France Métro : 5,95 Eur - BEL : 6,85 Eur - CH : 12 FS - CAN : 11 $ - LUX : 6,85 Eur - PORT.CONT : 6,85 Eur- MAR : 60 DH

KERNEL Episode 2

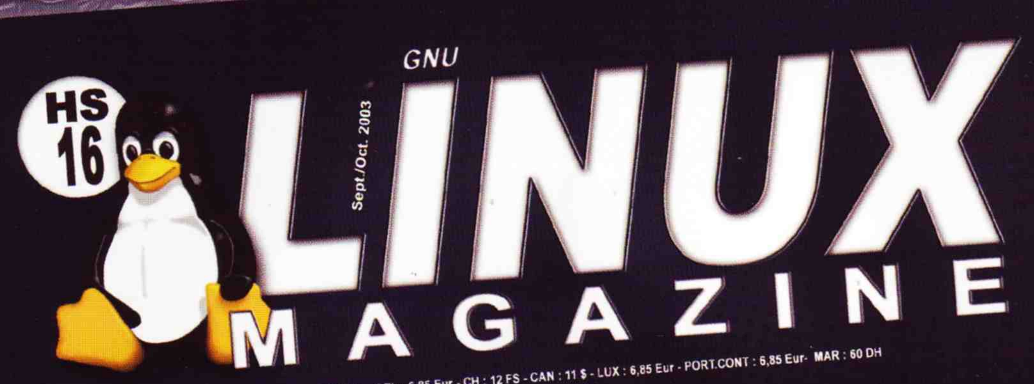
Voyage au centre du noyau

- Découvrir **UML**, ou comment mettre des **Linux** dans son **Linux**
- Autopsie du **boot** Linux
- Le noyau et le réseau : Comment **repousser** les **limites** de la **connectivité**
- Développez vos **pilotes** de périphériques **USB**
- **SELinux**, l'agence de **sécurité** du noyau

L 15066 - 17 - F: 5,95 € - RD

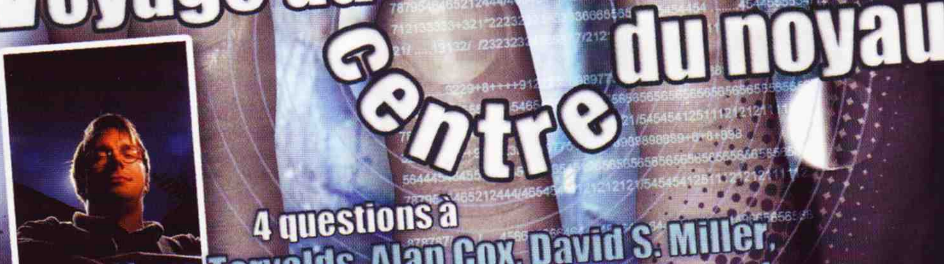
Le magazine en français 100% LINUX

Commandez
Linux Magazine Hors-Série
N°s 17 sur :



France Métro : 5,95 Eur - BEL : 6,85 Eur - CH : 12 FS - CAN : 11 $ - LUX : 6,85 Eur - PORT.CONT : 6,85 Eur - MAR : 60 DH

KERNEL
Episode 1



Architecture des noyaux (µ, monolithique, modulaire, exo)
Les nouveautés du 2.6 : visite guidée
Tutoriel de programmation noyau

Et aussi : Patcher son noyau, Ext2/Ext3, ...

Le magazine en français 100% LINUX

www.ed-diamond.com

Voyage au centre du noyau - épisode 2

Bienvenue dans ce hors série de Linux Magazine,

Après avoir découvert les premiers éléments dans le précédent hors série, voici que nous "remettons le couvert" pour poursuivre nos explorations du monde étrange et fascinant du kernel Linux.

Si le HS 16 trône encore à côté de votre ordinateur, ou mieux encore, à côté de votre lit, vous risquez de ne pas être déçu.

En effet, le présent numéro est quelque peu plus pratique et abordera des concepts directement applicables tant au niveau de ce qui existe qu'en ce qui concerne ce que vous pouvez faire. Ainsi, après avoir assimilé les articles qui suivent cet éditorial, vous pourrez comprendre encore davantage le fonctionnement du noyau, mais aussi développer ou commencer le développement de vos propres drivers ou modules.

On notera que la sécurité n'est pas en reste avec une présentation détaillée de SELinux, de ce qu'il apporte et du prix qu'il en coûte. Dans le même domaine d'application, UML fait également l'objet d'un article relativement poussé.

Mais j'en ai déjà trop dit, je vous laisse en compagnie des auteurs du présent magazine en espérant que vous prendrez autant de plaisir que moi à le lire...

Sommaire

14 4 QUESTIONS À ...
Ingo Molnar

8 Autopsie du boot Linux

20 Découvrir UML, ou comment mettre des Linux dans son Linux

26 Le noyau et le réseau : Comment repousser les limites de la connectivité

40 Développez vos pilotes de périphériques USB

46 Débogage du noyau Linux avec kGDB

58 Linux Security Modules

68 SELinux, l'agence de sécurité du noyau

La rédaction n'est pas responsable des textes, illustrations et photos qui lui sont communiqués par leurs auteurs. La reproduction totale ou partielle des articles publiés dans Linux Magazine France est interdite sans accord écrit de la société Diamond Editions. Sauf accord particulier, les manuscrits, photos et dessins adressés à Linux Magazine France, publiés ou non, ne sont ni rendus, ni renvoyés. Les indications de prix et d'adresses figurant dans les pages rédactionnelles sont donnés à titre d'information, sans aucun but publicitaire.

PRINTED IN Germany/ Imprimé en Allemagne / Dépôt légal: 3e Trimestre 1998 / **N° ISSN :** 1291-78 34 / **Commission Paritaire :** 09 08 K78 976 / **Périodicité :** Mensuel / **Prix de vente :** 5,95 Euros.

ont participé à ce N° :

- Frédéric RAYNAL
- Philippe BIONDI
- Frédéric Combeau
- Samuel Dralet
- Thomas Meurisse
- Michaël Hervieux
- Xavier Montrichard
- Stelian Pop
- Mathieu Blanc
- Laurent Oudot

Processus virtuel sous UML

Processus noyau UML

Appel système

Exécution de l'appel système en mode utilisateur

Retour

Hors Série

Linux Magazine France

est édité par Diamond Editions
B.P. 121 - 67603 Sélestat Cedex
Tél. : 03 88 58 02 08
Fax : 03 88 58 02 09
E-mail : lecteurs@linuxmag-france.org
Service commercial :
abo@linuxmag-france.org
Site : www.linuxmag-france.org

Directeur de publication :
Arnaud Metzler

Rédacteur en chef :
Denis Bodor

Secrétaire de rédaction :
Carole Durocher

Conception graphique :
Franck Toussaint - Katia Paquet

Impression :
VPM DRUCK
www.vpm-druck.de
Printed in Germany/
Imprimé en Allemagne

Responsable publicité :
Véronique Wilhelm
Tél. : 03 88 58 02 08

Distribution France :
(uniquement pour les dépositaires de presse)
MLP Réassort :
Plate-forme de
Saint-Barthélemy-d'Anjou.
Tél. : 02 41 27 53 12
Plate-forme de
Saint-Quentin-Fallavier.
Tél. : 04 74 82 63 04
Service des ventes : Distri-médias :
Tél. : 05 61 72 76 24

Distribution Belgique :
Tondeur Diffusion Avenue
Van Kalken, 9
1070 Bruxelles
Press@tondeur.be

Service abonnement :
Tél. : 03 88 58 02 08

4 Questions à

Ingo Molnar

Pourquoi passer tant de temps (libre ?) sur le noyau ?

Parce que j'y prends beaucoup de plaisir, et que c'est le plus grand projet technologique de l'humanité.

Quel(s) conseil(s) adresser à un débutant voulant s'initier au monde du noyau Linux ?

La réponse courte est de récupérer un livre sur les mécanismes internes du noyau, et de le lire. La réponse plus longue est de prendre un livre, le lire, faire des comparaisons avec ce qui est dans les sources, se documenter sur le Web (par exemple avec www.kernelnewbies.org), et de contacter d'autres développeurs. Et écrire du code aussi ! :-)

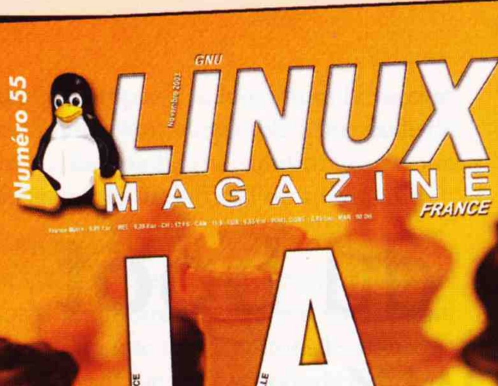
Site : www.kernelnewbies.org

Quelles sont, selon vous, les trois meilleures killer features du noyau 2.6 ?

De mon point de vue, c'est la réécriture de la couche IO, <pub>NTPL (les nouveaux threads) et le nouvel ordonnanceur :-)</pub> Mais c'est totalement injuste de sélectionner seulement trois fonctionnalités : le noyau 2.6 ne serait certainement pas moins intéressant sans ces trois fonctionnalités.

Qu'aimeriez-vous voir apparaître dans le noyau 2.8 ? Et dans le 3.0 ?

Pour le noyau 2.8, j'aimerais voir encore plus de technologies pour les petits et les énormes systèmes. Pour le 3.0, j'aimerais voir des contributions significatives au code en provenance des restes de Microsoft :-)



Interview :
- Gérard Onesta nous donne son point de vue sur les " brevets logiciels "

Dossier :
- L'intelligence artificielle des jeux de stratégie classiques
- Créer un jeu d'échecs en C/GTK+

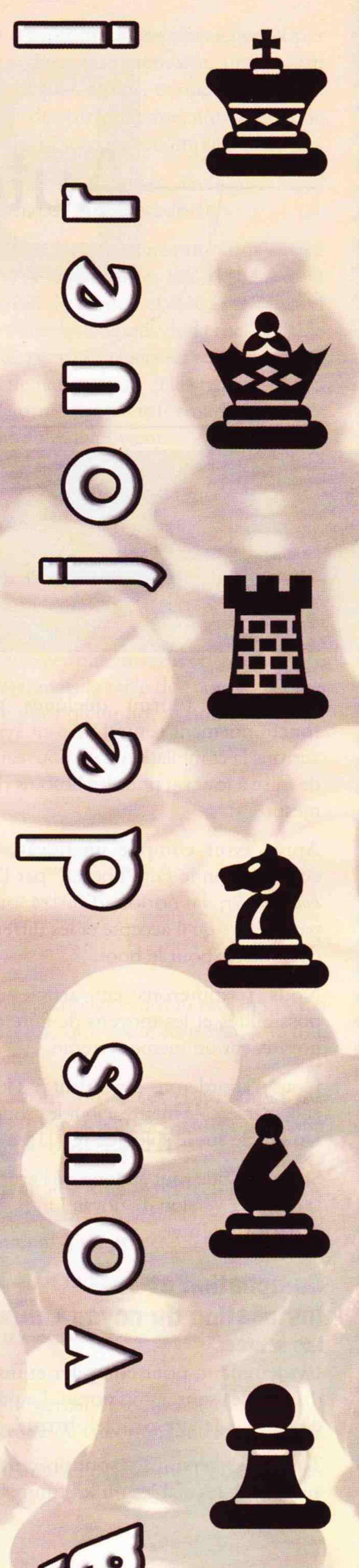
Systeme :
- x86-64 : L'arme 64 bits d'AMD
- Utilisez Windows pour "booter" Linux

Serveur :
- Autoconfiguration des adresses IPv6
- Créez votre module de vote en PHP

Développement :
- Découvrez la programmation fonctionnelle avec Erlang
- Développez des applications PostgreSQL en C
- Réalisez une "vraie" application Qt
- Lancez des processus à partir de Perl
- Rappels & glossaire de g++

Sommaire

En kiosque



Autopsie du **boot** Linux

Le noyau Linux est le système de base de toute distribution Linux. Il permet la gestion du matériel, de la mémoire, des entrées/sorties, la distribution du CPU pour l'exécution des processus et bien d'autres choses encore. Mais il reste souvent le côté obscur du système, ce qu'il ne faut surtout pas toucher, sous peine de ne plus rien voir fonctionner.

Cet article fournit quelques précisions sur le fonctionnement central de tout système Linux. Nous verrons la compilation d'un nouveau noyau pour les cas de mise à jour ou pour les besoins de noyaux taillés sur mesure.

Après avoir compilé un noyau, nous montrerons comment on le fait "booter" par l'intermédiaire d'un *boot loader*, la notion d'`initrd` ou de *ramdisks*, les paramètres qu'il accepte et les différents périphériques disponibles pour le boot.

Nous terminerons cet article en examinant les possibilités et les moyens de faire cohabiter plusieurs noyaux sur un même système.

Nous n'aborderons cependant pas le système des *patches* et la manière de mettre à jour les sources de votre noyau. Un article sur le sujet (cf réf. [1]) a déjà été écrit.

Signalons que tout au long de l'article, nous utiliserons la dernière version du noyau Linux, la version 2.4.22 sur plate-forme Intel.

Compilation et installation du noyau Linux

Les sources

Évidemment, pour compiler et installer, il nous faut d'abord les sources du noyau. La première solution est d'aller sur le site officiel, **http://www.kernel.org**.

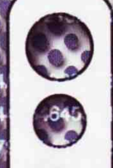
Toutes les versions y sont présentes. Que ce soit les anciennes, les stables ou les futures versions du noyau (branche 2.6), vous les trouverez toutes sur ce site (cf réf. [2] pour le système de numérotage des versions de noyaux).

Actuellement, nous sommes à la version stable 2.4.22. Voici donc comment récupérer les sources et vérifier l'intégrité de l'archive récupérée (cf. **http://www.kernel.org/signature.html**) :

```
# pwd
/usr/src
# wget http://www.kernel.org/pub/linux/kernel/v2.4/linux-2.4.22.tar.bz2
[...]
# wget http://www.kernel.org/pub/linux/kernel/v2.4/linux-
2.4.22.tar.bz2.sign
[...]
# gpg – keyserver wwwkeys.pgp.net –recv-keys 0x517D0F0E
gpg: requesting key 517D0F0E from wwwkeys.pgp.net ...
gpg: key 517D0F0E: public key imported
gpg: Total number processed: 1
gpg:               imported: 1
# gpg –verify linux-2.4.22.tar.bz2.sign linux-2.4.22.tar.bz2
gpg: Signature made Mon Aug 25 13:54:52 2003 CEST using DSA key ID
517D0F0E
gpg: Good signature from "Linux Kernel Archives Verification Key <ftpad-
min@kernel.org>"
Could not find a valid trust path to the key.  Let's see whether we
can assign some missing owner trust values.

No path leading to one of our keys found.

gpg: WARNING: This key is not certified with a trusted signature!
gpg:          There is no indication that the signature belongs to the
owner.
gpg: Fingerprint: C75D C40A 11D7 AF88 9981  ED5B C86B A06A 517D 0F0E
```

Ne faites pas attention au *warning*, `gpg` nous indique bien que c'est la bonne signature.

Moins universelle comme méthode, les distributions proposent des *packages* de sources de noyaux ou de noyaux pré-compilés. Le problème généralement est qu'il n'y a pas la dernière version.



Debian par exemple propose au maximum la version 2.4.18 (connue comme étant vulnérable à une faille de sécurité `ptrace()`). La meilleure solution est donc évidemment d'aller sur le site officiel ou sur un miroir.

La compilation du noyau Linux

Vous avez tout d'abord besoin d'être `root` et de vous positionner dans le répertoire `/usr/src` (par convention, c'est dans ce répertoire que nous mettons les sources) pour désarchiver le *tarball* que nous venons de récupérer.

Avec les anciennes versions de noyaux (versions 2.2.x et antérieures), l'installation des sources créait le répertoire `/usr/src/linux`. Il fallait alors prendre soin de le renommer par `/usr/src/linux-x.y.z` si vous décidiez d'installer de nouvelles sources.

Avec les nouvelles versions du noyau (version 2.4.x et plus), vous n'avez plus ce souci puisque le répertoire `/usr/src/linux-x.y.z` est créé à chaque installation.

A partir de maintenant et tout au long de l'article, nous ferons référence au répertoire `/usr/src/linux-x.y.z` par la variable `$SRC_LINUX`.

Procédons ensuite à la compilation des sources. La première chose à faire est d'utiliser la commande `make mrproper`. Elle permet d'effacer les anciens fichiers objets `.o`, les dépendances et le fichier de configuration si une configuration a déjà été faite précédemment ou si elle comporte des erreurs.

Au contraire, elle n'est pas utile si vous souhaitez le garder ou si aucune configuration n'est déjà présente. Vous pouvez savoir exactement ce que fait cette commande en éditant le fichier `Makefile` à la racine des sources du noyau. Il en est de même pour les commandes suivantes.

Il s'agit ensuite de configurer votre noyau avec la commande `make config`. Si vous trouvez l'aspect trop rudimentaire, vous pouvez utiliser les commandes `make menuconfig` qui offrent une interface Ncurses en mode console ou `make xconfig` avec une interface Tcl/Tk sous XFree.

La configuration consiste à choisir ce que supportera votre noyau, comme une carte graphique ou encore le système de fichiers NTFS. Nous ne détaillerons pas ici chaque option, une aide très bien faite est disponible pour chacune et nous vous conseillons de la lire avant d'activer ou désactiver une option.

Chacune de ces commandes génère le fichier `$SRC_LINUX/.config` que vous pouvez éditer à la main. Une dernière commande concernant la configuration est `make oldconfig` qui modifie votre fichier de configuration pour qu'il contienne les paramètres par défaut.

Toujours pendant la configuration, vous avez la possibilité de choisir une option en tant que module. Les avantages sont d'avoir un noyau plus léger, et par conséquent moins d'espace mémoire utilisé, un démarrage du système plus rapide, et une plus grande facilité de mise à jour de votre système, notamment de vos drivers sans avoir besoin de recompiler le noyau.

Une fois la configuration terminée, vous pouvez la vérifier avec la commande `make checkconfig`.

Après la configuration de votre noyau, vous devez construire les fichiers de dépendances `$SRC_LINUX/.hdepend` (à la racine des sources du kernel) et `$SRC_LINUX/.depend` (dans chaque répertoire des sources du kernel) à l'aide de la commande `make dep`. Ces fichiers servent d'une part à indiquer de quels fichiers `include .h` les fichiers objets `.o` ont besoin, et d'autre part les informations sur les symboles que les modules exportent.

Un petit nettoyage avec la commande `make clean` efface les binaires créés lors d'une précédente compilation. Cette étape est quasi indispensable.

Vient ensuite la compilation du noyau à l'aide de la commande `make bzImage` qui construit une image compressée du noyau.

Nous pouvons aussi créer une image non compressée à l'aide de `make zImage`, mais la taille des noyaux actuels est trop importante pour qu'un noyau non compressé soit supporté :

```
# make zImage
[...]
Root device is (3, 1)
Boot sector 512 bytes.
Setup is 4652 bytes.
System is 861 kB
System is too big. Try using bzImage or modules.
[...]
#
```

Une autre manière de compiler le noyau est d'utiliser la commande `make bzdisk`, qui copie le noyau sur une disquette directement.

Pour savoir ce que font toutes ces commandes qui servent à créer une image du noyau, éditez le fichier `$SRC_LINUX/arch/i386/boot/Makefile`.

La compilation terminée, vous trouverez l'image du noyau `bzImage` dans le répertoire `$SRC_LINUX/arch/i386/boot` (dans le cas évidemment d'un processeur i386).

La commande `file` nous indique la nature du fichier :

```
$ file bzImage
bzImage: Linux kernel x86 boot executable RO-rootFS,
root_dev=0x301, Normal VGA
```

Si vous avez configuré certains modules, il est nécessaire de les compiler à l'aide de la commande `make modules`.

Voilà pour la configuration et la compilation de votre noyau.

Le conseil que nous pouvons vous donner est d'abord d'éliminer au maximum ce dont vous n'avez pas besoin dans le noyau.

Cela l'allégera davantage. Ensuite les fonctionnalités nécessaires continuellement sont configurées nativement dans le noyau et non en tant que modules. Nos modules sont plutôt les composants peu utilisés.

L'installation du noyau Linux

Comme nous l'avons dit, nous partons du principe qu'il n'y a pas d'autres noyaux compilés sur la machine. Nous n'avons donc pas à prendre de précaution quant à la sauvegarde de l'ancien noyau et des anciens modules.

L'installation consiste d'abord à copier le fichier `bzImage` dans le répertoire `/boot` et à lui donner le nom que vous voulez.

Ensuite, il faut copier le fichier `$SRC_LINUX/System.map` dans le même répertoire que l'image du kernel. Ce fichier est le fichier de symboles du noyau, c'est-à-dire qu'il contient tous les symboles du noyau et leurs adresses respectives.

Un fichier manquant ou une mauvaise version de noyau peut vous rendre dans l'impossibilité de charger un module ou vous donner ce genre d'erreur :

```
$ ps fauxwwwwwww
Warning: /boot/System.map not parseable as a System.map
{seq_startplay} {sound_timer_init}
Warning: /System.map does not match kernel data.
[...]
$
```

Il est donc indispensable d'avoir la bonne version de ce fichier dans le répertoire `/boot`. Si par mégarde, vous avez perdu ou effacé ce fichier, vous pouvez le reconstruire à l'aide du script suivant :

```
#!/bin/sh

PATH_VMLINUX=$1

if [ -z "$PATH_VMLINUX" ];
then
    echo "[!] usage: ./create_system_map.sh [path of vmlinux].."
    exit 0
fi

nm $PATH_VMLINUX | grep -v '\(compiled\)\|\(\.o$\)\|\( [aU]
\)\|\(\.\.ng$\)\|\(LASH[RL]DI\)' | sort
> System.map
```

Le fichier `vmlinux` se trouve généralement à la racine des sources du noyau.

Ensuite, pour que votre système démarre sur le noyau fraîchement compilé, vous avez deux solutions. La première est d'utiliser les commandes `make zlilo` ou `make install` qui feront tout pour vous : installation du noyau et configuration du boot loader LILO.

Elles font même plus que cela. Regardez le fichier `$SRC_LINUX/arch/i386/boot/Makefile` pour en savoir davantage, mais nous ne sommes pas en fait partisans des procédures automatiques et nous préférons la deuxième solution qui est de tout faire manuellement. Vous verrez cela dans le chapitre suivant.

Si vous avez configuré des modules et que vous les avez compilés, il vous reste à les installer avec la commande `make modules_install`.

Les modules seront copiés dans le répertoire `/lib/modules/version_de_votre_noyau` (`/lib/modules/2.4.22` pour nous). Ce répertoire contient aussi le fichier `modules.dep` qui contient les dépendances entre modules. Il est recréé à chaque boot de la machine avec la commande `depmod -a`.

Le Boot du Kernel

Si vous avez bien suivi la section précédente, vous devriez avoir un noyau flambant neuf. C'est bien ! Mais un noyau que l'on boote et dont on se sert, c'est encore mieux.

Pour cela, nous allons tenter de découvrir les secrets du boot d'un noyau Linux. Nous verrons tout d'abord les boot loaders et aborderons ensuite la notion de `initrd` ou ramdisk.

Nous passerons ensuite aux paramètres qu'il est possible de passer au noyau, juste avant qu'il ne se lance. Nous terminerons par les différents supports qui permettent à un noyau de s'amorcer.

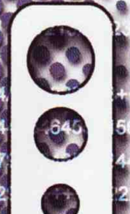
Boot loader ou le chargement de noyau en mémoire

Qu'est-ce qu'un boot loader ? C'est un programme qui est lancé sur les architectures i386, par le BIOS, pour charger un système d'exploitation. Lorsque le PC démarre, au niveau matériel (à la mise sous tension), le BIOS prend la main.

Il vérifie que tout fonctionne correctement, grâce à des tests (mémoire, CPU...), et fait l'inventaire des ressources disponibles (disques durs, lecteurs de CD-ROM, lecteurs de disquette...). Lorsque le BIOS a terminé, il donne la main à un lecteur de boot, tel que défini dans la configuration du BIOS.

Le plus souvent, c'est un disque dur. Mais il est possible de configurer un lecteur de disquette, de CD-ROM, un disque USB ou la carte réseau pour booter. Le BIOS définit une liste de périphériques pour le boot. Le BIOS essaiera les périphériques définis dans cette liste, dans l'ordre, jusqu'à ce qu'il en trouve un qui boote.

Selon le périphérique, la notion de boot est différente, mais pour un disque dur, il s'agit du MBR (*Master Boot Record*) qui contient un bout de code assurant le lancement d'un boot loader.



Si le MBR ne contient rien, mais qu'une partition est estampillée avec le *flag* "bootable", c'est cette partition qui porte le boot loader (c'est le cas, par exemple, des systèmes d'exploitation Windows).

Mais revenons à nos moutons, ou plutôt à notre boot loader. Le BIOS charge le MBR en mémoire et l'exécute. La première partie (ou stage 1) du boot loader est alors en mémoire et s'exécute.

C'est lui qui va ensuite aller chercher sur le disque la seconde partie (ou stage 2) du boot loader et charger sa configuration. Selon cette configuration, il permettra de charger en mémoire tel ou tel système d'exploitation (Windows, Linux, FreeBSD, OpenBSD, BeOS...). En résumé, le boot loader assure le lancement d'un système d'exploitation.

Sous Linux, deux boot loaders (entre autres) existent. Ils sont largement utilisés, aussi bien l'un que l'autre. Il s'agit de LILO (LInux LOader) et GRUB (Grand Unified Boot Loader). Il en existe d'autres, mais nous nous concentrerons sur ces deux-là.

LILO

Chronologiquement, LILO est plus ancien que GRUB, mais il est encore très souvent installé. Il se décompose en deux éléments : l'exécutable `/sbin/lilo` et le fichier de configuration `/etc/lilo.conf`.

Le but de cet article n'est pas de commenter toutes les fonctions possibles de LILO (le `man` devrait suffire pour cela), mais plutôt de présenter un fichier de configuration standard, et de l'expliquer, pour que le lecteur puisse acquérir les connaissances de bases pour aller, ensuite, plus loin. Cependant, les références suivantes ([4] et [5]) apporteront plus d'informations à ceux qui le souhaitent.

Prenons donc un fichier de configuration de base.

```
# LILO configuration file
# /etc/lilo.conf
# Start LILO global section
# Où doit-on installer LILO
boot = /dev/hda
# rapide, mais ne fonctionne pas partout
compact
# L'utilisateur est autorisé à entrer quelque chose
prompt
# Mais il n'a que 5 secondes pour cela
timeout = 50
# Configuration à lancer par défaut
default=Linux
# End LILO global section
# Linux bootable partition config begins
# Le noyau Linux à charger
image = /boot/vmlinuz
# La partition root de notre linux
  root = /dev/hda4
# Le nom LILO de cette configuration
  label = Linux
# Le noyau Linux à charger
image = /boot/vmlinuz
# Le fichier initrd à charger avec le noyau
  initrd = /boot/initrd.img
# La partition root de notre Linux
  root = /dev/hda4
# Le nom LILO de cette configuration
  label = Linux-initrd
# Linux bootable partition config ends
# Windows bootable partition config begins
# On désire booter sur cette partition
other = /dev/hda1
# Le nom LILO de cette configuration
label = WinXP
# Pour Windows, il suffit que la partition soit bootable
  table = /dev/hda
# Windows bootable partition config ends
```

Le fichier de configuration comporte deux sections : la section globale et la section des différents systèmes d'exploitation à booter. Dans la section globale, nous définissons où doit être installé LILO. Ici, il le sera sur `/dev/hda`, donc, dans le MBR. Ensuite, nous définissons que le mode de lecture du noyau sur le disque doit être compact.

Cela signifie que les requêtes de lecture sont regroupées pour que le chargement du noyau en mémoire soit plus rapide. Cette option est particulièrement utile sur les périphériques lents comme les lecteurs de disquettes. Elle augmente considérablement la vitesse de chargement, même sur un disque dur.

Par contre, sur ce dernier type de périphérique, cette option risque d'être incompatible avec le mode d'accès LBA32 (accès sur 32 bits pour les disques durs récents). Le mode LBA32 est le mode d'accès par défaut pour LILO. Si une incompatibilité est trouvée, pour un disque dur, la première chose à faire est de désactiver l'option `compact`.

Les deux paramètres suivants définissent le comportement de LILO vis-à-vis de l'utilisateur. `prompt` permet d'interagir avec LILO (pour choisir une configuration ou ajouter des paramètres à une configuration) et `timeout` définit un *timer* (en dixième de seconde) au bout duquel la configuration par défaut sera lancée. Le dernier paramètre de la section globale pointe sur la configuration par défaut à lancer, lorsque le timer est expiré.

Après la section de configuration globale, trois configurations sont définies. La première intègre le fichier du noyau Linux (image), la partition `root` de cette configuration (partition racine de la distribution) et le nom que nous donnons à cette configuration, qui apparaîtra au boot.

La seconde configuration est identique à la première, exceptée qu'elle intègre la définition d'un fichier `initrd`. Enfin, la dernière configuration définit un système Windows.

Elle détermine quelle partition doit être bootée, le label (le nom) de cette configuration, tel qu'il apparaîtra sur l'écran de LILO et quel disque dur contient la partition à booter.

Lorsque le fichier de configuration est terminé, il ne faut pas oublier d'installer LILO. Pour cela, il suffit de lancer la commande `lilo -v`. Le fichier de configuration par défaut est `/etc/lilo.conf` (le connecteur `-C` permet de spécifier un autre fichier). LILO lit le fichier de configuration et s'installe.

Le problème majeur de LILO est qu'il nécessite de le relancer à chaque fois que le fichier est modifié. Si, par exemple, vous modifiez une configuration ou ajoutez une nouvelle configuration de boot, il ne faut pas oublier de relancer LILO. Sinon, les modifications apportées au fichier de configuration ne seront pas prises en compte.

GRUB

GRUB est arrivé après LILO. Il ne souffre pas de la limitation énoncée à la fin de la partie consacrée à LILO. Avec GRUB, il est possible de modifier le fichier de configuration, sans avoir à relancer l'installation de GRUB. GRUB lit le fichier de configuration tel qu'il existe sur le système de fichiers, à la différence de LILO qui contient le fichier de configuration codé dans son installation sur le disque.

Pour GRUB, il suffit donc d'installer GRUB sur le MBR ou sur la partition voulue, une seule fois, puis de mettre à jour le fichier de configuration comme on le souhaite.

De plus, GRUB offre la possibilité de créer ou modifier, directement au boot, la configuration d'un système. Typiquement, si ce que l'on vient de définir dans le fichier de configuration ne marche pas, il est possible de le modifier.

Ainsi, on réussira à booter son Linux. Bien sûr, il ne faudra pas oublier de modifier le fichier de configuration, sinon au prochain reboot, on se retrouvera dans la même situation (la modification au boot d'une configuration ne met pas à jour le fichier de configuration).

Comme pour LILO, le but de cet article n'est pas de présenter toutes les options de GRUB, mais plutôt un exemple standard pour expliquer les concepts de base, et être opérationnel assez vite. Cependant, pour plus d'informations, la consultation des pages de `man` et d'`info` correspondantes, ainsi que [6], sera appréciable.

GRUB s'utilise en deux étapes. La première étape installe GRUB sur le MBR du disque principal (ou un autre endroit, selon sa configuration) en lui spécifiant le fichier de configuration à lire.

La seconde étape est la création du fichier de configuration qui sera lu par GRUB lors du démarrage.

L'installation de GRUB se fait de la manière suivante. Il faut lancer GRUB par la commande `grub`. On se retrouve sous GRUB dans ce qui ressemble à un shell. Il faut alors dire à GRUB de s'installer. Pour cela, on prendra la commande `install` de GRUB.

Nous allons définir ce qu'il faut installer et où. La syntaxe de la commande est la suivante :

```
install <stage1> <où_mettre_le_stage1> <stage2> p
<chemin_fichier_configuration>.
```

`<stage1>` est le fichier qui contient le code exécutable de la première partie du boot loader. Il s'agit de `/boot/grub/stage1`. Mais il faut spécifier à GRUB sur quel disque et quelle partition il doit trouver ce fichier. GRUB définit les disques par `hd0`, `hd1`, `hd2` qui correspondent sous Linux, respectivement, à `hda`, `hdb`, `hdc`.

Les partitions sont définies par un nombre, en commençant par 0. Donc `hda1` s'écrit, sous GRUB, `(hd0,0)` ; `hda2` s'écrit `(hd0,1)` ; `hdb3` s'écrit `(hd1,2)`. Si le fichier `/boot/grub/stage1` se trouve sur la partition `hda1`, on écrira `(hd0,0)/boot/grub/stage1`. Si ma partition `/boot` est une partition spécifique qui réside sur `hda1`, on écrira `(hd0,0)/grub/stage1`, car le répertoire `/boot` n'existe qu'à partir de la partition racine `/`. Le paramètre `<où_mettre_le_stage1>` définit un disque ou une partition où le `stage 1` doit être installé. Si l'on veut installer GRUB sur le MBR de `hda`, on écrira `(hd0)`. Le paramètre `<stage2>` définit le chemin d'accès (comme pour le `stage 1`) du `stage 2`. Par défaut, celui-ci se trouve dans `/boot/grub/stage2`. Dans notre exemple précédent, on écrira `(hd0,0)/boot/grub/stage2`.

Il ne nous reste plus qu'à spécifier le chemin du fichier de configuration GRUB, avec la même syntaxe que les deux premiers paramètres. Si le fichier de configuration se trouve sur `hda1` et qu'il se nomme `/boot/grub/menu.conf`, on écrira `(hd0,0)/boot/grub/menu.conf`. Pour résumer, la commande totale est donc :

```
grub> install (hd0,0)/boot/grub/stage1 (hd0)
(hd0,0)/boot/grub/stage2
p (hd0,0)/boot/grub/menu.conf
```

Après le lancement de cette commande, on sort de GRUB, et il ne reste plus qu'à construire le fichier de configuration.

Le fichier de configuration `/boot/grub/menu.conf` est composé de deux parties. La première partie comporte les paramètres globaux à GRUB, puis viennent les différentes configurations pour chaque système bootable.

Un fichier standard de configuration GRUB ressemble à ceci (à titre de comparaison, nous allons définir la même configuration que dans le paragraphe consacré à LILO).

```
# Timer fixé à 5 secondes
timeout 5
```

```
# Ce fichier est à afficher à l'écran de GRUB
i18n (hd0,3)/boot/grub/messages
# On définit une map clavier azerty
keytable (hd0,3)/boot/fr-latin1.klt
# La configuration par défaut est la première définie (Linux)
default 0
# Nom de la configuration
title Linux
# Définition du fichier noyau et
# de la partition racine
  kernel (hd0,3)/boot/vmlinuz root=/dev/hda4
# Nom de la configuration
title Linux-initrd
# Définition du fichier noyau et
#  de la partition racine et
#  du fichier de ramdisk
  kernel (hd0,3)/boot/vmlinuz root=/dev/hda4
initrd=/boot/initrd.img
# Nom de la configuration
title WinXP
# Partition contenant l'OS
  root (hd0,0)
# Cette partition est à activer
  makeactive
# Et à booter
  chainloader +1
```

Comme dans le fichier de configuration de LILO, le fichier de configuration de GRUB se compose de deux parties : une partie globale et une partie regroupant les configurations bootables. La partie globale définit le timer d'attente pendant lequel l'utilisateur a l'opportunité d'intervenir.

Ensuite, on trouve le fichier texte comportant le message à afficher lors du choix par l'utilisateur de la configuration à booter. Le *mapping* du clavier à prendre en compte est ensuite donné.

Comme nous avons un clavier français, on préférera un mapping Azerty français. Enfin, on a la configuration bootable par défaut, celle qui sera chargée si l'utilisateur n'intervient pas pendant le timer défini plus haut. Il s'agit de la configuration baptisée "Linux".

Après la configuration globale, on trouve les trois configurations définies dans le paragraphe LILO. La configuration "Linux" contient un nom, une référence au fichier du noyau et le paramètre passé au noyau concernant la partition root de cette distribution.

Dans la configuration "Linux-initrd", on reprend la configuration précédente, à laquelle on ajoute un fichier `initrd`. Pour la troisième et dernière configuration, appelée "WinXP", on trouve la partition à booter et à activer. Avec ce fichier, au reboot, l'utilisateur devrait voir une fenêtre contenant ces trois configurations.

Il lui est possible de se déplacer entre ces trois configurations (avec les flèches du clavier), pour choisir celle qu'il souhaite voir lancer.

Juste avant de terminer avec les boot loaders, nous souhaitons vous donner une mise en garde en ce qui concerne la sécurité. En effet, le boot loader est une partie critique de la sécurité d'un système. Dans le cas où un utilisateur est autorisé à modifier la configuration du chargement d'un système Linux, celui-ci peut être booté avec des paramètres n'assurant plus un niveau de sécurité acceptable.

Le cas le plus flagrant consiste à faire booter la machine sur un CD-ROM (sans modifier la configuration du BIOS) avec le noyau défini dans GRUB (le noyau sur le disque dur), mais en modifiant le paramètre `root`. Dans ce cas, on obtient un système Linux sur CD-ROM, contrôlé par l'utilisateur. Il ne lui reste plus qu'à monter la partition `root` du système Linux et modifier le mot de passe `root`. Il reboote la station et passe `root` sur le système grâce au mot de passe qu'il vient de modifier. Il faut donc verrouiller la configuration du boot loader d'une station, ce qui est possible, aussi bien avec LILO qu'avec GRUB, en définissant des mots de passe pour pouvoir modifier, voire même choisir une configuration à booter. Moins on donne de latitude à l'utilisateur lors du boot, plus la sécurité sera assurée. Ceci est également valable au niveau du BIOS et de sa possible modification par un utilisateur.

Initrd ou l'art des ramdisks

Derrière ce mot un peu barbare se cache un concept tout simple : comment faire exécuter quelque chose au noyau Linux avant qu'il lance le processus `init` ? Mais reprenons depuis le début : que se passe-t-il lorsque l'on boote sous Linux ?

Les différentes phases du boot d'un système Linux

Tout d'abord, le boot loader est chargé en mémoire (pour le boot loader, c'était le paragraphe précédent). Celui-ci permet de choisir un noyau à lancer et de lui passer des paramètres. Ceci fait, le noyau s'exécute, détecte le matériel, charge les différents supports matériels, charge les couches réseaux, etc. Lorsque tout est bon au niveau du matériel, il monte ensuite une partition (la partition `root` ou `/`), puis donne la main à `/sbin/init`. Si le noyau ne trouve pas de partition `/` ou de `init`, celui-ci nous insulte avec son célèbre cri, le fameux "kernel panic". C'est `/sbin/init` qui va terminer le processus de boot, en lançant les différents processus (les consoles virtuelles en mode texte), en configurant les adresses IP, en chargeant une configuration de pare-feu, en lançant les services réseaux (`http`, `nfs`, `inetd`...). `init` est le père de tous les autres processus et gère le démarrage et l'arrêt de la machine grâce aux *run levels* définis dans `/etc/inittab` et aux scripts dans `/etc/rc.d` et `/etc/init.d`.

Initrd et les phases de boot

Et où trouve-t-on `initrd` dans ce processus de boot ? On le trouve au moment où le noyau monte la partition

`root`. C'est à ce moment que le noyau prend en compte le paramètre `initrd` passé par le boot loader. Ce paramètre est `inirtd=<nom_de_fichier>`. S'il est présent, le noyau doit trouver, en mémoire, un fichier chargé par le boot loader (en même temps que le noyau lui-même). Pour le noyau, ce fichier est la partition racine qu'il essaye de monter. Il décompresse le fichier dans le *device* `/dev/ram0`. Puis, le noyau essaie de monter `/dev/ram0` comme partition.

`initrd` est donc un fichier qui contient un système de fichiers compressé et qui est monté par le noyau au boot. Cette partition un peu spéciale montée, le noyau cherche le fichier `/linuxrc`. S'il le trouve, il l'exécute. A la fin de l'exécution de `/linuxrc`, le noyau démonte la partition `initrd` et essaie de monter la partition `root` définie dans le boot loader comme un des paramètres du noyau. A partir de ce moment, on revient dans la procédure de boot du noyau, au niveau du montage de la partition `root` et du lancement de `/sbin/init`.

Si, dans la partition `root` (celle du disque dur), le répertoire `/initrd` existe, le système de fichiers contenu dans `/dev/ram0` (donc notre `initrd`) est monté sous ce répertoire. Le reste de la procédure de boot se déroule normalement.

Tout ceci est résumé dans la **figure 1**.

