PNL - 4I402

TP 08 – Premiers pas dans le VFS

Maxime Lorrillere et Julien Sopena

janvier 2016

Le but de ce TP est de se familiariser avec les abstractions du VFS et le . Il permettra aussi d'aborder le fonctionnement du cache des *dentry* et de son importance dans les performances du système.

Environnement : ce TP reprend l'environnement de programmation mis en place lors du deuxième TP. Il suppose entre autre l'utilisation dans qemu de l'image *nmv-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 4.2.3 et que vous ayez dans `/tmp` un exemplaire des sources.

Exercice 1 : Patcher son noyau à l'ancienne

Question 1

Pour faire ce TP nous allons devoir travailler sur une version légèrement modifiée du noyau. Commencez par charger le patch `tp7-linux-4.2.3.patch.xz`, puis appliquez le sur les sources de votre noyau. Que fait-il ?

Question 2

Nous allons aussi avoir besoin de la variable `d_hash_shift`. Trouvez cette variable dans le noyau. Pouvez vous l'utiliser dans l'un de vos modules ? Faites les modifications nécessaire si ce n'est pas le cas.

Question 3

Afin de pouvoir refaire ce TP facilement, générez une nouvelle version du patch `tp7-linux-4.2.3.patch.xz`. Lors de la génération vous veillerez à faire attention à la création de nouveaux fichiers (même si ici ce n'est pas le cas), à utiliser le format unifié et à bien comparer toute l'arborescence.

Exercice 2 (rootkit 2) : Un cache pas très discret

Question 1

Le cache des *dentry* repose sur une table de hachage. En regardant les sources du noyau, et plus particulièrement la fonction `d_hash`, déterminez la taille de cette dernière, *i.e.*, le nombre de listes qui la composent.

Question 2

Réalisez un module `weasel` qui affiche l'adresse de la table ainsi que sa taille.

Question 3

Modifiez votre module pour qu'il affiche le nombre de *dentry* contenu dans le cache au chargement du module. Affichez aussi la taille de la liste la plus longue. Que pensez vous de la taille de cette table.

Question 4

Afin d'obtenir des informations en dehors du chargement du module, nous allons utiliser le *procfs*. Dans un premier temps, modifiez votre module pour qu'il crée un répertoire `/proc/weasel` ainsi qu'un fichier `/proc/weasel/whoami` qui affiche *"I'm a weasel!"* lorsque l'on fait un `cat` dessus. Vous utiliserez pour cela les fonctions `proc_create`, `remove_proc_entry`.

Question 5

On souhaite maintenant afficher la liste de tous les *dentry* du cache dans le fichier `/proc/weasel`. Vous utiliserez pour cela une version simplifiée des `seq_files`. Il n'est ici pas nécessaire de créer une `struct seq_operations`, vous pouvez simplement appeler la fonction `single_open` lors de l'ouverture de votre fichier. Vous trouverez de multiples exemples d'utilisation de cette méthode dans le noyau Linux.

Question 6

Tentez d'exécuter dans votre terminal une commande inexistante puis affichez le contenu de `/proc/weasel`. Déduisez-en votre `PATH`. Pourquoi ces erreurs sont-elles enregistrées dans le cache ?

Question 7

Il arrive parfois à un utilisateur, après un echec lors de l'authentification d'une commande commande `su`, de ré-entrer directement le mot passe. Celui si est alors interprété comme une commande par le shell.

Ajoutez au *procfs* de votre module, un fichier `/proc/weasel/pwd` qui affiche lorsqu'on le lit la liste des commandes qui n'ont pas été trouvées dans le `$PATH`.

Exercice 3 (rootkit 3) : Cacher un processus - naïf

Le but de cet exercice est d'implémenter un module permettant de cacher à un utilisateur, utilisant des commandes de type `ps`, la présence du processus dont le *pid* sera passé en paramètre.

Les techniques employées permettront d'étudier : le **Virtual File Système**, la notion de **dentry** et le fonctionnement du **procfs**.

Question 1

Les commandes de type `ps` affichent le processus listé dans le répertoire `/proc`. Cacher un processus à un utilisateur revient donc à dissimuler l'entrée correspondant au processus. Nous allons donc modifier la lecture de ce répertoire.

Comme nous l'avons vu en cours, le noyau masque les différents systèmes fichier au moyen d'une interface générique appelée **Virtual File Système (VFS)**. Cette couche d'abstraction manipule quatre types d'objet : *fichier*, *dentry*, *inode* et *superblock*.

Dans cet exercice, nous allons manipuler les objets de type *dentry* pour modifier les fonctions du *procfs* permettant d'accéder au `/proc`.

Pour commencer, utilisez la fonction `filp_open` qui retourne une `struct file` correspondant au fichier `/proc`.

Question 2

Le fonctionnement du *VFS* consiste à rediriger les fonctions d'accès aux fichiers vers leur implémentation dans le système de fichier de leur partition. L'ensemble des pointeurs de fonction correspondant à un système est enregistré dans une `struct file_operations`.

Pour cacher le processus, vous allez rediriger l'appel à la fonction `iterate` du *procfs*, dont le rôle est de parcourir la liste des fichiers d'un répertoire, vers une autre version (`my_iterate`). Cette nouvelle version doit reprendre la signature suivante :

```
int my_iterate(struct file *fp, struct dir_context *ctx)
```

L'implémentation originale de cette fonction utilise la fonction transmise par l'intérmédiaire de la `struct dir_context` (*actor*), sur chaque élément du répertoire. Cette fonction de "callback", de type `filldir_t`, remplit un buffer (une `struct dirent`) à partir du nom de l'élément.

Pour simplifier la réécriture de la fonction `iterate`, vous utiliserez la version originale en remplaçant au préalable l'*actor* de la `struct dir_context` par une fonction `my_actor` à vous) qui retournera 0 si le nom correspond au `pid` à cacher. Cette fonction devra respecter la signature suivante :

```
int my_actor (struct dir_context *ctx, const char *name, int nlen, loff_t off, ino_t ino, unsigned x)
```

Attention : votre implémentation doit permettre de retrouver le comportement original si le nom ne correspond pas au *pid*.