra retourner la taille de la chaîne générée, ou un code d'erreur négatif si l'exécution de l'appel système échoue.

Exercice 2 (rootkit 4) : Détourner un signal

Dans cet exercice, on cherchera dérouter les signaux adressés aux processus dissimulés grâce au TP précédent. Les techniques employées permettront d'étudier : le fonctionnement d'un appel système sous Linux, le rôle de la table des appels systèmes et la table des symboles du noyau.

Question 1

Après avoir lancé une commande `sleep 9000 &`, envoyez lui un signal `SIGCONT` puis renouvelez l'opération après l'avoir masqué.

En comparant vos observations au résultat de l'envoi du même signal à un processus inexistant, proposez une méthode permettant de lister les processus cachés.

Question 2

Pour contrer cette détection nous allons dérouter le signal au niveau des appels systèmes. Lancez la commande `strace` sur un `kill`. Quels appels systèmes doit-on dérouter ?

Question 3

L'ensemble des appels systèmes passe par l'interruption `int 0x80`. Pour les différencier, on leur associe un numéro (*system call number*). Une fois basculé en mode noyau, ce numéro est utilisé pour chercher dans la **table des appels systèmes** l'adresse de l'appel système.

Intercepter un appel système correspond donc à remplacer, dans la table des appels systèmes, l'adresse originale par celle de votre fonction. Cette méthode nécessite de connaitre le début de la table noté par le symbole `sys_call_table`. Malheureusement, depuis la version 2.6, ce symbole n'est plus exporté. Nous allons tout de même tenter de la retrouver sans utiliser la fonction `kallsyms_lookup_name` vue dans le troisième TP.

Pour commencer rechercher tous les appels systèmes exportés du noyau. Les symboles de ces derniers étant toujours préfixés par `sys_xxx`.

Question 4

Pour chercher la table nous allons nous baser sur deux faits :

1. la table se trouve forcément dans l'espace mémoire noyau, *i.e.*, au dessus de l'adresse définie par la macro `PAGE_OFFSET`.
2. la table contient une liste des adresses des appels systèmes, tel que l'adresse du `sys_xxx` se trouve dans la case `__NR_xxx` (`<linux/unistd.h>`).

Implémentez une fonction `find_sys_call_table` qui retourne l'adresse de la table en recherchant dans la mémoire l'adresse d'un appel système exporté.

Question 5

Maintenant que vous avez l'adresse de la table, essayez de remplacer par `NULL` l'adresse du premier appel système de la table. Que se passe-t-il ?

Question 6

En fait la table des appels systèmes se trouve sur une page protégée en écriture.

Notre code s'exécutant en mode noyau on devrait quand même pouvoir la modifier, mais *Intel* a ajouté un mécanisme de protection basé sur le bit `wp` du registre de contrôle (control register) `cr0`.

Vérifiez la présence de ce mécanisme en consultant le `/proc/cpuinfo`.

Question 7

Sachant que le bit `wp` est le 16eme du registre `cr0`, utilisez les fonctions `read_cr0` et `write_cr0` pour vous redonner les droits et testez vos nouveaux pouvoirs.

Vous veillerez naturellement à restaurer le registre après modification de la table des appels systèmes.

Question 8

Vous pouvez maintenant modifier la table de façon à intercepter le `sys_kill` avec une fonction qui retourne `-ESRCH` si le *pid* correspond au processus à cacher.

Attention : votre module doit conserver le fonctionnement normal des signaux pour les autres processus. D'autre part vous devez rétablir le comportement original lors du déchargement du module.