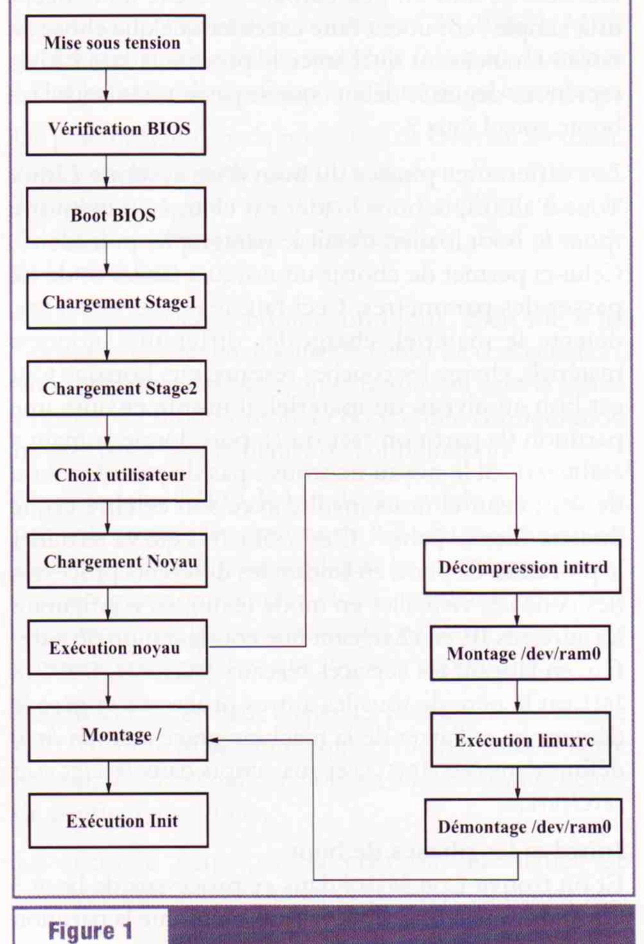


Figure 1

initrd et mkinitrd

`initrd` est l'abréviation de INITial RamDisk. Un fichier `initrd` est donc un système de fichiers stocké en RAM, monté par le noyau au boot et dans lequel ont été placés des outils (commandes, bibliothèques, fichiers de configuration...) afin de réaliser des actions spécifiques au démarrage du noyau.

Mais à quoi cela sert-il ? Considérons la commande `mkinitrd` (un `man mkinitrd` donnera toutes les informations, sinon la réf. [3] vous permettra de lire le `man` si la commande n'est pas installée sur votre machine). Cette commande crée un fichier `initrd` à mettre en mémoire lors du boot pour pouvoir charger des modules du noyau.

Mais prenons plutôt un exemple pour éclaircir les idées. Supposons que vous possédiez un PC sous Linux, avec un disque dur SCSI. Malheureusement, votre noyau ne supporte votre carte SCSI que s'il a chargé des modules nécessaires.

Cela signifie que votre noyau n'est pas capable de booter sur votre disque SCSI. Il ne pourra pas trouver votre disque pour monter la partition `root` et lancer `/sbin/init`. Alors comment faire ? `initrd` est votre ami. Il suffit de créer un système de fichiers qui contient les modules de votre noyau afin de supporter votre disque SCSI et de les charger dans le noyau.

Pour reprendre la procédure de boot du système, le noyau monte le fichier `initrd` en tant que `root`, le fichier `/linuxrc` est un script shell qui insère les modules du noyau dans le but de gérer la carte SCSI de votre système et le disque dur qui y est attaché. Le fichier `/linuxrc` est une liste de `insmod <nom_du_module>` pour insérer les modules du noyau (modules présents dans le système de fichiers `initrd`).

A la fin de l'exécution de `/linuxrc`, le noyau sait gérer votre disque SCSI grâce aux modules qui viennent d'être chargés. Il est alors capable de monter le système de fichiers racine présent sur votre disque dur SCSI et de continuer la procédure de démarrage sans problème.

Comment créer un ramdisk

Un fichier `initrd` est utile pour réaliser beaucoup d'autres choses. Par exemple, il est possible d'y stocker le système de fichiers nécessaire à l'installation d'une distribution Linux, ou pour toute configuration qui a besoin d'un système de fichiers résidant en mémoire.

Si on ne veut pas lancer `mkinitrd` pour créer un fichier `initrd`, comment faire ? Voici le principe de base pour créer un tel système de fichier.

Tout d'abord, il faut s'assurer que le noyau est capable de gérer les ramdisks. Sinon il faudra le recompiler en ajoutant cette gestion.

Lors de la configuration du noyau, il faut aller dans "Block devices" et mettre "Ram disk support" à "yes", définir la taille initiale d'un disque RAM (4096 octets par défaut) selon la taille du fichier `initrd` (lorsqu'il est décompressé) et ajouter le support `initrd` en mettant "Initial RAM disk (initrd) support" à "yes".

Lorsque l'on est sûr que le noyau gère le système de fichiers `initrd`, il faut créer un fichier qui contiendra le système de fichiers par la commande `dd if=/dev/zero of=<fichier> bs=512 count=<taille_du_fichier>`. Cette commande crée le fichier `<fichier>` dont la taille sera `<taille_du_fichier>`*512 octets.

Lorsque le fichier est créé, il faut créer un système de fichiers dans celui-ci grâce à la commande `mkfs -t ext2 <fichier>`. Cette commande crée un système de fichiers de type Ext2 dans le fichier `<fichier>`.

Ensuite, on monte le système de fichiers, fraîchement créé, dans un répertoire local avec la commande `mount -t ext2 -o loop <fichier> /mnt/initrd`. Cette commande monte le système de fichiers stocké dans `<fichier>`, de type Ext2, dans le répertorie `/mnt/initrd`.

A partir de là, il suffit de mettre dans le répertoire `/mnt/initrd` tous les outils, bibliothèques... que vous souhaitez voir dans votre `initrd`. Pour commencer, il faut recréer une arborescence standard avec `/bin`, `/sbin`, `/lib`, `/etc`, `/usr`...

Puis il faut y copier les différents composants systèmes nécessaires à ce que vous voulez faire. La commande `/usr/bin/ldd` est votre amie, puisqu'elle permet de trouver les bibliothèques nécessaires à un exécutable. Il ne faut pas oublier le fichier `/linuxrc` qui initiera toute la procédure.

Il suffit alors de démonter le système de fichiers (`umount /mnt/hd`) et de compresser le fichier par la commande `gzip -9 <fichier>`. Votre fichier `initrd <fichier>` est prêt à l'emploi.

Il faut ajouter une entrée dans le fichier de configuration `/etc/lilo.conf`, par exemple, en spécifiant la ligne `initrd=<fichier>` pour le bloc d'un noyau (ne pas oublier de lancer `lilo -v`, pour prendre en compte vos modifications). Et voilà, au prochain démarrage, votre noyau bootera avec un fichier `initrd`.

Paramètres du noyau

Le noyau Linux comporte des paramètres. Ces paramètres sont passés par le boot loader lors du chargement du noyau. C'est au moment de l'exécution du noyau que celui-ci regarde ces paramètres et modifie son comportement selon eux.

Il suffit de regarder les logs du noyau lors du boot (par la commande `/bin/dmesg` par exemple) pour vérifier la liste des paramètres. On aura une ligne comme celle-ci : `Kernel command line: BOOT_IMAGE=Linux ro root=306`.

On y trouve le nom de la configuration du boot loader (Linux), la définition de la partition racine (`root=306`) et le fait que cette partition doit être montée en lecture seule (c'est la procédure de démarrage gérée par `init` qui la remontera en lecture/écriture).

La liste des paramètres qu'il est possible de passer au noyau est assez grande et est définie dans le fichier `/usr/src/linux/Documentation/kernel-parameters.txt`. Les paramètres intéressants sont les suivants (cette liste n'est pas exhaustive) :

- `hdc=ide-scsi` : définit que le périphérique IDE `hdc` doit être traité comme un périphérique SCSI (permet l'utilisation de `cdrecord` sur un graveur IDE) ;
- `init` : donne l'exécutable à lancer après avoir monté la partition racine ;
- `initrd` : définit le fichier contenant le système de fichiers ramdisk ;
- `mem` : donne la taille de la mémoire physique lorsque le noyau n'est pas capable de la détecter entièrement ;
- `nfsroot` : donne le système de fichiers NFS root que le noyau doit monter avant de lancer `init` ;
- `noinitrd` : définit qu'il ne faut pas prendre en compte de ramdisk, même si un ramdisk a été fourni ;
- `panic` : définit ce que doit faire le noyau en cas de kernel panic ;
- `ramdisk_size` : donne la taille d'un ramdisk ;
- `ro` : définit que la partition racine doit être montée en lecture seule ;
- `root` : donne la partition racine à monter ;
- `rootflags` : définit les paramètres pour monter la partition racine ;
- `rootfstype` : donne le type (Ext3, Ext2, Minix...) de partition racine ;
- `rw` : définit que la partition racine doit être montée en lecture/écriture ;
- `S` : définit que `init` doit être lancé en mode *single user* ;
- `vga` : donne le mode vidéo à configurer.

Les autres paramètres du noyau concernent principalement la configuration de *drivers* matériels tels que les cartes sonores, les cartes SCSI... D'autres paramètres concernent la gestion du bus PCI ou ISA et la configuration de carte ISA *plug and play*.

Mais comment passer des paramètres au noyau ? Il suffit de penser aux boot loaders. Pour LILO, le paramètre `append` (à insérer dans la configuration d'un système bootable) définit une chaîne de caractères qui sera passée au noyau. Cette chaîne de caractères sera considérée par le noyau comme une liste de paramètres. Pour GRUB, on rajoutera simplement la liste des paramètres à passer au noyau sur la ligne `kernel`.

Par exemple, la ligne suivante `kernel (hd0,3)/boot/vmlinuz root=/dev/hda4 ro initrd=/boot/initrd.img S vga=711` passera les paramètres `root`, `ro`, `initrd`, `S` et `vga` au noyau, directement.

Supports de Boot

Désormais, nous savons réaliser un ramdisk, configurer deux boot loaders et passer des paramètres au noyau. Et si on essayait de booter un Linux à partir d'autres choses qu'un disque dur !

Quels sont les supports physiques qui permettent de booter un système Linux ? En fait, il y en a beaucoup. Nous n'en prendrons que quelques-uns pour illustrer cet article : une disquette, un CD-ROM, une clé USB de type DiskOnKey et une carte réseau compatible PXE.

Booter depuis une disquette

Booter un système Linux depuis une disquette peut ne pas être très différent par rapport à un disque dur, excepté au niveau de l'espace disponible. Cependant, trois méthodes existent : la méthode classique du boot loader de type LILO ou GRUB, la méthode plus spécifique pour la disquette du SYSLINUX sur un système de fichiers FAT et enfin la méthode un peu brutale de la copie du noyau sur la disquette.

Il est possible de voir une disquette comme un disque dur de petite taille. Un boot loader pourra donc répondre à ce besoin. La configuration reprend ce que nous avons montré dans les paragraphes précédents. La disquette doit posséder un système de fichiers (Ext2, Minix, FAT...), sur lequel on copie les fichiers nécessaires au boot (noyau, `initrd`, messages...).

Cette disquette sera montée dans un répertoire. On crée un fichier de configuration, par exemple pour LILO. Ce fichier de configuration comportera une section globale spécifiant l'installation de LILO sur la disquette, puis un (ou des) configuration(s) de boot pointant sur les fichiers de la disquette. On installe LILO avec la commande `lilo -v` (ne pas oublier le `-C` pour définir le bon fichier de configuration). On vérifie avec les messages de sortie que tout est bon et on reboote sur cette disquette.

La deuxième méthode se sert de SYSLINUX. Des informations complémentaires sont disponibles sur le site Internet dédié [7]. SYSLINUX est un boot loader pour Linux, fonctionnant avec des systèmes de fichiers FAT.

L'intérêt de SYSLINUX réside dans le fait qu'il suffit de créer une disquette au format FAT, d'y installer SYSLINUX, de copier les fichiers de boot et de créer un fichier de configuration définissant les systèmes à booter. Ceci fait, la disquette devient un support de boot pour ces systèmes. La philosophie ressemble un peu à GRUB, sans la partie interactive. Par contre, la disquette reste une disquette FAT. De plus, les options de configuration de SYSLINUX sont assez nombreuses et permettent de faire ce que l'on veut.

La troisième et dernière méthode n'est disponible que si le noyau Linux à booter le permet. Cette méthode consiste à copier directement le noyau sur la disquette par une commande de type `cp bzImage /dev/fd0` (on notera que l'on n'a absolument pas monté la disquette dans un répertoire local).

Lors de la création du noyau, un message précise si le noyau n'est pas utilisable avec cette méthode. On trouve ce message à la fin de la compilation du noyau (par un `make bzImage`). Pour un noyau non compatible avec cette méthode, on aura (c'est l'avant-dernière ligne qui est intéressante) :

```
[...]
Root device is (3, 6)
Boot sector 512 bytes.
Setup is 4764 bytes.
System is 998 kB
warning: kernel is too big for standalone boot from floppy
make[1]: Leaving directory `/usr/src/linux-2.4.22/arch/i386/boot'
```

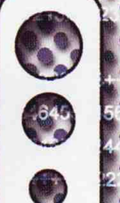
Si le `warning` n'est pas présent, le noyau est utilisable avec cette méthode. Il est impossible, par cette méthode, de définir des configurations particulières, des paramètres, un ramdisk. Elle servira seulement à booter un noyau, point final.

Les paramètres possibles sont ceux stockés dans le noyau, modifiables par la commande `rdev`. Cette méthode est peu paramétrable, mais sera utile pour tester rapidement si un noyau répond aux besoins.

Booter depuis un CD-ROM

Le boot depuis un CD-ROM est possible grâce à deux techniques différentes. La première technique reprend la notion de CD-ROM bootable de type El-Torito.

La seconde technique se sert d'un outil spécifique, nommé ISOLINUX, qui fait partie de la suite SYSLINUX dont nous venons de parler. Des informations complémentaires sur les CD-ROM El-Torito et ISOLINUX sont disponibles en [8], [9] et [10].



Un CD-ROM El-Torito est un CD bootable. Un CD-ROM bootable émule une disquette, dont le contenu est stocké dans un fichier. Deux fichiers spécifiques sont nécessaires, le fichier de boot et le fichier catalogue qui pointe sur le premier fichier. Le principe de création d'un CD-ROM bootable est simple. Il faut tout d'abord créer une disquette bootable (les trois méthodes définies dans le paragraphe précédent sont possibles). On transfère le contenu de cette disquette dans un fichier. Une commande de type `dd if=/dev/fd0 of=<fichier_boot>` est suffisante.

Ensuite, il ne reste plus qu'à créer une image de CD-ROM de type El-Torito. Pour cela, on lance la commande `mkisofs` avec les connecteurs `-b <fichier_boot> -c <fichier_catalogue>`. Le fichier de catalogue est créé par la commande `mkisofs`. Il ne reste plus qu'à graver l'image de CD-ROM avec la commande `cdrecord` et on obtient un CD-ROM El-Torito avec lequel on bootera sa machine.

La seconde technique impose que l'on se serve de ISOLINUX. Cet outil est un boot loader pour Linux sur architecture i386 qui sert à booter depuis des systèmes de fichiers ISO9660 (norme des CD-ROM), sans recourir au mode émulation. Il n'est plus nécessaire de devoir créer une disquette de boot, comme dans la technique précédente, et à la stocker dans un fichier d'émulation d'image disque.

En fait, l'exécutable de ISOLINUX remplace le fichier d'émulation d'image disque. Puis, c'est cet exécutable qui lira le fichier de configuration ISOLINUX contenu sur le CD-ROM et affichera les messages à l'écran pour que l'utilisateur puisse choisir la configuration à booter.

On utilise toujours les connecteurs `-b` et `-c` lors de la création de l'image du CD-ROM par `mkisofs`, mais, dans ce cas, ce sera toujours `-b isolinux/isolinux.bin`. Le fichier de configuration `isolinux.cfg` sera lu au boot du CD-ROM. Cette méthode est très répandue pour les distributions Linux, lors de l'installation par CD-ROM.

Booter depuis une clé USB

Pour les machines qui supportent le boot depuis un périphérique USB, le fait de booter depuis ce type de périphérique ou un autre ne change rien. L'accès à ce périphérique (que ce soit un disque dur, un lecteur de disquette, un lecteur de CD-ROM ou une clé USB) est transparent, la couche USB étant gérée par le BIOS.

Ceci signifie que si l'on boote depuis un lecteur de CD-ROM USB ou IDE, il n'y a aucune différence, et ce que nous venons de dire dans le paragraphe précédent, pour les CD-ROM, est toujours valable.

En revanche, la clé USB de type DiskOnKey est spécifique à cette interface. Y a-t-il des spécificités à connaître pour mettre en œuvre ce type de périphérique ?

La réponse est simple : non. Il n'y a aucune différence. Voici le principe dont nous nous sommes servis pour faire booter un portable IBM X31 sur une clé USB.

Nous avons connecté le périphérique sur un port USB. Ce périphérique a été reconnu comme un dispositif SCSI en tant que `/dev/sda`. Nous avons créé une partition et un système de fichiers sur cette partition (`fdisk /dev/sda et mkfs -t ext3 /dev/sda1`).

Nous avons monté cette partition (`mount -t ext3 /dev/sda1 /mnt/usb`) et copié les fichiers nécessaires sur cette partition. Nous avons créé une configuration LILO pour ce périphérique avec l'installation dans le MBR (`/dev/sda`), mais avec les paramètres suivants, dans la configuration globale :

```
disk=/dev/sda
bios=0x80
```

Lors du boot, le périphérique USB sera vu comme le premier disque de boot du BIOS (0x80), alors que ce n'est pas le cas actuellement, au moment de l'installation de LILO. On lance LILO (`lilo -v -C /mnt/usb/etc/lilo.conf`), on démonte le périphérique (`umount /mnt/usb`) et on reboote.

On configure dans le BIOS du portable que l'on souhaite booter sur un périphérique USB, et le portable boote sur le périphérique USB, afin, par exemple, d'installer une distribution Linux à partir de ce périphérique.

Booter depuis une carte réseau

De plus en plus de PC possèdent une carte réseau et parmi ces cartes réseau, de plus en plus sont compatibles avec la spécification PXE (*Pre-Execution Environment*) d'Intel. Mais qu'est-ce que c'est qu'une carte réseau PXE ?

C'est une carte réseau qui possède une ROM conforme à la spécification PXE afin de booter la machine par le réseau. Comment tout cela fonctionne-t-il ?

Le BIOS définit la carte réseau comme un périphérique de boot ; celle-ci émet une requête BOOTP sur le réseau. Un serveur BOOTP répond à cette requête (selon l'adresse MAC de la carte) en donnant les informations de configuration IP de la carte réseau (adresse IP, masque de réseau, passerelle, DNS...).

Le serveur BOOTP transmet aussi un nom de fichier à télécharger via le protocole TFTP. Lorsque le client a configuré sa carte réseau, il récupère le fichier via TFTP sur le serveur BOOTP. Ce fichier correspond à un stage 2 au niveau d'un boot loader.

Ce fichier fait partie du projet SYSLINUX, dont nous avons parlé dans les paragraphes sur la disquette et le CD-ROM, et plus particulièrement de PXELINUX ([11]). Le fichier se nomme `pxelinux.0` et il est placé dans le répertoire `/tftpboot` du serveur.

Dans ce répertoire, un répertoire doit être créé. Il s'agit de `pxelinux.cfg`. Il contient le fichier `default` définissant une configuration de boot, comme on a pu en voir avec LILO.

Lorsque `pxelinux.0` est chargé par le client, celui-ci charge le fichier de configuration de boot `/tftpboot/pxelinux/cfg/default`. Ce fichier contient le nom du fichier de noyau Linux, les paramètres de boot et, le cas échéant, un fichier `initrd`.

Selon cette configuration, le client télécharge, toujours par TFTP, ces fichiers (qui doivent se trouver dans le répertoire `/tftpboot` du serveur) et lance le noyau comme le ferait LILO. A partir de ce moment, la procédure de boot redevient la même que pour un amorçage à partir d'un disque dur.

Cette méthode de boot offre la possibilité d'installer des PC sous Linux sur un réseau à grande échelle. Il suffit de les bancher sur le réseau, de booter via la carte réseau et, l'installation et la configuration de la station peuvent être entièrement automatisées.

La station boot lance un script qui crée les partitions sur le disque dur et les systèmes de fichiers, installe les packages qu'elle récupère sur le réseau, configure la station, met en place un boot loader et reboote la station à partir du disque dur.

Comment faire cohabiter plusieurs versions de noyau

Nous avons vu jusqu'à présent la manière de compiler un noyau, de l'installer, de le faire booter sur divers supports physiques, etc. Le problème qui se pose maintenant est de pouvoir compiler et installer un nouveau noyau (dans le cas d'un test, ou parce que vous avez besoin d'une fonctionnalité bien spécifique), tout en gardant votre ancien noyau.

Il est même conseillé de garder l'ancien noyau au cas où le nouveau ne fonctionnerait pas au boot du système pour une raison quelconque.

Nous allons donc reprendre pratiquement chaque étape de la compilation et de l'installation, et nous verrons à chaque fois les précautions à prendre.

Au moment de la configuration

Deux cas de figure se présentent. Le premier est que vous modifiez quelques options dans la version actuelle du noyau.

Dans ce cas-là, nous vous conseillons de sauvegarder le fichier de configuration `.config` pour pouvoir éventuellement revenir en arrière.

Le second cas de figure est que vous vous voulez installer un nouveau noyau. Dans ce cas-là, aucune sauvegarde n'est à faire.

Un autre conseil est de configurer la variable `EXTRAVERSION` dans le fichier `$SRC_LINUX/Makefile` :

```
# cat /usr/src/linux-2.4.22/Makefile
VERSION = 2
PATCHLEVEL = 4
SUBLEVEL = 22
EXTRAVERSION =

KERNELRELEASE=$(VERSION).$(PATCHLEVEL).$(SUBLEVEL)$(EXTRAVERSION)
[...]
```

L'idée par exemple est de mettre comme valeur la date de compilation de votre noyau. Nous verrons plus tard pourquoi il est utile de configurer cette variable.

Au moment de l'installation

Rappelez-vous, l'installation consiste à copier les fichiers `$SRC_LINUX/System.map` et `$SRC_LINUX/arch/i386/bzImage` dans le répertoire `/boot` et à faire un `make modules_install` pour installer les modules compilés.

En ce qui concerne les deux fichiers à copier, il est évident qu'il faut les renommer pour chaque version différente de noyau.

Parfois, certains utilitaires ont besoin du fichier `/boot/System.map` ; le mieux est de créer un lien symbolique avec le fichier du noyau le plus utilisé.

Conclusion

Au terme de cet article (et plus généralement, des deux derniers hors séries de Linux Magazine France), nous espérons avoir quelque peu démystifié le noyau Linux, cette couche basse qui supporte tout le reste. La compilation du noyau, son lancement, son paramétrage, l'utilisation des ramdisks et des périphériques de boot ne devraient plus vous poser de problèmes.

Nous espérons également vous avoir montré que les capacités du noyau Linux, en termes de paramétrage et d'adaptation à des besoins spécifiques, sont assez incroyables. Il est, en effet, possible de se construire un noyau Linux sur mesure, pour la ou les fonction(s) dont on a besoin et seulement celle(s)-ci. De plus, il est possible de disposer d'un grand nombre de périphériques pour toute une palette de systèmes Linux différents. Bienvenue dans le monde de la "customization" de noyau Linux !!!

Voici à quoi peut alors ressembler votre répertoire /boot au final :

```
$ ls -alp /boot
[...]
lrwxrwxrwx    1 root    root         20 Sep 15 18:41 System.map -> System.map-2.2.22
-rw-r--r--    1 root    root     197819 Nov  3  2001 System.map-2.2.20
-rw-r--r--    1 root    root     202930 Sep 18  2002 System.map-2.2.22
lrwxrwxrwx    1 root    root         23 Sep 18  2002 bzImage -> /boot/bzImage-2.2.22
-rw-r--r--    1 root    root     618421 Nov  3  2001 bzImage-2.2.20
-rw-r--r--    1 root    root     634781 Sep 18  2002 bzImage-2.2.22
[...]
```

Il reste à installer les modules. C'est maintenant que la variable $EXTRAVERSION est utile. D'ordinaire, la commande make modules_install installe les modules dans le répertoire /lib/modules/x.y.z. C'est gênant si vous avez décidé de modifier certaines options de la même version du noyau, car ils seront copiés au même endroit que ceux de votre précédente installation.

Certes, vous pouvez faire une sauvegarde du répertoire avant. Mais à chaque fois que vous voudrez démarrer sur une version différente de noyau, il faudra auparavant faire une restauration du répertoire correspondant à la version de votre noyau.

Avouez que cela est assez contraignant. Par contre, le fait d'avoir configuré la variable $EXTRAVERSION fera que les modules seront copiés dans le répertoire /lib/modules/x.y.z$EXTRAVERSION, et le noyau aura pour nom bzImage-2.4.22$EXTRAVERSION si vous faites un uname -a (évidemment si le fichier /boot/bzImage-2.4.22 est présent).

Au démarrage du système, le noyau saura à quel endroit aller chercher les modules.

Au moment de la configuration du boot et de initrd

Gérer plusieurs noyaux avec les boot loaders est assez facile. Nous avons vu, que ce soit avec GRUB ou avec LILO, que leur configuration contenait une partie concernant le boot du système avec le nom d'image du noyau, la racine root de cette configuration, et le nom de la configuration.

Donc, si nous voulons gérer plusieurs noyaux et avoir le choix de choisir celui que nous voulons au démarrage du système, il suffit de faire un copier-coller de ces lignes et de modifier les paramètres nécessaires.

Nous relançons ensuite /sbin/lilo et le tour est joué. Voici un exemple concret avec le fichier lilo.conf :

```
# cat /etc/lilo.conf
[...]
image=/boot/bzImage-2.4.22-wlan
        label=main
        read-only

image=/boot/bzImage-2.4.22-sspax
        label=sspax
        read-only

image=/boot/bzImage-2.4.22-test
        label=test
        read-only
[...]
```

Si, pour chaque configuration, nous voulons aussi charger un fichier initrd, il suffit de lui donner un nom différent pour chaque version de noyau (il est préférable de toujours utiliser la convention nom_de_fichier-x.y.z) et d'inclure, comme nous avons vu dans le paragraphe précédent, la ligne adéquate dans les fichiers de configuration des boot loaders.

Frédéric Combeau
fred@rstack.org

Samuel "zorgon" Dralet
zorgon@miscmag.com

Bibliographie

[1] "Patcher les sources de son noyau" - Cédric Blancher, Linux Magazine HS Kernel Linux n°1

[2] "Linux: histoire d'un noyau" - Christophe Blaess, Linux Magazine HS Kernel Linux n°1

[3] http://www.rt.com/man/mkinitrd.8.html

[4] http://www.freenix.fr/unix/linux/HOWTO/mini/LILO.html

[5] http://gd.tuwien.ac.at/opsys/linux/lilo/Version21_docs/tech.pdf

[6] http://www.linuxgazette.com/issue64/kohli.html

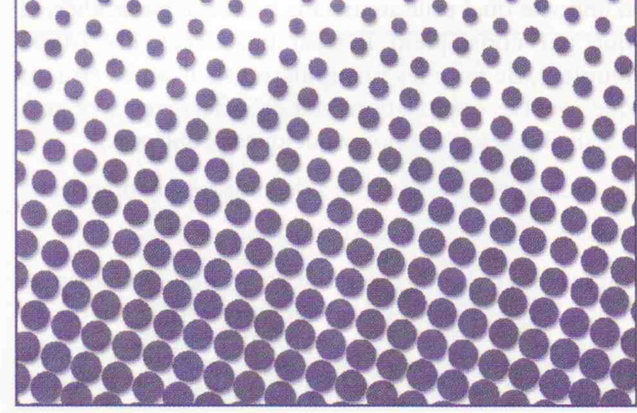
[7] http://syslinux.zytor.com/index.php

[8] http://www.phoenix.com/resources/specs-cdrom.pdf

[9] http://www.tldp.org/HOWTO/Bootdisk-HOWTO/cd-roms.html

[10] http://syslinux.zytor.com/iso.php

[11] http://syslinux.zytor.com/pxe.php



Découvrir UML, ou comment mettre des Linux dans son Linux

Il n'est pas rare de vouloir compiler et tester les derniers noyaux sortis. Toutefois, il est difficile de faire confiance d'emblée à de tels noyaux, et faire tourner sa machine dessus peut paraître hasardeux. En outre, recompiler et rebooter à chaque nouvelle modification peut vite devenir pénible. User Mode Linux (UML)[1] offre une alternative à tous ces problèmes.

UML est un projet libre qui permet de faire tourner un Linux dans une machine virtuelle depuis n'importe quelle distribution Linux. Dans les nouveaux noyaux 2.6, UML fait même partie intégrante des sources du noyau (`arch/um`). Nous allons donc présenter ici comment tester son nouveau noyau avec UML et quelles utilisations sont possibles.

Compiler son UML pour un noyau 2.6

Le noyau 2.6 étant encore en phase de tests, certains bogues persistent, notamment en ce qui concerne l'architecture UML. On se basera tout au long de cet article plus particulièrement sur un noyau 2.6-test3 et on verra les problèmes encore existants et comment y faire face.

Nul doute que lors de la sortie du noyau, ces problèmes seront résolus. On remarquera également que l'intérêt d'UML est également de pouvoir être utilisé par n'importe quel utilisateur sans privilège particulier, et que par conséquent il n'est pas nécessaire (donc inutile :), de faire les manipulations suivantes en tant que *root*. Les compilations ont été effectuées avec gcc-3.2 (le 3-3 semblant poser quelques problèmes).

La première étape consiste à décompresser son noyau :

```
tom@hote:tar -zxf linux-2.6.0-test3.tar.gz
```

En se plaçant dans le répertoire des sources, on remarque alors la présence du répertoire `arch/um` qui décrit l'architecture UML. Pour préciser donc que c'est à cette architecture qu'on s'intéresse, il faudra ajouter `ARCH=um` pour chaque étape de la compilation. Par exemple, on peut configurer les options de son noyau grâce à la commande :

```
tom@hote:make menuconfig ARCH=um
```

Il est recommandé d'appliquer le dernier *patch* UML disponible sur le site aux sources du *kernel*, celles-ci n'ayant pas l'air d'avoir pris en compte toutes les modifications. On applique alors ce patch grâce à la commande :

```
tom@hote:bzcat uml-patch-2.6.0-test3-1.bz2 | patch -p1
```

La partie réseau comporte encore un bogue dans le noyau 2.6-test3 qu'on va tout de suite corriger. Dans le fichier `arch/um/drivers/net_kern.c`, il suffit de commenter la ligne `dev-hard_header = uml_net_hard_header;` (ligne 339).

En effet, sans cette modification, les *headers* des paquets réseaux ne seront pas complets et il ne sera donc pas possible de communiquer avec d'autres machines.

Il ne nous reste alors plus qu'à compiler en sélectionnant les différentes options que l'on souhaite tester. Il faut cependant remarquer que certains composants ne marchent pas : le support pour les modules (`CONFIG_MODULES`), SCSI et MTD (`MTD_BLKMTD`). Il ne faut donc pas choisir ces composants, sinon le noyau ne pourra pas être compilé.

Sachant tout ça, il nous est maintenant possible de compiler notre noyau sans difficulté. Tout d'abord, on nettoie nos sources :

```
tom@hote:make mrproper
```

(si on oublie cette commande, le *linkage* final risque de ne pas marcher après plusieurs compilations). On prend les options par défaut :

```
tom@hote:make defconfig ARCH=um
```

Toutefois, certains des modules qui ne fonctionnent pas sont sélectionnés. Il suffit de les commenter dans le `.config` (`CONFIG_MODULES=y`, `CONFIG_SCSI=y`, `CONFIG_MTD`

_BLKMTD=m), ou de les modifier avec un make menuconfig ARCH=um.

La phase de compilation proprement dite peut alors être lancée grâce à la commande :

```
tom@hote:make linux ARCH=um
```

Après un petit moment d'attente, on obtient l'exécutable linux.

Lancement

Tout d'abord il faut récupérer un rootfs. Plusieurs sont disponibles sur le site de UML. Nous avons choisi Debian-3.0r0.ext2. Pour lancer UML, il suffit de taper :

```
tom@hote:./linux root=/dev/ubd/0 ubd0=Debian-3.0r0.ext2
devfs=mount
```

(on remarquera le root=/dev/ubd/0 qu'on peut remplacer par root=6200, obligatoire sur la ligne, contrairement aux versions précédentes, et qui évitera une erreur au lancement : Kernel panic: VFS: Unable to mount root fs on unknown-block(0,0)). Ça y est, on a enfin un UML qui fonctionne :)

```
tom@uml:uname -a
Linux uml 2.6.0-test3-1um #1 Fri Sep 26 18:35:56 CEST 2003 i686
unknown
```

Réseau

Pour lancer le réseau, il suffit d'ajouter à la ligne de lancement de l'UML : eth0=tuntap,,,192.168.0.2. Il est également nécessaire d'avoir les uml_utilities d'installées, et que le binaire uml_net se trouve bien dans le *path* et ait les bonnes permissions. Le réseau se configure ensuite comme sous n'importe quel Linux :

```
tom@uml:ifconfig eth0 192.168.0.3 up
```

Les 2 machines peuvent alors communiquer : (l'hôte a pour IP 192.168.0.4)

```
tom@uml:ping 192.168.0.4
PING 192.168.0.4 (192.168.0.4): 56 data bytes
64 bytes from 192.168.0.4: icmp_seq=0 ttl=64 time=0.9 ms
64 bytes from 192.168.0.4: icmp_seq=1 ttl=64 time=0.6 ms
```

Par défaut, l'hôte joue le rôle de *proxy arp* pour l'UML, mais on peut bien entendu ensuite le configurer pour qu'il serve de passerelle ou de *bridge*.

Utiliser UML

UML est un simple programme, exécutable par un simple utilisateur sans privilèges.

Une fois compilé, on a l'exécutable linux. Celui-ci possède plusieurs arguments possibles, dont voici quelques-uns :

- root : comme pour un vrai noyau, on indique sur quelle partition on va monter la racine. /dev/ubd0 correspond au majeur 98 (62 en Hexa) et au mineur 0 (00 en Hexa) ; donc on peut l'écrire root=6200 ;
- ubd0 : indique le *filesystem* à monter comme racine, si on a choisi /dev/ubd0 dans l'option précédente ;
- debug : pour déboguer le noyau UML ;
- devfs=mount : pour monter une partition virtuelle contenant des *devices* par défaut contenant par exemple /dev/ubd0 ;
- umid : on donne un nom à chaque système UML pour les différencier si on en élève plusieurs ;
- mode=tt : si le mode skas est présent sur l'hôte mais qu'on ne veut pas l'utiliser ;
- jail : pour activer la protection mémoire du mode jail ;
- honeypot : pour permettre à certains exploits de fonctionner et de faire croire qu'on serait bien sur un vrai noyau ;
- mem : pour indiquer la taille mémoire allouée au système UML ;
- eth0 : pour configurer un élément réseau, les arguments indiquent à quoi est relié sur l'hôte cet élément.

Comment ça marche ?

Le fonctionnement habituel d'un processus sous Linux se fait par l'intermédiaire du noyau, qui est le lien avec le matériel. L'idée avec User-Mode-Linux, c'est de placer le noyau en tant que simple programme. Les processus virtuels feront donc leurs appels systèmes vers ce programme émulant un noyau.

La différence avec un noyau habituel n'est pourtant pas très grande, puisque tout d'abord, on l'a vu, il se construit à partir des mêmes sources communes du noyau linux. La différence tient juste dans l'architecture. Avec UML on aura un matériel hardware émulé.

Une fois compilé et lancé, UML se comporte donc comme un simple programme, qui fonctionne comme un noyau, mais qui n'a pas un réel accès à l'hardware. Etant lancé en mode utilisateur, cela permettra que ce qui s'y passe à l'intérieur n'ait pas de répercussions sur le matériel.

Ainsi le noyau UML, au lieu de s'adresser directement au matériel, s'adresse au noyau réel comme un programme normal (**voir Figure 1**).

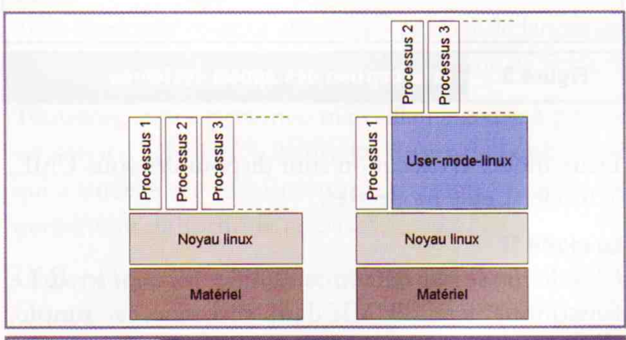


Figure 1 UML n'a pas de lien direct avec le matériel

Linux supporte la surveillance des "Threads" en redirigeant les appels systèmes. UML se sert donc de cette fonctionnalité. Ainsi les appels systèmes des processus sous UML sont surveillés pour être traités par le noyau UML (**voir Figure 2**). UML fonctionne donc avec IPC, pour la communication inter-processus et se sert donc de l'appel système `ptrace`.

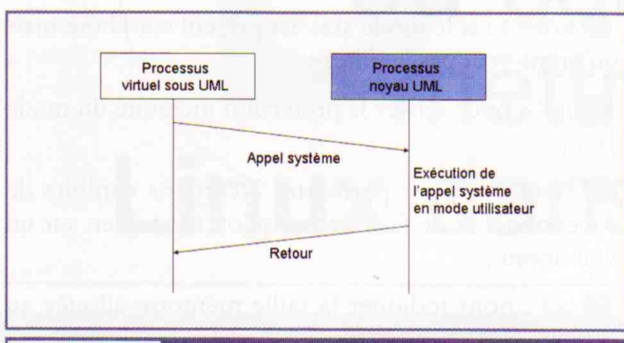


Figure 2 Interception des appels systèmes

ptrace

Ptrace fournit au processus parent un moyen de contrôler l'exécution d'un autre processus et d'éditer son image mémoire.

L'utilisation primordiale de cette fonction est l'implémentation de points d'arrêt pour le *debugging*. Un processus suivi se déroule jusqu'à l'arrivée d'un signal. Ensuite il s'interrompt, même si le signal est ignoré, et son père sera averti grâce à la fonction `wait`.

Il peut inspecter et modifier le processus fils pendant son arrêt. Le parent peut également faire continuer l'exécution de son fils. Quand le père a fini le suivi, il peut terminer le fils ou le faire continuer normalement.

Ainsi, UML peut prendre la place du vrai noyau.

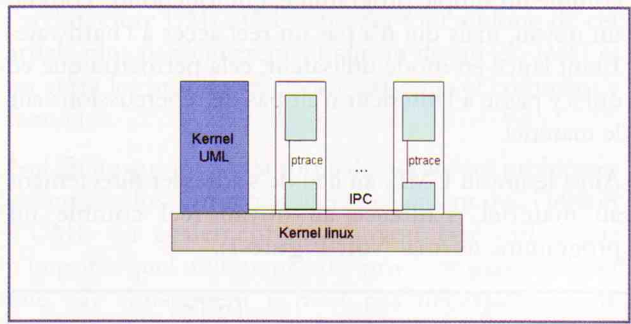


Figure 3 Interception des appels systèmes

Deux modes d'exécution sont disponibles sous UML, le mode `tt` et le mode `skas`.

Le mode tt

C'est le mode par défaut. `tt` signifie *Tracing Thread*. Le fonctionnement d'UML dans ce mode est simple, chaque processus virtuel sous UML a un processus correspondant sur l'hôte. Il y a alors une Thread spéciale qui permet de relier les appels systèmes au processus UML noyau. On peut lister les processus sur l'hôte :

```
kl@hote:ps -f -C linux
UID        PID  PPID  C STIME TTY          TIME CMD
kl         309   305  1 15:51 ttyp1    00:00:04 linux  [(tracing
thread)]
kl         314   309  0 15:51 ttyp1    00:00:00 linux  [(kernel
thread)]
kl         321     1  0 15:51 ttyp1    00:00:00 linux  [(kernel
thread)]
kl         323     1  0 15:51 ttyp1    00:00:00 linux  [(kernel
thread)]
kl         325     1  0 15:51 ttyp1    00:00:00 linux  [(kernel
thread)]
kl         327     1  0 15:51 ttyp1    00:00:00 linux  [(kernel
thread)]
kl         333     1  0 15:51 ttyp1    00:00:00 linux  [(Unknown)]
kl         647     1  0 15:51 ttyp1    00:00:00 linux  [(Unknown)]
...
```

Si on tue un de ces processus sur l'hôte, on peut s'attendre à un blocage du système UML. En effet, le processus noyau le surveillant attendra indéfiniment que celui-ci se manifeste.

Le noyau UML est présent dans l'espace adresse de chacun de ses processus qui sont accessibles en écriture.

C'est évidemment un problème de sécurité, puisque, avec l'accès en écriture aux données du kernel, un processus pourrait ainsi aller jusqu'à l'hôte.

