PNL - MU4IN402

TP 03 – Premiers pas dans le noyau

Redha Gouicem, Maxime Lorrillere et Julien Sopena

février 2020

Exercice 1 : Environnement de développement de l'UE.

Question 1

Dans le cadre de cette UE, nous allons utiliser une instance de la machine virtuelle **qemu** comme plateforme de développement. Pour commencer, **comme avant chaque début de TP**, copiez depuis `/usr/data/sopena/pnl/`, l'image `pnl-tp.img` dans `/tmp`. Cette image contient une distribution archlinux avec un noyau *Linux* et un compte root sans mot de passe.

Que se passe-t-il lorsqu'on la lance avec la commande suivante :

```
qemu-system-x86_64 -drive file=pnl-tp.img,format=raw
```

Question 2

Votre image étant stockée dans le `/tmp`, elle sera perdue à chaque redémarrage de la machine. Vous allez donc ajouter un autre disque qui sera monté dans `/root` (le répertoire de travail de l'utilisateur *root*). Générez une image personnelle à l'aide des commandes suivantes :

```
dd bs=1M  count=50  if=/dev/zero of=${HOME}/myHome.img
/sbin/mkfs.ext4 -F ${HOME}/myHome.img
```

Question 3

Récupérez maintenant le script de lancement qemu-run.sh qui se trouve dans `/usr/data/sopena/pnl/`. Ouvrez-le puis corrigez si besoin les chemins menant aux différents fichiers en fonction de votre configuration. Vous pouvez maintenant lancer votre VM et vous connecter en tant que root (pas de mot passe).

Exercice 2 : Configuration et compilation du noyau Linux

Nous allons maintenant créer notre propre version du noyau sur la machine hôte et l'utiliser pour lancer notre VM. Cette technique permet non seulement d'éviter de compiler le noyau dans la VM, mais simplifiera aussi le débogage (voir TP 03).

Pour commencer, décompressez l'archive `linux-4.19.3.tar.xz` dans le répertoire `/tmp`

```
tar -xvJf /usr/data/sopena/pnl/linux-4.19.3.tar.xz -C /tmp/
```

Question 1

Pour simplifier ce TP et accélérer la compilation, nous avons préparé une configuration allégée du noyau. Copiez cette configuration dans le répertoire de votre noyau sous le nom **.config**.

```
cp /usr/data/sopena/pnl/linux-config-pnl /tmp/linux-4.19.3/.config
```

Accédez à la configuration du noyau Linux à l'aide de la commande `make nconfig`. Familiarisez vous avec l'interface en parcourant quelques options. Dans quelle catégorie allons nous trouver des options de débogage que nous pourrons utiliser dans le TP 03 ?

Question 2

Compiler le noyau est une étape simple mais qui peut s'avérer très longue. Si optimiser sa configuration permet de limiter la taille du code compilé, il est nécessaire d'utiliser au mieux les ressources disponibles.

Commencez par trouver le nombre de cœurs de votre machine. Quelle doit alors être la valeur du paramètre `-j` lorsque vous compilerez votre noyau.

Question 3

Lancez la compilation de votre noyau puis affichez les informations de votre image à l'aide de la commande `file`

```
file arch/x86/boot/bzImage
```

Question 4

Comme pour le script `qemu-run.sh` corrigez si besoin les variables du script qemu-run-externKernel.sh, puis utilisez-le pour démarrer une machine virtuelle. Affichez le nom du noyau en cours d'exécution à l'aide de la commande `uname -r`. Pouvez-vous être sûr qu'il s'agit de votre noyau ?

Question 5

Dans la section *General setup*, modifiez l'option de configuration qui permet de modifier le nom du noyau pour qu'il se termine par `-pnl`.

Recompilez votre noyau, démarrez une machine virtuelle sur celui-ci et vérifiez à l'aide de la commande `uname -r` qu'il s'agit bien du votre.

Question 6

Dans votre machine virtuelle, affichez la liste des modules chargés. En étudiant la configuration du noyau, expliquez le résultat obtenu.

Exercice 3 : Le processus *init*

Le but de cet exercice est de comprendre le rôle du processus init dans le démarrage d'un système Linux, et notament de comprendre qu'il s'agit d'un programme comme un autre qui peut être remplacé par n'importe quel exécutable.

Question 1

Pour commencer implémentez **dans la VM** un programme `hello` qui attendra 5 secondes pour se terminer après le classique *Hello World*. Vous veillerez à placer l'exécutable à la racine du système.

Question 2

Le kernel dispose d'une option `init=xxx` pour modifier le binaire *init* à exécuter. Cette option peut être définie dans la configuration du boot loader (ici grub), mais nous allons plus simplement le faire en passant par notre script `qemu-run-externKernel.sh`. Modifier ce dernier pour que le noyau exécute votre programme `hello` comme processus *init*.

Question 3

Expliquez pourquoi le système fini par crasher.

Question 4

Puisque l'on peut remplacer l'*init* original par n'importe quel programme, il est possible de lancer un shell en tant que processus 1. Testez, en lançant quelques commandes, cette astuce qui permet dans bien des cas de récupérer un système corrompu.