Le mode `jail` (ce mode est détaillé plus loin dans l'article) permet de résoudre ce problème, mais dans le cadre d'un *honeypot*, cela est visible.

De plus, l'aller/retour du signal est lent dans le cadre du mode `tt`.

Le mode skas

`skas` signifie *Separate Kernel Address Space*.

Pour utiliser ce mode, il faut patcher **[5]** le noyau de l'hôte.

Ensuite, au lancement d'UML, celui-ci détectera automatiquement la présence de l'hôte patché ou non. S'il ne trouve pas le support `skas` de l'hôte, il passera en mode `tt`.

Le mode `skas` est plus sûr que le mode `tt` et il est aussi plus rapide. Pour le système UML vu de l'intérieur, il n'apporte aucun changement.

Pour l'hôte, les différences sont que désormais il n'y a plus que 4 processus en tout par UML. Il permet donc de moins polluer en présence de processus l'hôte, surtout si on place plusieurs systèmes UML dessus.

```
tom@hote:ps -f -C linux
UID        PID  PPID  C STIME TTY          TIME CMD
```

```
tom       1469   578 43 15:15 pts/1    00:00:02 linux [(Unknown)]
tom       1471  1469  5 15:15 pts/1    00:00:00 [linux]
tom       1477  1469  1 15:15 pts/1    00:00:00 linux [(Unknown)]
tom       1478  1469  0 15:15 pts/1    00:00:00 linux [(Unknown)]
```

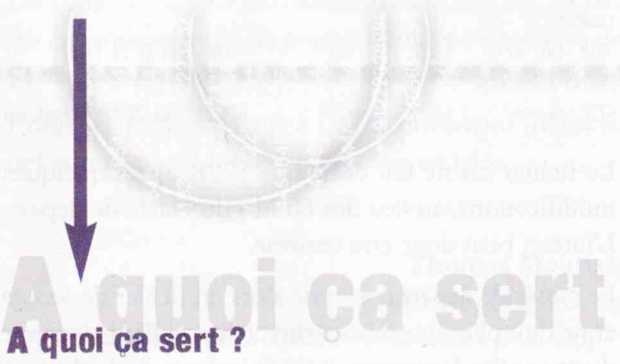
`kernel thread`, qui fonctionne dans l'espace adresse séparé du noyau, exécute le code du noyau et fait l'interception des appels systèmes des processus UML.

`userspace thread`, qui exécute tous les codes des processus UML, passe d'un contexte de processus UML à un autre.

`ubd driver asynchronous IO thread` est le driver entrée/sortie d'`ubd` pour la gestion du filesystem virtuel.

`write SIGIO emulation thread` gère l'émulation pour l'écriture des signaux d'entrée/sortie.

Enfin, le mode `skas` permet de diminuer la charge lors de l'exécution de plusieurs systèmes UML.



A quoi ça sert ?

Nous allons présenter ici différentes utilisations d'UML. Celles-ci sont diverses, mais elles reposent toutes sur le principe voulant donner le moindre privilège à une utilisation quelconque.

Faire des tests

Vous voulez tester une nouvelle version du noyau Linux, une version encore en test ou patchée, ou votre version du noyau modifiée par vos soins.

Vous voulez tester une distribution, voir le comportement après un `rm -R /`, créer un système prenant le moins de place, un noyau le plus petit possible...

Autant de challenges ou de problèmes que vous voulez résoudre, avec la possibilité de faire ces tests facilement et de façon plus agréable (i.e. pas besoin de *rebooter* après chaque test et chaque *kernel panic*) et sans craindre pour son filesystem ou son matériel.

UML permet aussi de tester facilement des processus dangereux dans un système isolé.

Debugging

Puisque UML représente un noyau Linux et qu'il se comporte comme un simple programme, eh bien, on va pouvoir déboguer un noyau comme un programme habituel.

Pour cela, rien de plus simple, il suffit d'ajouter au lancement d'UML l'option `debug`, ce qui permet d'avoir un accès à un terminal contenant `gdb`, qui pourra suivre le fonctionnement du noyau virtuel. Ainsi, quand on voudra déboguer un noyau, on pourra penser à utiliser UML.

Pour la version d'UML du noyau 2.6.0, le noyau est automatiquement prêt pour le débogage. Si vous utilisiez une ancienne version d'UML, veillez à activer l'option `CONFIG_DEBUGSYM` au moment de la compilation (elle permet de compiler le noyau avec l'option `-g` de `gcc`).

Il faut signaler qu'un mécanisme de débogage a été ajouté à UML, étant donné que les processus sont déjà "ptracés" et que `gdb` ne pourrait donc le faire (il faut bien penser à activer l'option `CONFIG_PT_PROXY` lors de la compilation du noyau UML pour permettre à `gdb` de pouvoir lui-même ptracer les processus).

Pour le mode `skas`, le débogage est sensiblement identique, hormis le fait que l'option `debug` n'est pas nécessaire. En effet, il suffit de lancer `gdb` sur le binaire d'UML.

```
kl@hote:gdb linux
```

Puis ensuite, on lance l'exécution et on inhibe la prise en compte des signaux `SIGSEGV` et `SIGUSR1` qui sont à l'usage propre d'UML.

```
(gdb) r root=6200 ubd0=/uml/FileSystem/Debian-3.0r0.ext2
devfs=mount

(gdb) handle SIGSEGV pass nostop noprint

(gdb) handle SIGUSR1 pass nostop noprint
```

Ensuite, le débogage se fait comme à l'habitude avec les commandes `att` (pour attacher un programme au débogage), `b` (pour déposer un point d'arrêt), `c` (pour continuer l'exécution), `n` (pour exécuter la ligne suivante)...

Jail

Tout le monde connaît `chroot`, qui permet de lancer un service dans une partie restreinte du système de fichiers.

Toutefois, si une personne malveillante arrive à pirater un service "chrooté", il aura les droits de l'utilisateur qui a lancé le service. En outre, différentes techniques permettent de sortir de ce `chroot`.

UML propose une solution intéressante à ce problème. Comme nous l'avons dit, UML peut être lancé par un utilisateur quelconque du système.

En le lançant par un utilisateur qui a un minimum de droits, on va ainsi limiter les risques, puisque même si un pirate arrive à devenir `root` sur notre UML, il ne sera finalement que l'utilisateur qui a lancé le processus sur la machine hôte.

En outre, UML propose une option sur sa ligne de commande : `jail`.

Ceci permet un meilleur contrôle de la mémoire, mais bien entendu ralentit l'exécution des processus. Pour utiliser cette option, il suffit de la préciser sur la ligne de commande.

Toutefois, cela nécessite que certaines options du noyau soient désactivées, ce qui semble logique du point de vue de la sécurité :

```
'jail' may not used used in a kernel with CONFIG_SMP enabled
'jail' may not used used in a kernel with CONFIG_HOSTFS enabled
'jail' may not used used in a kernel with CONFIG_MODULES enabled
```

Pour utiliser cette option, on recompile donc le kernel en enlevant ces options. Il semble également logique d'utiliser le mode `skas`, plus sûr, d'UML. Pour ce faire, il suffit juste d'appliquer le patch correspondant au noyau de l'hôte :

```
tom@hote:patch -pl <host-skas3.patch>
```

Une fois tout cela fait, on peut démarrer un `rootfs` dans une `jail`. Toutefois, pour faire une bonne prison, il est inutile de mettre des binaires qui ne servent à rien.

La meilleure solution consiste alors à se construire soi-même son propre `rootfs`. Le plus simple est d'utiliser les *loopdevices* (il faut avoir `CONFIG_BLK_DEV_LOOP=y` pour le noyau de l'hôte).

On crée et monte alors simplement le filesystem `jail_apachefs.ext2` (10 Mo) :

```
tom@hote:dd if=/dev/zero of=jail_apachefs.ext2 seek=$[10*1024]
count=1 bs=1k
tom@hote:losetup /dev/loop0 jail_apachefs.ext2
tom@hote:mkfs.ext2 /dev/loop0
tom@hote:mount -o loop /dev/loop0 /mnt/uml/
```

Il ne reste plus qu'à copier les fichiers nécessaires au lancement de notre Linux virtuel (un script de démarrage `/sbin/init`, les fichiers `/etc/passwd`, `/etc/groups`...), ainsi que les binaires et bibliothèques nécessaires (`ldd` le binaire pour connaitre les bibliothèques utiles) au lancement de l'application que l'on souhaite sécuriser. Un exemple de jail DNS est disponible sur le site d'UML.

Copy On Write

UML propose autre chose de très intéressant : *Copy On Write* ou Cow. En effet, il peut être intéressant de lancer plusieurs UML différents avec des filesystems très voisins.

Cow permet alors des gains de place importants. En effet, il est possible de lancer plusieurs UML simultanément et de travailler sur un filesystem identique, et de ne finalement sauvegarder que les modifications de chacun séparément.

Pour utiliser cette technique, il faut lancer l'UML en passant sur la ligne de commande `ubd0=fichier.cow,Debian-3.0r0.ext2`.

Le fichier `fichier.cow` sera alors créé et il contiendra les différentes modifications apportées. Lorsqu'on voudra le réutiliser, il suffira de passer sur la ligne de commande `ubd0=fichier.cow`, le chemin du `rootfs` de départ étant en fait écrit dans le fichier `cow`.

```
tom@hote:ls -lsh
 61M -rwxr-xr-x    1 tom      tom            60M 2003-09-26 17:37
Debian-3.0r0.ext2
572K -rw-r-r-    1 tom      tom            60M 2003-09-26 17:39 root.cow
```

Le fichier `cow` ne fait donc que 572K après quelques modifications, au lieu des 60 Mo du `rootfs` de départ. L'intérêt peut donc être énorme.

Le `rootfs` de départ constitue alors un fichier de *backup* et ne doit plus être relancé directement. Pour s'assurer de ne pas l'endommager, il paraît intéressant de changer les droits qui lui sont associés :

```
tom@hote:chmod 444 Debian-3.0r0.ext2
```

Il se peut toutefois qu'une imprudence nous oblige à modifier ou recopier ce fichier de backup. L'UML ne veut alors plus se lancer : `0mtime mismatch (1064590679 vs 1064591338) of COW header vs backing file`. Il suffit de corriger cela grâce à la commande :

```
mtime=1064590679 ; unset LC_ALL ;
touch -date="`date -d 1970-01-01\ UTC\ $mtime\ seconds`" Debian-
3.0r0.ext2
```

Il est également possible de réassembler le fichier `cow` et le fichier de backup pour faire une sauvegarde par exemple. Ceci se fait avec la commande (`uml_moo` fait partie des `uml_utilities`) :

```
tom@hote:uml_moo root.cow Debian-3.0r0.ext2.backup
```

On peut alors ensuite relancer ce `rootfs` comme un `rootfs` normal.

Pour finir

L'avantage est avant tout que si UML a un problème, le système principal est encore en pleine santé. On peut donc très bien l'utiliser comme honeypot **[6]** avec tous

les avantages de surveillance par l'hôte qu'il propose.

Les projets en cours

Aujourd'hui, les évolutions d'UML sont surtout au niveau du portage. D'une part, pour l'instant, UML permet d'émuler une machine de type i386 ; il pourrait être intéressant d'émuler pour différentes architectures.

Une autre direction concerne le portage au niveau de l'hôte. Il faudrait porter UML sur différents OS comme Unix ou même Windows.

D'autres projets concernent des améliorations d'`hostfs` (le filesystem permettant de remonter le / de l'hôte sous l'UML).

On pourrait aussi avoir un UML émulant un système multiprocesseur.

Un projet concerne la standardisation d'UML en tant que bibliothèque afin de permettre aux programmes qui y serait liés d'accéder à toutes les fonctionnalités du noyau.

Enfin, les projets relatifs à UML concernant toutes les utilisations possibles qu'on pourrait en faire.

Thomas Meurisse
tmeurisse@mailme.org,
jeune diplômé ENSEIRB à la recherche d'un emploi

Michaël Hervieux
michael.hervieux@free.fr

Références

[1] http://user-mode-linux.sourceforge.net

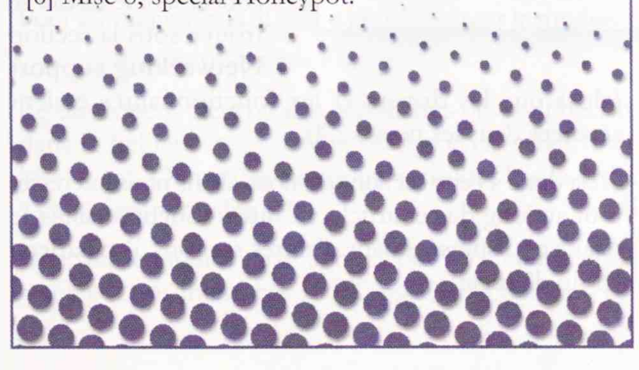
[2] http://usermodelinux.org

[3] http://hints.linuxfromscratch.co.uk/hints/uml.txt

[4] http://www.netwinder.org/~tpot/nano-linux.html

[5] http://user-mode-linux.sourceforge.net/dl-sf.html#Host%20patches

[6] Misc 8, spécial Honeypot.



GNU LINUX MAGAZINE FRANCE Hors-Série N°12
LE FIREWALL VOTRE MEILLEUR ENNEMI acte I
Introduction au Firewalling
Netfilter/iptables Le fonctionnement du pare-feu selon Linux
Pénétrations de réseaux et backdoors furtives
Firewalls et applications architecture et sécurisati...
GNU LINUX MAGAZINE FRANCE Hors-Série N°13
LE FIREWALL VOTRE MEILLEUR ENNEMI Acte II
Classification Outils
Architecture Limites
misc POWERED
Le magazine en Français 100% LINUX
2 Collectors
Linux Magazine Hors-Série 12
Linux Magazine Hors-Série 13
à commander sur notre site :
www.ed-diamond.com
misc POWERED
Le magazine 100% Sécurité Informatique

Le **noyau** et le **réseau** : Comment repousser les limites de la **connectivité**

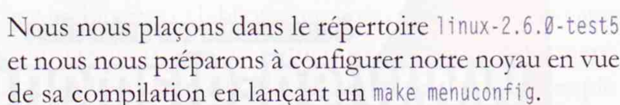
Parmi les domaines dans lesquels excelle le noyau Linux, il y a incontestablement le réseau. Qu'on regarde les matériels supportés via les drivers présents dans le noyau, les protocoles disponibles ou les nombreuses fonctionnalités, on ne peut que se rendre à une évidence : Linux est un des systèmes d'exploitation le plus interopérable du marché au niveau réseau.

Organisation des supports réseau

Tout au long de cet article, je ferai référence au menu de configuration d'un noyau de la série 2.6, le `2.6.0-test5`. Comme à l'accoutumée, l'arbre des sources se trouvera dans `/usr/src/linux-2.6.0-test5` avec un petit lien symbolique `linux` pointant vers ce répertoire.

On n'aura évidemment pas oublié de vérifier la signature de l'archive avant de commencer à travailler.

```
cbr@elendil:/usr/src$ ls -la
total 40520
drwxr-xr-x   3 root   src         4096 2003-09-28 18:09 ./
drwxrwxrwt   6 root   root        4096 2003-09-28 18:05 ../
lrwxrwxrwx   1 root   src           17 2003-09-28 18:05 linux -> linux-2.6.0-test5/
drwxr-xr-x  18 root   src         4096 2003-09-21 19:54 linux-2.6.0-test5/
-rw-r--r--   1 root   src     41430080 2003-09-08 22:33 linux-2.6.0-test5.tar.gz
-rw-r--r--   1 root   src          248 2003-09-08 22:33 linux-2.6.0-test5.tar.gz.sign
```



Nous nous plaçons dans le répertoire `linux-2.6.0-test5` et nous nous préparons à configurer notre noyau en vue de sa compilation en lançant un `make menuconfig`.

En cas de doute, n'hésitez pas à consulter l'aide et les fichiers de documentation fournis dans `/usr/src/linux/Documentation/networking/`.

Nous commencerons par activer le support des parties de codes expérimentales ou incomplètes (Code maturity level options, Prompt for development and/or incomplete code/drivers). De nombreuses fonctionnalités réseau dépendent en effet de cette option de configuration (`CONFIG_EXPERIMENTAL`), bien qu'elles soient suffisamment stables pour une utilisation sur des plates-formes de production.

Selon ses goûts, les supports pourront être compilés à même le noyau (`builtin`) ou en modules.

Nous ne discuterons pas ici des avantages et inconvénients de l'une ou l'autre de ces deux approches, mais rappellerons que l'utilisation des modules pour tester des fonctionnalités peut être une bonne idée.

Enfin, nous activerons le support Sysctl (configuration de paramètres noyau par modification de valeurs dans `/proc/sys/`, cf General setup, Sysctl support) puisque de très nombreux paramètres réseau sont ajustables de cette manière.

Dans les noyaux 2.6, l'ensemble des options concernant le réseau se trouve sous la section **Networking support** (alors que les drivers et les fonctionnalités étaient séparées dans les noyau 2.4).

C'est donc à cette section que nous allons nous intéresser tout au long de cet article. D'autres fonctionnalités se trouvent dispersées dans d'autres sections, j'en parlerai en fin d'article.

Nous activons donc le support réseau (`CONFIG_NET`) et entrons dans le vif du sujet, comme montré dans la figure 1.

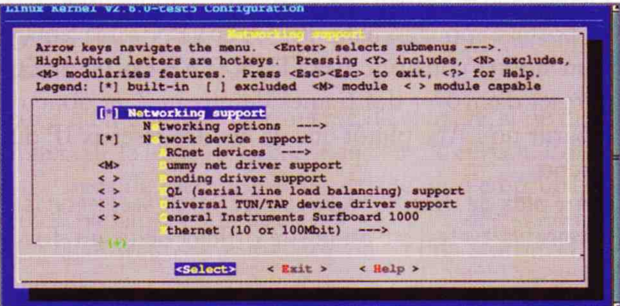


Figure 1 **Configuration du réseau**

Cet écran de configuration comprend deux sections majeures :

- **Networking options**, qui regroupe les fonctionnalités réseau ;
- **Networking device support**, en dessous de laquelle on trouvera les pilotes réseau.

L'article n'a pas pour objet la description exhaustive de toutes les options et sous-options de la section réseau. Pour chacune d'entre elles, je vous invite à lire l'aide accessible via le bouton <Help> en bas à droite de l'écran de configuration. Cependant, nous allons nous attarder sur certaines qui me semblent particulièrement intéressantes.

Les pilotes réseau

Nous trouvons ici tous les types d'interface réseau que peut supporter Linux, des désormais omniprésentes cartes Ethernet (10/100Mbps, 1Gbps ou 10Gbps) aux interfaces WAN en passant par les cartes Token Ring et les liens PPP. Nous allons nous focaliser sur certains d'entre eux.

Bonding driver support

Le pilote `bonding` permet de réaliser l'agrégat de plusieurs interfaces Ethernet (EtherChannel dans la littérature Cisco).

Ainsi, si vous disposez de trois interfaces Ethernet 100Mbps, vous pouvez les agréger en une seule interface logique disposant d'une bande passante de 300Mbps. Pour configurer cela, il faut d'abord charger le module :

```
cbr@elendil:~# modprobe bonding
```

Ce driver peut recevoir des paramètres qui sont décrits dans le fichier `/usr/src/linux/Documentation/networking/bonding.txt`. On pourra aussi aller consulter le site dédié à cette fonctionnalité [1].

Une fois le module chargé, nous pouvons commencer à agréger nos interfaces 100Mbps. Cette opération est réalisée par l'outil `ifenslave`, dont le source est distribué avec le noyau (`/usr/src/linux/Documentation/networking/ifenslave.c`).

```
cbr@elendil:~# ifconfig bond0 192.168.1.1 netmask
255.255.255.0 broadcast 192.168.1.255
cbr@elendil:~# ifconfig eth0 up
cbr@elendil:~# ifenslave bond0 eth0
cbr@elendil:~# ifconfig eth1 up
cbr@elendil:~# ifenslave bond0 eth1
cbr@elendil:~# ifconfig eth2 up
cbr@elendil:~# ifenslave bond0 eth2
```

Nous disposons maintenant d'une interface nommée `bond0` disposant d'une bande passante de 300Mbps. La distribution des trames sur les interfaces se fait par *round robbin* en utilisant comme adresse MAC l'adresse de la première interface ajoutée à l'agrégat (`eth0` dans cet exemple).

Bien évidemment, pour fonctionner au mieux, cette fonctionnalité suppose la présence à l'autre bout d'un équipement convenablement configuré. Sur un Cisco 2950 [2], nous aurons besoin d'une configuration de ce genre :

```
2950# configure terminal
2950(config)# interface port-channel 1
2950(config)# interface range fastethernet0/1 -3
2950(config-if)# channel-group 1 mode on
2950(config-if)# end
```

Enfin, nous pouvons consulter les information relatives à notre agrégat dans le répertoire `/proc/net/bond0` :

```
root@elendil:/proc/net/bond0# ls
info
root@elendil:/proc/net/bond0# cat info
Bonding Mode: load balancing
MII Status: up
MII Polling Interval (ms): 0
Up Delay (ms): 0
Down Delay (ms): 0

Slave Interface: eth2
MII Status: up
Link Failure Count: 0

Slave Interface: eth1
MII Status: up
Link Failure Count: 0

Slave Interface: eth0
MII Status: up
Link Failure Count: 0
```

Notons que ce driver supporte la tolérance aux pannes à l'aide d'un monitoring MII (paramètre `miimon`) et le *failover* sur un ou plusieurs *switches* (paramètre mode=1), ainsi que le protocole 802.3ad (*Dynamic link aggregation*). Je vous renvoie à la documentation pour de plus amples détails.

Universal TUN/TAP device driver

Ce driver permet la création d'une interface logique de type point à point (`tun0` ou `tap0`). Le trafic qui transite

par cette interface se retrouve en entrée ou sortie du fichier /dev/net/tun sous forme de paquets IP (TUN) ou des trames Ethernet (TAP). Ainsi, une application peut injecter du trafic dans le noyau pour le voir apparaître à travers l'interface associée ou récupérer les paquets émis par cette interface, comme illustré en figure 2.

D'un côté, l'application servant de base à l'interface doit ouvrir le périphérique :

```
#include <net/if.h>
#include <linux/if_tun.h>

int tun_alloc(char *dev)
{
        struct ifreq ifr;
        int fd, err;

        if ( (fd = open("/dev/net/tun",O_RDWR)) < 0)
                return tun_alloc_old(dev);

        memset(&ifr, 0, sizeof(ifr));

        ifr.ifr_flags = IFF_TUN;
        if ( *dev )
                strncpy(ifr.ifr_name, "dev", IFNAMSIZ);

        if( (err=ioctl(fd, TUNSETIFF, (void *)&ifr) < 0){
                close(fd);
                return err;
        }
        strcpy(dev, ifr.ifr_name);
        return fd;
}
```

Ensuite, elle écrira ses trames, au format désiré, selon la valeur du paramètre ifr.ifr_flags (IFF_TUN, IFF_TAP ou IFF_NO_PI). De l'autre côté, nous obtenons une interface de type point à point (tap0) ou de type Ethernet (tun0), que nous pouvons configurer comme une interface classique :

```
cbr@elendil:~# ifconfig tun0 192.168.1.1 netmask
255.255.255. broadcast 192.168.1.255
```

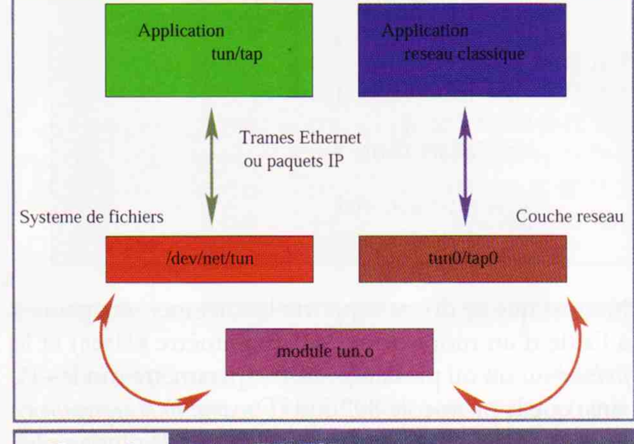


Figure 2 Driver TUN/TAP

Ce type de configuration est utilisé pour des simulations d'environnement réseau, à des fins de développement par exemple, ou pour des applications visant à mettre en place des tunnels non standards comme tun-proxy [3], vtun [4] ou encore OpenVPN [5]. Notons qu'un driver similaire existe pour Solaris, FreeBSD, et même pour les environnements Win32 (distribué avec le portage pour Win32 de OpenVPN). L'avantage d'un tunnel construit avec TUN/TAP est qu'il vous permet d'envoyer des trames de niveau 2, et donc de répartir un LAN, plutôt que de lier des réseaux IP différents.

Pour plus de détails, voir /usr/src/linux/Documentation/networking/tuntap.txt ou le site officiel dédié au driver [6].

PPP (point-to-point protocol)

Le protocole PPP est bien évidemment supporté par Linux, ainsi que les options courantes qui l'accompagnent comme la compression (de type Deflate ou BSD) et la gestion des liens synchrones (HDLC) et asynchrones. Le support du PPPoE est intégré au noyau, ce qui permet, moyennant l'utilisation d'un pppd [7] adéquat (v2.4.2 en CVS), de supporter ce type d'encapsulation au niveau noyau plutôt qu'en espace utilisateur avec un client comme le RoaringPenguin [8] et ainsi gagner en performances.

L'option **PPP filtering** permet quant à elle de configurer des filtres sur les trames PPP directement au niveau de pppd (cf la page man de pppd), permettant ainsi de choisir les trames pouvant circuler sur le lien, tant au niveau client qu'au niveau serveur.

Enfin, l'option **PPP multilink** permet d'agréger des liens PPP physiques (canaux B en RNIS par exemple) ou logiques en une seule interface. Ainsi, si vous montez un canal PPP sur IP et que vous disposez de plusieurs routes possibles pour atteindre l'autre partie, vous pouvez lancer un pppd avec l'option multilink par route et obtenir un agrégat de bande passante supérieure.

Comme pour le driver bonding vu précédemment, la distribution des trames se fait par *round robbin* entre les instances de pppd participant au lien agrégé. Ce mode de gestion a son importance. Si vous disposez en effet de deux liens de latence très différente, votre lien PPP global s'en trouvera fortement affecté, le lien lent ralentissant l'ensemble de l'agrégat.

Traffic Shaper

Le module shaper vous permet d'obtenir un outil de limitation de bande passante simple, mais limité. À vous de voir s'il pourra répondre à vos besoins, mais en ce qui me concerne, je vous recommande plutôt l'utilisation des options de qualité de service disponibles sous **QoS and/or fair queueing**.

Pour s'en servir, on utilise l'outil shapecfg (disponible en paquetage dans la plupart des distributions) :

```
cbr@elendil:~# modprobe shaper
cbr@elendil:~# shapecfg attach shaper0 eth0
```

```
cbr@elendil:~# shapecfg speed shaper0 64000
cbr@elendil:~# ifconfig shaper0 192.168.1.1 netmask
255.255.255. broadcast 192.168.1.255
```

Ensuite, il vous suffira de router le trafic à limiter à travers `shaper0`. Simple, rustique, mais moyennement efficace.

Parce qu'il utilise le mécanisme de routage, il ne peut limiter que le trafic sortant, et non le trafic entrant. Enfin, il n'y a pas de mécanisme de partage ou d'emprunt de bande passante comme c'est le cas avec des systèmes plus élaborés comme CBQ ou HTB, que je vous conseille.

Pour plus de détails, voir `/usr/src/linux/Documentation/networking/shaper.txt`.

Outils de diagnostic et d'optimisation

Deux outils vous seront probablement utiles pour contrôler l'état de vos interfaces et éventuellement en modifier des paramètres. Le premier s'appelle `mii-diag` [9]. Il sert à lire et modifier les registres MII [10] de votre interface (vitesse, duplex, type de médium).

```
root@elendil:~# mii-diag eth0
    Basic registers of MII PHY #1:  1000 786d 2000 5c10 01e1
    40a10005 2801.
    The autonegotiated capability is 00a0.
    The autonegotiated media type is 100baseTx.
    Basic mode control register 0x1000: Auto-negotiation enabled.
    You have link beat, and everything is working OK.
    Your link partner advertised 40a1: 100baseTx 10baseT.
    End of basic transceiver informaion.
```

Le second est `ethtool` [11]. Cet outil utilise l'IOCTL ETHTOOL qui permet, sur certains drivers (tous ne le supportent pas), d'obtenir des informations très complètes sur l'état de l'interface et de son driver, et d'interférer avec en modifiant ces valeurs.

```
root@elendil:~# ethtool -i eth1
    driver: 8139too
    version: 0.9.26
    firmware-version:
    bus-info: 07:00.0
    root@elendil:~# ethtool eth1
    Settings for eth0:
        Supported ports: [ TP MII ]
        Supported link modes:     10baseT/Half10baseT/Full
                                  100baseT/Half 100baseT/Full
        Supports auto-negotiation: Yes
        Advertised link modes:  10baseT/Half 10baseT/Full
                                100baseT/Half 100baseT/Full
        Advertised auto-negotiation: Yes
        Speed: 100Mb/s
        Duplex: Full
        Port: MII
        PHYAD: 32
        Transceiver: internal
        Auto-negotiation: on
        Supports Wake-on: pumbg
        Wake-on: d
        Current message level: 0xffffffff (-1)
        Link detected: yes
```

Un driver ne supportant pas ETHTOOL ne rendra pas d'information (ou une erreur) quand il sera interrogé.

```
root@elendil:~# ethtool -i eth0
    driver: 3c59x
    version: LK1.1.18-ac
    firmware-version:
    bus-info: 03:00.0
    root@elendil:~# ethtool eth0
    Settings for eth0:
    No data available
```

Les fonctionnalités réseau

Maintenant que nous avons pu voir les interfaces, physiques ou logiques, que pouvait supporter notre noyau, voyons ce que nous pouvons en faire, en nous focalisant sur le monde IP. Nous entrons donc dans la section **Networking options**. Là encore, je ne détaillerai pas toutes les options et vous renvoie donc aux descriptions accessibles via l'aide.

Options de base

Pour obtenir un noyau capable de réaliser un nombre suffisant d'opérations, nous allons activer certaines options :

- **Packet socket** : *sockets* de type `pf_packet` utilisées pour accéder directement aux périphériques réseau, dont se servent par exemple les bibliothèques de capture `libpcap` [9] ou de capture/injection `libnet` [10] ;
- **Packet socket: mmapped IO** : optimisation ;
- **Netlink device emulation** : option de compatibilité destinée à disparaître permettant aux applications utilisant `/dev/tap0` (ancien driver EtherTap) ou `/dev/route` (démon de routage dynamique) de fonctionner ;
- **Unix domain sockets** : les fameuses sockets de type Unix ;
- **TCP/IP networking** : on veut faire de l'IP, donc on active le support adéquat ;
- **IP Multicasting** : si vous voulez faire du multicast IP ;
- **Network packet filtering** : support de Netfilter[11].

Avec ceci, vous avez de quoi faire de l'IP dans de bonnes conditions. Nous allons maintenant explorer des fonctionnalités plus avancées pour transformer votre Linux en bête de réseau.

IP advanced router

Votre Linux est déjà capable de router des paquets IP. Il suffit pour cela d'activer l'IP forwarding via le Sysctl (et de vérifier que vos tables de routage et votre filtrage IP sont cohérents) :

```
cbr@elendil:~# echo 1 > /proc/sys/net/ipv4/ip_forward
```

Vous disposez alors d'un routeur simple, c'est-à-dire n'utilisant que l'adresse de destination de paquets pour choisir une route. En activant l'option "Advanced Router", nous allons pouvoir aller beaucoup plus loin et contrôler notre routage de manière très fine. Les options suivantes sont à notre disposition :

- **Policy routing** : cette option nous permet d'introduire de nouveaux critères de routage, comme le routage sur l'adresse source des paquets. En outre, nous voyons apparaître deux nouvelles options : **Use netfilter MARK value as routing key** et **fast network address translation**. La première est particulièrement intéressante. En marquant les paquets à l'aide de la cible `MARK` de Netfilter [11], nous sommes capables de modifier le routage de nos paquets. En clair, cela veut dire que tout élément reconnaissable par Netfilter peut nous servir de critère de routage. La seconde permet de mettre en place du NAT au niveau de la couche de routage. Cette traduction d'adresses est nettement moins souple que celle fournie dans Netfilter, mais se révèle particulièrement rapide. Ceux qui avaient l'habitude de faire du NAT (pas seulement du *masquerading*) avec les noyaux 2.2 ont déjà utilisé ce mécanisme.

- **Equal cost multipath** : cette option vous permet d'avoir à un même instant deux routes de même poids vers la même destination. Le choix de la route est fait de manière non déterministe (la décision dépend entre autres de l'état du cache de routage) parmi toutes les routes disponibles. C'est la base des configurations de partage de charge s'appuyant sur le routage.

- **Use TOS value as routing key** : le critère de routage est le champ TOS des paquets IP. Intéressant. On pourra noter que la valeur du champ TOS peut être fixée par Netfilter avec la cible TOS de la même manière qu'on marque les paquets. On peut ainsi transformer le champ TOS en marque exportable sur le réseau (ce qui n'est pas le cas des marques Netfilter).

- **Verbose route monitoring** : cette option permet d'obtenir des messages supplémentaires dans les journaux d'activité (via `klogd`) sur les problèmes de routage. Extrêmement utile pour déboguer un routage dynamique qui ne fonctionne pas.

L'utilisation d'un système Linux en routeur avancé nécessite l'installation du paquetage `iproute2` [12]. Je vous recommande chaudement la lecture (et la relecture) de la mine d'information qu'est le *Linux Advanced Routing and Traffic Control HOWTO* [13], plus communément appelé LARTC.

N'oubliez pas, sauf configuration particulière (routage asymétrique), d'activer le **Reverse Path Filtering** sur vos interfaces, c'est-à-dire le contrôle, par rapport à la table de routage, de l'interface d'arrivée de chaque paquet. Toute incohérence entraînera la destruction du paquet. C'est la base de l'*antispoofing*. Et comme vous voulez loguer ces échecs, activez la journalisation des "martiens" :

```
for f in /proc/sys/net/ipv4/conf/*/rp_filter; do
        echo 1 > $f
done
for m in /proc/sys/net/ipv4/conf/*/log_martians; do
        echo 1 > $m
done
```

Baladez-vous un peu dans `/proc/sys/net`. Vous verrez qu'on peut y agir sur de nombreuses fonctionnalités (ainsi que leurs paramètres éventuels) de votre noyau.

IP kernel level autoconfiguration

Cette fonctionnalité est intéressante pour les stations devant monter leur système de fichiers racine par le réseau (stations *diskless* en particulier). Le noyau de ces stations pourra ainsi obtenir sa configuration IP directement du réseau, en DHCP, BOOTP ou RARP, sans avoir à passer par un quelconque fichier de configuration ou autre client applicatif. Un *must have* pour "booter" sur le réseau.

IP tunneling et GRE tunnels over IP

Ces deux options permettent respectivement l'utilisation de tunnel IP sur IP (IPIP) et GRE (*Generic Routing Encapsulation*) sur IP. Ce sont des tunnels simples, sans chiffrement ni authentification forte, mais permettant de résoudre des problématiques comme les migrations de sites. Pour monter un tunnel, vous avez besoin du paquetage `iproute2` [12].

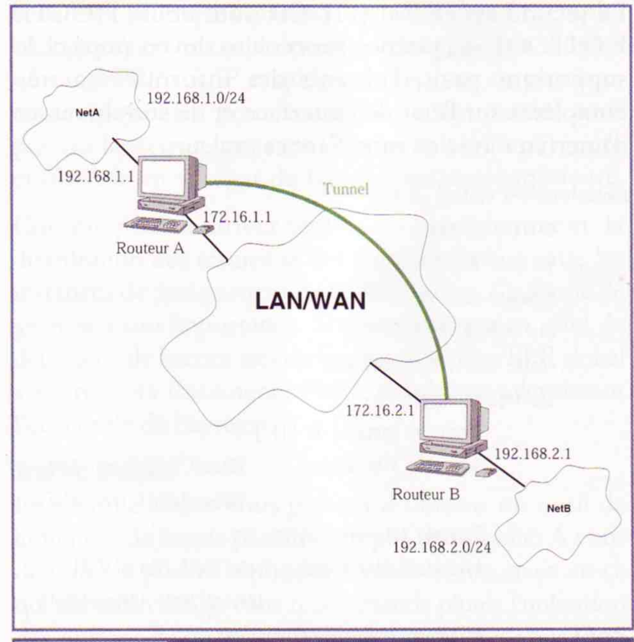


Figure 3 Mise en place d'un tunnel

Pour monter un tunnel IPIP, nous devons avant toute autre chose charger les modules concernés :

```
cbr@elendil:~# modprobe ipip
cbr@elendil:~# modprobe new_tunnel
```

À ce stade-là, nous disposons d'une interface `tunl0` (et une seule) que nous pouvons configurer. Supposons que nous disposons de la configuration décrite en figure 3. Nous voulons monter un tunnel reliant les réseaux NetA et NetB via les routeurs A et B. Sur le routeur A, nous configurons le tunnel ainsi :

```
cbr@routerA:~# ifconfig tunl0 10.0.0.1 pointopoint 172.16.2.1
cbr@routerA:~# route add -net 192.168.2.0/24 dev tunl0
```

Sur le routeur B, nous aurons :

```
cbr@routerB:~# ifconfig tunl0 10.0.0.2 pointopoint 172.16.1.1
cbr@routerB:~# route add -net 192.168.1.0/24 dev tunl0
```

Et c'est bon, on peut y aller. On doit évidemment activer le routage et mettre en place un filtrage cohérent avec l'utilisation du tunnel. Mais le mode IPIP est limité. En effet, comme vous avez pu le constater, nous n'avons qu'un tunnel sous la main.

Si nous voulons en mettre plusieurs en place, il faudra passer par un applicatif, s'appuyant par exemple sur le driver TUN/TAP. En outre, un tunnel IPIP, comme son nom l'indique, ne peut faire passer que de l'IP, seulement sur IP (M. de la Palisse ne m'aurait certainement pas contredit sur ce point) et uniquement à destination d'un autre système Linux, ce mode d'encapsulation lui étant (pour autant que je sache) spécifique. Malgré ces limitations, il n'en reste pas moins un mode de *tunneling* simple et efficace.

Pour des besoins d'interopérabilité ou pour encapsuler de l'IPv6, on s'orientera plutôt vers GRE, un standard initialement développé par Cisco et qu'on retrouve sur beaucoup de routeurs.

GRE a été conçu pour répondre à un maximum de besoins d'encapsulation. Il permet en fait d'encapsuler n'importe quel protocole de niveau 3 dans n'importe quel autre protocole de niveau 3. Puissant... Mais sous Linux, GRE se retrouve sur IP seulement. Reprenons l'architecture décrite en figure 3 et montons notre tunnel GRE. Sur A, nous exécutons :

```
cbr@routerA:~# modprobe ip_gre
cbr@routerA:~# ip tunnel add netb mode gre remote
172.16.2.1 local 172.16.1.1 ttl 255
cbr@routerA:~# ip link set netb up
cbr@routerA:~# ip addr add 10.0.0.1 dev netb
cbr@routerA:~# ip route add 192.168.2.0/24 dev netb
```

Et sur B :

```
cbr@routerA:~# modprobe ip_gre
cbr@routerA:~# ip tunnel add neta mode gre remote
172.16.1.1 local 172.16.2.1 ttl 255
cbr@routerA:~# ip link set neta up
cbr@routerA:~# ip addr add 10.0.0.2 dev neta
cbr@routerA:~# ip route add 192.168.1.0/24 dev neta
```

Et c'est parti. Notons que nous avons donné aux interfaces de tunneling des noms explicites (`neta` et `netb`). GRE permettant en effet de monter plusieurs tunnels, il est bon de savoir sur quel réseau vont déboucher les paquets envoyés par une interface rien qu'en regardant son nom.

GRE supporte aussi l'encapsulation IPv6. Dans ce cas, le tunnel doit être monté en mode SIT :

```
cbr@routerA:~# modprobe ip_gre
cbr@routerA:~# ip tunnel add netb6 mode sit remote
172.16.2.1 local 172.16.1.1 ttl 255
cbr@routerA:~# ip link set netb6 up
```

Ensuite, nous configurons l'interface `netb6` avec une adresse IPv6 et routons l'adressage IPv6 du réseau B à travers `netb6`. Enfantin. Si vous activer le support IPv6, vous aurez aussi la possibilité de faire des tunnels IPv6 sur IPv6 (`CONFIG_IPV6_TUNNEL`) conformément à la RFC 2473.

Le tunneling est donc une chose aisée à réaliser au niveau noyau sous Linux. Nous ne parlerons pas des implémentations en espace utilisateur tellement il y en a (nous en avons déjà vu trois s'appuyant sur le pilote TUN/TAP).

Il ressort que le mode de tunneling le plus souple est GRE et qu'on le préférera à IPIP, en particulier pour son interopérabilité. Sur un système aussi ouvert que Linux, il serait dommage d'utiliser des modes de communication qui lui sont spécifiques...

IP ARP daemon, TCP Explicit Congestion Notification et TCP syncookie

Trois options à manipuler avec précaution :

- **ARP daemon** : permet l'utilisation d'un démon ARP en espace utilisateur, `arpd`, via `/dev/arpd`. Pour les réseaux nécessitant beaucoup d'enregistrements ARP, le cache peut devenir très volumineux, consommant une grosse quantité de mémoire noyau, d'où l'idée de déléguer une partie du stockage des entrées à un applicatif. Dans ce cas, le cache ARP se trouve limité à 256 entrées, les suivantes étant gérées par le démon `arpd`. Pratique, mais si jamais le démon n'est pas lancé, votre cache ARP se trouvera *de facto* limité à une taille inférieure à sa taille habituelle !

- **TCP Explicit Congestion Notification** : support du TCP ECN, mécanisme permettant aux routeurs de signaler les congestions réseau aux utilisateurs dans le but de limiter les pertes de paquets induites, et donc améliorer les performances du réseau. De nombreux firewalls refusent les connexions de machines ayant cette fonctionnalité activée. Donc, en attendant que leurs administrateurs se soient mis à la page, on devra désactiver le mécanisme :

```
cbr@elendil:~# echo 0 >
/proc/sys/net/ipv4/tcp_ecn
```

- **TCP syncookie** : ce mécanisme apporte une protection contre les dénis de service de type SYN flood. En utilisant un mécanisme de challenge cryptographique connu sous le nom de SYN Cookies, un utilisateur

légitime peut accéder à un système "floodé" en outrepassant la mise en file (puisque la file est pleine) par un système de challenge/réponse sur les numéros de séquences initiaux (ISN) des deux protagonistes. La contrepartie étant que, comme une partie de l'ISN sert de challenge, vous réduisiez d'autant la force de votre générateur d'ISN et devenez plus vulnérables à des attaques nécessitant de deviner l'ISN fourni par le serveur. Pour plus d'information sur les SYN Cookies, voyez la page [14] de DJ Bernstein sur le sujet. En tout état de cause, ce mécanisme est désactivé par défaut, et je vous conseille de le laisser dans cet état, d'autant que Linux encaisse très bien ce type d'attaques. Si l'envie vous prend de l'activer :

```
cbr@elendil:~# echo 1 >
/proc/sys/net/ipv4/tcp_syncookies
```

IPSEC, IPv6 et IPVS

La grande nouveauté de la couche réseau des noyaux 2.6 est l'intégration du support IPSEC sous forme de transformations. Certains seront nostalgiques de la gestion par interfaces comme c'était le cas avec FreeS/WAN [15], mais c'est un grand pas en avant que d'avoir, enfin, une implémentation officielle et de qualité d'IPSEC dans le noyau. Le support IPSEC s'active via les deux options que sont **AH transformation**, **ESP transformation**.

En outre, nous pourrons activer la compression par le choix de **IPComp transformation**. La configuration de cette couche se fait à l'aide des outils IPSEC [16] portés du projet KAME qui utilisent des sockets spéciales, les `PF_KEY`, que nous devons donc activer avec l'option `PF_KEY sockets` au début de la section **Networking options**. Une interface de configuration spécifique à Linux est également disponible, pour peu que l'option **IPsec user configuration interface** soit donc activée (et nous l'activerons, évidemment).

Notons que les fonctions cryptographiques nécessaires à la couche IPSEC sont assurées par la CryptoAPI [17] qui est maintenant intégrée aux noyaux officiels. Lorsque nous activons le support IPSEC, la CryptoAPI est activée avec les modules HMAC, MD5, SHA1, DES/3DES (ainsi que le module Deflate si nous choisissons IPComp). Là encore, c'est une avancée certaine de ne plus avoir qu'une couche cryptographique au sein du noyau. Cela va grandement faciliter, par exemple, l'utilisation de périphériques d'accélération cryptographique.

La configuration de la couche IPSEC se fait avec l'outil `setkey` qui sert à définir les paramètres des associations de sécurité (SA) et des fichiers de configuration. Le noyau supporte les SA manuelles et les SA automatiques. Ces dernières peuvent se faire sur la base d'une clé partagée ou d'un certificat 509, par l'intermédiaire du démon IKE `racoon`. Pour plus de détails, voir le chapitre 7, *IPSEC: secure IP over the Internet*, du LARTC [13]. En l'espèce, je vais reprendre l'exemple de la figure 3 pour monter rapidement un tunnel IPSEC manuel. Il s'agit donc de faire communiquer NetA et NetB via nos deux passerelles A et B. Sur A, nous configurons le tunnel :

```
root@routeurA:~# setkey flush
root@routeurA:~# setkey spdflush
root@routeurA:~# setkey 172.16.1.1 172.16.2.1 esp 300
-m tunnel -E 3des-cbc "123456789012123456789012"
root@routeurA:~# setkey spadd 192.168.1.0/24
192.168.2.0/24 any -P out ipsec esp/tunnel/172.16.1.1
172.16.2.1/require
root@routeurA:~# setkey spadd 192.168.2.0/24
192.168.1.0/24 any -P in ipsec esp/tunnel/172.16.1.1
172.16.2.1/require
```

Ainsi, nous spécifions d'abord que nous montons un tunnel entre A et B, en utilisant 3DES avec la clé `123456789012123456789012` et `300` comme SPI pour ESP. Ensuite, nous disons que le trafic sortant de NetA vers NetB doit passer par le tunnel précédemment défini, et que le trafic en retour de NetB vers NetA doit arriver en utilisant ce même tunnel. Nous ne devons pas oublier de configurer notre filtrage IP pour laisser passer ESP entre A et B (protocole IP 50).

```
root@routeurA:~# iptables -A INPUT -p 50 -s 172.16.2.1 -j ACCEPT
root@routeurA:~# iptables -A OUTPUT -p 50 -d 172.16.2.1 -j ACCEPT
```

Sur B, les règles sont les mêmes, au sens de paquets près :

```
root@routeurB:~# setkey flush
root@routeurB:~# setkey spdflush
root@routeurB:~# setkey 172.16.1.1 172.16.2.1 esp 300
-m tunnel -E 3des-cbc "123456789012123456789012"
root@routeurB:~# setkey spadd 192.168.1.0/24
192.168.2.0/24 any -P in ipsec esp/tunnel/172.16.1.1
172.16.2.1/require
root@routeurB:~# setkey spadd 192.168.2.0/24
192.168.1.0/24 any -P out ipsec esp/tunnel/172.16.1.1
172.16.2.1/require
root@routeurB:~# iptables -A INPUT -p 50 -s 172.16.1.1
-j ACCEPT
root@routeurB:~# iptables -A OUTPUT -p 50 -d
172.16.1.1 -j ACCEPT
```

La couche IPv6 a été fortement retravaillée, en particulier pour y inclure une couche de sécurité s'appuyant sur IPSEC.

Nous avons donc à disposition les supports des extensions de confidentialité définies par la RFC 3041 (assignement périodique de nouvelles adresses à l'interface), de AH, de ESP et d'IPComp. Comme vu précédemment, les tunnels IPv6 sur IPv6 sont également supportés.

Autre nouveauté, l'inclusion dans les sources officielles du projet IPVS [18] qui fournit à Linux les capacités nécessaires à la constitution de fermes de serveurs avec partage de charge et haute disponibilité. IPVS supporte le partage de charge pour TCP, UDP, AH et ESP, ainsi que, à l'aide d'un *helper* spécifique, FTP.

Ce projet pourra intéresser tous ceux qui veulent implémenter des firewalls redondants sous Linux avec synchronisation d'état, puisque cette fonctionnalité est disponible pour le *load balancer* IPVS (*director*), alors que le projet Netfilter n'a toujours rien d'implémenté à ce niveau-là. Pour plus de détails sur IPVS, voyez la documentation sur le site du projet.

802.1d Ethernet Bridging, Frame Diverter, Network packet filtering et 802.1Q VLAN Support

Intéressons-nous un peu aux fonctions de niveau 2 concernant le support des ponts Ethernet (conformes à la norme IEEE 802.1d) et le support de l'encapsulation 802.1q pour la transport des VLANs. "Mais que vient faire le filtrage de paquets là-dedans ?" me direz-vous. Je le place ici parce que les nouveautés dans cette section sont l'inclusion dans le noyau du filtrage de paquet au-dessus d'un pont (*bridging firewall*) et le filtrage de niveau 2.

En activant l'option **802.1d Ethernet Bridging**, nous permettons à notre système Linux de se comporter comme un commutateur Ethernet (i.e. un switch) entre plusieurs interfaces physiques. Cette fonctionnalité se configure avec l'outil `brctl` qu'on trouve dans le paquetage `bridge-utils` [19] :

```
root@elendil:~# modprobe bridge
root@elendil:~# brctl addbr brØ
```

Nous disposons à présent d'une interface, `brØ`, matérialisant ce que nous appellerons par la suite un pont (appellation abusive mais commode). Notons qu'une machine peut supporter plusieurs ponts (`brØ`, `br1`, etc.) mais, en toute logique, une interface physique ne peut appartenir qu'à un seul pont à la fois. Bien que la fonction de pontage ne nécessite aucune configuration IP (fonction de niveau 2), nous pouvons tout de même configurer `brØ` comme n'importe quelle interface classique, rendant ainsi notre machine joignable sur le réseau Ethernet auquel appartient notre pont par une IP.

Dans certains cas, comme la mise en place de "firewalls transparents", nous n'affecterons pas d'IP au pont. Nous allons à présent lier des interfaces à notre pont :

```
root@elendil:~# ifconfig ethØ up
root@elendil:~# brctl addif brØ ethØ
root@elendil:~# ifconfig eth1 up
root@elendil:~# brctl addif brØ eth1
```

En examinant le log kernel (par `dmesg` par exemple), nous voyons que nos interfaces passent en mode `promiscuous` :

```
root@elendil:~# dmesg
[...]
NET4: Ethernet Bridge ØØ8 for NET4.Ø
Bridge firewalling registered
ethØ: Setting promiscuous mode.
device ethØ entered promiscuous mode
eth1: Setting promiscuous mode.
device eth1 entered promiscuous mode
```

En effet, nos interfaces doivent à présent s'intéresser à toutes les trames qu'elles reçoivent, quelle que soit leur adresse MAC destination, pour pouvoir les traiter. Enfin, nous activons notre pont en montant l'interface associée :

```
root@elendil:~# ifconfig brØ 192.168.1.1 netmask
255.255.255.Ø broadcast 192.168.1.255
```

Si nous ne voulons pas affecter d'IP, un `ifconfig brØ up` suffira. Les logs confirment que le pont est activé :

```
root@elendil:~# dmesg
[...]
brØ: port 2(eth1) entering learning state
brØ: port 1(ethØ) entering learning state
brØ: port 2(eth1) entering forwarding state
brØ: topology change detected, propagating
brØ: port 1(ethØ) entering forwarding state
brØ: topology change detected, propagating
```

Notons que l'adresse MAC utilisée par le pont lorsqu'il communique est celle de la première interface qui lui a été associée, à savoir celle mentionnée comme étant le port 1. Notons aussi que ce driver supporte le Spanning Tree (STP) et donc que votre machine Linux peut s'intégrer dans une architecture de commutation Ethernet mettant en œuvre ce protocole. Dans le doute, je vous conseille de le désactiver, juste après avoir créé votre pont :

```
root@elendil:~# brctl stp brØ off
```

L'état de vos ponts et leurs statistiques peuvent être consultés à l'aide de la commande `brctl` (rien dans `/proc/net` cette fois-ci) :

```
root@elendil:~# brctl show
bridge name     bridge id               STP enabled     interfaces
brØ             8ØØØ.ØØØ1Ø24Ø2eØa       yes             ethØ
                                                        eth1
root@elendil:~# brctl showmacs brØ
port no mac addr                is local?       ageing timer
  1     ØØ:Ø1:Ø2:4Ø:2e:Øa       yes                0.ØØ
  2     ØØ:Ø1:Ø5:dd:ef:43       yes                Ø.ØØ
  1     ØØ:Ø2:7e:22:c3:48       no                 3.6Ø
  1     ØØ:Øa:b7:58:33:94       no                 Ø.71
  1     ØØ:1Ø:a4:bb:4a:d3       no                15.97
  2     ØØ:1Ø:a4:bb:8a:51       no                37.3Ø
  1     ØØ:1Ø:b5:81:8a:d2       no               156.98
  1     ØØ:eØ:ØØ:59:cb:Ø8       no               279.86
  1     ØØ:eØ:ØØ:5c:88:33       no               126.71
root@elendil:~# brctl showstp brØ
brØ
 bridge id              8ØØØ.ØØØ1Ø24Ø2eØa
 designated root        2Øac.ØØØ5dc19fcØØ
 root port                 1                    path cost                  1Ø4
 max age                2Ø.ØØ                   bridge max age           2Ø.ØØ
 hello time              2.ØØ                   bridge hello time         2.ØØ
 forward delay          15.ØØ                   bridge forward delay     15.ØØ
 ageing time           3ØØ.ØØ                   gc interval               4.ØØ
 hello timer             Ø.ØØ                   tcn timer                 Ø.ØØ
 topology change timer   Ø.ØØ                   gc timer                  3.83
 flags
```

```
eth0 (1)
 port id              8001                 state                  forwarding
 designated root      20ac.0005dc19fc00    path cost                     100
 designated bridge    80ac.000ab7583380    message age timer            1.67
 designated port      8014                 forward delay timer          0.00
 designated cost         4                 hold timer                   0.00
 flags

eth1 (2)
 port id              8002                 state                  forwarding
 designated root      20ac.0005dc19fc00    path cos                      100
 designated bridge    8000.000102402e0a    message age timer            0.00
 designated port      8002                 forward delay timer          0.00
 designated cost      104                  hold timer                   0.67
 flags
```

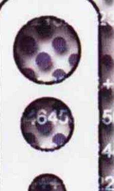
L'option **Frame Diverter** permet à un pont de rediriger vers lui des trames au lieu de les "forwarder". C'est l'équivalent au niveau Ethernet de la cible `REDIRECT` de Netfilter. Cette option nous permet donc de mettre en place des *proxies* transparents directement au niveau 2. À titre d'exemple succinct, nous allons configurer un proxy HTTP transparent au moyen de Squid [20] sur une machine possédant un pont `br0` sur ses deux interfaces `eth0` et `eth1`. Tout d'abord, Squid doit être configuré pour supporter le mode transparent. Nous le plaçons en écoute sur le port 3128 :

```
http_port 3128
httpd_accel_host virtual
httpd_accel_port 80
httpd_accel_with_proxy  on
httpd_accel_uses_host_header on
```

Le *diverter* se configure avec l'outil `divert` disponible sur le site [21] dédié à cette fonctionnalité. Si vous voulez utiliser la version fournie avec le noyau, à savoir la version stable 0.46, il vous faudra utiliser la version 0.221 de `divert`. Si vous voulez utiliser la toute dernière version, la 0.52, vous devrez utiliser `divert 0.32`. Non seulement les versions ne sont pas compatibles, mais en outre, elles n'ont pas la même syntaxe (le numéro de la version compilée dans le noyau courant est contenue dans le fichier `/proc/sys/net/core/divert_version`). Nous utiliserons ici la version du noyau officiel.

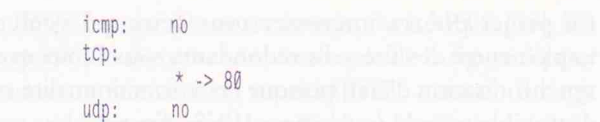
Nous allons remonter au niveau IP les trames contenant les requêtes HTTP qui nous intéressent. L'interface considérée est celle par laquelle les requêtes vont arriver au pont. Nous redirigeons donc les trames contenant un segment TCP à destination du port 80 que nous recevons sur l'interface `eth0` :

```
root@elendil:~# divert on eth0 enable
root@elendil:~# divert on eth0 tcp add dst 80
```



Nous pouvons à tout moment vérifier notre configuration courante en appelant la commande sur l'interface considérée sans argument :

```
root@elendil:~# divert on eth0
version: 0.46
status:  active
ip:      no
```



```
icmp:    no
tcp:
         * -> 80
udp:     no
```

Une fois les paquets IP contenus de ces trames redirigés vers la couche IP, nous n'avons plus qu'à les rediriger vers notre proxy Squid avec une simple règle de NAT :

```
root@elendil:~# iptables -t nat -A PREROUTING -p tcp –dport 80 -j REDIRECT –to-ports 3128
```

Notons qu'il y a maintes façons de mettre en place un proxy transparent, que ce soit en utilisant un routage avancé à base de marques ou encore une NAT sur la destination des requêtes HTTP. C'en est une parmi d'autres, que je trouve plutôt élégante quand on utilise un pont.

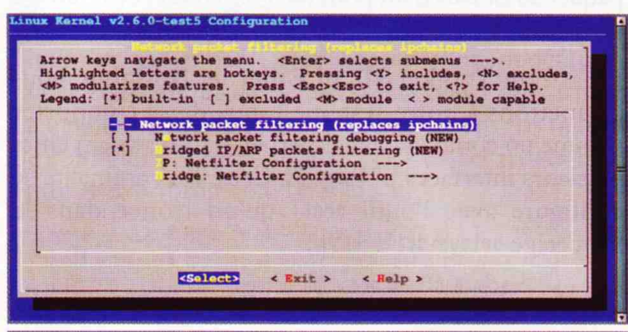


Figure 4 Menu de configuration du filtrage de paquets

Une fois l'option `bridging` activée, nous constatons que deux nouvelles entrées sont présentes dans le sous-menu de configuration du filtrage de paquets, **Network packet filtering**, comme le montre la figure 4 :

- **Bridged IP/ARP packets filtering** : cette option permet de voir apparaître dans la chaîne `FORWARD` de Netfilter la charge des trames traitées par notre pont si celle-ci est de l'ARP ou de l'IP. Nous pouvons donc filtrer ces paquets, créant ainsi un pont filtrant, base des "firewalls transparents". Le principe est fort simple et si vous désirez en savoir plus sur cette fonctionnalité, je vous conseille la lecture d'un article [22] sur le sujet paru dans le Hors Série 13 de Linux Magazine France.

- **Bridge : Netfilter Configuration** : cette option permet de configurer le filtrage de niveau 2 au niveau de vos ponts (et seulement sur des ponts !) à l'aide de l'outil `ebtables` [23]. Ce dernier vous permettra de filtrer les trames sur la base de leurs en-têtes de niveau 2 (adresses MAC, type de la charge, etc.) pour différents protocoles (Ethernet, 802.3, 802.1q et STP), un filtrage complet des paquets ARP, un filtrage simple sur quelques en-têtes IP, du *mangling* (dont le marquage de trames) et du NAT. Pour ARP et IP en particulier, `ebtables` vous permettra d'imposer des règles de validation MAC/IP non négligeables dans certains environnements.

Du côté de Netfilter proprement dit, il n'y a pas grand-chose de nouveau sous le soleil au niveau de l'architecture générale. Je vous renvoie donc à mon article [24] sur le sujet paru dans le Hors Série 12. Côté fonctionnalités, de nouvelles concordances et cibles ont été introduites. Enfin, le support du protocole ARP (filtrage et *mangle*) dispose maintenant d'un outil de configuration, `arptables`, disponible sur le site de `ebtables` [23].

L'option **802.1Q VLAN Support** permet à notre système de supporter le protocole de transport de VLAN par encapsulation (IEEE 802.1q) sur des liens couramment appelés *trunks* (terminologie Cisco). Comme le montre la figure 5, la trame 802.1q se différencie de la trame Ethernet classique par l'insertion d'un champ de 2 octets permettant d'une part d'identifier le VLAN auquel appartient la trame (champ VLAN ID sur 12 bits) et d'autre part de lui affecter une priorité sur le trunk (champ Prio), et d'un nouveau champ Type (le premier désigne la trame comme trame 802.1q, le second définissant le type de charge).

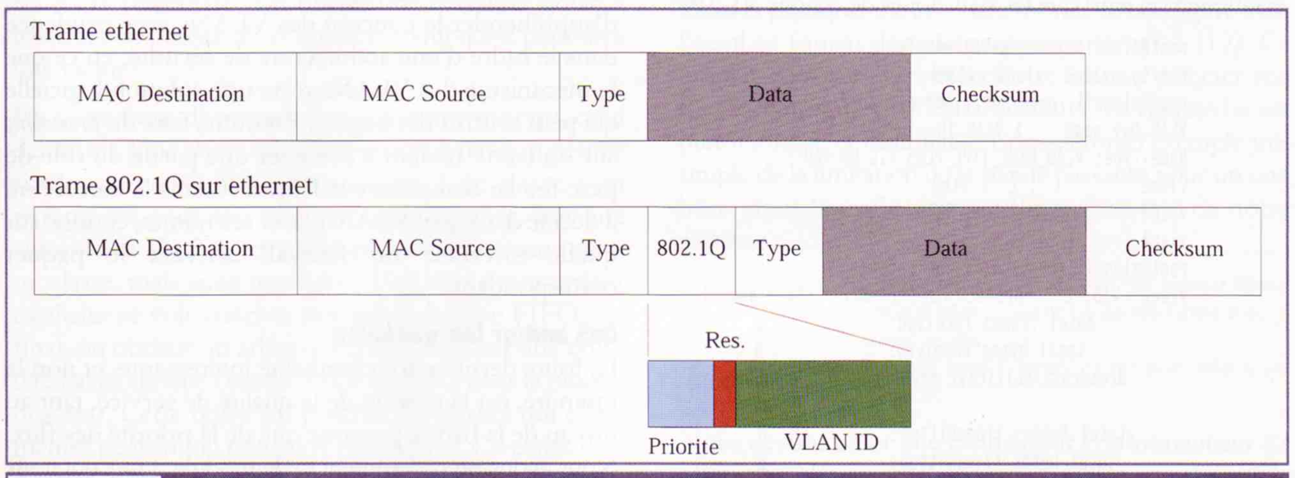


Figure 5 Trames Ethernet et trame 802.1q

En reliant une interface Ethernet de notre système à un commutateur supportant l'encapsulation 802.1q, nous sommes capables de recevoir plusieurs VLAN. Pour notre couche réseau, chaque VLAN sera vu comme une interface logique. La configuration 802.1q se fait avec l'outil `vconfig` disponible sur le site [25] dédié au support des VLAN. Si nous voulons configurer notre machine comme décrit en figure 6, nous procéderons comme suit. Commençons donc par configurer nos VLAN sur l'interface destinée à les recevoir :

```
root@elendil:~# modprobe 8021q
root@elendil:~# ifconfig eth0 up
root@elendil:~# vconfig set_name_type
VLAN_PLUS_VID_NO_PAD
Set name-type for VLAN subsystem. Should be visible in
/proc/net/vlan/config
root@elendil:~# vconfig add eth0 2
Added VLAN with VID == 2 to IF -:eth0:-
root@elendil:~# vconfig add eth0 3
Added VLAN with VID == 3 to IF -:eth0:-
root@elendil:~# vconfig add eth0 4
Added VLAN with VID == 4 to IF -:eth0:-
root@elendil:~# vconfig add eth0 5
Added VLAN with VID == 5 to IF -:eth0:-
```

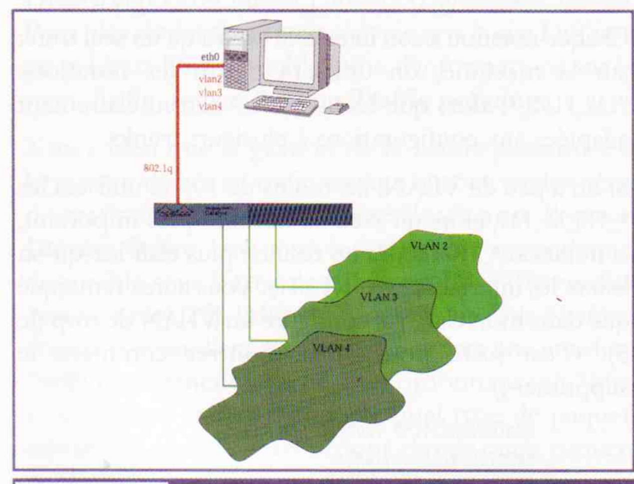


Figure 6 Architecture à base de VLANs

Nous n'utilisons pas le VLAN 1 à dessein. En effet, ce VLAN présente chez certains constructeurs (Cisco en particulier) des particularités qui interdisent son utilisation à des fins de transport de trafic réseau, le réservant à un usage uniquement administratif.

On obtient donc sur `eth0` ce qu'on appelle un trunk, un lien destiné à transporter des VLAN, c'est-à-dire des trames 802.1q. La commande `vconfig set_name_type VLAN_PLUS_VID_NO_PAD` permet de choisir le modèle de nommage des interfaces logiques associées aux différents VLAN. Quatre modèles sont disponibles :

- `VLAN_PLUS_VID` : `vlan` suivi du numéro de VLAN sur quatre chiffres, i.e. `vlan0005` ;
- `VLAN_PLUS_VID_NO_PAD` : `vlan` suivi du numéro court de VLAN, i.e. `vlan5` (c'est ce que j'ai choisi ici) ;

DEV_PLUS_VID : interface suivie du numéro de VLAN sur quatre chiffres, i.e. eth0.0005 ;

DEV_PLUS_VID_NO_PAD : interface suivie du numéro court de VLAN, i.e. eth0.5.

Chaque notation a son intérêt. Si on n'a qu'un seul trunk sur la machine, on utilisera plutôt les notations VLAN_PLUS_*, alors que les DEV_PLUS_* seront clairement adaptées aux configurations à plusieurs trunks.

Si on a peu de VLAN, i.e. moins de 10, on utilisera les *_VID_NO_PAD, alors qu'avec un nombre plus important, la notation *_VID rendra un résultat plus clair lorsqu'on listera les interfaces avec ifconfig. Vous aurez remarqué que dans mon élan, j'ai configuré un VLAN de trop (le 5). C'est juste pour vous montrer comment le supprimer ;)

```
root@elendil:~# vconfig rem vlan5
Removed VLAN -:vlan5:-
```

Une fois votre trunk mis en place, nous pouvons aller jeter un coup d'œil dans /proc/ pour en regarder la configuration ainsi que les statistiques de chaque VLAN :

```
root@elendil:/proc/net/vlan# ls
config  vlan2  vlan3  vlan4
root@elendil:/proc/net/vlan# cat config
VLAN Dev name    | VLAN ID
Name-Type: VLAN_NAME_TYPE_PLUS_VID_NO_PAD
vlan2          | 2  | eth0
vlan3          | 3  | eth0
vlan4          | 4  | eth0
root@elendil:/proc/net/vlan# cat vlan2
vlan2  VID: 2    REORDER_HDR: 1  dev->priv_flags: 1
         total frames received:           0
          total bytes received:           0
     Broadcast/Multicast Rcvd:            0

      total frames transmitted:           0
       total bytes transmitted:           0
            total headroom inc:           0
           total encap on xmit:           0
Device: eth0
INGRESS priority mappings: 0:0  1:0  2:0  3:0  4:0
5:0  6:0 7:0
EGRESSS priority Mappings:
```

Bien entendu, vous devez disposer en face d'un commutateur configuré en conséquence, ce qui, sur un Cisco 2950 [2] nous donne quelque chose dans ce genre-là (mode trunk pur, sans négociation possible) :

```
2950# configure terminal
Enter configuration commands, one per line. End with CNTL/Z.
2950(config)# interface fastethernet0/1
2950(config-if)# switchport mode trunk
2950(config-if)# switchport nonegotiate
2950(config-if)# switchport trunk native vlan 100
2950(config-if)# switchport trunk allowed vlan add 2-4
2950(config-if)# end
```

Notons qu'il est tout à fait possible de créer un trunk au-dessus d'un agrégat, aussi bien du côté Linux que du côté du commutateur Ethernet. Il est intéressant de considérer cette éventualité si on ne dispose pas de carte GigabitEthernet dans la mesure où les trunks sont par essence des liens très chargés sur une architecture commutée.

Notons aussi que nous pouvons toujours émettre ou recevoir des trames Ethernet classiques en utilisant directement eth0, mais que ce type d'utilisation (lien hybride) est clairement déconseillée.

Et maintenant que vos interfaces sont en place, vous pouvez activer le routage entre les différents VLAN et configurer un filtrage de paquets ou de la gestion de bande passante, bref, tout ce que vous pouviez faire de vos interfaces physiques. Au sein d'une architecture réseau, l'utilisation des VLAN vous permet de rationaliser l'utilisation des ports de vos commutateurs Ethernet même lorsque le nombre de réseaux IP augmente.

En outre, au niveau d'un pare-feu, vous pouvez multiplier les DMZ à moindre coût et donc augmenter votre niveau de segmentation. Il convient cependant d'appréhender le concept des VLANs avec prudence dans le cadre d'une architecture de sécurité, en ce que le mécanisme des VLANs reste une solution logicielle qui peut souffrir des bogues. En outre, faire du *firewalling* sur un trunk revient à déléguer une partie du rôle de pare-feu au commutateur Ethernet dans la mesure où il décide dans quel VLAN placer ses trames, et donc sur quelle interface du firewall arrivera le paquet correspondant.

QoS and/or fair queueing

La toute dernière fonctionnalité intéressante, et non la moindre, est la gestion de la qualité de service, tant au niveau de la bande passante que de la priorité des flux. Nous avons précédemment vu le module shaper qui nous permettait de limiter de la bande passante de manière fort simple. Les méthodes que nous allons voir à présent sont nettement plus puissantes, mais aussi plus difficiles à appréhender.

Globalement, la section **QoS and/or fair queueing** est divisée en trois groupes :

- Les ordonnanceurs (*schedulers*), chargés de la gestion des files d'attente (*queue*) pour une interface donnée. Nous disposons de divers algorithmes dont les deux plus courants sont CBQ (*Class Based Queueing*) et HTB (*Hierarchical Token Buckets*). Ces deux ordonnanceurs permettent de créer une hiérarchie de files d'attentes avec des notions de parenté et d'héritage.

- Les files d'attentes, qui sont chargées d'émettre les paquets selon un comportement bien spécifique à chaque type. Les deux types le plus souvent retrouvés sont SFQ (*Stochastic Fairness Queueing*) et TBF (*Token Bucket Filter*)

Les classificateurs (*classifiers*), sous **Packet classifier API**, dont le rôle est de ventiler les paquets entre les classes (cf ci-après) disponibles. Les plus utilisés sont U32, qui permet de "matcher" les en-têtes des paquets à l'aide de masques hexadécimaux ; Route, qui permet de trier en fonction de la route suivie ; et enfin Firewall, qui s'appuie sur la reconnaissance de la sacro-sainte marque Netfilter. On peut aussi filtrer en fonction de la valeur du champ TOS, puisque la QoS est sa raison d'être.

Ce qu'il faut bien comprendre, c'est le concept de *queueing discipline* (*qdisc*), ou gestion de queue. Les ordonnanceurs et les files d'attentes sont des qdiscs, à savoir des méthodes d'envoi des paquets. Seulement, les ordonnanceurs diffèrent des files en ce qu'ils peuvent avoir des enfants. Ils servent donc à envoyer des paquets vers ces enfants (classes). C'est pourquoi on les appelle *classful qdisc*. Les autres sont des *classless qdisc*, qui ne peuvent pas avoir d'enfant et servent à envoyer les paquets sur le réseau. De fait, une classless qdisc est toujours terminale. En outre, une classless qdisc a toujours un parent, alors que la classful qdisc peut-être une racine.

En outre, nous disposons d'un quatrième concept, celui de classes. Une classe sert à imposer la limitation désirée sur les paquets qui y sont envoyés. Une classe a un parent, qui peut soit être une classful qdisc, soit une autre classe. De fait, une classe peut avoir comme enfant un classe, mais aussi une qdisc. Une classe sans enfant explicite se voit attacher une qdisc de type FIFO. Au final, on obtient un arbre qui commence par une qdisc dite racine de type classful (HTB ou CBQ dans la plupart des cas). Cette qdisc est parente de classes qui elles-mêmes peuvent ou non avoir des enfants. Certaines n'en ont pas, d'autres ont des classes comme filles, d'autres, enfin, sont parentes de qdisc de type classless. Nous disposons en parallèle d'une série de classificateurs dont le rôle est de ventiler les paquets dans les différentes classes.

Le fonctionnement est le suivant. Une qdisc est une manière d'envoyer les paquets qui sont dans une file d'attente. La qdisc racine a pour rôle d'envoyer les paquets vers les classificateurs qui vont ventiler les classes filles. Celles-ci vont alors devoir envoyer leurs propres paquets, soit à une autre classe (auquel cas on parcourt l'arbre en utilisant un autre classificateur), soit sur l'interface en utilisant une qdisc classless, FIFO par défaut ou la qdisc imposée par l'utilisateur (généralement TBF ou SFQ).

Si vous comptez aller plus loin et vous tourner vers la qualité de service avec priorités sur les flux et réservation de bande passante, il vous faudra activer les options **QoS support** et **Rate estimator**. Cette dernière permet au noyau de mesurer le trafic présent sur une interface donnée et donc d'adapter la qualité de service en fonction ; c'est une fonctionnalité indispensable à une bonne implémentation de la QoS. Vous aurez à votre disposition un classificateur RSVP (*Resource Reservation Protocol*) et pourrez utiliser DiffServ (*Differentiated Services*). Pour plus de détails, je vous renvoie vers le site DiffServ pour Linux [26], véritable mine d'informations sur le sujet. Enfin, activez l'option **Traffic policing**.

Notez bien que la gestion de la bande passante est largement dédiée au trafic sortant. Si vous voulez gérer du trafic entrant, vous devez sélectionner l'option **Ingress Qdisc**. Une autre fonctionnalité sympathique, disponible sous forme de patch, est IMQ (*Intermediate queueing device*) [27]. IMQ vous fournit une cible Netfilter dont le rôle est d'envoyer les paquets vers une interface (`imq0`) directement attaché à un ordonnanceur. Vous pouvez ainsi traiter n'importe quel type de paquet, entrant ou sortant, mais surtout choisir quels paquets vont passer par la gestion de QoS et quels paquets qui n'y passeront pas.

Dans la pratique, tout ce beau monde se configure avec l'outil `tc` fourni dans le paquetage `iproute2` [12]. Ce dernier vous servira à créer votre arbre, y affecter vos files et configurer les classificateurs. La démarche est plutôt longue et fastidieuse. Considérons l'exemple très simple de la limitation de la bande passante pour un seul hôte, dont le trafic sort par l'interface `eth0` de notre système :

```
tc qdisc add dev eth0 root handle 1: cbq avpkt 1000 bandwidth 100mbit
tc class add dev eth0 parent 1: classid 1:1 cbq rate 512kbit allot
1500 prio 5 bounded isolated
tc filter add dev eth0 parent 1: protocol ip prio 16 u32 match ip src
192.168.1.10 flowid 1:1
```

Nous devons créer une racine pour l'ordonnanceur (`tc qdisc add`). Nous greffons sur cette racine une classe fille limitée à 512kbps (`tc class add`). Enfin, nous configurons notre classificateur pour qu'il envoie sur cette dernière classe les paquets en provenance de 192.168.1.10 (`tc filter add`). Et ceci ne concerne qu'un seul sens. Si je veux mettre en place une limitation bidirectionnelle, il faut mettre un jeu de règles similaires sur `eth1` :

```
tc qdisc add dev eth1 root handle 1: cbq avpkt 1000 bandwidth 100mbit
tc class add dev eth1 parent 1: classid 1:1 cbq rate 512kbit allot
1500 prio 5 bounded isolated
tc filter add dev eth1 parent 1: protocol ip prio 16 u32 match ip dst
192.168.1.10 flowid 1:1
```

Si en plus, je veux utiliser une gestion de file d'attente particulière plutôt qu'une simple file FIFO (comportement par défaut), je le précise ainsi :

```
tc qdisc add dev eth0 parent 1:1 handle 10: sfq perturb 10
tc qdisc add dev eth1 parent 1:1 handle 10: sfq perturb 10
```

De cette manière, les paquets qui vont passer par la classe `1:1` limitée à 512kbps seront traités par le mécanisme SFQ.

On peut vérifier le tout en demandant à `tc` de nous afficher les paramètres en cours sur `eth0` par exemple :

```
root@elendil:/etc/cbq# tc qdisc show dev eth1
qdisc sfq 10: quantum 1514b perturb 10sec
qdisc cbq 1: rate 100Mbit (bounded,isolated) prio no-transmit
root@elendil:/etc/cbq# tc class show dev eth1
class cbq 1: root rate 100Mbit (bounded,isolated) prio no-transmit
class cbq 1:1 parent 1: leaf 10: rate 512Kbit (bounded,isolated) prio 5
root@elendil:/etc/cbq# tc filter show dev eth1
filter parent 1: protocol ip pref 16 u32
filter parent 1: protocol ip pref 16 u32 fh 800: ht divisor 1
filter parent 1: protocol ip pref 16 u32 fh 800::800 order 2048 key ht
800 bkt 0 flowid 1:1
  match c0a8010a/ffffffff at 16
```

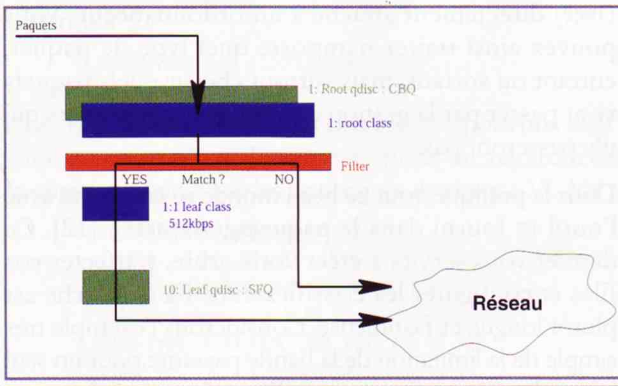


Figure 7 Configuration CBQ simple

Voyons la **figure 7**. Nous avons une qdisc racine nommée 1: de type CBQ (classful) et une qdisc `10:` de type SFQ (classless). 1: a une classe fille également appelée `1:`. Celle-ci est créée automatiquement et prend toute la bande passante disponible sur la qdisc parente puisqu'elle en traitera tous les paquets. La classe `1:1` a pour parente la classe `1:` (`parent 1:`) et pour fille la qdisc classless `10:` (`leaf 10:`) de type SFQ. Les paquets qui passeront par elle sont choisis par le troisième filtre (`flowid 1:1`), qui est attaché à la classe `1:` (`parent 1:`). La bande passante de 1:1 est limitée à 512kbps et les paquets sont envoyés en utilisant la qdisc SFQ (`10:`).

Le choix entre SFQ et TBF est relativement simple. TBF vous permet d'obtenir une limitation très précise du trafic, mais ne fait pas de distinction entre les flux qui le composent, pouvant conduire à la monopolisation de la bande passante par une application au détriment des autres. SFQ est moins précise, mais elle distingue les flux de niveau 4 (typiquement UDP et TCP) et tente de les traiter de manière égale. Ainsi, on ne risque pas de voir une application prendre le dessus et gêner les autres. La classe `1:1` est enfin bornée (`bounded`), ce qui signifie qu'elle ne peut emprunter de bande passante à sa classe parente lorsqu'elle atteint sa limite, même si cette dernière dispose de bande passante libre, et isolée (`isolated`), ce qui signifie qu'elle ne peut pas prêter de bande passante disponible à des sœurs qui ne seraient pas elles-mêmes bornées. Les concepts de classe bornée et de classe isolée nous permettent de mettre en place des schémas de gestion de bande passante plus dynamiques avec, par exemple, la notion de bande passante minimum.

Je ne vous cache pas que dès que vous avez besoin de spécifier plusieurs politiques de limitations, les choses commencent à devenir de plus en plus compliquées, comme illustré en figure 8. Heureusement, nous avons à notre disposition deux scripts qui nous permettent de concevoir des architectures de gestion de trafic de manière raisonnablement simple et souple. Le premier, `CBQ.init` [28], s'appuie sur la qdisc CBQ tandis que le second, `HTB.init` [29], s'appuie plutôt sur HTB. Ils supportent tous les deux les classificateurs U32, Route et Firewall, et les qdisc classless TBF et SFQ. En ce qui concerne leur configuration, le mode d'emploi figure en commentaire au début des scripts eux-mêmes. *So, read the source, Luke, read the source.* Les scripts utilisent des fichiers de configuration très simples, de format très proche. Si on sait se servir de l'un, il ne faut guère de temps pour apprendre à utiliser l'autre.