Si la plupart des commandes sont actives, pourquoi ne peut-on pas lancer directement une commande `ps` dans le shell ainsi obtenu ?

Question 5

Corrigez le problème et testez dans le shell les commandes `ps aux` et `pstree 0`.

Question 6

Pour finir, nous allons tenter de retrouver un fonctionnement normal en finalisant le processus de démarrage. Trouvez un moyen de lancer le script `init` original depuis votre shell.

Exercice 4 : Comprendre le fonctionnement du *initramfs*

Dans cet exercice, nous allons étudier le principe de l'*initramfs* et voir comment ce mécanisme de chargement est à la base de nombreuses distributions.

Question 1

Pour commencer, nous allons analyser le contenu de l'`initramfs` de la distribution Linux de l'*hôte*. Après avoir créé un repertoire dans `/tmp`, désarchivez l'`initramfs` présent dans le répertoire `/boot` grâce à la commande suivante. En parcourant l'arborescence obtenue, vous remarquerez la présence d'un exécutable `init` à la racine de l'archive.

```
cd /tmp/test
zcat /boot/initrd.img-3.16.0-4-amd64 | cpio -i -d -H newc --no-absolute-filenames
```

Question 2

Pour utiliser votre `helloWorld` comme `init` dans un `initramfs`, vous allez devoir le recompiler en utilisant l'option `-static` de `gcc`. A quoi sert elle ? Pourquoi est elle nécessaire dans ce cas ?

Question 3

Nous allons maintenant créer notre propre `initramfs` dans l'hôte puis démarer la *VM* dessus. Après avoir créé un repertoire `racine` contenant une version de votre programme `helloWorld` nommé `init`, vous allez générer une archive cpio à l'aide des commandes suivantes :

```
cd racine
find . | cpio -o -H newc | gzip > ../my_initramfs.cpio.gz
```

Question 4

Pour tester votre archive, il vous suffit d'utiliser l'option `-initrd` de qemu (avant l'option `-append`).

Exercice 5 : Complément sur le chargement dynamique de bibliothèques

Dans cet exercice, nous allons étudier les différentes fonctionnalités offertes par les bibliothèques dynamiques qui sont à la base de la programmation des modules.

Question 1

Pour commencer, récupérez depuis `/usr/data/sopena/pnl/TP-03` les fichiers : `cron_func.c`, `func.h`, et `nothing.c` ainsi que le `Makefile` associé et exécutez le programme `./cron_func` après l'avoir compilé. Ouvrez ensuite les sources pour comprendre son fonctionnement.

Question 2

Dans un premier temps, on veut pouvoir modifier le comportement du programme sans le recompiler. L'idée est d'embarquer l'implémentation de `func()` dans une bibliothèque dynamique `libfunc.so`.

Modifiez en conséquence votre `Makefile` et vérifiez que l'on puisse réimplémenter la fonction `func` dans le fichier `nothing.c` sans avoir à recompiler le programme `./cron_func`.

Question 3

Si l'utilisation massive de bibliothèques dynamiques permet d'économiser de la mémoire, elle peut poser des problèmes de sécurité. Il est en effet possible de rediriger un exécutable vers une version modifiée d'une bibliothèque.

Dans un premier temps, modifiez le comportement de `cron_func` pour que tous les appels à la fonction read affichent le message `"Tchao !!!"` et terminent le programme en retournant le caractère `'e'` (sans vraiment le lire).

Pour réaliser cette injection, vous utiliserez la variable `LD_PRELOAD` :

```
LD_PRELOAD=./libread.so ./cron_func
```

Question 4

Remplacer brutalement une fonction conduit souvent au crash du programme et donc à la fin du hack. Il est souvent plus utile d'appeler la fonction originale depuis la fonction injectée et de modifier son résultat.

Modifiez maintenant le comportement de `cron_func` pour que que la saisie d'un 'r' exécute le code d'une insertion (action du caractère 'i'), tout en conservant tous les autres comportements.

Pour réaliser cette nouvelle injection, vous associerez la `dlsym` à l'utilisation de la variable `LD_PRELOAD`. Utilisez `man dlsym` pour trouver comment récupérer le bon `"handle"`. Vous veillerez aussi à bien inclure toutes les libraries au moment de la compilation (voir `man dlsym`).

Question 5

Comment peut-on se prémunir de ce type d'attaques et à quel prix ?

Testez votre solution avec la commande `ltrace` :

```
ltrace -o log.txt ./cron_func
```

Question 6

On veut maintenant pouvoir modifier le comportement d'un programme en cours d'exécution, sans avoir à le redémarrer. A cette effet, l'appel système `dlopen` permet de recharger un symbole depuis une bibliothèque passée en paramètre.

Après avoir implémenté une nouvelle version de la fonction `func()` (respectant le prototype défini dans `func.h`), modifiez le `cron_func` pour qu'un `'i'` charge votre fonction et qu'un `'r'` restore l'implémentation originale.