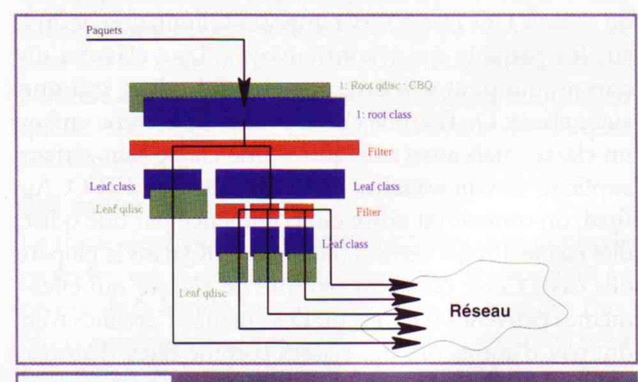


Figure 8 Configuration CBQ étoffée

On saisit bien toute la puissance de la gestion de QoS sous Linux, mais aussi toute la difficulté qui l'accompagne. Même si le travail se trouve facilité par l'utilisation des scripts cités, les concepts sous-jacents sont tout de même (en tout cas pour moi) un peu ardus à appréhender au début. Je vous conseille donc vivement la lecture du chapitre 9, *Queueing Disciplines for Bandwidth Management*, de l'incontournable LART [13] avant de vous lancer dans l'aventure.

Autres fonctionnalités

De nombreuses fonctionnalités en relation avec le réseau sont disséminées dans d'autres sections de l'interface de configuration du noyau. On pourra citer :

- **Network block device** dans la section Block devices, qui a fait l'objet d'un article [30] dans Linux Magazine ;

Ethernet over 1394 dans la section IEEE 1394 (FireWire), qui vous permettra de créer un lien réseau haut débit sur FireWire ;

Amateur Radio support, qui vous permet de faire du X25 avec des équipements radio-amateur ;

IrDA (infrared), qui possède une extension IrLAN vous permettant de communiquer via votre interface infrarouge (en plus du PPP sur IrCOMM) ;

ISDN support pour le support des modem RNIS ;

Le support des systèmes de fichiers en réseau avec NFS, SMB, CIFS, NCP, Coda, InterMezzo et AFS ;

Multi-purpose USB Networking Framework dans la section USB qui vous fournit la possibilité de faire du réseau sur un câble USB adéquat, en plus de tous les contrôleurs réseau et modems USB supportés disponibles dans la même section ;

Et enfin la possibilité de monter des réseau, Ethernet sur BlueTooth avec le support BNET.

J'ajouterai à tout cela le support des LSM ou (*Linux Security Modules*), qui font l'objet d'un article [31] dans ce même numéro, pouvant induire de nouvelles fonctions de sécurité sur l'accès des applications à la couche réseau, en particulier à travers l'option **Socket and Networking Security Hooks**.

Au tout début de cet article, je vous conseillais l'activation du support du Sysctl. Nous avons en effet pu voir qu'il pouvait servir à configurer certaines options ou à récupérer des données (statistiques, état, etc.). Je ne peux pas vous faire ici la liste exhaustive détaillée de toutes les options proposées par le Sysctl pour configurer le comportement de votre machine vis-à-vis du réseau. C'est pourquoi je vous renvoie au chapitre 13, *Kernel network parameters*, du LARTC [13].

Enfin, les capacités réseau de Linux ne s'arrêtent pas à ce que supporte le noyau Linux. En effet, de nombreuses fonctionnalités sont implémentées sous forme d'applications, parmi lesquelles ont pourra citer le support complet du routage dynamique (RIP, OSPF ou BGP) avec Zebra [32], ou encore la gestion du *failover* en réseau avec VRRP en utilisant KeepAlived [33].

Conclusion

J'espère vous avoir montré, sinon démontré, au travers de ce descriptif sommaire, l'extrême versatilité du noyau Linux dans l'univers des réseaux. Il peut se connecter à pratiquement tous les réseaux existants ou ayant existé, communiquer en utilisant une large palette de protocoles, s'interfacer avec de nombreux systèmes et enfin remplir un nombre de fonctions particulièrement important. Cela fait de Linux un outil d'une incroyable utilité pour tous ceux qui veulent jouer avec les réseaux et expérimenter des fonctions normalement réservées aux gros systèmes, et, à l'occasion, faire pâlir de jalousie vos collègues devant votre portable que vous pouvez configurer en pont/routeur/firewall/shaper en trois minutes chrono en main.

P.S. : Je n'ai malheureusement pas pu me procurer le switch Cisco 2950 pour valider les exemples fournis que je donne de mémoire, avec le manuel sur les genoux ; je réclame donc toute votre indulgence en la matière.

Références

[1] Linux Channel Bonding, http://sourceforge.net/projects/bonding/

[2] Cisco Catalyst 2950 Series Switches Documentation, http://www.cisco.com/en/US/products/hw/switches/ps628/index.html

[3] Tunneling over UDP using tun/tap, http://www.cartel-securite.fr/pbiondi/projects/tuntap_udp.html

[4] VTun, http://vtun.sourceforge.net/

[5] OpenVPN, http://openvpn.sourceforge.net/

[6] Universal TUN/TAP Driver, http://vtun.sourceforge.net/tun/

[7] PPP, http://www.samba.org/ppp/

[8] RoaringPenguin PPPoE, http://www.roaringpenguin.com/pppoe/

[9] Libpcap, http://www.tcpdump.org/

[10] Libnet Packet Assembly, http://www.packetfactory.net/projects/libnet/

[11] Netfilter, http://www.netfilter.org/

[12] iproute2, ftp://ftp.inr.ac.ru/ip-routing/

[13] Bert Hubert et autres auteurs, *Linux Advanced Routing and Traffic Control HOWTO*, http://lartc.org/

[14] DJ Bernstein, *SYN Cookies*, http://cr.yp.to/syncookies.html

[15] FreeS/WAN, http://www.freeswan.org/

[16] IPsec-Tools, http://ipsec-tools.sourceforge.net/

[17] The GNU/Linux CryptoAPI site, http://www.kerneli.org/

[18] Linux Virtual Server Project, http://www.linuxvirtualserver.org/

[19] Linux Ethernet Bridging, http://bridge.sourceforge.net/

[20] Squid Web Proxy Cache, http://www.squid-cache.org/

[21] Frame Diverter, http://diverter.sourceforge.net/

[22] Loïc Minier, *Un bridge "firewallant"*, Linux Magazine France HS13

[23] ebtables, http://ebtables.sourceforge.net/

[24] Cédric Blancher, *Netfilter/iptables*, Linux Magazine France HS12

[25] 802.1Q VLAN implementation for Linux, http://www.candelatech.com/~greear/vlan.html

[26] Differentiated Services on Linux, http://diffserv.sourceforge.net/

[27] The intermediate queueing device, http://trash.net/~kaber/imq/

[28] CBQ.init, http://sourceforge.net/projects/cbqinit/

[29] HTB.init, http://sourceforge.net/projects/htbinit/

[30] Yves Bailly, *NBD*, http://www.linuxmag-france.org/LINU_ndb.pdf et Linux Magazine France 54

[31] Philippe Biondi et Frédéric Raynal, *Les LSM du Futur*, Linux Magazine France HS 17

[32] Zebra, http://www.zebra.org/

[33] KeepAlived, http://keepalived.sourceforge.net/

Développez vos pilotes de périphériques USB

Les périphériques USB se sont largement répandus : scanner, appareil photo, disques durs externes... Cette connectique offre en effet de nombreux avantages (configuration "à chaud", débit de 480 Mbits/s pour la version USB2.0) et semble à présent incontournable. Cet article est une introduction à la gestion de l'USB sous Linux et à l'écriture de pilotes USB. Il s'inscrit dans la lignée des articles déjà parus dans Linux Magazine 17 (introduction à l'écriture driver) et Linux Magazine 42 (écriture de pilotes PCI).

Après quelques précisions sur les spécifications USB (version 1.1), nous décrirons l'interface USB présentée par le noyau et détaillerons l'écriture d'un mini-driver nous permettant de lire directement les données d'une souris USB.

Connaissances et matériel requis :

- Langage C
- Une souris USB
- Noyau 2.4.x

Aperçu de l'architecture USB et concepts clefs

Un ordinateur ne comporte qu'un seul contrôleur USB implanté sous forme matérielle et logicielle. Cet unique contrôleur intègre un hub racine fournissant un ou plusieurs points d'attachements.

Les périphériques USB sont de deux sortes :

- Les hubs offrant de nouveaux points d'attachements ;
- Les "fonctions" offrant de nouvelles fonctionnalités : scanner, souris...

Tous présentent la même interface USB au travers de :

- Leur compréhension du protocole USB ;
- Leur réponse aux opérations USB standards : configuration, reset... ;
- Leur manière de se décrire.

Topologie du bus

Topologie physique : l'USB connecte des périphériques USB à un hôte USB. L'interconnexion physique décrit une topologie en étoile sur plusieurs niveaux (**Fig. 1**).

Topologie logique : L'hôte communique avec chaque périphérique comme s'il était directement connecté au hub racine.

Support physique

L'USB transfère signal et alimentation dans un câble à quatre fils. Il autorise deux taux de transfert : *full-speed* (12 Mb/s) et *low-speed* (1,5 Mb/s) pour les périphériques ne nécessitant pas une large bande passante, comme les souris par exemple.

Flux de communication

Les communications USB se déroulent au travers de deux interfaces logicielles :

- *Host Controller Driver* (HCD), couche logicielle permettant de faire abstraction du contrôleur USB matériel. Elle fournit une SPI (*System Programming Interface*) permettant d'interagir avec le contrôleur USB, et de cacher les spécificités de l'implantation matérielle.
- *USB Driver* (USBD), le pilote USB qui fournit les fonctionnalités propres au périphérique.

Périphérique logique

Un périphérique USB apparaît au système comme une collection de "endpoints". Ce sont d'uniques parties adressables du périphérique et sont sources ou cibles du flux de communication entre le périphérique et l'hôte. Chaque "endpoint" est caractérisé par un identifiant unique (fixé par le constructeur) appelé *endpoint number* et constitue une connexion supportant un flux de données du périphérique vers l'hôte ou inversement.

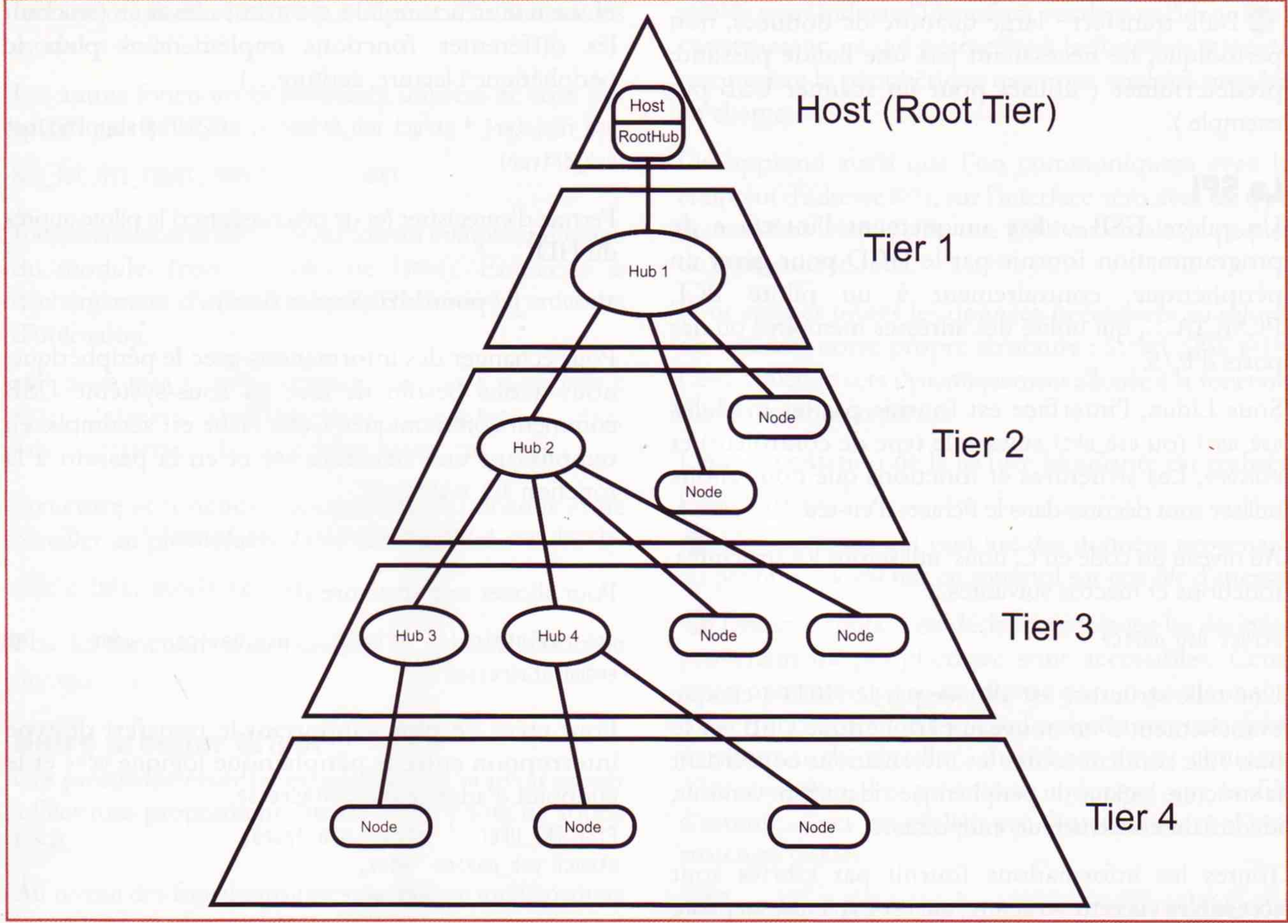


Figure 1 **Un hub est au centre de chaque étoile.**

Chaque endpoint possède une direction de flux prédéterminée (IN/OUT) et différents paramètres décrivant les caractéristiques des transferts qu'il est susceptible de supporter :

- Temps de latence et fréquence d'accès au bus ;
- Bande passante ;
- Endpoint address ;
- Gestion d'erreur ;
- Taille maximale des paquets qu'il peut recevoir et envoyer ;
- Type de transfert ;
- Direction des transferts.

Les "pipes" sont l'abstraction logique représentant l'association entre un "endpoint" et l' USBD.

Les endpoints sont regroupés en "*endpoints sets*" pour former des interfaces. Une interface peut être assimilée à une vue du périphérique.

Chaque périphérique USB implémente de manière obligatoire une méthode de contrôle par défaut utilisant un endpoint particulier possédant le numéro 0.

A cet endpoint est associé un tube de contrôle par défaut (*Default Control Pipe*), qui fournit un accès aux informations de configuration du périphérique.

Ce tube supporte des transferts de contrôle permettant de configurer le périphérique. Les endpoints de numéro 0 sont accessibles dès que le périphérique est attaché au bus.

Sous Linux, ces informations sont lisibles soit directement via `/proc/bus/usb/devices`, soit par des outils tels que `lsusb` (mode texte) ou `usbview` (mode graphique).

Types de transfert

L'USB définit quatre types de transfert, optimisés en termes de temps de latence, contraintes de taille… suivant le type de service demandé.

- Transfert de contrôle : ponctuel, non périodique, à l'initiative de l'hôte, utilisé typiquement pour les opérations de configuration et de contrôle de statut ;
- Transfert synchrone : périodique, continu et pouvant nécessiter une certaine bande passante (utilisé pour une camera USB par exemple) ;
- Transfert d'interruption : données de petite taille, faible temps de latence, faible fréquence (utilisé pour une souris USB par exemple) ;

Bulk transfert : large quantité de données, non périodique, ne nécessitant pas une bande passante prédéterminée (utilisés pour un scanner USB par exemple).

La SPI

Un pilote USB utilise uniquement l'interface de programmation fournie par le HCD pour gérer un périphérique, contrairement à un pilote PCI, PCMCIA..., qui utilise des adresses mémoires ou des ports d'E/S.

Sous Linux, l'interface est fournie par les modules `usb_uhci` (ou `usb_ohci` suivant le type de contrôleur) et `usbcore`. Les structures et fonctions que nous allons utiliser sont décrites dans le fichiers d'en-tête `linux/usb.h`.

Au niveau du code en C, nous utiliserons les structures, fonctions et macros suivantes :

```
struct usb_device
```

Une telle structure est allouée par le HCD à chaque branchement d'un nouveau périphérique USB sur le bus. Elle contient toutes les informations concernant la structure logique du périphérique : identifiant vendeur, identifiant constructeur, endpoints...

Toutes les informations fournit par `usbview` sont accessibles via cette structure, qui est transmise au pilote afin de lui permettre de reconnaître dans un premier temps le périphérique qu'il va prendre en charge puis de communiquer avec.

```
static void *probe (struct usb_device *udev,unsigned ifnum,
const struct usb_device_id *prod)
```

Cette fonction est appelée à chaque nouvel attachement d'un périphérique au bus USB. L'appel de cette fonction est réalisé avec la structure `usb_device` décrite précédemment, correspondant au nouveau périphérique.

Le rôle de cette fonction est de vérifier les informations de configuration du périphérique et de s'attacher le périphérique le cas échéant. L'attachement du pilote au périphérique est réalisé si la fonction retourne un pointeur non nul (généralement un pointeur sur une structure de données spécifique au pilote et allouée au sein de celui-ci).

```
static void disconnect(struct usb_device *udev, void *ptr)
```

Fonction appelée lors du débranchement du périphérique. Son rôle est la libération des ressources mobilisées par le driver. `Ptr` est un pointeur sur la structure de données retournée par `probe`.

```
struct usb_driver
```

Structure représentant le driver USB. Contient le nom du driver, un pointeur sur les fonctions `probe`, `disconnect` et sur une structure `file_operations` classique (stockant les différentes fonctions implémentées pour le périphérique : lecture, écriture...)

```
usb_register( * struct usb_driver ), usb_deregister (*struct
usb_driver)
```

Permet d'enregistrer (et de désenregistrer) le pilote auprès du HDC.

`struct urb` (pour *USB Request Block*)

Pour échanger des informations avec le périphérique, nous avons besoin de dire au sous-système USB comment communiquer. Cette tâche est accomplie en remplissant une structure `urb`et en la passant à la fonction `usb_submit_urb`.

```
struct urb * usb_alloc_urb(int iso_frames)
```

Pour allouer une structure `URB`.

```
usb_rcvintpipe(* struct usb_device udev ,int
endpointadresse)
```

Pour créer un pipe supportant le transfert de type interruption entre le périphérique logique `udev` et le endpoint d'adresse `endpointadresse`.

```
FILL_INT_URB(      struct urb *purb,
struct usb_device *udev,
                   int pipe,
                   (void*) tampon,
int size,
(* fonction_de_rappel),
void *context),
                    int interval)
```

Cette macro permet de remplir une structure `urb` qui spécifiera au HCD que le type de transfert sera "interruption".

Le périphérique concerné est pointé par `udev`, l'`urb` utilisé pour les transfert de données est pointé par `purb`. `pipe` représente le *pipe* que l'on aura créé par la macro précédente. `Fonction_de rappel` est un pointeur vers une fonction de prototype `void fonction_de_rappel (struct urb *purb)`.

Cette fonction sera appelée après chaque interruption spécifiant l'achèvement d'un transfert de données du périphérique vers l'hôte. (Elle doit notamment respecter les contraintes spécifiques au code appelé dans un contexte d'interruption).

`Interval` représente l'intervalle en ms entre les interruptions. `tampon` est pointeur sur une zone de mémoire non "swappable" qui contient les données transférées depuis le périphérique. (Une version de cette macro existe pour chaque type de transfert : `FILL_BULK_URB`, `FILL_ISO_URB`...).

```
usb_inc_dev(struct usb_driver *dev), usb_dev_dev(struct
usb_driver *dev)
```

Incrémente et décrémente le compteur d'utilisation des modules du HCD.

Les autres fonctions et structures utilisées ne sont pas spécifiques à l'USB :

`MOD_INC_USE_COUNT, MOD_DEC_USE_COUNT`

Incrémentation et décrémentation du compteur d'usage du module. (colonne `used` de `lsmod`). Empêche le déchargement d'un driver lorsque celui-ci est en cours d'utilisation.

`wait_queue_head_t, init_waitqueue_head( wait_queue_head_t *p), interruptible_sleep_on(wait_queue_head_t *p), wake_up_interruptible (wait_queue_head_t *p)`

Structure et fonctions permettant d'endormir et de réveiller un processus sur une file d'attente.

`module_init, module_exit`

Fixe les fonctions d'initialisation et de destruction du driver.

Notre premier driver

Ces précisions étant faites, nous allons pouvoir passer à l'écriture proprement dite de notre pilote de souris USB.

Au niveau des fonctionnalités, notre pilote implémentera seulement une lecture en mode bloquant.

La première tâche à effectuer est la découverte des informations de configuration de la souris USB dont on veut écrire le pilote. Je branche ma souris sur un port USB. `usbview` me donne alors les informations (**Fig. 2**) :

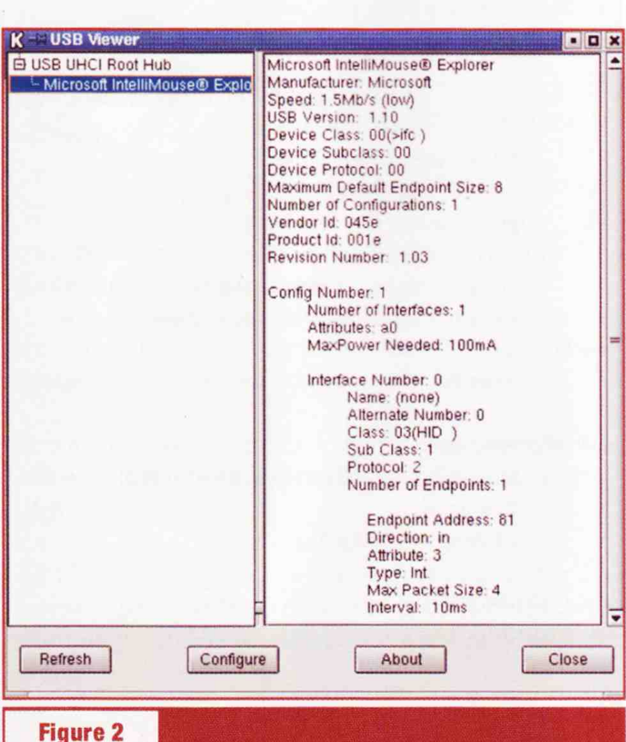


Figure 2

`usbview` nous indique l'identifiant vendeur et l'identifiant constructeur, ce qui permettra à la fonction `probe` de reconnaître le périphérique que nous voulons prendre en charge.

On apprend aussi que l'on communiquera avec le endpoint d'adresse `0x81`, sur l'interface zéro avec un type de transfert `interruption`, une taille maximale de paquet de quatre octets, etc.

Pour stocker toutes les données nécessaires au pilote, on utilisera notre propre structure : `Struct LMAG_priv`. Cette structure sera dynamiquement allouée à la fonction d'initialisation du pilote.

L'implémentation de la lecture bloquante est réalisée comme suit :

- Un processus qui veut lire des données provenant du périphérique est mis en sommeil sur une file d'attente.
- Une interruption est déclenchée lorsque les données provenant du périphérique sont accessibles. Cette interruption est prise en charge par un gestionnaire d'interruption dont le rôle est d'ordonnancer une tâche permettant de réveiller, en dehors de ce contexte d'interruption, les processus endormis sur la file d'attente. Ceci est réalisé par l'intermédiaire d'une structure `tasklet`.
- Lorsqu'un processus lecteur est réveillé, cela signifie que des données provenant de la souris sont accessibles. Il ne reste alors qu'à les rendre accessibles dans le contexte de l'utilisateur.

L'accès au périphérique sera effectué grâce à un fichier spécial de numéro majeur 183 (pour tous les périphériques USB) créé par :

```
mknod  LMAG_mouse c 183 251
```

par exemple (si le numéro mineur 251 n'est pas utilisé par un autre périphérique USB).

CODE SOURCE

```
#include<linux/module.h>
#include<linux/kernel.h>
#include<linux/init.h>
#include<linux/usb.h>
#include<linux/malloc.h>
#include<linux/sched.h>
#include<asm/uaccess.h>

struct LMAG_priv { /** structure de données privées **/
        char mess[20] ;/** pour stocker les données de la souris **/
        struct urb  *lmagurb;
        struct usb_device *udev;
        wait_queue_head_t readwait; /** file d'attente lecture **/
        int plugged; /** souris branchée **/
        int opened; /** souris ouverte **/
};
```

```
static struct LMAG_priv *ppriv =NULL;

void LMAG_interruption2( unsigned long param) {
        wake_up_interruptible(&(ppriv->readwait));}/** reveil des
processus lecteur **/

DECLARE_TASKLET(int_tasklet,LMAG_interruption2,0);

void LMAG_interruption1 (struct urb *ptr) {/** fonction de gestion des
interruptions **/
        tasklet_schedule(&int_tasklet);}/** ordonnacement de la fonc-
tion de gestion **/

static void *LMAG_probe (struct usb_device *udev,unsigned ifnum, const
struct usb_device_id *prod){
        printk(KERN_DEBUG" LMAG_driver: entrée dans probe\n");
        if(udev->descriptor.iManufacturer != 1 || udev-
>descriptor.idVendor != 1118)
                return 0;
        usb_inc_dev_use(udev);
        if ((ppriv->lmagurb=usb_alloc_urb(0))==0) {
                printk(KERN_DEBUG" echeC ALLOCATION URB\n");}
        ppriv->udev=udev;
        (ppriv->plugged)++;
        return (ppriv);}

int LMAG_open(struct inode *inode,struct file *filp){
        int res;
        if(!ppriv->plugged==0) return -ENODEV;
        if (ppriv->opened==0 ){/** installation du mode de communica-
tion par d'interruption     ******/
                FILL_INT_URB(ppriv->lmagurb,
                        ppriv->udev,
                    usb_rcvintpipe(ppriv->udev,0x81),
                        (void*) ppriv->mess,
                        4,
                        LMAG_interruption1,
                        ppriv,
                    0x10);
                if ((res=usb_submit_urb((ppriv->lmagurb)))!=0) {
                        printk(KERN_DEBUG" LMAG_driv:Unable to
allocate INT URB.erreur :%i",res);
                return -1;}
                else printk(KERN_DEBUG" LMAG_driv: allocate INT
URB.ok:\n");}
        MOD_INC_USE_COUNT ;
        (ppriv->opened)++;
        return 0;}

static void LMAG_disconnect(struct usb_device *udev, void *ptr){
        int res;
        printk(KERN_DEBUG" LMAG_driver: entrée dans disconnect\n");
        wake_up_interruptible(&(ppriv->readwait));
        (ppriv->plugged)-;
        if(ppriv->opened !=0 ) {
                        if((res=usb_unlink_urb(ppriv->lma-
gurb))!=0)
                                        printk(KERN_DEBUG"
LMAG_driv: echec unlink urb:\n");}
        ppriv->opened=0;
        usb_free_urb(ppriv->lmagurb);
        usb_dec_dev_use(udev);}

int LMAG_release(struct inode *inode,struct file *filp){
        printk(KERN_DEBUG" LMAG_driver: entrée dans release\n");
        ppriv->opened-;
        if(ppriv->opened ==0)
                usb_unlink_urb(ppriv->lmagurb);
        MOD_DEC_USE_COUNT;
        return 0;}

ssize_t LMAG_read(struct file *fp,char * buf, size_t count,loff_t *pos){
        interruptible_sleep_on(&(ppriv->readwait));
        if(copy_to_user(buf,ppriv->mess,ppriv->lmagurb-
>actual_length))
                return -EFAULT;
        return (ppriv->lmagurb->actual_length);}

struct file_operations LMAG_fops= {
        open:LMAG_open,
        release:LMAG_release,
        read:LMAG_read};

static struct usb_driver LMAG_usb={
        name : "LMAG_usb",
        probe: LMAG_probe,
        disconnect:LMAG_disconnect,
        fops:&LMAG_fops,
        minor:251};

int LMAG_init (void) {
        if(!(ppriv=(struct LMAG_priv*)kmalloc(sizeof *ppriv,GFP_ATO-
MIC))) {
                printk(KERN_DEBUG" echeC ALLOCATION LMAG_priv\n");
                return -1;}
        if(usb_register(&LMAG_usb) <0){
                return (-1);        }
        ppriv->opened=0;
        ppriv->plugged=0;
        init_waitqueue_head (&(ppriv->readwait));
        printk(KERN_DEBUG" LMAG_driver: enregistrement du driver
OK\n");
        return 0;}

void LMAG_cleanup (void) {
        printk(KERN_DEBUG" LMAG_driver: liberation du pilote\n");
        kfree(ppriv);
        usb_deregister(&LMAG_usb);}

module_init(LMAG_init);
module_exit(LMAG_cleanup);

/***********fin du driver*******************/
```

Le makefile

```
KERNELDIR=/usr/src/linux

include ${KERNELDIR}/.config

CFLAGS=-D__KERNEL__ -DMODULE -I${KERNELDIR}/include -O -Wall

ifdef CONFIG_SMP
 CFLAGS+=-D__SMP__ -DSMP
endif

all:LMAG_driver.o


LMAG_driver.o: LMAG_driver.c
        gcc ${CFLAGS} -c LMAG_driver.c -o LMAG_driver.o

clean :
        rm -f LMAG_driver.o
```

Fonctionnement

Après compilation (utiliser le `Makefile`), nous obtenons le fichier `LMAG_driver.o`, code objet du pilote de notre souris. On crée le fichier spécial `LMAG_mouse` (ou tout autre nom) par la commande

```
mknod  LMAG_mouse c 183 251
```

Pour éviter qu'un driver pré-existant soit directement chargé par le système lors du branchement de la souris, il faut "killer" le démon `usbd` s'il est présent sur la machine (`Killall usbd`).

REMARQUES IMPORTANTES :

Afin de le simplifier, le driver a été écrit spécifiquement pour la souris que j'ai utilisée (une Microsoft IntelliMouse Explorer) puisque ses données de configurations (`idvendor`, `endpointaddress`...) ont été codées en dur dans le driver.

Un véritable driver devrait lire les informations de configuration (comme `usbview`) et s'y adapter.

Si vous en avez compris le fonctionnement, vous pouvez toutefois l'adapter sans peine à votre souris USB.

Un véritable pilote de souris devrait travailler les données provenant de la souris pour les présenter selon un protocole prédéfini (`man mouse` pour plus de détails).

On charge le driver par `insmod LMAG_driver.o` et on le décharge par `rmmod LMAG_driver`. On peut indifféremment brancher la souris avant de charger le driver ou charger le driver puis brancher la souris. Si la souris USB est branchée, on peut lire les données qu'elle envoie en ouvrant le fichier spécial `LMAG_mouse`. Par exemple :

```
Od -x -v -w4 < LMAG_mouse
```

affiche par groupe de quatre octets en hexadécimal les données de la souris.

La commande `dmesg` permet de voir les différents messages de débogage envoyés par le pilote et de tracer le passage dans les différentes fonctions.

Xavier Montrichard
xavier.montrichard@libertysurf.fr

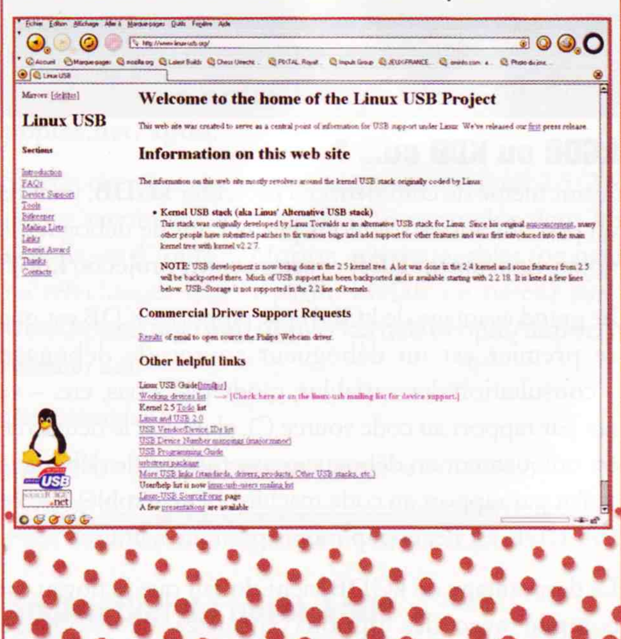
Bibliographie et liens

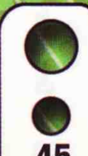
www.linux-usb.org

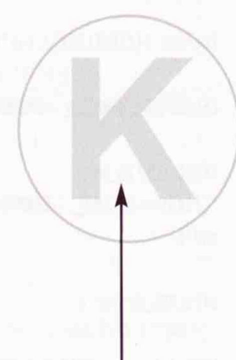
Le noyau Linux, Bovet & Cesati (Editions O'Reilly)

Linux device drivers, Alessandro Rubini (Editions O'Reilly)

URB.txt dans la documentation du noyau.







Débogage du noyau Linux avec kGDB

Dans le hors série précédent, en conclusion de son article dédié à l'introduction à la programmation noyau sous Linux, *Philippe Biondi* passait rapidement en revue les différentes techniques permettant de déboguer le noyau Linux.
L'une de ces techniques, kGDB, mérite que l'on s'y intéresse de plus près, car il s'agit d'une méthode largement utilisée par certains des développeurs noyau (dont par exemple *Andrew Morton*, le très probable futur mainteneur du prochain noyau stable 2.6).

kGDB ou KDB ou... ?

Avant même de commencer à présenter **kGDB**, parlons un peu des autres solutions permettant de déboguer le noyau, notamment **KDB : oss.sgi.com/projects/kdb/**.

Le grand avantage de kGDB par rapport à KDB est que ce premier est un débogueur source (le débogage – consultation des variables, mode pas à pas, etc. – se fait par rapport au code source C), alors que le deuxième est uniquement un débogueur assembleur (le débogage se fait par rapport au code machine désassemblé). De ce fait, KDB est beaucoup moins pratique à utiliser.

Le désavantage de kGDB vient du fait que le noyau est compilé avec des options spéciales de débogage (utilisation de l'option `-g` et compilation avec *frame pointers*), ce qui aura des effets en termes d'empreinte mémoire et de performances, pouvant même aller jusqu'à modifier la dynamique du système et empêcher la reproduction des bogues que l'on recherche.

KDB, lui, ne nécessite aucune option spécifique de compilation et ne consomme pas ou peu de ressources quand il est présent.

Un autre désavantage de kGDB est son besoin d'une infrastructure assez lourde (deux machines reliées par un câble série, comme on le verra un peu plus loin), alors que KDB est embarqué directement sur la machine à déboguer (le débogueur est intégré dans le gestionnaire d'interruptions du clavier).

Pour toutes ces raisons, les développeurs noyau s'accordent pour dire que kGDB est une solution recommandée lors du développement, mais que l'on utilisera plutôt KDB pour le débogage d'une machine en production.

Mentionnons cependant qu'il existe d'autres solutions permettant de déboguer le noyau Linux. Il peut être intéressant de les explorer, afin de choisir celle la mieux adaptée à votre besoin.

Parmi ces solutions, on trouve **DProbes** (www-124.ibm.com/linux/projects/dprobes), **Kprobes** (www-124.ibm.com/linux/projects/kprobes), **UML** (user-mode-linux.sourceforge.net), **Linux Kernel Crash Dump** (lkcd.sourceforge.net), **Linux Trace Toolkit** (www.opersys.com/LTT/index.html), **OProfile** (oprofile.sourceforge.net), etc.

Préparation des machines et du lien série

Pour utiliser kGDB, il faut disposer de deux machines : la *machine de développement* et la *machine cible* (souvent appelée par son nom anglais, la *target*).

La *machine de développement* est, comme son nom l'indique, celle sur laquelle on effectue le développement : la modification du code source et sa compilation.

Cette dernière peut éventuellement être croisée (en utilisant un ensemble d'outils de *cross-compilation*) lorsque

l'architecture pour laquelle on développe (l'architecture de la *cible*) n'est pas la même que l'architecture de la *machine de développement*. (Cependant, dans la suite de cet article, les exemples seront basés sur une architecture *x86*.)

La *machine de développement* et la *cible* doivent être reliées par un câble série de type *null-modem*. (Il est donc nécessaire de disposer d'au moins un port série sur chacune des deux machines, chose qui devient cependant de plus en plus rare aujourd'hui, les ports série étant en train d'être abandonnés au profit de ports USB ou IEEE1394.

Heureusement, des travaux sont en cours pour rendre kGDB utilisable sans devoir passer obligatoirement par un lien série, tel qu'on le verra un peu plus tard).

Les machines ayant été mises en place, la prochaine étape est de tester que le lien série fonctionne correctement (il vaut mieux prendre quelques dizaines de secondes pour le tester maintenant plutôt que de perdre plusieurs dizaines de minutes plus tard, en essayant de faire marcher kGDB sans succès, avant de découvrir que le câble est défectueux...).

On peut faire le test simplement, sans utiliser de logiciel particulier. Sur la première machine, faites (en supposant que vous utilisez le premier port série, d'où `/dev/ttyS0`, sinon remplacez `ttyS0` par `ttyS1` pour le deuxième port série) :

```
$ stty -F /dev/ttyS0 9600
$ cat /dev/ttyS0
```

Et sur la deuxième :

```
$ stty -F /dev/ttyS0 9600
$ echo "foobar" > /dev/ttyS0
```

Si vous voyez le message apparaitre sur la première machine, et que vous pouvez également refaire l'opération dans l'autre sens, alors la liaison est proprement configurée et on peut passer à la suite. Sinon, bidouillez les câbles jusqu'à ce que ça marche :)

Téléchargement du patch kGDB

Le noyau officiel standard ne contient pas par défaut le code permettant son débogage par un lien série.

Pour être plus précis, certaines architectures disposent du code kGDB (parmi ces architectures, on peut noter MIPS, PPC, SH), mais pas l'architecture x86, qui fait principalement l'objet de cet article.

La raison de ce manque est le fait que Linus Torvalds n'a jamais aimé les débogueurs, considérant que lorsque l'on utilise un débogueur on aura plutôt tendance à corriger les effets des bogues qu'à essayer de comprendre le code d'abord et corriger les causes des bogues ensuite.

De ce fait, il a toujours refusé d'inclure le *patch* kGDB dans le noyau officiel, en tout cas dans sa version x86 (architecture qui est directement maintenue par lui).

D'autres mainteneurs d'architectures ont un avis différent, et c'est pour cela que certaines architectures supportent kGDB sans modifications, alors que pour l'x86 il faut appliquer un patch supplémentaire. Le patch kGDB pour les architectures x86 a été écrit initialement par Dave Grothe.

Plusieurs développeurs ont contribué à son écriture par la suite, et il est actuellement maintenu par Amit S. Kale qui le propose en téléchargement sur le site **kgdb.sourceforge.net**.

Hélas, Amit S. Kale ne propose pas de patch pour tous les noyaux récents, mais uniquement pour certains d'entre eux (le patch pour le noyau RedHat 2.4.20-18.7 est le dernier en date).

Bien qu'il soit possible, avec relativement peu de difficultés, d'adapter ce patch pour n'importe quelle version du noyau, il peut être plus facile de télécharger un patch déjà mis à jour sur mon site personnel, situé à **popies.net/kgdb**.

Si vous cherchez une version adaptée aux noyaux 2.5/2.6, il faut savoir que le patch kGDB est inclus dans les noyaux `-mm` d'Andrew Morton, téléchargeables (on peut ne télécharger que la partie kGDB du patch) sur : **www.kernel.org/pub/linux/kernel/people/akpm/patches/2.6**.

Par exemple :

```
$ wget http://www.kernel.org/pub/linux/kernel/v2.4/linux-2.4.22.tar.bz2
$ wget http://popies.net/kgdb/kgdb-1.6-for-2.4.22.patch.bz2
```

Application du patch kGDB

Une fois les sources du noyau et le patch kGDB téléchargés, on peut appliquer ce dernier facilement (pour des raisons de clarté, une fois les sources du noyau extraites, on renommera le répertoire les contenant pour prendre en compte le fait que l'on compile un noyau modifié par kGDB) :

```
$ bzcat linux-2.4.22.tar.bz2 | tar xf -
$ mv linux-2.4.22 linux-2.4.22-kgdb
$ cd linux-2.4.22-kgdb
$ bzcat ../kgdb-1.6-for-2.4.22.patch.bz2 | patch -p1
```

```
patching file arch/i386/config.in
patching file arch/i386/kernel/entry.S
patching file arch/i386/kernel/gdbstart.c
patching file arch/i386/kernel/gdbstub.c
patching file arch/i386/kernel/Makefile
patching file arch/i386/kernel/nmi.c
patching file arch/i386/kernel/signal.c
patching file arch/i386/kernel/traps.c
patching file arch/i386/Makefile
patching file arch/i386/mm/fault.c
patching file Documentation/Configure.help
patching file Documentation/i386/gdb-serial.txt
patching file Documentation/sysrq.txt
patching file drivers/char/gdbserial.c
patching file drivers/char/Makefile
patching file drivers/char/serial.c
patching file drivers/char/sysrq.c
patching file drivers/char/tty_io.c
patching file include/asm-i386/ioctls.h
patching file include/asm-i386/page.h
patching file include/asm-i386/processor.h
patching file include/linux/dcache.h
patching file include/linux/gdb.h
patching file include/linux/sched.h
patching file init/main.c
patching file kernel/ksyms.c
patching file kernel/sched.c
patching file Makefile
```

On peut d'ores et déjà remarquer que le patch kGDB ne modifie qu'un petit nombre de fichiers, et que, parmi ces modifications, un bon nombre sont triviales.

Fonctionnellement, le patch est composé de plusieurs parties :

- Des modifications dans le code du noyau à plusieurs endroits clés (*handlers* des exceptions du processeur, ordonnanceur, traitement des séquences `sysrq`) afin de pouvoir activer kGDB et effectuer les traitements demandés lors du débogage, notamment l'exécution pas à pas ;
- Un driver simplifié pour piloter un port série (dans `drivers/char`) ;
- L'implémentation du protocole GDB pour le débogage série (dans `arch/i386/kernel/gdbstub.c`) ;
- Un utilitaire, `gdbstart` permettant de paramétrer et d'activer le débogage si celui-ci est lancé sur la machine *cible* (dans `arch/i386/kernel/gdbstart`) ;
- Des modifications dans le système de configuration et de compilation du noyau (ajout des nouveaux fichiers, ajout de l'option `-g` et enlèvement de l'option `-fomit-frame-pointers`) permettant de prendre en charge les nouvelles fonctionnalités ;
- De la documentation (dans `Documentation/i386/gdb-serial.txt`).

Le patch kGDB modifie aussi la version du noyau, en ajoutant `-kgdb` à la fin de la version (en obtenant donc une version `2.4.22-kgdb` dans notre exemple), nous permettant d'installer facilement le noyau modifié par kGDB à côté d'un noyau non modifié.

Configuration du noyau patché

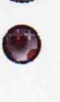
Le noyau étant patché, on peut maintenant le configurer :

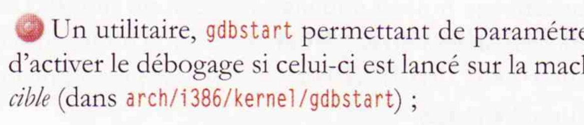
```
$ make config
...
*
* Kernel hacking
*
KGDB: Remote (serial) kernel debugging with gdb (CONFIG_X86_REMOTE_DEBUG) [N/y/?] (NEW) y
KGDB: Thread analysis (CONFIG_KGDB_THREAD) [N/y/?] (NEW) y
KGDB: Console messages through gdb (CONFIG_GDB_CONSOLE) [N/y/?] (NEW) y
...
```

Lors de la configuration, comme on le voit plus haut, trois nouvelles questions sont posées par l'outil :

- `CONFIG_X86_REMOTE_DEBUG` : cette option détermine si le code kGDB va être compilé dans le noyau. En répondant *non* à cette question, on obtiendrait le même noyau que si l'on n'avait même pas appliqué le patch. Bien évidemment, on va donc répondre *oui* ;
- `CONFIG_KGDB_THREAD` : cette option permet, lors du débogage, d'utiliser les commandes GDB prévues pour les *threads* (comme par exemple `info threads`, `thread N`) pour consulter des informations sur les processus tournant sur la *cible*. L'activation de cette option va cependant induire un petit surcoût dans l'ordonnanceur ;
- `CONFIG_GDB_CONSOLE` : cette option va nous permettre d'utiliser une fonctionnalité intéressante de GDB : en plus des commandes de débogage, GDB peut faire passer des messages supplémentaires par la ligne série et les afficher dans la fenêtre.

L'activation de cette option permettra de faire passer les messages de la console du noyau par GDB et donc de les afficher dans la même fenêtre que celle où l'on fait le débogage. C'est particulièrement utile lorsque, comme c'est souvent le cas, la *cible* est dépourvue d'écran...





Compilation du noyau

Il ne nous reste plus qu'à lancer la compilation du noyau :

```
$ make dep bzImage modules
```

...et à aller boire un petit café (selon la puissance de la machine de chacun et des options choisies lors de la configuration du noyau, il peut s'agir d'un tout petit café ou bien d'un café double assorti d'un croissant :)).

Au bout de quelques minutes, la compilation sera finie et l'on pourra passer à l'étape d'installation.

Installation du noyau

Le choix de la meilleure façon d'installer le noyau sur la machine *cible* dépend beaucoup du type de la cible : dans le cas d'une carte embarquée, on transférera le noyau par le même lien série que celui qui servira au débogage, en utilisant un chargeur particulier.

Lorsque la *cible* dispose d'une carte réseau, on préférera son utilisation, car on peut ainsi atteindre des vitesses de transfert beaucoup plus élevées.

L'une des façons qui permettent d'installer le noyau sur la *cible* lorsque l'on dispose d'une connectivité réseau est d'avoir un serveur NFS sur la *machine de développement*.

Le serveur exporte le répertoire contenant les sources (compilées) du noyau, on monte ce répertoire sur la *cible*, et on effectue l'installation directement à partir de la *cible*:

```
$ ssh root@cible
cible> mount dev:/home/stelian/linux-2.4.22-kgdb /mnt
cible> cd /mnt
cible> cp arch/i386/boot/bzImage /boot/vmlinuz-2.4.22-kgdb
cible> make modules_install
```

On peut en profiter pour installer aussi `gdbstart`, l'utilitaire de paramétrage/activation qui a été compilé en même temps que le noyau (cette installation n'est à faire qu'une seule fois, car le binaire `gdbstart` est indépendant de la version de noyau compilée et on peut utiliser le même pour les prochains noyaux) :

```
cible> cp arch/i386/kernel/gdbstart /sbin/
```

Si l'on ne peut pas se connecter à la *cible* par un répertoire partagé par NFS, il faut d'abord effectuer l'installation du noyau et des modules sur la *machine de développement* dans un répertoire temporaire :

```
$ mkdir /tmp/kgdb
$ mkdir /tmp/kgdb/boot
$ cp arch/i386/boot/bzImage /tmp/kgdb/boot/vmlinuz-2.4.22-kgdb
$ make modules_install INSTALL_MOD_PATH=/tmp/kgdb
```

puis transférer, d'une façon ou d'une autre, l'intégralité du contenu du répertoire `/tmp/kgdb` sur la *machine cible*.

A partir de ce moment, le noyau kGDB et ses modules ont été correctement installés sur la machine *cible*. Il ne reste plus qu'à modifier le gestionnaire de démarrage de la machine (`lilo` ou `grub`) afin d'ajouter le noyau fraîchement installé dans le menu de démarrage de la machine.

Lors du redémarrage de la *cible* sur le nouveau noyau, on ne verra rien d'exceptionnel, car kGDB n'est pas actif initialement.

Paramétrage et activation de kGDB

Sur la *cible*, les paramètres par défaut de kGDB sont :

- Le port utilisé est le deuxième port série (`/dev/ttyS1`) ;
- La vitesse de ce port est initialisée à 38400 bps.

Si l'on veut changer ces paramètres, on peut le faire :

- Soit au démarrage du noyau, en rajoutant les paramètres suivants sur la ligne de commande du noyau :

```
gdbttyS=0 gdbbaud=9600
```

- Soit une fois que le noyau est démarré, en utilisant `gdbstart` :

```
cible> gdbstart -t /dev/ttyS0 -s 9600
```

(à la différence que cette deuxième méthode activera aussi kGDB tout de suite, ce qui peut ne pas être l'action désirée).

Lorsque les paramètres de kGDB correspondent à notre installation, on peut commencer à l'utiliser. L'activation de kGDB (c'est-à-dire l'interruption du noyau de la *cible* en donnant le contrôle au GDB tournant sur la *machine de développement*) peut se faire de plusieurs façons :

- Si une erreur quelconque se produit en mode noyau, kGDB est automatiquement activé ;
- L'utilitaire `gdbstart` permet de faire une activation manuelle ;
- Une autre activation manuelle peut être effectuée en faisant la combinaison de touches *SysRq* + *g* sur la console (sur un PC, cela se traduit par la séquence de touches *AltGr* + *Print Screen* + *g*) ;
- L'envoi du caractère *Control* + *C* sur le port série active aussi le débogueur :

```
$ echo -e "\003" > /dev/ttyS0
```

- Enfin, si l'on souhaite démarrer la machine avec kGDB déjà activé (avec l'avantage que l'on peut déboguer du code s'exécutant très tôt lors de la procédure de démarrage), il suffit de rajouter le paramètre `gdb` sur la ligne de commande du noyau.

kGDB pourra donc être activé sur la machine *cible* à tout moment, à condition que les interruptions soient actives. Si la machine est bloquée et que les interruptions sont désactivées, il n'y a aucun moyen logiciel permettant de la réveiller.

La seule façon possible de le faire est d'attaquer directement le matériel et de générer une interruption NMI. Comme son nom l'indique, une *Non Maskable Interrupt* sera prise en compte par le processeur même si celui-ci se trouve dans une section de code où les interruptions ont été désactivées, et cela réveillera kGDB.

La génération d'une NMI peut être réalisée avec plus ou moins de matériel électronique ; on peut par exemple le faire avec un simple trombone sur les cartes mères ayant au moins un *slot* ISA en court-circuitant les *pins* A1 et B1 de ce slot !

Lancement de GDB

Lorsque kGDB a été activé, la *machine cible* se retrouve complètement bloquée, en attendant d'être pilotée par un GDB distant au moyen du lien série.

Sur la *machine de développement*, on va donc lancer `gdb` en lui passant en paramètre le binaire `vmlinux` se trouvant à la racine des sources du noyau. `vmlinux` est le même noyau que celui se trouvant dans le fichier `arch/i386/boot/bzImage` et que l'on a installé sur la *cible*, le premier étant le binaire ELF alors que le deuxième contient une version compressée de ce binaire ainsi que du code permettant le chargement initial en mémoire de ce noyau.

Une fois `gdb` lancé, on va utiliser les commandes `set remotebaud` et `target remote` pour dire à `gdb` que le binaire ne s'exécute pas localement mais à distance :

```
$ gdb vmlinux
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you
are welcome to change it and/or distribute copies of it under certain
conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for
details.
This GDB was configured as "i386-redhat-linux-gnu"...
(gdb) set remotebaud 9600
(gdb) target remote /dev/ttyS0
Remote debugging using /dev/ttyS0
breakpoint () at gdbstub.c:1344
1344        }
warning: shared library handler failed to enable breakpoint
```

Bingo ! `gdb` a bien réussi à se connecter à la *cible*, qui est interrompue à la ligne 1344 de `gdbstub.c`, en attendant cette connexion.

Le message de *warning* de `gdb` concernant les bibliothèques partagées n'est pas important (après tout, le noyau que l'on débogue n'utilise pas de bibliothèques partagées !) et peut être ignoré.

Afin de simplifier les futurs lancements, on pourra créer, à la racine des sources du noyau, le fichier `.gdbinit` qui sera lu et évalué par `gdb` à chaque lancement, et qui contiendra les deux lignes de paramétrage du lien série.

On peut en profiter pour rajouter dans ce même fichier d'autres macro-commandes utiles, comme la macro `ps` permettant de connaitre la liste des processus, `lsmod` qui affiche la liste des modules chargés, ainsi que des macros qui seront utilisées pour poser des points d'arrêt matériels (et que l'on utilisera un peu plus loin dans cet article).

Toutes ces macros sont téléchargeables sur le site de kGDB : **kgdb.sourceforge.net/gdbmacros.html**.

Il peut être utile de mentionner que l'utilisation d'un *front-end* graphique à `gdb`, comme par exemple `ddd`, est tout à fait possible et compatible avec toutes les opérations que l'on va faire.

Maintenant que la *cible* est bien sous le contrôle de `gdb`, on peut la débloquer en utilisant la commande `cont` :

```
(gdb) cont
Continuing.
```

Lorsque l'on veut reprendre la main dans le débogueur, il suffit de taper *Control* + *C* :

```
^C
Program received signal SIGTRAP, Trace/breakpoint trap.
breakpoint () at gdbstub.c:1344
1344    }
(gdb)
```

Si on veut arrêter le débogage et donc quitter `gdb` tout en laissant le noyau de la *cible* en train de tourner, on se retrouve devant un petit dilemme : si on utilise *Control* + *C* pour reprendre la main dans `gdb` afin de quitter correctement, le noyau de la *cible* sera laissé bloqué.

La solution est de quitter `gdb` en tapant *Control* + *Z*, puis, une fois revenu au *shell*, tuer le processus `gdb` :

```
(gdb) cont
Continuing.
^Z
[1]+  Stopped                 gdb vmlinux
$ kill %1
```

```
[1]+ Stopped                  gdb vmlinux
[1]+ Terminated               gdb vmlinux
```

On peut bien entendu par la suite relancer `gdb` à tout moment et reprendre la main sur le noyau de la cible en utilisant la même procédure que pour l'activation initiale.

De temps en temps, il arrive aussi que `gdb` devienne un peu confus quant au fil de l'exécution, et dans ce cas il est préférable de le quitter et de le relancer...

Utilisation de kGDB

L'utilisation de kGDB sur le noyau ressemble beaucoup à l'utilisation de GDB sur un programme standard. On peut consulter la valeur des variables, poser des points d'arrêt, tracer l'exécution en mode pas à pas, connaître la pile d'appel à un instant donné, etc.

Par rapport à ce type d'utilisation standard, kGDB apporte quelques fonctionnalités avancées, comme l'utilisation des points d'arrêts *hardware* qui permettent de surveiller une adresse mémoire par le processeur, afin de détecter des accès en lecture ou écriture à cette adresse, ou bien la possibilité de consulter la pile d'appel de chaque processus ordonnancé par le noyau.

Dans la suite de cet article, nous allons présenter rapidement quelques-unes de ces fonctionnalités au travers d'exemples réels d'exécution. Pour une documentation plus complète sur les commandes `gdb`, veuillez vous rapporter au manuel `info`, très complet, de `gdb`.

Consultation du source et des données

La consultation du code source peut se faire à tout moment grâce à la commande `list` de `gdb`. Sans argument, cette commande affiche le code source correspondant à l'endroit où le pointeur d'exécution se trouve, mais on peut sélectionner un autre fichier source, une autre fonction ou un autre numéro de ligne en les passant en paramètre à la commande `list` :

```
^C
Program received signal SIGTRAP, Trace/breakpoint trap.
breakpoint () at gdbstub.c:1344
1344    }
(gdb) list
1339
1340    void breakpoint(void)
1341    {
1342            if (initialized)
1343                    BREAKPOINT();
1344    }
1345
1346    #ifdef CONFIG_GDB_CONSOLE
1347    char    gdbconbuf[BUFMAX];
1348
(gdb) list sched.c:0
1    /*
2     *  linux/kernel/sched.c
3     *
4     *  Kernel scheduler and related syscalls
5     *
6     *  Copyright (C) 1991, 1992  Linus Torvalds
7     *
8     *  1996-12-23  Modified by Dave Grothe to fix bugs in semaphores and
9     *              make semaphores SMP safe
10    *  1998-11-19  Implemented schedule_timeout() and related stuff
(gdb) list sys_sethostname
1041            up_read(&uts_sem);
1042            return errno;
1043    }
1044
1045    asmlinkage long sys_sethostname(char *name, int len)
1046    {
1047            int errno;
1048            char tmp[__NEW_UTS_LEN];
1049
1050            if (!capable(CAP_SYS_ADMIN))
(gdb)
```

La consultation des variables peut se faire en utilisant la commande `print` :

```
(gdb) print system_utsname
$1 = {sysname = "Linux", '\0' <repeats 59 times>,
  nodename = "cible", '\0' <repeats 59 times>,
  release = "2.4.22-kgdb", '\0' <repeats 53 times>,
  version = "#2 Wed Oct 15 13:22:54 CEST 2003", '\0' <repeats 32 times>,
  machine = "i686", '\0' <repeats 60 times>,
  domainname = "(none)", '\0' <repeats 58 times>}
(gdb) print system_utsname.nodename
$2 = "cible", '\0' <repeats 59 times>
```

La commande `x` (abréviation de *examine*) permet de consulter une zone mémoire, en précisant la taille de cette zone, le format des données à lire et le format à utiliser à l'affichage. Par exemple, pour afficher en format

hexadécimal les 8 premiers octets se trouvant à l'adresse de la variable `system_utsname.nodename` on fera :

```
(gdb) x/8xb system_utsname.nodename
0xc0269a21 <system_utsname+65>: 0x63    0x69    0x62    0x6c
0x65    0x00    0x00    0x00
```

La commande `print` peut aussi servir à modifier dynamiquement une variable. Dans l'exemple suivant, nous allons modifier le nom d'hôte de la *cible* :

```
(gdb) print system_utsname.nodename="target"
$3 = "target", '\0' <repeats 58 times>
```

Par la suite, toute application effectuant l'appel système `gethostname` aura en retour le nouveau nom d'hôte.

Points d'arrêt

Les points d'arrêt peuvent être posés à n'importe quel endroit du code en utilisant la commande `break`, à l'exception de :

- La phase initiale de démarrage du noyau (correspondant à l'initialisation de la plate-forme, la mise en place des *mappings* mémoire, l'initialisation de tous les sous-systèmes et pilotes du noyau, etc.). Si l'on veut cependant activer kGDB le plus tôt possible lors du démarrage du noyau, on peut utiliser l'option `gdb` sur la ligne de commande du noyau comme précisé plus haut.
- Les parties du noyau utilisées par kGDB pour sa gestion interne (le driver série simplifié, l'implémentation du protocole de kGDB, le code permettant d'intercepter les exceptions et interruptions à destination de kGDB).

Par exemple, on peut mettre un point d'arrêt sur l'implémentation de l'appel système `stat64`, puis l'effacer une fois qu'il sera atteint par l'un des processus ordonnancés sur la *cible*:

```
(gdb) break sys_stat64
Breakpoint 1 at 0xc0143c6c: file stat.c, line 339.
(gdb) cont
Continuing.

Breakpoint 1, sys_stat64 (filename=0x804dc61 "/dev/initctl",
    statbuf=0x804dc61, flags=2) at stat.c:339
339             error = user_path_walk(filename, &nd);
(gdb) delete 1
```

Mode pas à pas

Le mode pas à pas est effectué grâce aux commandes `step` et `next` de `gdb`. La différence entre les deux est que, si la prochaine ligne du code source est un appel de fonction, la commande `step` exécutera en mode pas à pas la fonction appelée, alors que la commande `next` exécutera l'appel en un seul pas et s'arrêtera sur la ligne de code suivant l'appel de la fonction.

L'exécution en mode pas à pas peut être relativement difficile à suivre parfois, car le noyau est compilé en mode optimisé (et donc le compilateur a quelques fois réarrangé l'ordre les lignes du code source lors de la génération du code), et l'on utilise très souvent des fonctions *inline* ou des macro-commandes...

En reprenant l'exemple où l'on avait posé un point d'arrêt dans l'appel système `stat64`, on peut tracer cette fonction en mode pas à pas (et remarquer le déroulement de la trace, influencé à la fois par l'optimisation du code et par l'exécution de la fonction inline `kdev_t_to_nr`) :

```
339             error = user_path_walk(filename, &nd);
(gdb) next
340             if (!error) {
...
(gdb) next
343                     error = cp_new_stat64(nd.dentry->d_inode,
statbuf);
(gdb) print *nd.dentry->d_inode
$11 = {i_hash = {next = 0xcbf25a20, prev = 0xcbf25a20}, i_list = {
...
    i_ino = 411, i_count = {counter = 1}, i_dev = 6,
    i_mode = 4480, i_nlink = 1, i_uid = 0, i_gid = 0, i_rdev = 0, i_size = 0,
    i_atime = 1066323791, i_mtime = 1066323791, i_ctime = 1066323791,
    i_blkbits = 10, i_blksize = 1024, i_blocks = 0, i_version = 0, i_bytes =
0,
...
(gdb) step
cp_new_stat64 (inode=0x0, statbuf=0xcbf25a20) at stat.c:280
280             memset(&tmp, 0, sizeof(tmp));
(gdb) next
276     {
(gdb) next
280             memset(&tmp, 0, sizeof(tmp));
(gdb) next
93              return (MAJOR(dev)<<8) | MINOR(dev);
(gdb) next
92      static inline unsigned int kdev_t_to_nr(kdev_t dev) {
(gdb) next
282             tmp.st_ino = inode->i_ino;
(gdb) next
284             tmp.__st_ino = inode->i_ino;
(gdb) next
286             tmp.st_mode = inode->i_mode;
```

```
(gdb) next
287             tmp.st_nlink = inode->i_nlink;
(gdb) next
288             tmp.st_uid = inode->i_uid;
(gdb) next
289             tmp.st_gid = inode->i_gid;
(gdb) next
93              return (MAJOR(dev)<<8) | MINOR(dev);
(gdb) next
92      static inline unsigned int kdev_t_to_nr(kdev_t dev) {
(gdb) next
291             tmp.st_atime = inode->i_atime;
(gdb) next
292             tmp.st_mtime = inode->i_mtime;
(gdb) next
293             tmp.st_ctime = inode->i_ctime;
(gdb) next
294             tmp.st_size = inode->i_size;
(gdb) next
313             if (!inode->i_blksize) {
(gdb) next
328                     tmp.st_blocks = inode->i_blocks;
(gdb) next
329                     tmp.st_blksize = inode->i_blksize;
(gdb) next
328                     tmp.st_blocks = inode->i_blocks;
(gdb) next
331             return copy_to_user(statbuf,&tmp,sizeof(tmp)) ? -EFAULT : 0;
(gdb) print tmp
$12 = {st_dev = 6, __pad0 = "\000\000\000\000\000\000\000\000\000\000",
  __st_ino = 411, st_mode = 4480, st_nlink = 1, st_uid = 0, st_gid = 0,
  st_rdev = 0, __pad3 = "\000\000\000\000\000\000\000\000\000\000", st_size = 0,
  st_blksize = 1024, st_blocks = 0, __pad4 = 0, st_atime = 1066323791,
  __pad5 = 0, st_mtime = 1066323791, __pad6 = 0, st_ctime = 1066323791,
  __pad7 = 0, st_ino = 411}
(gdb)
```

Accès à la pile d'appel des processus

La pile d'appel du noyau peut être consultée à n'importe quel moment par la commande `backtrace` ou son équivalent `where` :

```
(gdb) backtrace
#0  cp_new_stat64 (inode=0xcb6eba20, statbuf=0x400) at stat.c:331
#1  0xc013fe2f in sys_stat64 (filename=0x400 <Address 0x400 out of bounds>,
    statbuf=0x400, flags=-1073743088) at stat.c:343
```

De plus, si lors de la configuration du noyau on a choisi d'activer l'option `CONFIG_KGDB_THREAD`, alors on peut utiliser les commandes `gdb` prévues pour gérer les *threads*, et on pourra connaître la pile d'appel de chaque processus se trouvant en mode noyau :

```
(gdb) info threads
  26 Thread 1108  schedule_timeout (timeout=2147483647) at sched.c:424
  25 Thread 1107  schedule_timeout (timeout=2147483647) at sched.c:424
  24 Thread 1106  schedule_timeout (timeout=2147483647) at sched.c:424
...
  13 Thread 934  do_syslog (type=6184, buf=0x804d6a0 "Initdefault level
'%c' is invalid", len=4095) at printk.c:190
...
  5 Thread 4  kswapd (unused=0x0) at current.h:7
  4 Thread 3  ksoftirqd (__bind_cpu=0x0) at current.h:7
  3 Thread 2  context_thread (startup=0xc02a7f24) at context.c:101
  2 Thread 1  cp_new_stat64 (inode=0xcb6eba20, statbuf=0x400) at
stat.c:331
* 1 Thread 0  cpu_idle () at process.c:138
(gdb) thread 2
[Switching to thread 2 (Thread 1)]#0  cp_new_stat64 (inode=0xcb6eba20,
    statbuf=0x400) at stat.c:331
331             return copy_to_user(statbuf,&tmp,sizeof(tmp)) ? -EFAULT
: 0;
(gdb) bt
#0  cp_new_stat64 (inode=0xcb6eba20, statbuf=0x400) at stat.c:331
#1  0xc013fe2f in sys_stat64 (filename=0x400 &mt;Address 0x400 out of
bounds>,
    statbuf=0x400, flags=-1073743088) at stat.c:343
(gdb) thread 5
[Switching to thread 5 (Thread 4)]#0  kswapd (unused=0x0) at current.h:7
7       {
(gdb) bt
#0  kswapd (unused=0x0) at current.h:7
#1  0xc010728a in arch_kernel_thread (fn=0xc0259cc0 <contig_page_data>,
    arg=0x1, flags=9) at process.c:492
(gdb) thread 13
[Switching to thread 13 (Thread 934)]#0  do_syslog (type=6184,
    buf=0x804d6a0 "Initdefault level '%c' is invalid", len=4095)
    at printk.c:190
190                     error = wait_event_interruptible(log_wait,
(log_start - log_end));
(gdb) bt
#0  do_syslog (type=6184, buf=0x804d6a0 "Initdefault level '%c' is inva-
lid",
    len=4095) at printk.c:190
#1  0xc0158d7d in kmsg_read (file=0xcb4c42c0,
    buf=0x1828 <Address 0x1828 out of bounds>, count=6184,
ppos=0xcb4c42e0)
```

```
    at kmsg.c:35
#2  0xc0138d17 in sys_read (fd=6184,
    buf=0x1828 <Address 0x1828 out of bounds>, count=4095) at
read_write.c:177
```

Points d'arrêt matériels

Le patch kGDB permet d'utiliser les facilités matérielles de débogage offertes par les processeurs x86. En effet, ces processeurs ont des registres spéciaux de débogage et on peut les utiliser pour installer des points de surveillance sur des adresses mémoire.

Lorsqu'un accès en lecture, écriture ou exécution est détecté à cette adresse, le processeur lèvera une exception permettant à kGDB de reprendre le contrôle sur le noyau.

Les points de surveillance sont au nombre de 4 maximum, numérotés de 0 à 3, et ils peuvent s'appliquer à une zone mémoire de 1, 2 ou 4 octets. La mise en place et la suppression des points de surveillance ne se fait pas en utilisant les commandes standards de `gdb` (les commandes `watch`), mais se fait grâce à des macro-commandes `gdb` (ces macros peuvent être téléchargées sur le site de kGDB et mises dans le fichier `.gdbinit` pour les charger en mémoire automatiquement).

On dispose alors de :

- `hwebrk` : installe un point de surveillance d'exécution ;
- `hwwbrk` : installe un point de surveillance d'écriture ;
- `hwabrk` : installe un point de surveillance d'accès ;
- `hwrmbrk` : supprime un point de surveillance ;
- `exinfo` : affiche des informations sur la dernière interruption du noyau précisant si elle a été causée par un point d'arrêt ou par un point de surveillance.

Dans l'exemple suivant, on va installer un point de surveillance sur la variable contenant le nom d'hôte de la machine, et on va déclencher l'exception en modifiant ce nom par le lancement, sur la *cible*, de la commande `hostname` :

```
(gdb) p system_utsname.nodename
$1 = "cible", '\0' <repeats 59 times>
(gdb) p &system_utsname.nodename
$2 = (char (*)[65]) 0xc0269a21
(gdb) hwabrk 0 4 c0269a21
sending: "Y0,1,4,c0269a21"
received: "OK"
(gdb) c
Continuing.
Program received signal SIGTRAP, Trace/breakpoint trap.
0xc01275f9 in sys_sethostname (name=0x0, len=3) at string.h:202
202       __asm__ __volatile__(
(gdb) exinfo
sending: "qE"
received: "Hardware breakpoint 0"
(gdb) bt
#0  0xc0125570 in sys_sethostname (name=0x0, len=3) at string.h:202
(gdb) hwrmbrk 0
sending: "y0"
received: "OK"
```

Débogage des modules

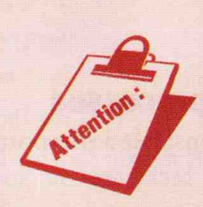
Le débogage des modules noyau est assez difficile, car, lors du lancement de `gdb`, on lui fournit uniquement le code de `vmlinux`.

Quand on chargera des modules plus tard dans le noyau de la *cible*, `gdb` ne pourra pas le détecter et il ne pourra pas résoudre les adresses appartenant au module fraîchement chargé.

Il est donc conseillé d'utiliser les modules le moins possible lorsque l'on fait du débogage en utilisant kGDB, ou au moins de ne pas en charger dynamiquement lors du démarrage de la machine, mais de le faire manuellement par la suite en utilisant une procédure spéciale.

Le but de cette procédure est de récupérer, pour chaque module, lors de son insertion dans le noyau, les adresses où le module a été chargé.

Puis, en utilisant ces adresses et la commande `add-symbol-file` de `gdb`, on fait connaître à ce dernier l'existence du module dans l'espace d'adressage du noyau.



Attention, pour insérer le module sur la *cible*, il faut que le noyau tourne (il faut donc avoir sélectionné `cont` dans `gdb`). Mais après, pour insérer les symboles dans `gdb` il faut rendre la main au débogueur (en interrompant l'exécution du noyau par *Control* + *C*).

Un script appelé `loadmodule.sh` est fourni sur le site de kGDB et permet d'automatiser au maximum ces opérations, en effectuant :

- Une copie du module sur la *cible* en utilisant `rsh` ;
- L'insertion du module dans le noyau de la *cible* ;
- La récupération des informations sur les adresses d'insertion du module ;
- La sauvegarde de ces informations, en utilisant une syntaxe directement compréhensible par `gdb` dans un fichier temporaire.

Une fois ce script fini, il suffira de charger le fichier temporaire dans `gdb` en utilisant sa commande source afin de faire prendre en compte le module par la suite.

Le module étant chargé, toutes les opérations sur son code sont permises, comme la consultation de son code, la mise de points d'arrêt, etc.

Voyons un exemple avec le module `loop` (on utilisera un point d'arrêt que l'on atteindra en exécutant `mount -o loop image.iso /mnt` sur la *cible*) :

```
$ loadmodule.sh cible drivers/block/loop.o
Copying drivers/block/loop.o to cible
Loading module drivers/block/loop.o
Generating script /tmp//loadcibleloop.o
$ cat /tmp//loadcibleloop.o
add-symbol-file drivers/block/loop.o 0xcc878060 -s .fixup 0xcc879fef -s
.rodata.str1.32 0xcc87a0c0 -s .rodata.str1.1 0xcc87a2ca -s __ex_table
0xcc87a328 -s __ksymtab 0xcc87a474 -s .rodata 0xcc87a4a4 -s __archdata
0xcc87a4c0 -s __kallsyms 0xcc87a4c0 -s .data 0xcc87ac80 -s .bss
0xcc87ad28
$ gdb vmlinux
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
...
(gdb)
breakpoint () at gdbstub.c:1344
1344    }
warning: shared library handler failed to enable breakpoint
(gdb) source /tmp/loadcibleloop
add symbol table from file "drivers/block/loop.o" at
        .text_addr = 0xcc86c060
        .fixup_addr = 0xcc86deb5
        .rodata.str1.32_addr = 0xcc86df80
        .rodata.str1.1_addr = 0xcc86e183
        __ex_table_addr = 0xcc86e1e0
        __ksymtab_addr = 0xcc86e32c
        .rodata_addr = 0xcc86e35c
```


PEARL

Le spécialiste du périphérique informatique

www.pearl.fr

www.pearl.fr

Plus de 5000 références parmis lesquelles un grand choix de cartouches compatibles

Demandez gratuitement votre Catalogue 124 pages

Tél. 03 88 58 02 02
Fax 03 88 58 02 07

3615 Pearl (0,34 €/mn) • www.pearl.fr
PEARL Diffusion 6, rue de la Scheer - Z.I. Nord
B.P. 121 - 67603 SELESTAT Cedex

0,12 €/mn
N° Indigo 0 820 822 823

```
        __archdata_addr = 0xcc86e370
        __kallsyms_addr = 0xcc86e370
        .data_addr = 0xcc86eb40
        .bss_addr = 0xcc86ebe8
warning: section __archdata not found in /home/stelian/linux-2.4.22-
kgdb/drivers/block/loop.o
warning: section __kallsyms not found in /home/stelian/linux-2.4.22-
kgdb/drivers/block/loop.o
(gdb) break lo_open
Breakpoint 2 at 0xcc86d6bf: file loop.c, line 911.
(gdb) c
Continuing.

Breakpoint 2, lo_open (inode=0xc9998280, file=0xc9cb8220) at loop.c:911
911             if (!inode)
(gdb) n
913             if (MAJOR(inode->i_rdev) != MAJOR_NR) {
(gdb) n
917             dev = MINOR(inode->i_rdev);
(gdb) n
918             if (dev >= max_loop)
(gdb) n
921             lo = &loop_dev[dev];
(gdb) where
#0  lo_open (inode=0xc9998280, file=0xc9cb8220) at loop.c:921
#1  0xc013f194 in do_open (bdev=0xc1394a60, inode=0x0, file=0x0)
    at block_dev.c:571
#2  0xc013f2e5 in blkdev_open (inode=0xc9998280, filp=0xc9cb8220)
    at block_dev.c:623
#3  0xc017aab9 in devfs_open (inode=0xc9998280, file=0xc9cb8220) at
base.c:2790
#4  0xc013803d in dentry_open (dentry=0xc9b352a0, mnt=0xc121f320,
    flags=-912686464) at open.c:698
#5  0xc0137f3b in filp_open (filename=0x0, flags=32768, mode=-912686464)
    at open.c:656
#6  0xc0138270 in sys_open (filename=0x0, flags=0, mode=0) at open.c:798
```

Si l'on veut décharger le module et le recharger par la suite, il faut quitter et relancer gdb, puis refaire la procédure de chargement du module, car il n'y a pas de moyen de libérer une partie de symboles dans la mémoire de gdb.

Désassemblage du code source

Lors du débogage, il faut parfois descendre au niveau du code assembleur généré lors de la compilation afin de détecter certains problèmes d'optimisation.

Pour ce faire, gdb dispose des commandes `info line` et `disassamble`. La première permet de consulter le mapping entre le code source et les adresses en mémoire.

La seconde effectue le désassemblage d'une fonction entière ou d'un intervalle d'adresses :

```
(gdb) info line cp_new_stat64
Line 280 of "stat.c" starts at pc 0xc0145500 and ends at
0xc0145619

(gdb) disassamble cp_new_stat64
Dump of assembler code for function cp_new_stat64:
0xc014f550 <cp_new_stat64+0>:    push   %esi
0xc014f551 <cp_new_stat64+1>:    push   %ebx
0xc014f552 <cp_new_stat64+2>:    sub    $0x6c,%esp
0xc014f555 <cp_new_stat64+5>:    mov    0x78(%esp,1),%ebx
0xc014f559 <cp_new_stat64+9>:    movl   $0x60,0x8(%esp,1)
0xc014f561 <cp_new_stat64+17>:   lea    0xc(%esp,1),%esi
0xc014f565 <cp_new_stat64+21>:   mov    %esi,(%esp,1)
0xc014f568 <cp_new_stat64+24>:   movl   $0x0,0x4(%esp,1)
0xc014f570 <cp_new_stat64+32>:   call   0xc014f6e0
<__constant_c_and_count_memset>
...
```

Futur de kGDB

Il y a eu des discussions à plusieurs reprises sur la liste de diffusion du noyau (www.tux.org/lkml) sur la possibilité de faire évoluer kGDB afin de s'affranchir de l'obligation de posséder un port série sur la machine.

Ces discussions ont récemment donné lieu à de réels travaux sur ce sujet, et on trouve aujourd'hui une implémentation de kGDB fonctionnant avec certaines cartes réseau (en attaquant la carte à un niveau très bas, en mode *polling*, sans utilisation des interruptions ni des couches réseau).

Ces développements récents sont intégrés dans la version de kGDB distribuée avec les noyaux 2.6-mm d'Andrew Morton et se basent sur la même couche d'abstraction que NetConsole (qui permet de connecter une console sur la machine en utilisant une interface réseau) et NetDump (qui permet, en cas de *crash*, de sauvegarder l'image mémoire du noyau sur une autre machine du réseau pour être analysée ultérieurement).

Toujours dans la version maintenue par Andrew Morton, on retrouve un certain nombre d'améliorations du patch kGDB apportées par George Anzinger (la possibilité de déboguer du code se trouvant encore plus tôt dans la séquence de démarrage du noyau, à la première ligne de code C !, l'amélioration de la sortie de `info threads`, la possibilité de faire des traces événementielles dans le noyau et de les récupérer par kGDB, etc.).

Ces améliorations n'ont cependant pas (encore) été réintégrées dans le patch kGDB de Amit S. Kale.

Soulignons aussi le fait que, selon les rumeurs qui courent parmi les développeurs noyau, une fois le noyau stable 2.6 sorti, Linus Torvalds se consacrera au développement de la nouvelle branche de développement – la 2.7 –, et le maintien du noyau stable incombera à ... Andrew Morton.

Si ces rumeurs s'avèrent fondées, il est fort possible que l'intégration de kGDB dans le version officielle du noyau revienne à l'ordre du jour...

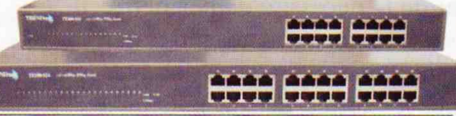
Stelian Pop
stelian@popies.net

Conclusion

Nous voici à la fin de cette présentation de la mise en place de kGDB et des fondements de son utilisation pour le débogage du noyau Linux.

J'espère que cet article a rempli son rôle de démystification de kGDB et qu'il vous aura donné l'envie de l'utiliser lors de vos prochains développements de code noyau.

HUB SWITCH 10/100 BASE T RACKABLE
16 PORTS Réf.P1005 Prix : 119,90 €TTC/786,49F TRENDnet
24 PORTS Réf.P1006 Prix :169,90 €TTC/1 114,47F



HUB MEDIA GIGABIT 16 PORTS RACKABLE 10/100/1000MBITS TRENDnet
Ce switch cuivre rackable de haute performance bénéficie d'une technologie d'auto négociation qui lui permet de sélectionner automatiquement la vitesse de transfert adaptée : 10Base-T, 100Base-TX et 1000Base-T, aussi bien en mode half duplex qu'en full duplex. Réf.PE8365
Prix : 799,90€TTC/5247,00F

HUB 3COM SUPERSTAK II 3000
Un hub 100base-T de grande marque à un prix exceptionnel ! ► 8 ports 100 Mbp/sec ► Format 19" rackable ► LED de contrôle de connection Réf.S500
Prix : 79,90€TTC/524,11F

HUB SWITCH 9 PORTS (8 EN 10/100MBITS ET 1 EN 10/100/1000MBITS) TRENDnet
Vous pouvez connecter un serveur Gigabit à un port Gigabit pour augmenter la performance de votre réseau ou relier deux switchs Gigabit ensemble afin de créer une haute densité de données.
Réf.PE8366 Prix : 149,90 €TTC/983,28F

CARTE GIGABIT PCI TRENDnet
Utilisez cette carte pour connectez vos PC à l'aide d'un réseau très haut débit. Idéal pour transférer de gros fichiers. Se connecte à un port PCI 32 bits. Réf. PE8363 Prix : 39,90€TTC/261,73F

CARTE RÉSEAU 10/100 BASE T TRENDnet
Une réserve de puissance pour votre réseau grâce à cette carte 32 bits au débit de 100 Mbits. Cette carte réseau se configure automatiquement en 10Mbits ou 100Mbits selon le réseau. ► Architecture PCI Busmaster ► Compat. NE2000 ► Connexion RJ45 ► 10Mbits ou 100Mbits ► DMA Buffer management. Réf. PE223 Prix : 14,90 €TTC/97,74F

PRISE RJ45 CATÉGORIE5 BLINDÉE (à sertir) : Réf.PE261 Prix : 1,22 €TTC/8,- F

CÂBLE RJ45 CATÉGORIE 5E BLINDÉ
Au mètre Réf.PE262 Prix : 1,37 €TTC/9,-F
100 m Réf.PE268 Prix : 59,90€TTC/392,92F
100 m en rigide pour prises murales Réf.PE275 Prix : 69,90€TTC/458,51F

BOÎTIERS MURAUX
Boîtiers en saillies catégorie 5 blindés
BOITIER 1xRJ45 Réf. P1200
Prix : 9,90 €TTC/64,94F
BOITIER 2xRJ45 Réf. P1201
Prix : 19,67 €TTC/129,-F

PINCE À SERTIR (RJ45)
Réf. PE2558
Prix : 14,90€TTC/97,74F

TESTEUR DE CÂBLES RÉSEAUX
Pour Câbles BNC et RJ45 ► Livré avec un bouchon pour les câbles BNC et une terminaison pour le type RJ45 ► Pochette de transport fournie.
Réf. PE40 Prix : 69,90€TTC/458,51F

PEARL

PEARL Professionals™

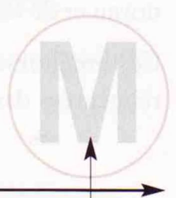
Découvrez tous nos produits professionnels : *Accessoires réseaux rackables (panneaux de brassage, hubs...), onduleurs, connectique...*
PEARL Diffusion 6, rue de la Scheer - Z.I. NordB.P. 121 - 67603 SELESTAT Cedex

www.pearl.fr

0,12 €/mn N° Indigo 0 820 822 823

Demandez gratuitement notre Catalogue 124 pages

Linux Security Modules



Historique

Lors du Kernel Summit de San Jose, en mars 2001, Peter Loscocco présenta *Security Enhanced Linux* [SELinux], un patch pour le noyau Linux implémentant une architecture de contrôle d'accès, tous deux développés par la NSA.

SE Linux n'était pas le seul correctif de la sorte. D'autres projets comme RSBAC, Medusa DS9, LIDS, SubDomain, DTE ou encore LoMAC [rsbac, medusa, lids, subdomain, dte, lomac] implémentent d'autres contrôles d'accès ou d'autres charpentes de contrôle d'accès.

Linus Torvalds, conscient de la nécessité de permettre d'autres types de contrôles d'accès en standard, autres que ceux déjà présents (1), et plutôt que de privilégier un type de contrôle d'accès sur tous les autres, décida alors qu'il serait prêt à intégrer dans la distribution standard du noyau 2.6 une charpente commune laissant chacun choisir la méthode de contrôle d'accès de son choix.

Ainsi naquit le projet Linux Security Modules (LSM). La charpente prit la forme d'un ensemble de points de contrôle (*hooks* ou ancres) dans le noyau Linux, afin de brancher différents types de contrôles d'accès. On y trouve actuellement une implémentation de SE Linux, LIDS, DTE, et même les *capabilities* ont été sorties du noyau pour s'appuyer sur les LSM.

Il est à noter que le mot module dans LSM ne signifie pas que le contrôle d'accès doit être implémenté sous forme de module noyau (LKM), mais que la charpente est assez bien faite pour que tout code de contrôle d'accès l'utilisant soit modulaire. On peut ainsi le compiler sous forme de module si on le désire, mais rien ne nous y oblige.

(1)c'est-à-dire le *Discretionary Access Control* (DAC) (les droits UNIX que tout le monde connaît) et les *capabilities*).

Quelques mots sur le contrôle d'accès

Lorsqu'on s'intéresse à la sécurité d'un système, il est nécessaire de répondre à de nombreuses questions. Qui est qui ? Qui fait quoi ? Qui peut faire quoi (et quand) ? etc. Le contrôle d'accès est la réponse à certaines de ces questions en renforçant certaines contraintes de sécurité, comme la confidentialité, l'intégrité ou la disponibilité.

Modèle de sécurité

Pour cela, le contrôle d'accès apporte des solutions précises et fiables. Comme son nom l'indique, le contrôle d'accès cherche à déterminer qui peut accéder à quoi sur un système, et donc définir ce qu'on entend par "qui" et "quoi" :

- Qui : un sujet est une entité active sur le système, comme un processus agissant pour le compte d'un utilisateur ;
- Quoi : un objet est une entité passive sur le système, comme un fichier.

Selon les cas, il est possible qu'un même élément d'un système soit un sujet ou un objet. Si on appelle une commande permettant de prendre l'identité d'un autre utilisateur (su), alors l'utilisateur qui appelle su sera un sujet, mais l'identité cible un objet.

Enfin, il reste à définir ce qu'on entend par "accéder". Les privilèges (ou permissions ou droits) constituent le dernier élément intervenant dans le contrôle d'accès. Les droits auxquels nous sommes habitués sous Unix (read, write, execute) ne sont qu'un exemple parmi tant d'autres.

Un modèle général de sécurité est présenté en figure 1 :

- L'authentification a pour objectif de garantir l'identité du demandeur qui présente des *credentials* : un mot de passe (je sais), une Smartcard (je possède), la biométrie (je suis) ;

- Une fois la légitimité du demandeur reconnue, sa requête pour accéder à un objet est confrontée à la base des autorisations et au contrôle d'accès ;

- L'audit est le mécanisme qui supervise tout ce qui se passe, enregistre les événements afin de déterminer les tentatives de violations de la politique de sécurité (*offline* après les faits, ou en temps réel pour de la détection d'intrusion).

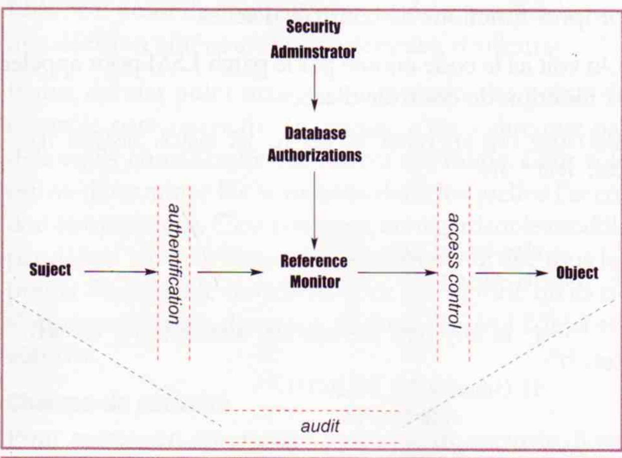


Figure 1 **Contrôle d'accès**

Ainsi, des sujets (ex: utilisateurs) détiennent des privilèges sur des objets (ex: fichiers, *devices*) en accord avec une politique (ex. : modèle de Biba) de contrôle d'accès mise en œuvre par le *reference monitor*.

Politiques et mécanismes

Le contrôle d'accès détermine qui est autorisé ou non à accéder à quoi. Cette distinction entre autorisation et rejet est effectuée en adéquation avec une politique de contrôle d'accès.

Le contrôle d'accès est un problème délicat, en particulier lorsqu'on compare plusieurs sujets :

- Il existe des rôles différents : utilisateurs normaux vs super-utilisateur ;

- Il existe des groupes ou hiérarchiee : étudiants vs professeurs ;

- Il y a besoin de déléguer ses droits et pouvoirs.

Pour tenter de simplifier la résolution de ces problèmes, on distingue la politique de sécurité (*Discretionary Access Control* - DAC, *Mandatory Access Control* - MAC, *Role-Based Access Control* - RBAC), des mécanismes (matrice d'accès, liste de contrôle d'accès - ACL, liste de *capabilities*) mises en œuvre pour appliquer la politique de sécurité.

Le mécanisme de contrôle d'accès le plus complet est la matrice d'accès [Lampson74]. Pour un système, il s'agit d'identifier tous les sujets (lignes) et tous les objets (colonnes), puis de définir les relations qui les unissent. Bien évidemment, une telle matrice est énorme et très creuse, ce qui la rend très peu pratique. En fait, cette matrice sera représentée et utilisée sous forme de liste :

- *Access Control List* (ACL) : il s'agit des colonnes de la matrice de contrôle d'accès, c'est-à-dire que pour chaque objet du système, on conserve la liste des utilisateurs qui peuvent y accéder et des opérations qu'ils peuvent effectuer dessus ;

- *Capability List* : il s'agit des lignes de la matrice de contrôle d'accès, c'est-à-dire que pour chaque sujet du système, on conserve la liste des objets auxquels ils peuvent accéder et des opérations qu'ils peuvent effectuer dessus.

La politique de contrôle d'accès la plus simple est celle qui est laissée à discrétion de l'utilisateur, d'où son nom *Discretionary Access Control*. Il s'agit par exemple de celle à laquelle chaque utilisateur d'Unix est habitué, avec les droits en lecture, écriture et exécution, qu'il peut changer à souhait sur les objets qu'il possède. Toutefois, une telle politique possède d'importants défauts. Tout d'abord, la politique globale de sécurité peut être compromise par un seul utilisateur s'il commet une erreur. Ensuite, le flux d'information n'est pas du tout contrôlable : un utilisateur capable de lire des données peut ensuite les transmettre à un utilisateur qui n'était pas censé les voir. C'est pourquoi on préfère en général d'autres modèles, plus évolués mais un peu moins flexibles.

Le *Role-Based Access Control* (RBAC) est une autre forme de contrôle d'accès. Avec la matrice d'accès, on considère uniquement des sujets et des objets. Ici, on met en place des classes d'équivalence qui regroupent plusieurs sujets (on parle de groupe ou de rôle). Cela est particulièrement utile lorsque les sujets changent fréquemment car on évite ainsi de recalculer la matrice.

Le modèle de rôles le plus utilisé est celui proposé dans [SC96]. Il s'agit en fait d'une "indirection" entre les sujets et les permissions, illustrée en figure 2 : les sujets n'ont pas directement de droits, ce sont les rôles qui en ont mais les sujets sont affectés à des rôles. Des modèles de RBAC proposent également des hiérarchies de rôles, et des contraintes sur les rôles.

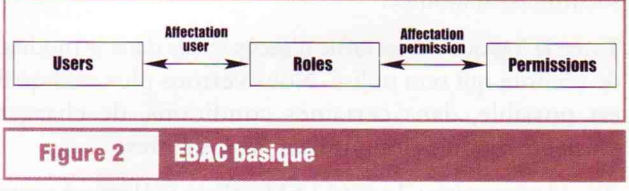


Figure 2 **EBAC basique**

Enfin, Le *Mandatory Access Control* (MAC) s'appuie sur l'utilisation de labels associés aux objets et sujets :

- Le label d'un sujet (*clearance* ou accréditation) indique le niveau de confiance de ce sujet ;

Le label d'un objet spécifie le niveau minimal que doit avoir un sujet pour accéder à l'objet en question. Un cas particulier de MAC est donné par les politiques de sécurité multi-niveaux. Définissons par exemple 4 labels (top secret, secret, confidentiel et non classifié), ainsi que 2 règles qui seront systématiquement appliquées :

- *read down* : l'accréditation d'un sujet doit dominer le niveau de sécurité de l'objet qu'il lit ;
- *write up* : l'accréditation d'un sujet doit être dominée par le niveau de sécurité de l'objet qu'il écrit.

De cette manière, on garantit (mathématiquement) la confidentialité des objets en fonction des niveaux d'accréditation. Ce modèle, élaboré en 1975, s'appelle Bell-La Padula (BLP) [BLP73], du nom de ses concepteurs. Le modèle "inverse" (`read up`, `write down`, modèle de Biba [Biba77]), garantit quant à lui l'intégrité des données.

À la différence des permissions habituelles sous Unix, ici, l'utilisateur n'est plus maître même des objets qu'il possède. Ainsi, il est possible de baisser les capacités de root, comme nous le verrons dans l'exemple en fin d'article.

Il existe de nombreux autres modèles de contrôle d'accès, chacun avec ses objectifs. Lorsque Linus Torvalds a voulu que la sécurité entre dans la noyau, il ne voulait néanmoins pas imposer un modèle précis, chacun ayant ses propres besoins.

Ainsi, les LSM constituent non pas un modèle de contrôle d'accès, mais une charpente, dans laquelle chacun peut brancher ce qu'il veut.

Signalons que les LSM ne constituent pas une nouveauté. De telles charpentes existaient déjà par ailleurs, comme le *Generalized Framework for Access Control* (GFAC) de RSBAC [`rsbac`], ou le *Flux Advanced Security Kernel* (FLASK, [`flask`]) implémenté, entre autres, par SE Linux.

Survol de la charpente LSM

La charpente en elle-même ne fait que rajouter des points de contrôle aux endroits critiques dans le code du noyau, ainsi que des champs destinés à contenir des informations de sécurité dans les structures internes du noyau. Aucune sécurité n'est ajoutée.

Toute la logique du contrôle d'accès réside dans le module de sécurité qui sera utilisé. Nous verrons plus tard qu'il est possible, dans certaines conditions, de charger plusieurs modules les uns derrière les autres.

Anecdotiquement, le *patch* LSM enlève également toute la logique de contrôle associée aux *capabilities* du noyau pour la mettre dans un module de sécurité utilisant la charpente.

Points de contrôle

Un point de contrôle se présente sous la forme d'un appel à une fonction dont l'adresse est stockée dans une structure pointée par la variable globale `security_ops`. Cette structure est du type `security_operations` :

```
struct security_operations {
          int (*sethostname) (char *hostname);
          int (*setdomainname) (char *domainname);
          int (*reboot) (unsigned int cmd);
          [...]
};
```

Chaque module de sécurité fournit une structure de ce type. Lorsqu'il est chargé, il s'enregistre pour faire pointer la variable `security_ops` vers sa structure référençant ses propres fonctions de contrôle d'accès.

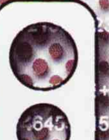
On voit ici le code rajouté par le patch LSM pour appeler la fonction de contrôle d'accès.

```
asmlinkage long sys_reboot(int magic1, int magic2, unsigned int
cmd, void * arg
)
 {

          char buffer[256];
+         int retval;

          /* We only trust the superuser with rebooting the sys-
tem. */
          if (!capable(CAP_SYS_BOOT))
                    return -EPERM;

+         retval = security_ops->reboot(cmd);
+         if (retval) {
+                   return retval;
+         }
+
          /* For safety, we require "magic" arguments. */
          if (magic1 != LINUX_REBOOT_MAGIC1 ||
               (magic2 != LINUX_REBOOT_MAGIC2 && magic2 !=
LINUX_REBOOT_MAGIC2A &&
```

Dans la version pour les noyaux 2.5/2.6, le point de contrôle se présente sous la forme d'un appel à une fonction *inline* qui se charge de déréférencer `security_ops` et d'appeler la fonction du module de sécurité utilisé. Ce double appel est équivalent à la version des noyaux 2.4 après optimisation par le compilateur, et est plus propre :

```
static inline int security_bprm_set (struct linux_binprm *bprm)
{
          return security_ops-bprm_set_security (bprm);
}
Et on trouvera donc au lieu de

     retval = security_ops-bprm_set_security (bprm);
le code suivant :

     retval = security_bprm_set(bprm);
```

Les points de contrôle ne sont pas ajoutés au hasard dans le code. En fait, ils se situent systématiquement "au même endroit", c'est-à-dire après tout un tas de vérifications préalables. Par exemple, dans le cas de la fonction `vfs_symlink()`, on commence par vérifier le droit de créer



le lien symbolique dans le répertoire destination. Ensuite seulement, on appelle le *hook* du LSM. Enfin, on crée le lien symbolique, sous réserve que l'autorisation ait été accordée (cf partie "champ de sécurité" pour voir le code complet de cette fonction).

Cela a plusieurs conséquences. Tout d'abord, cela implique que les arguments reçus par les points de contrôle soient déjà présents en espace noyau et des contrôles d'erreur aient déjà été effectués. Ce sont autant de choses qui ne sont plus à faire. Ensuite, il faut bien comprendre que les tests du LSM arrivent dans les derniers : si un seul des tests précédents échoue, on quitte la fonction avant d'avoir le temps de demander l'avis du LSM. Un point de contrôle ne peut donc pas surcharger une décision antérieure (ni postérieure, d'ailleurs).

Enfin, dernier point mais non le moindre, les points de contrôle sont restrictifs (*restrictive*), c'est-à-dire que par défaut, ils considèrent que l'accès est valide. Leur rôle est de déterminer les situations dans lesquelles l'accès doit être empêché. C'est pourquoi, en regardant le module par défaut `security/dummy.c` vous constaterez que tous les points de contrôle renvoient 0, ce qui signifie qu'ils ne s'opposent pas à la demande, et donc l'accès à l'objet est autorisé.

Champs de sécurité

Pour mettre en œuvre une politique de sécurité, il est souvent nécessaire de conserver des données, servant généralement à prendre les décisions sur l'accès à l'objet. Par exemple, dans le modèle de BLP vu précédemment, chaque objet possède un *label*, son niveau de sécurité. La solution la plus simple est alors de prévoir pour chaque objet du noyau un champ (un `void *`) destiné à contenir ces informations.

Objet	Structure
Programme	`struct linux_binprm`
Processus	`struct task_struct`
Système de fichiers	`struct super_block`
Fichier, *pipe*, *socket*, *inode*	`struct inode` et `struct file`
Paquet et *device* réseau	`struct sk_buff` et `struct net_device`
IPC	`struct kern_ipc_perm` et `struct msg_msg`

Tab. 1 : Structure contenant un champ de sécurité en fonction du type d'objet

Le champ de sécurité se présente donc sous forme d'un pointeur, initialisé à `NULL` (cf Tab. 1). Il faut donc prévoir des points de contrôle spécifiques destinés à la gestion de ces pointeurs : allocation de mémoire et initialisation, libération, modification, etc. Pour cela, il existe des *hooks* spécifiques pour chaque structure contenant un champ de sécurité, appelés `*_alloc_security()` et `*_free_security()` (respectivement pour allouer et libérer les champs de sécurité), où `*` correspond à la structure. Ces fonctions retournent 0 en cas de réussite, et prennent comme argument la structure à laquelle on souhaite attacher une structure de sécurité.

Prenons comme exemple la structure `inode` (`include/linux/fs.h`). Elle contient un champ supplémentaire destiné à la sécurité :

```
struct inode {
        ...
        void            *i_security;
        ...
};
```

Parmi les pointeurs sur fonction contenus dans la `security_ops`, on trouve ceux destinés à gérer ce champ, soit pour la structure `inode` :

```
int (*inode_alloc_security) (struct file * inode);
```

Les concepteurs des LSMs ont prévu dans le code du noyau des points d'entrée qui permettent au programmeur d'un module de sécurité spécifique d'allouer le champ de sécurité :

```
        struct *inode;

        ...
        error = security_ops->inode_alloc_security(inode);
        if (!error) {
                ...
        }
```

Toutefois, s'il est indispensable d'allouer ces champs de sécurité, il l'est tout autant de pouvoir les mettre à jour. Pour le moment, les points de contrôle que nous avons évoqués intervenaient dans la chaîne de contrôle d'accès pour autoriser ou non l'accès à un objet. Pour remédier à cela, des hooks `<objet>_post_<operation>` sont prévus, où `<objet>` est un objet du système (*file*, *inode*, *task*, etc.) et `<operation>` ... une opération sur l'objet en question.

Revenons à notre objet `inode`. Il existe une fonction de contrôle associée à la création d'un lien symbolique `inode_symlink()`. Dans le cas où la création du lien est autorisée et se passe sans erreur, une seconde fonction de contrôle permet de mettre à jour le champ de sécurité : `inode_post_symlink()`. On trouve alors dans le code du noyau :

```
int vfs_symlink(struct inode *dir, struct dentry *dentry, const char
*oldname)
{
        int error;

        down(&dir->i_zombie);
        error = may_create(dir, dentry);
        if (error)
                goto exit_lock;

        error = -EPERM;
```

```
        if (!dir->i_op || !dir->i_op->symlink)
                goto exit_lock;

        error = security_ops->inode_symlink(dir, dentry, oldname);
        if (error)
                goto exit_lock;

        DQUOT_INIT(dir);
        lock_kernel();
        error = dir->i_op->symlink(dir, dentry, oldname);
        unlock_kernel();

exit_lock:
        up(&dir->i_zombie);
        if (!error) {
                inode_dir_notify(dir, DN_CREATE);
                security_ops->inode_post_symlink(dir, dentry,
oldname);
        }
        return error;
}
```

Communication avec le *user space*

Ajouter des labels ou autres aux objets du système se fait assez facilement par le biais des champs de sécurité comme nous venons de le voir. Toutefois, cela se passe uniquement en espace noyau. Pourtant, un utilisateur pourrait très bien souhaiter connaître, si la politique de sécurité le permet, ses propres permissions, ou celles nécessaires sur certains objets. On imagine aisément une commande `lsperm`, renvoyant les permissions.

Pour réaliser cela, il apparaît clairement qu'une interface est nécessaire entre l'espace utilisateur (là où est lancée la pseudo-commande `lsperm`) et l'espace noyau (là où sont stockées les informations de sécurité).

Comme il n'existe pas 50 manières de faire communiquer *kernel space* et *user space*, la première solution envisagée fut la mise en place d'un appel système `sys_security()`. Celle-ci ne fut finalement pas retenue pour éviter de mettre en place un appel système non standard d'une part, mais surtout qui servirait à tout et rien puisque sa fonction serait propre à chaque module.

La solution retenue actuellement repose sur le système de fichiers `/proc`, mais chaque LSM est libre de faire comme bon lui semble. La charpente intègre dans la structure `security_operations` deux pointeurs sur fonction destinés à communiquer avec le *user space* via le `/proc` :

```
int (*getprocattr)(struct task_struct *p, char *name, void *value, size_t size);
int (*setprocattr)(struct task_struct *p, char *name, void *value, size_t size);
```

Les répertoires `/proc/<pid>/` contiennent maintenant un nouveau répertoire, `attr/`, qui contient les entrées correspondant aux labels de sécurité que l'on souhaite connaître, ou même fixer.

Ces entrées appellent directement les fonctions précédentes, avec le paramètre `name` valant le nom du fichier de `attr/` auquel nous accédons.

Tests de consistance

Le plus gros problème des politiques de contrôle d'accès en général et des charpentes de politique de sécurité en particulier est d'arriver à prouver leur consistance.

Dans notre cas, il faut parvenir à montrer que la charpente proposée par les LSM permet une telle consistance, c'est-à-dire qu'elle n'oublie pas de contrôler un point particulier qui permettrait de contourner un autre point de contrôle. Par exemple, contrôler l'appel système `ioperm()` ne sert à rien si l'appel `iopl()` ne peut pas être filtré.

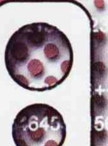
Cette consistance n'est pas réellement prouvable au sens mathématique, vu le travail que nécessiterait la modélisation du noyau Linux. En revanche, des travaux très intéressants ont été menés par Zhang `et al`. [ZEJ02], pour arriver à un degré de confiance correct en cette charpente.

Cette vérification se déroule en deux étapes. Tout d'abord, il s'agit de s'assurer de la complétude du principe de médiation (*complete mediation*, aussi appelée *reference monitor property*) [SS75] : le mécanisme de contrôle d'accès doit être capable d'intercepter et potentiellement d'empêcher tous les accès à une ressource. Dans le cas contraire, la sécurité ne peut être garantie. Prenons comme analogie un firewall qui protège un réseau interne d'Internet : si une seule route permet d'atteindre un élément du réseau interne sans passer par le firewall, alors ce dernier ne sert à rien.

Il faut donc que chaque objet subisse bien une opération de contrôle de la part d'un hook fourni par les LSM avant d'être utilisé. Seuls les objets accessibles en espace utilisateur sont impliqués ici comme nous l'avons vu précédemment.

Cette vérification est effectuée à l'aide des opérations suivantes :

- 1. détermination des fonctions qui initialisent les objets ;
- 2. détermination des fonctions qui utilisent les objets ;
- 3. détermination des fonctions qui autorisent l'accès aux objets ;
- 4. vérification que toutes les manipulations d'objets dans les fonctions d'autorisation sont réalisées après le contrôle d'accès ;
- 5. vérification qu'il n'y ait pas d'affectation de l'objet après son contrôle d'accès ;
- 6. détermination des chemins d'exécution entre l'initialisation et l'utilisation de l'objet ;
- 7. vérification de la présence d'au moins une opération de contrôle d'accès dans chaque chemin.)



L'autre problème est de s'assurer que toutes les opérations de contrôle ont bien été effectuées avant l'utilisation d'un objet, pour tous les chemins possibles entre l'initialisation et l'utilisation : il s'agit donc de vérifier la complétude de l'autorisation. Cette opération coule relativement de source une fois le problème de médiation résolu. L'approche employée dans [ZEJ02] repose sur une analyse statique du noyau. Si elle ne permet pas de prouver mathématiquement la sécurité de la charpente des LSM (personne n'est à l'abri d'une erreur de codage, pas même les *kernel hackers*), elle a néanmoins permis d'identifier trois types d'erreur :

- Inconsistance de la vérification : la variable qui est vérifiée n'est pas celle qui sera utilisée par la suite. Les tests conduits par les auteurs ont ainsi permis d'identifier une condition de concurrence (*race condition*) pouvant conduire à une exploitation. En effet, la vérification était effectuée sur un objet de type `struct file`, mais la suite des opérations sur un *file descriptor*, à partir duquel le fichier était récupéré.
- Modification d'un objet contrôlé sans vérification de sécurité : cela se produit lorsque qu'un objet est censé avoir été contrôlé avant utilisation, mais que cela n'a pas été le cas. En particulier, l'analyse des auteurs a permis d'identifier un problème lorsqu'une *page fault* se produisait : l'objet `file` manipulé dans ce cas n'a pas subi de contrôle d'accès. Il est toutefois passé à la fonction `page_cache_read()` qui appelle une sous-fonction pour lire la page en question, en prenant l'objet `file` en argument.

Cet exemple illustre que la lecture d'un fichier "mmap()é" ne se soucie pas d'éventuelles modifications des attributs de sécurité du fichier, une fois celui-ci en mémoire. Cela a généré quelques mails sur la liste de développement des LSM, dont la conclusion fut que le contrôle ne serait systématique que pour les fichiers non mmap()és, les fichiers mmap()és n'étant contrôlés qu'au moment de la lecture pour chargement.

- Problème de typage : l'analyse statique employée par les auteurs s'appuie sur le typage des objets, qui sont soit *checked*, soit *unchecked*. Ce problème survient lorsqu'une opération est réalisée sur un objet d'un type donné, alors que c'est l'autre type qui est attendu.

Le problème de l'approche des auteurs est un taux très élevé de faux positifs, c'est-à-dire d'endroits où l'analyseur détecte une faille alors qu'il n'y en a pas. Malgré cela, ces travaux ont permis de donner une garantie de sécurité, certes non absolue mais suffisante, à la charpente des LSMs.

Enregistrement et empilage de modules de sécurité

Nous avons vu qu'un module de sécurité est en fait un tableau de pointeurs sur des fonctions. Il reste à préciser au noyau qu'il faut l'utiliser. Cela se fait au chargement du module par l'appel à la fonction `register_security()`.

Dans certaines situations, il est souhaitable de combiner plusieurs politiques de contrôle d'accès. Pour cela, il faut pouvoir enregistrer plusieurs modules de sécurité. Lorsqu'un module est déjà enregistré, l'appel à la fonction `register_security()` échouera. Il est alors possible de s'enregistrer auprès du module déjà enregistré en appelant `mod_reg_security()`. Libre à lui d'accepter notre module sur son dos. S'il accepte, nous recevrons toutes les demandes d'autorisation qu'il souhaitera nous transmettre, mais il n'est ni obligé de nous consulter à chaque fois, ni obligé de tenir compte de notre avis.

En résumé, le code d'enregistrement d'un LSM ressemblera le plus souvent à ça :

```
/* 1 si présence d'un autre module avant */
static int secondary;

/* définition des opérations de notre charpente */
static struct security_operations facist_ops = {
         ...
};

static int init_module (void)
{
         ...
         /* enregistrement dans la charpente */
         if (register_security (&facist_ops)) {
                  printk (KERN_INFO "échec de l'enregistrement\n");
                  /* tentative en contactant le module précédent */
                  if (mod_reg_security (MY_NAME, &facist_ops)) {
                           printk (KERN_INFO "échec auprès module
précédent\n");
                           return -EINVAL;
                  }
                  secondary = 1;
         }
         printk (KERN_INFO "facist LSM chargé\n");
         return 0;
}
```

Dans la même lignée, le déchargement du module dépend de la position dans la chaîne de sécurité :

```
static void exit_module (void)
{
         /* on se retire de la charpente */
         if (secondary) {
                  if (mod_unreg_security (MY_NAME, &facist_ops))
                           printk (KERN_INFO "échec du déchargement
secondaire\n");
                  return;
         }

         if (unregister_security (&facist_ops)) {
                  printk (KERN_INFO "échec du déchargement\n");
         }
}
```

Lorsque plusieurs modules sont chargés, il faut voir cela comme une liste. Chaque module est alors en charge d'appeler, s'il le souhaite, les contrôles du module suivant.

Par exemple, si un module souhaite interdire systématiquement le *reboot* sans en référer à un éventuel successeur, la fonction de contrôle sera :

```
static int facist_reboot (unsigned int cmd)
{
	return -EPERM;
}
```

En revanche, s'il souhaite suivre l'avis d'un éventuel successeur, la fonction ressemblera à :

```
/* pointeur sur les hooks du successeur */
struct security_operations *next_ops;

static int facist_reboot (unsigned int cmd)
{
	int res = -EPERM;
	if (next_ops->reboot)
		res = next_ops->reboot(cmd);
	return res;
}
```

Mais il faut alors prévoir que cet éventuel successeur aura besoin de s'enregistrer :

```
static struct security_operations *next_security_ops;

static int mylsm_register(const char *name, struct security_operations *ops)
{
	int res = 0;
	MOD_INC_USE_COUNT;
		if (next_security_ops) {
				res = next_security_ops-register_security(name, ops);
				printk(KERN_INFO "asked for %s registration. returned
%d.\n",
				name, res);
		}
		else {
		next_security_ops = ops;
		printk(KERN_INFO "Registering security module %s.\n", name);
	}
		if (res) MOD_DEC_USE_COUNT;
		return res;
}

static int mylsm_unregister(const char *name, struct security_operations *ops)
{
	int res = -EINVAL;
		if (!next_security_ops) {
				printk(KERN_INFO "no secondary module %s.\n", name);
		} else if (ops == next_security_ops) {
				next_security_ops = NULL;
				printk(KERN_INFO "unregistering module %s.\n", name);
				res = 0;
		} else res = next_security_ops-unregister_security(name, ops);
	if (!res) MOD_DEC_USE_COUNT;
	return res;
}
```

La pratique

Ça a l'air bien, je veux essayer

Pour les noyaux 2.4, il suffit de télécharger le patch [lsm] et de l'appliquer au noyau correspondant. Pour les noyaux 2.6, les LSM sont en standard, donc il n'y a rien d'autre à faire que de télécharger son noyau.

Les répertoires

Le patch LSM va, en plus des ancres réparties un peu partout dans le code, rajouter un fichier d'*includes* `include/linux/security.h`, qui définit la structure `struct security_operations`, ainsi que déclarer l'existence d'une demi-douzaine de fonctions, dont celles nécessaires pour s'enregistrer auprès de la charpente ou d'un module de sécurité.

Le reste du code est placé dans le répertoire `security`, à la racine de l'archive. Il contient le `Makefile` et le menu de configuration `Config.in` (`Kconfig` pour les 2.6), le fichier `security.c` qui implémente les fonctions annoncées dans `include/security.h`, le module `dummy.c` qui est la politique par défaut, et `capability.c` qui implémente les *capabilities*.

Nous avons également quelques exemples ou projets, qui varient selon les patches. Pour les noyaux 2.4, on y trouve un portage de certaines fonctionnalités d'Openwall dans `owlsm.c`, et DTE, LIDS et SELinux dans leurs répertoires respectifs. Pour les 2.6, on y trouve seulement SELinux et `root_plug`, un petit exemple qui permet d'interdire les processus de tourner avec un `egid` valant 0 si une clef USB n'est pas branchée.

Les modules de sécurité

Nous venons d'énumérer les modules de sécurité présents. Le seul qui soit vraiment important est `dummy.c`, car il implémente le contrôle d'accès par défaut, c'est-à-dire celui qui est appliqué lorsqu'aucun module n'est chargé.

Le module `capability.c` est aussi important car il implémente les contrôles qui manquent à `dummy.c` pour retrouver un contrôle d'accès tel qu'on le connaît dans un noyau sans LSM.

Application : création d'un jeu de règles rudimentaire

Afin d'illustrer notre propos, nous avons créé un petit LSM, `dictator`, dont le rôle est de restreindre l'accès à certains objets en fonction du sujet. En fait, cela est bien plus simple (et notre approche bien plus simpliste) qu'il ne semble. Les sources complètes du module sont disponibles à **http://www.security-labs.org/Download/dictator.tgz**.

Les structures

Comme une des caractéristiques des dictateurs est d'appliquer des lois à la tête du client, nous allons faire de même. Nous définissons quelques structures dont nous avons besoin pour décrire nos règles :

```
/* Define the structure for the rules  */
struct restriction {
	char *	r_name;
	u_long	r_inode;
	kdev_t	r_dev;
	u_int	r_perms;
};

/* These mean this right is removed */
#define DONT_READ	0x1
#define DONT_WRITE	0x2
#define DONT_EXEC	0x4
```


MISC POWERED Le magazine 100% Sécurité Informatique

```
/* (Capabilities like, i.e. sorted by subjects) */
struct laws {
    uid_t l_uid;              /* uid to restrict */
    struct restriction *l_rules;    /* inodes uid cant access */
};
```

La structure `restriction` correspond en fait à une loi. Elle indique les permissions (`r_perms`) qu'on retire à un `inode` donné (`r_inode`, `r_dev`). Le nom est uniquement là pour faire joli lors des affichages des logs car nous ne pouvons pas l'utiliser si nous voulons que notre politique de sécurité soit fiable.

En effet, plusieurs fichiers peuvent avoir le même nom. Le seul identifiant unique pour un fichier est son numéro d'inode sur un device donné, c'est pourquoi nous fondons notre politique dessus. Il est possible d'obtenir ces informations par la commande `ls` :

- L'option `-i` donne le numéro d'inode d'un fichier ;
- Dans le répertoire des devices, l'option `-l` affiche les numéros de majeur et mineur servant au calcul du numéro de device.

Par exemple :

```
phil:~$ mount
[...]
/dev/sda2 on /tmp type ext3 (rw,errors=remount-ro)
[...]
phil:~$ ls -al /dev/sda2
brw-rw—    1 root    disk     8,   2 Apr 15  2001 /dev/sda2
phil:~$ touch /tmp/toto
phil:~$ ls -i /tmp/toto
     38 /tmp/toto
```

permet de conclure que, pour l'arborescence telle qu'elle est montée, le couple (device=0x0802, inode=38) représente de manière unique les données contenues dans le fichier, ou plutôt devrait-on dire l'inode. Un lien sur ce fichier aura le même inode, donc nous sommes à l'abri des chemins détournés. Ce qui est important, après tout, ce sont les données, et pas le nom qu'on utilise pour y accéder.

Pour éviter de remplir les informations à la main, nous définissons également la fonction `init_policies()` qui initialise les règles définies en transformant le nom de fichier en paire (`r_inode`, `r_dev`) (cf ci-après). Si une des règles contient une entrée qui n'existe pas, le module n'est pas chargé, et une erreur est émise vers les logs.

Les politiques à la tête du client

Rappelez-vous que les LSM sont restrictifs. Nous allons donc nous contenter d'ajouter des interdits supplémentaires.

Pour l'utilisateur `joker` (uid=500) : tout d'abord, nous protégeons le répertoire `/home/bwayne/tmp` en interdisant l'accès en lecture ou écriture. De plus, nous bloquons tout accès au fichier `/home/bwayne/secret`. Enfin, pour être tranquille, nous interdisons aussi l'accès à `/etc/shadow` :

```
static struct restriction joker_rules[] = {
        /* dir/file                 inode dev  removed perms
*/
        {"/home/bwayne/tmp",         0,  0, DONT_READ|DONT_WRITE},
        {"/home/bwayne/secret",      0,  0, DONT_READ|DONT_WRITE},
        {"/etc/shadow",              0,  0,
DONT_READ|DONT_WRITE|DONT_EXEC},
        {0,                          0,  0, 0} /* trick to have
a null at the end in find_rule_* */
};

static struct laws joker_policy = {
        .l_uid  = 500,
        .l_rules = joker_rules
};
```

De même, on définit une politique pour root, en lui interdisant l'accès aux *homedir* de ses utilisateurs :

```
static struct restriction root_rules[] = {
        /* dir/file             inode  dev  removed perms
*/
        {"/home",                 0, 0,
DONT_READ|DONT_WRITE|DONT_EXEC},
        {0,                       0,  0, 0} /* trick to have a
null at the end in find_rule_* */
};

static struct laws root_policy = {
        .l_uid  = 0,
        .l_rules = root_rules
};
```

Nous pouvons maintenant définir la politique à appliquer sur le système. La politique par défaut ci-après ne sert à rien si ce n'est à indiquer que c'est la dernière "politique à la tête du client" disponible.

```
static struct restriction default_rule[] = {
      {0,                     0,  0, 0}
};

static struct laws default_policy = {
        .l_uid   = -1,
        .l_rules = default_rule
};

/* politiques pour les utilisateurs */
static struct laws *policy[] = {
        &joker_policy,
        &root_policy,
        &default_policy /* must ALWAYS be the last one */
};
```

Mise en place

Maintenant que nous avons tout ce qu'il nous faut, il nous reste à choisir les opérations que notre dictateur va superviser. Comme il est très simple, on se contente de tout ce qui est lié aux inodes, et nous allons donc définir notre comportement dans le hook `dictator_inode_permission()`.

Pourquoi ce choix ? En fait, ce hook est utilisé pour presque toutes les opérations liées aux inodes, ou plus exactement dès qu'on veut accéder à un inode, avant même son ouverture. Ainsi, qu'on veuille se déplacer

dans un répertoire ou créer un lien vers un fichier, on passera forcément par ce hook.

Il existe d'autres points de contrôle pour des opérations spécifiques (création de lien, effacement de fichier, etc.) ou bien pour la manipulation de fichiers. Mais après tout, nous construisons un dictateur, et il ne veut pas qu'on puisse même ouvrir un inode s'il a décidé de l'interdire.

Dernière précision : pourquoi ne pas utiliser `dictator_file_permission()` ? En fait, ce dernier n'est appelé que dans le cas d'une lecture ou d'une écriture dans un fichier. Notre dictateur est plus strict et général que ça.

```
static int dictator_inode_permission (struct inode *inode, int mask)
{
        struct laws *law;
        struct restriction *rule;

        if (!(S_ISLNK(inode->i_mode) || S_ISREG(inode->i_mode) ||
S_ISDIR(inode->i_mode)))
                return 0;

        get_policy(current->euid, law);

        /* permissive default policy */
        if (law == &default_policy) {
                return 0;
        }

        get_rule_by_inode(inode->i_ino, inode->i_dev, law, rule);
        if (rule->r_name) {
                check_perm(mask & MAY_READ, rule->r_perms &
DONT_READ, "read", rule->r_name);
                check_perm(mask & MAY_WRITE, rule->r_perms &
DONT_WRITE, "write", rule->r_name);
                check_perm(mask & MAY_EXEC, rule->r_perms &
DONT_EXEC, "exec", rule->r_name);
        }
        return 0;
}
```

La macro `get_policy()` initialise le pointeur `law` en fonction de l'uid passé en argument. La macro `get_rule_by_inode()` initialise le pointeur `rule` en fonction du numéro d'inode et de device auquel on tente d'accéder. Si on trouve une règle à appliquer, la macro `check_perm()` prend la décision finale en fonction des droits. Si l'accès est interdit, elle retourne `-EPERM` (0 si aucun problème).

En route vers le pouvoir

Une fois compilé et chargé, regardons ce qui se passe. Nous avons bien pris soin de mettre dans nos macros des `printk()` pour nous faire part des décisions prises.

```
root@gotham# insmod dictator.o && tail -f /var/log/kernel/all
08:54:08 gotham kernel: init policy for uid=500
08:54:08 gotham kernel: path_walk() succeed for (/home/bwayne/tmp 43402
2049)
08:54:08 gotham kernel: path_walk() succeed for (/home/bwayne/secret 43406
2049)
08:54:08 gotham kernel: path_walk() succeed for (/etc/shadow 43415 2049)
08:54:08 gotham kernel: init policy for uid=0
08:54:08 gotham kernel: path_walk() succeed for (/home 14911 2049)
08:54:08 gotham kernel: dictator LSM initialized (second=0)
```

En tant que root, nous tentons une approche subtile dans le répertoire `/home/joker` :

```
root@gotham# cd /home/joker
bash: cd: /home/joker: Operation not permitted
Aug 24 09:03:18 gotham kernel: dictator_inode_permission exec
denied for (0, /home, 14911, 2049)
```

La sanction ne se fait pas attendre. Vous remarquerez que nous avions simplement interdit l'accès à `/home` avec l'inode adéquat, mais que cette interdiction porte également sur les répertoires situés en-dessous. En fait, cela est dû à la fonction `link_path_walk()` (dans `fs/namei.c`), qui parcourt chaque élément du chemin en vérifiant si les permissions nécessaires sont présentes (fonctions `permission()` dans `fs/namei.c`). Or, notre ancre est justement située dans cette fonction : nous n'avons donc pas besoin de nous préoccuper de l'arborescence et nous interdisons d'un coup un nœud et ses descendants (quel cruel dictateur n'est-ce pas ? ;-)

En tant qu'utilisateur `joker` maintenant, tentons d'aller voler le secret caché dans `/home/bwayne/secret` :

```
joker@gotham$ cat /home/bwayne/secret
cat: /home/joker/secret: Operation not permitted
Aug 24 09:04:16 gotham kernel: dictator_inode_permission read
denied for (500, /home/bwayne/secret, 43406,  2049
```

En revanche, si nous pouvons bien entrer dans le répertoire `/home/bwayne/tmp`, nous ne pouvons rien faire dedans :

```
joker@gotham$ cd /home/bwayne/tmp
Aug 24 09:43:51 gotham kernel: dictator_inode_permission exec
granted for (500, /home/bwayne/tmp, 43402, 2049)
joker@gotham$ ls
ls: .: Operation not permitted
Aug 24 09:44:50 gotham kernel: dictator_inode_permission read
denied for (500, /home/bwayne/tmp, 43402, 2049)
```

Dernier point, tentons maintenant de lire le fichier de mots de passe :

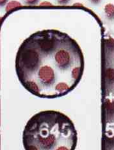
```
joker@gotham$ cat /etc/shadow
cat: /etc/shadow: Permission denied
```

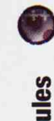
Certes, l'accès nous est interdit... mais rien dans les logs. Bien sûr, cela est normal. Les droits du fichiers sont 600, donc l'utilisateur `joker` ne peut pas lire le fichier. Donc l'accès est refusé bien avant d'arriver dans notre point de contrôle. Rappelez-vous, les points de contrôle sont testés en dernier.

Ce module n'est pas du tout convivial et performant : pour le moment, il n'a que 2 utilisateurs et très peu de restrictions. Donc, l'impact n'est pas trop lourd. Mais je vous laisse imaginer ce que ça pourrait être avec beaucoup plus de contraintes...

Pour les curieux qui se demandent encore quel est ce secret que `bwayne` cherche à protéger, voici la solution :

```
bwayne@gotham$ cat /home/bwayne/secret
batman
```





Conclusion

Ce tour d'horizon des LSM s'achève. La "philosophie" des LSM est de permettre, pas d'imposer. Si vous ne souhaitez pas les activer dans votre noyau, il suffit de le faire lors de la sélection des options. Et au pire, si c'est activé et que vous n'en voulez pas, vous avez les modules `dummy` et `capability` pour vous retrouver en terrain connu. Dans les noyaux 2.6, vous pouvez choisir de n'activer que certains hooks (système de fichiers ou réseau).

Bien évidemment, l'ajout de ces points de contrôle a un impact sur les performances, dont [WC02] donne une estimation sur 3 tests (microbenchmark : LMBench, macrobenchmark : kernel compilation, macrobenchmarks : Webstone). Signalons que ces tests ont été réalisés sur un noyau 2.5.15, et que les choses ont dû évoluer depuis. Dans l'ensemble, les pertes sont inférieures à 2%. En revanche, la manipulation de fichiers (`open()`, `close()`, `unlink()`, `stat()`) subit un impact légèrement supérieur, ce qui se comprend avec notre module `dictator`. En effet, comme nous l'avons montré, lors de la résolution du chemin pour accéder à un inode, les permissions de chaque élément sont testées. L'autre perte concerne le protocole TCP : la fonction `select()` perd 5% et les sockets TCP 8.6%. En conséquence, le taux de connexion est également touché par cette perte de performance (dans les 7% de charge en plus, qui passent même à 16% avec SELinux). Mais répétons-le : ces tests sont vieux et mériteraient sans doute d'être ré-évalués. Enfin, pour conclure, ne croyez pas que les LSM vont résoudre tous les problèmes de sécurité possibles et imaginables. D'abord, personne n'est à l'abri d'une erreur de configuration, et quand on voit la finesse de certaines charpentes, c'est loin d'être facile. Ensuite, il est des choses qu'un LSM ne fait pas. Par exemple, les protections dites "segment non exécutable" (PaX ou OpenWall) ou l'ASLR (*Address Space Layout Randomization*) ne rentrent pas dans ce cadre-là, tout comme certaines modifications introduites dans GRSecurity [`GRSecurity`] (renforcement du `chroot`, amélioration de l'aléa...).

Frédéric RAYNAL
pappy@miscmag.com

Philippe BIONDI
phil@secdev.org
philippe.biondi@arche.fr

Bibliographie

[dte] *Domain And Type Enforcement For Linux*
http://www.cs.wm.edu/~hallyn/dte

[flask] FLASK
http://www.cs.utah.edu/flux/fluke/html/flask.html

[Biba77] K.J. Biba. *Integrity considerations for secure computer systems.* Rapport technique 3153 du MITRE, 1977.

[BLP73] D. E. Bell et L. J. LaPadula. *Secure computer systems: a mathematical model.* Rapport technique 2547 du MITRE, vol. II, 1973.

[GRSecurity] B. Spengler - GRSecurity
http://www.grsecurity.org

[Lampson74] B. W. Lampson. *Protection.* ACM Operating Systems Rev., 8(1) : p. 18-24, 1974.

[lids] LIDS homepage
http://www.lids.org

[lsm] Linux Security Modules homepage
http://lsm.immunix.org

[LSMIntro] Linux Security Modules: *General Security Hooks for Linux.*
http://lsm.immunix.org/docs/overview/linuxsecuritymodule.html

[medusa] Medusa DS9 Security System
http://medusa.fornax.sk

[rsbac] Rule Set Based Access Control (RSBAC) for Linux
http://www.rsbac.org

[SC96] R. S. Sandhu, E. J. Coyne, H. L. Feinstein et C. E. Youman. *Role-based access control models.* IEEE Computer, 29(2) : p. 38-47, 1996.

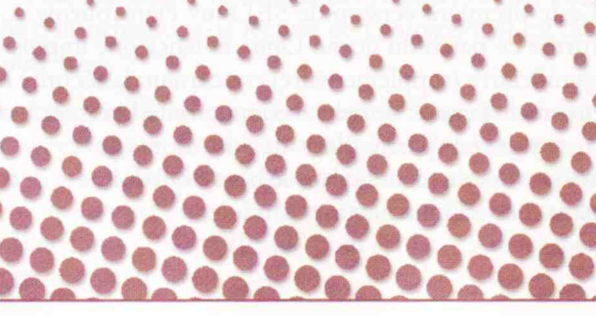
[selinux] Security-Enhanced Linux
http://www.nsa.gov/selinux

[SS75] J. H. Saltzer et M. D. Schroeder. *The protection of information in computer systems.* In Proc. of the IEEE, 9(63) : p. 1278-1308, 1975.

[subdomain] SubDomain
http://www.immunix.org/subdomain.html

[WC02] C. Wright, C. Cowan, J Morris, S. Smalley, G. Kroah-Hartman. *LSM: General Security Support for the Linux Kernel.* In Proc. of USENIX 2002.
http://lsm.immunix.org/docs/lsm-usenix-2002

[ZEJ02] X. Zhang, A. Edwards et T. Jaeger. *Using CQUAL for Static Analysis of Authorization Hook Placement.* In USENIX Security Symposium, San Francisco, CA, August 2002.



SELinux, l'agence de sécurité du noyau

SELinux ou Security Enhanced Linux est une solution de sécurité développée par la NSA qui s'intègre au noyau linux.

Généralités

SELinux est une solution de sécurité offrant un blindage du noyau Linux, en lui apportant des fonctionnalités de protection poussées supplémentaires comme le Mandatory Access Control, etc., que nous allons détailler dans l'article (cf Architecture).

Ce projet correspond à une initiative américaine et a été lancé par la NSA : *National Security Agency*. Cette agence de renseignement qui compte près de 40000 personnes est assez connue du grand public depuis les révélations concernant le projet Echelon, réseau planétaire de surveillance et d'interception de communications [Assemblée2000]. La NSA s'appuya initialement sur des piliers industriels comme MITRE, Secure Computing Corporation et NAI, avec une participation du tissu universitaire américain.

La première mouture publique de SELinux a été donnée à la communauté du logiciel libre (licence GPL) vers la fin 2000 par le *Information Assurance Research Office* qui pilote le projet. Cette entité de la NSA demeure responsable de la recherche et du développement de solutions de sécurité dans le domaine des technologies de l'information, pour des infrastructures sensibles avec une vocation gouvernementale et industrielle (projet à prendre au sérieux, donc).

Cette première version de SELinux était fournie sous forme de patch du noyau Linux et lança une impulsion significative (supplémentaire) concernant la sécurité bas niveau du fameux kernel. La communauté Linux (et Linus Torvalds en particulier) décida alors de créer un framework de sécurité dans le noyau qui donna plus tard naissance à Linux Security Modules.

Avec NAI, la NSA changea alors son implémentation de SELinux pour passer d'un patch à un simple module noyau comme nous le verrons plus loin dans l'article.

En effet, nous commencerons par introduire l'architecture globale de SELinux (Flask...) et son intégration au noyau existant (LSM...), pour ensuite détailler son fonctionnement interne et ce qui façonne ses atouts (TE, RBAC) avant d'aborder rapidement des aspects pratiques comme l'installation ou la configuration. Nous conclurons enfin sur ce que cela apporte concrètement en sécurité, les limites, et terminerons sur le potentiel futur de ce projet.

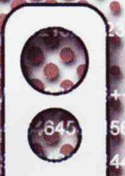
Architecture

SELinux a été conçu sur le modèle de l'architecture Flask [Spencer99]. Au début, l'implémentation avait été faite sous forme de patch pour le noyau Linux. Par la suite, avec la création de l'architecture Linux Security Module (LSM), SELinux a progressivement évolué vers un module de sécurité utilisant les nouvelles fonctions du noyau. Remarquons au passage que l'implémentation de SELinux sous forme de module a donné lieu à une énorme contribution au projet LSM [Smalley01].

Mandatory Access Control

La protection apportée par SELinux est fondée sur le principe de *Mandatory Access Control* (MAC), en français : contrôle d'accès obligatoire. Contrairement au contrôle d'accès typique d'un système Unix, *Discretionary Access Control* (DAC), qui laisse le problème du contrôle d'accès à une ressource à la discrétion de son propriétaire, le but de MAC est de vérifier la légitimité de tous les accès par des **sujets** (processus) à des **objets** (fichiers, sockets...), et ceci à partir d'une politique de sécurité définie au préalable (cf [LMHS16/LSM].

Cependant, le MAC, que l'on retrouve dans de nombreuses solutions de sécurité, n'est pas l'apport



principal de SELinux. En fait, ce qui caractérise vraiment la réalisation de la NSA est l'architecture **Flask**, déployée dans le noyau pour implémenter le MAC.

Nous allons maintenant voir plus en détail l'architecture Flask.

Flask

Flask est un modèle très flexible autorisant l'implémentation de différents modèles de sécurité. Il repose sur une séparation complète entre le code de décision et le code d'application de la politique de sécurité. Par conséquent, SELinux constitue une sorte de gouvernement où le pouvoir législatif et le pouvoir exécutif sont détenus par des entités distinctes.

La partie décisionnelle de l'architecture (*policy decision-making code*) de SELinux est contenue dans un composant particulier du noyau appelé le *security server* (en français, référentiel de sécurité). Cette partie constitue le "pouvoir législatif" dans SELinux. Elle établit les lois à partir de la politique de sécurité à appliquer sur le système.

La partie d'application (*policy enforcement code*) de la politique de sécurité est directement intégrée dans les composants du noyau gérant les processus, les systèmes de fichiers, les IPC et le réseau. C'est le "pouvoir exécutif" de SELinux, responsable de l'application des lois dans le système.

Enfin, SELinux comporte également un système de cache, pour un accès rapide aux décisions d'accès qui ont déjà été calculées par le référentiel de sécurité. Il s'agit de l'*Access Vector Cache* (AVC).

Pour que ce modèle soit indépendant de la politique de sécurité à appliquer, les interactions entre sujets et objets sont modélisées. Chaque entité du système reçoit un **label**, qui est en fait un contexte de sécurité. Celui-ci va contenir des informations sur ce qu'est cette entité, et permettra de déterminer avec quelles autres entités elle peut interagir et comment. En outre, chaque contexte de sécurité est associé à un entier appelé **SID** (*Security IDentifier* ou identifiant de sécurité), pour simplifier leur manipulation. Les significations des labels et SID sont connues uniquement par le référentiel de sécurité, mais c'est la partie "enforcement" qui va attribuer les SID aux entités.

Les objets du système ont également une **classe**, par exemple "file" ou "socket", à laquelle sont associées différentes permissions, comme "open" ou "link" pour un fichier, ou "listen" et "send" pour un socket.

Les décisions prises par le référentiel de sécurité reposent sur les contextes de sécurité attribués au couple (sujet, objet) en interaction, et sur la classe de l'objet. Elles sont de deux types : accès et transitions. Toutes ces décisions sont calculées en fonction de la politique appliquée sur le système (cf Modèles de sécurité).

Les décisions d'accès (*access decisions*) spécifient si une certaine permission est accordée ou refusée pour le couple de SID et la classe de l'objet cible. Les SID sont appelés SID source (celui du sujet) et SID cible (celui de l'objet). Concrètement, le référentiel de sécurité fournit un tableau de l'ensemble des permissions associées à la classe de l'objet pour le couple (SID source, SID cible). Ce tableau est enregistré dans l'AVC, puis la décision d'accès est transmise par l'AVC à la partie "enforcement" pour être appliquée. Le référentiel de sécurité fournit également deux tableaux spécifiant les besoins de journalisation (audit) des permissions associées à la classe d'objet. Un des tableaux (*auditallow*) concerne l'audit en cas de succès, l'autre (*auditdeny*) concerne les cas d'échec.

Les décisions de transition (*labeling decisions*) attribuent des labels aux nouveaux objets. Une décision de transition de processus est faite lors de l'exécution d'un nouveau programme, à partir du SID du processus demandant l'exécution et du SID du fichier à exécuter. Une décision de transition d'objet est faite lors de la création d'un nouvel objet, à partir du SID du programme créant l'objet et du SID d'un objet "parent", typiquement le dossier parent s'il s'agit d'un fichier. Par ailleurs, une application peut spécifier directement le label attribué à un nouvel objet, en utilisant des fonctions de SELinux. Par exemple, un utilisateur qui se connecte par SSH va recevoir un SID qui lui correspond, et non pas le SID par défaut. Il est important de noter, premièrement, que l'application doit être autorisée par SELinux à modifier le contexte de sécurité, et d'autre part, qu'une telle transition doit être autorisée par une décision d'accès correspondante (cf Modèles de sécurité).

Le cas des fichiers enregistrés sur des mémoires de masse (disques durs, etc.) est particulier. En effet, ceux-ci possèdent des SID persistants, ou **PSID**, avec un format identique aux SID normaux. Ces PSID sont enregistrés dans le système de fichiers, et ne sont donc pas perdus lors d'un redémarrage du système.

Modèles de sécurité

SELinux offre la possibilité d'utiliser différents modèles de sécurité. Suivant les modèles utilisés, les possibilités seront différentes, mais les principes de fonctionnement restent les mêmes, comme décrits dans la partie précédente. La politique de sécurité que l'on va pouvoir définir à partir de ces différents modèles est destinée à piloter le mécanisme de Mandatory Access Control. Pour plus d'informations, la lecture de [Amoroso94] est une bonne base.

Actuellement, le référentiel de sécurité implémenté dans SELinux est une combinaison de *Type Enforcement* et de *Role-Based Access Control*. Une implémentation de *Multi-Level Security* est en cours, ainsi qu'une implémentation de la norme CIPSO/FIPS188 pour l'application de labels au trafic réseau.

Comme expliqué dans la partie précédente, afin de pouvoir modéliser les interactions entre sujets et objets de façon efficace, SELinux attribue des labels (contextes de sécurité) aux entités du système. Il va s'agir d'une combinaison des identifiants déclarés par les différents modèles de sécurité. Par exemple : `moutane:sysadm_r:sysadm_t` est le label d'un processus lancé par l'utilisateur `moutane` lorsqu'il agit en tant qu'administrateur système. La signification des différents éléments est expliquée dans la suite de cette partie. Ce label sera associé arbitrairement à un entier appelé SID lors de la compilation de la politique de sécurité.

Type Enforcement

Le modèle de sécurité *Type Enforcement* (TE) implémenté dans SELinux est dérivé du modèle *Domain and Type Enforcement*. Dans DTE, tous les objets et processus du système reçoivent des attributs de sécurité. Ils sont appelés **domaines** pour les processus, et **types** pour les objets. Les domaines et types correspondent à des classes d'équivalence, les processus d'un même domaine et les objets d'un même type étant traités de façon identique. Ce modèle permet donc un contrôle très fin des exécutions et des transitions de domaines.

Le modèle TE diffère de l'original par le fait qu'il n'a qu'un seul type d'attributs de sécurité. Les domaines sont en fait des types qui peuvent être associés à des processus. Ainsi, en interne, SELinux ne fait pas de différence entre domaines et types.

Voici par exemple deux déclarations de types : la première concerne un processus, et la seconde concerne un fichier.

```
attribute domain;
attribute file_type;
attribute sysadmfile;
type syslogd_t, domain;
type resolv_conf_t, file_type, sysadmfile;
```

Après le mot clé `type`, le premier identifiant est le type déclaré, puis les identifiants suivants sont des **attributs** associés au type (on peut déclarer autant d'attributs que l'on veut). On construit ainsi des ensembles de types. Par exemple, l'attribut `domain` va être associé à tous les types de programmes. L'administrateur de sécurité peut alors distinguer domaines et types même si SELinux ne fait pas cette distinction.

Une autre différence de SELinux par rapport au modèle original est la notion de classe d'objet, qui provient de l'architecture Flask. Deux objets ayant le même type, mais des classes différentes, pourront être traités différemment. Par exemple, on pourra traiter différemment un *socket* TCP et un socket Raw IP.

Enfin, le modèle TE n'associe pas directement les utilisateurs à des domaines, mais plutôt à des rôles, comme expliqué dans la partie suivante. TE permet de définir des lois d'accès et des lois de transition. Si aucune loi n'existe pour une demande d'accès donnée, celle-ci est refusée. Quant aux lois de transition, les comportements par défaut sont la conservation du domaine pour les processus, et l'héritage du type du parent pour les objets.

```
type syslogd_t, domain;
type syslogd_exec_t, file_type, sysadmfile, exec_type;
domain_auto_trans(initrc_t, syslogd_exec_t, syslogd_t)
allow syslogd_t self:process { fork signal };
```

Voici quelques lignes tirées de la configuration de `syslogd`. En résumé, `syslogd_t` est le type associé au processus, `syslogd_exec_t` est le type associé au fichier exécutable. La macro `domain_auto_trans` utilisée ici spécifie qu'un programme de type `initrc_t`, typiquement un script de `/etc/rc.d`, peut exécuter le programme `syslogd` (lois d'accès) et qu'alors celui-ci prend le type `syslogd_t`. Enfin, la ligne débutant par `allow` est l'expression directe d'une loi d'accès, spécifiant ici que le programme a accès aux appels système `fork` et `signal`, et ce sur toute entité possédant un label identique (`self`).

Role-Based Access Control

Le modèle original de *Role-Based Access Control* (RBAC) assigne des rôles aux utilisateurs, et associe des permissions à ces rôles. Le modèle de SELinux associe à chaque utilisateur un ensemble de rôles, et associe à chaque rôle un ensemble de domaines TE. SELinux combine ainsi les facilités de gestion du RBAC, et la finesse des contrôles d'accès du modèle TE. Par exemple, la règle suivante indique que le rôle `system_r` (le rôle des processus du système) peut manipuler des objets de type `syslogd_t`.

```
role system_r types syslogd_t;
```

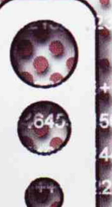
Les rôles sont inclus dans les contextes de sécurité associés aux sujets et objets du système. Pour les objets, il existe un rôle générique `object_r`. Pour les processus, la configuration de RBAC dans SELinux permet de définir des règles de transition de rôles reposant sur les rôles et les types. Il est préférable de limiter les possibilités de changement de rôle. En fait, la première des deux règles suivantes spécifie que seuls les domaines TE ayant l'attribut `privrole` peuvent effectuer cette opération. La seconde autorise la transition de `system_r` vers `sysadm_r`.

```
constrain process transition ( r1 == r2 or t1 == privrole );
allow system_r sysadm_r;
```

SELinux User Identity

Les UID traditionnelles de Linux ne conviennent pas pour SELinux. En effet, un même utilisateur peut utiliser plusieurs UID au cours de son activité sur le système, par exemple lorsqu'il utilise des programmes nécessitant des privilèges. Un programme lancé par `root` peut prendre n'importe quelle UID via les fonctions `set*uid`.

SELinux introduit donc un attribut reflétant l'identité de l'utilisateur dans le contexte de sécurité, complètement indépendant de l'UID. Il est ainsi possible pour SELinux d'effectuer des contrôles sur l'identité de l'utilisateur sans



perturber le système de contrôle d'accès discret (DAC) de Linux.

Dans la politique de sécurité standard de SELinux, seuls certains domaines ont la possibilité de modifier l'identité de l'utilisateur. Par exemple, les programmes `login` et `sshd` ont été modifiés afin d'utiliser les fonctions des bibliothèques fournies avec SELinux. Ainsi, les identités appropriées sont attribuées aux utilisateurs qui se connectent sur le système. En outre, un utilisateur donné conservera en permanence son identité SELinux, même s'il utilise une commande telle que `su`. On obtient ainsi une meilleure traçabilité.

```
user root roles { user_r sysadm_r };
user moutane roles { user_r sysadm_r };
user lolo roles { user_r };
constrain process transition ( u1 == u2 or t1 == privuser );
```

Voici quelques déclarations d'utilisateurs avec les rôles qu'ils peuvent prendre. La dernière ligne se trouve dans le fichier `constraints` et limite la possibilité de changer l'utilisateur aux domaines ayant l'attribut `privuser`.

Multi-Level Security

Le modèle de sécurité MLS est parmi le plu proche du modèle Bell-LaPadula [Bell73]. Il attribue des niveaux de confidentialité aux différents objets et utilisateurs du système. Assez proche de la notion du besoin d'en connaître du milieu militaire, ce modèle était par exemple implémenté dans certains systèmes Unix, notamment des systèmes d'exploitation Cray.

L'implémentation SELinux n'est pas encore terminée, mais il est déjà possible d'associer des niveaux de confidentialité à toutes les entités du système. Ensuite, le contrôle d'accès est simple : on n'a accès en lecture à un objet d'un certain niveau de classification que si l'on possède soi-même un niveau supérieur ou égal et l'on n'a le droit de modifier un objet que si l'on possède un niveau égal.

```
sensitivity unclassified alias u;
sensitivity confidential alias c;
sensitivity secret alias s;
sensitivity top_secret alias ts;
dominance { u c s ts }
```

L'exemple donné ici est la déclaration des différents niveaux de classification utilisés dans MLS, avec la hiérarchie associée.

Labeled Networking Support

SELinux comporte également un modèle de trafic réseau labellisé. Il repose sur une nouvelle option de l'en-tête IP décrite dans CIPSO/FIPS-188. L'objectif est de pouvoir associer les paquets IP à des SID, qui pourront être décodés par le destinataire du trafic. Ce décodage est fait par le protocole *Security Context Mapping Protocol* (SCMP). Ce modèle peut être mis en place sur un parc de machines où les politiques SELinux sont équivalentes, i.e. les utilisateurs, les types, les rôles et les niveaux MLS sont les mêmes sur chaque machine. Ce parc de machines, aussi appelé "périmètre de sécurité", doit être déclaré dans la configuration de la politique. Les SID sont alors transmis de façon transparente à l'intérieur de ce périmètre.

En revanche, ce modèle ne définit aucune protection du trafic. Elle est pourtant nécessaire, et doit être apportée par des moyens annexes, afin de garantir confidentialité, intégrité, imputabilité et protection contre le rejeu.

Intégration de SELinux...

L'intégration de SELinux dans une distribution GNU/Linux ne se fait pas seulement au niveau du noyau. Comme mentionné dans la partie sur RBAC, plusieurs démons et utilitaires doivent être modifiés afin de prendre en compte l'environnement SELinux.

... dans le noyau

A l'intérieur du répertoire contenant les sources du noyau (souvent `/usr/src/linux`), on trouve le dossier `security` contenant les modules utilisant l'architecture LSM. Le code spécifique à SELinux se trouve dans `security/selinux` : dans le sous-dossier `ss`, on trouve la partie "Security Server"; le dossier `selopt` contient l'implémentation de la norme CIPSO/FIPS188.

La partie "enforcement" de SELinux est assurée par les ancres (*hooks*) fournis par l'infrastructure Linux Security Modules. SELinux implémente l'ensemble des ancres, et les fonctions correspondantes sont définies dans le fichier `hooks.c`. On se reportera à [LMHS16/LSM] pour tout ce qui concerne LSM.

... dans les démons et utilitaires

SELinux comporte une bibliothèque, `libsecure`, permettant aux programmes normaux d'utiliser facilement les fonctions de communication avec l'environnement SELinux. En effet, SELinux se trouve dans l'espace noyau, et nécessite donc des mécanismes spéciaux fournis par LSM pour interagir avec les processus standards. Actuellement, cette communication se fait par des entrées spécifiques dans `/proc`.

Deux catégories de programmes utilisent cette bibliothèque : des démons et utilitaires classiques modifiés (`sshd`, `login`, `ls`, `ps`...), et de nouveaux utilitaires fournis avec SELinux (`runas`, `avc_toggle`, `checkpolicy`, `setfiles`...). On se reportera à la partie "Outils et configuration" pour plus de détails.

Guide d'installation

Dans cette partie, seules les étapes les plus significatives de l'installation seront détaillées. Le fichier `README` de l'archive SELinux est un très bon guide. Les chemins cités sont relatifs au dossier d'extraction de l'archive SELinux. Pour commencer, il faut télécharger l'archive contenant les sources de SELinux sur le site de la NSA [SELinux]. Plusieurs options sont possibles ; on choisit

ici de télécharger séparément les sources du noyau Linux 2.4.21, le patch `lsm-2.4` et l'archive `selinux`.

La première étape importante est l'application des patches du noyau. Le patch `lsm-2.4` va insérer les ancres de l'infrastructure Linux Security Module dans les sources du noyau. L'intégration définitive de ces ancres devrait se faire dans le noyau 2.6. Il est ensuite nécessaire d'appliquer un second patch spécifique à SELinux, pour ajouter certaines ancres absentes du patch `lsm-2.4`, ainsi que les options par défaut de SELinux dans la configuration du noyau. Ce patch devrait disparaître avec le noyau 2.6.

Une décision importante lors de la compilation du noyau SELinux est celle qui concerne l'option "NSA SELinux Development Support". Elle permet de faire fonctionner SELinux dans un mode `permissive`, dans lequel aucune action n'est bloquée par SELinux, mais les actions interdites sont enregistrées. Dans ce mode, il est possible de tester des politiques de sécurité sans que les applications ne soient bloquées à cause d'une erreur. Il est possible de passer en mode `enforcing` par la commande `avc_toggle` et vice-versa. Si le noyau est destiné à une machine de développement pour SELinux, il vaut mieux activer cette option. Si le noyau doit être utilisé sur une machine de production, alors il est fortement conseillé de ne pas l'activer.

Après la compilation du noyau SELinux, une autre étape importante est la compilation et l'installation de la politique de sécurité. Au préalable, il est souhaitable de modifier le fichier `policy/users` contenant les rôles associés à chaque utilisateur, en particulier retirer les exemples d'utilisateurs. Il faut également désactiver les gestionnaires de session tels que `gdm`, `kdm` ou `xdm`, qui n'ont pas encore été modifiés pour utiliser l'environnement SELinux. Ensuite, dans le dossier `policy`, la commande `make install-src install` va installer les sources et la version compilée de la politique de sécurité dans le dossier `/etc/security/selinux`.

Le fichier `policy.conf` est la version texte de la politique de sécurité, obtenue à partir des fichiers du dossier `policy`. Les fichiers `.te` contiennent les règles de Type Enforcement, en rapport avec le nom du fichier (par exemple, `admin.te` contient des règles spécifiques à l'administrateur SELinux). Ensuite, le fichier `policy.conf` est compilé par la commande `checkpolicy -o policy.version policy.conf`, la version du langage de configuration de la politique étant obtenue avec `checkpolicy -V` (actuellement `policy.12`).

L'étape suivante est la compilation de `libsecure`, la bibliothèque contenant les fonctions d'interaction avec l'environnement SELinux, puis la compilation des démons et utilitaires modifiés. Ensuite vient la compilation optionnelle des outils spécifiques à la configuration de SELinux (dont il faudra avoir ajouté les fichiers de configuration dans la politique de sécurité). Ces outils permettent de visualiser et de modifier la politique de sécurité de façon légèrement simplifiée, mais il ne faut pas s'attendre à quelque chose de miraculeux. Remarquons également l'installation de l'outil `setfiles`, très important puisqu'il met en place les PSID pour les fichiers. La mise en place de ces PSID est faite, dans le dossier `policy`, par la commande `make reset`, qui va en fait exécuter la commande `setfiles -R` en passant en paramètres la configuration des contextes de sécurité des fichiers et les points de montage des différents systèmes de fichiers. Les contextes de sécurité des fichiers sont contenus dans les fichiers `.fc`. Les PSID sont stockés dans le dossier `...security` à la racine de chaque système de fichier. Les formats supportés actuellement sont Ext2, Ext3 et ReiserFS.

Après la création des labels des fichiers, il est enfin possible de relancer le système. Ne pas oublier d'ajouter le noyau SELinux dans la configuration de son *boot manager*, sans effacer les anciens !

Configuration et outils

Cette partie va décrire rapidement comment configurer les contextes de sécurité des fichiers et modifier la politique de sécurité, puis quels sont les utilitaires spécifiques à SELinux et leurs intérêts.

Comment modifier la politique de sécurité

Les fichiers de configuration se trouvent dans le sous-dossier `policy` du dossier de travail pour l'installation. Le plus simple actuellement est de modifier les fichiers directement dans ce dossier, puis d'utiliser le `Makefile` pour appliquer les changements. Le but de cette partie n'est pas d'expliquer la grammaire de la configuration de SELinux, mais seulement d'indiquer des étapes importantes dans la modification de cette configuration. Pour plus de détails sur les étapes, on pourra se reporter au fichier `README`. Le document [PolicyConf] contenu dans le dossier `doc/policy` explique en détail la grammaire de définition de la politique de sécurité.

Pour modifier ou ajouter la politique associée à une application, deux étapes sont nécessaires. Il faut premièrement modifier les contextes des fichiers. Par convention, ces contextes sont placés dans un fichier `application.fc` situé dans le dossier `file_contexts/program`. On peut trouver des fichiers pour des applications non installées, ce n'est pas grave car le `Makefile` vérifie que chaque fichier `.fc` a bien un fichier `.te` correspondant, et la configuration des applications installées se fait donc plutôt à ce niveau-là.

Une fois que tous les contextes adéquats ont été ajoutés, on peut éditer les règles de Type Enforcement associées au programme. Le fichier concerné est `application.te` dans le dossier `domains/program`. Le sous-dossier `unused` contient des fichiers de configuration pour des

programmes non installés, et il suffit de déplacer un fichier de configuration de ce dossier dans le dossier parent pour ajouter dans la politique de sécurité l'application correspondante. Bien sûr, si l'application que l'on veut ajouter n'a pas de configuration disponible, il faudra l'écrire.

Concernant la déclaration des types pour les règles TE, il est intéressant de remarquer que le dossier `types` contient des types associés aux objets du système (fichiers, socket...) alors que le dossier `domains` contient plutôt les types associés aux processus. De plus, le dossier `macros` contient toutes les macros que l'on peut utiliser dans les fichiers de configuration.

Les outils de configuration

Dans le dossier d'installation, le dossier `tools/setools` contient des applications dédiées à la manipulation de la politique de sécurité : `apol`, `sepcut` et `seuser`. Tous ces utilitaires sont basés sur Tcl/Tk, il faudra donc en disposer pour pouvoir les compiler et les utiliser.

> `apol` est un outil d'analyse de la politique de sécurité. Il lit le fichier `policy.conf`, et permet de faire des recherches sur les types, les attributs, les rôles et les règles TE.
>
> `sepcut` sert à modifier et à tester des politiques de sécurité. Il suffit de lui faire ouvrir le dossier contenant les sources de la politique, et on peut alors lire et éditer tous les fichiers, puis charger la politique de sécurité modifiée pour faire des tests. Cependant, il ne mâche pas le travail : il faut savoir exactement ce qu'on fait lorsqu'on modifie un fichier.
>
> `seuser` est un gestionnaire d'utilisateurs pour SELinux, avec des fonctions d'ajout, de modification et de suppression d'utilisateurs dans la politique en cours d'utilisation. Il existe également trois outils en ligne de commande : `seuseradd`, `seusermod`, et `seuserdel`.

Enfin, on remarquera dans le dossier `scripts` la commande `newrules.pl`, qui peut générer un ensemble de règles à partir des logs générés par SELinux. Les logs de SELinux sont générés dans le noyau et remontés par `klogd`, ils sont donc situés dans le fichier de log destiné à recevoir les messages du noyau.

Les utilitaires classiques

Les utilitaires installés avec SELinux sont de deux sortes : les utilitaires classiques modifiés et les utilitaires spécifiques.

De nombreux utilitaires ont été modifiés pour afficher les informations relatives à SELinux. Voici une liste non exhaustive, mais déjà relativement longue : `ps`, `ls`, `id`, `tar`, `stat`, `strace`, `find`, `logrotate`, `cron`.

Le but est d'afficher les SID ou les contextes de sécurité grâce à de nouvelles options.

Voici des exemples d'exécution de commandes modifiées :

```
moutane@node1:~% /usr/local/selinux/bin/ls –context /
drwxr-xr-x  root   bin    system_u:object_r:bin_t       bin
drwxr-xr-x  root   root   system_u:object_r:boot_t      boot
drwxr-xr-x  root   root   system_u:object_r:device_t    dev
drwxr-xr-x  root   root   system_u:object_r:etc_t       etc
drwxr-xr-x  root   root   system_u:object_r:home_root_t home
drwxr-xr-x  root   root   system_u:object_r:lib_t       lib
drwxr-xr-x  root   root   system_u:object_r:file_t      mnt
dr-xr-xr-x  root   root   system_u:object_r:proc_t      proc
drwx--x---  root   root   system_u:object_r:sysadm_home_dir_t root
drwxr-xr-x  root   bin    system_u:object_r:sbin_t      sbin
drwxrwxrwt  root   root   system_u:object_r:tmp_t       tmp
drwxr-xr-x  root   root   system_u:object_r:usr_t       usr
drwxr-xr-x  root   root   system_u:object_r:var_t       var

moutane@node1:~% /usr/local/selinux/bin/ps ax –context
  PID  SID CONTEXT                          COMMAND
    1    7 system_u:system_r:init_t         init
    2    1 system_u:system_r:kernel_t       [keventd]
    3    1 system_u:system_r:kernel_t       [ksoftirqd_CPU0]
    4    1 system_u:system_r:kernel_t       [kswapd]
    5    1 system_u:system_r:kernel_t       [bdflush]
    6    1 system_u:system_r:kernel_t       [kupdated]
   78  207 system_u:system_r:syslogd_t      /usr/sbin/syslogd
   83  208 system_u:system_r:klogd_t        /usr/sbin/klogd -c 3 -x
   85  210 system_u:system_r:inetd_t        /usr/sbin/inetd
   88  212 system_u:system_r:sshd_t         /usr/sbin/sshd
  112  217 system_u:system_r:local_login_t  login – moutane
  113  216 system_u:system_r:getty_t        /sbin/agetty 38400 tty1 linux
  118  218 moutane:sysadm_r:sysadm_t        -zsh
  138  218 moutane:sysadm_r:sysadm_t        -su
  155  221 moutane:sysadm_r:honeyd_t        /usr/local/bin/honeyd -f
config.default

moutane@node1:~% id
uid=1000(moutane) gid=1000(moutane) groups=1000(moutane)
context=moutane:sysadm_r:sysadm_t sid=218
moutane@node1:~% su -
Password:
root@node1:~# id
uid=0(root) gid=0(root) groups=0(root),10(wheel)
context=moutane:sysadm_r:sysadm_t sid=218
```

La commande `ls` est lancée ici avec le modificateur `context`. En plus des informations classiques telles que les permissions, UID et GID, on obtient les contextes de sécurité des objets.

Par exemple, le dossier /root possède le contexte de sécurité system_u:object_r: sysadm_home_dir_t, avec la signification suivante : il est associé à l'utilisateur SELinux system_u, qui représente le système d'exploitation ; il a le rôle object_r, rôle générique des objets du système ; enfin il a le type sysadm_home_dir_t, qui caractérise le dossier de l'administrateur. Ce contexte de sécurité est défini dans le fichier types.fc de la politique de sécurité.

La commande ps lancée ici avec le modificateur context affiche les contextes de sécurité ainsi que les SID pour chaque processus.

L'exemple de la commande id montre que même si moutane devient root avec la commande su, il reste moutane pour SELinux.

De nouveaux utilitaires font également leur apparition. Certains sont très simples : runas va lancer une commande sous un certain contexte de sécurité, sous réserve d'accord dans la politique de sécurité.

run_init peut lancer une commande dans le contexte de sécurité du programme init. newrole lance un shell sous un nouveau contexte de sécurité. spasswd est l'utilitaire passwd modifié pour SELinux, pour tenir compte du contexte particulier du fichier /etc/shadow.

Les utilitaires de manipulation de la politique de sécurité sont très particuliers, et la plupart des opérations requiert le rôle sysadm_r. checkpolicy compile la politique de sécurité, et load_policy la charge. setfiles peut modifier les labels persistants des fichiers, avec deux modes : en temps normal, la commande déduit les SID des contextes de sécurité en faisant appel aux fonctions du noyau; mais lorsque l'on doit créer des labels sans que l'environnement SELinux ne soit chargé, par exemple lors de l'installation, il faut utiliser setfiles -R.

Enfin, lorsqu'on a compilé SELinux en mode "Development", la commande avc_toggle fait passer le système du mode permissive au mode enforcing et vice-versa. La commande avc_enforcing retourne le mode dans lequel se trouve SELinux.

Protection apportée

Sans détailler tous les aspects techniques présentant ce qu'apporte SELinux en sécurité, nous allons succinctement montrer en quoi une machine avec un tel système d'exploitation pourra résister face à la plupart des agressions connues.

Voici une question qui revient souvent quand on compare SELinux et d'autres solutions comme Grsecurity utilisant PAX : est-ce que SELinux bloque les *buffers overflows* et gêne les pirates avec des aléas dans les adresses de fonctions référencées dans des bibliothèques dynamiques, etc. ?

La réponse est négative. En effet, si SELinux ne bloque ou ne gère certaines possibilités utilisées par les agresseurs, il faut comprendre que le concept est différent : une fois l'attaque lancée vers un programme contenant des failles, elle sera limitée à ce que le programme est capable de faire.

On a donc une assurance du principe dit de *containment* permettant de contenir un intrus ayant réussi à percer certaines barrières qui ne dépendent pas de SELinux directement.

L'exemple donné dans [SYSADMIN03] rappelle ainsi que si un service Apache qui tournerait en root se faisait pirater sur un serveur SELinux (via buffer overflow, etc.), alors ce dernier ne pourrait modifier par exemple la configuration du serveur Bind local à ce serveur. En effet, le Security Server de SELinux, utilisant les politiques de sécurité qui ont été chargées dans le noyau, interdira au processus Apache agressé d'agir autrement que ce qu'on lui a imposé.

Usuellement, lorsqu'un pirate rentre sur une machine à distance, il essaie de lancer un shell pour pouvoir visiter à sa guise sa nouvelle conquête. Avec SELinux, l'appel à execl("/bin/sh"...) devra être autorisé à un processus pour que ce dernier puisse lancer un shell.

Typiquement, on va associer un type shell_exec_t aux programmes de type shell et n'autoriser leur exécution qu'aux processus qui ont le besoin, comme les programmes de login.

Par conséquent, pour assurer la sécurité, il suffit de configurer SELinux en respectant le principe de moindre privilège : les politiques de sécurité doivent donner le minimum nécessaire aux processus et utilisateurs pour qu'ils puissent effectuer les actions auxquelles ils peuvent normalement prétendre.

En cas d'intrusion à cause d'un problème applicatif, le noyau pourra contenir l'attaque. Et quand bien même Apache aurait besoin de lancer un shell (par exemple pour des scripts CGI), SELinux peut autoriser des changements de contexte pour ajuster au mieux les privilèges nécessaires en fonction des rôles et actions menées.

En guise d'exemple, nous avons étudié le démon **honeyd** (cf [MISC8]) sur un SELinux (sur une Slackware). Il s'agit d'un démon jouant le rôle de pot à miel en répondant à des requêtes réseau pour simuler des services sur lesquels les pirates perdront leur temps et montreront leurs techniques.

Honeyd est un démon Unix classique qui lit des fichiers de configuration, écoute le réseau, gère des requêtes/réponses réseau, lance des sous-programmes et écrit des traces de sécurité.

Concernant la "labellisation" des fichiers utilisés spécifiquement par honeyd, nous avons donc défini des contextes de fichiers suivants (honeyd.fc) :

```
# [attention on rappelle que ceci est montré pour illustration]
# Le binaire honeyd
/usr/local/bin/honeyd   system_u:object_r:honeyd_exec_t

# Les fichiers de configuration de honeyd
/usr/local/share/honeyd               system_u:object_r:honeyd_conf_t
/usr/local/share/honeyd/nmap.assoc    system_u:object_r:honeyd_conf_t
/usr/local/share/honeyd/xprobe2.conf  system_u:object_r:honeyd_conf_t
/usr/local/share/honeyd/nmap.prints   system_u:object_r:honeyd_conf_t
# Le fichier de paramétrage de honeyd que nous utilisions
# sachant qu'il faut développer le sien
/usr/local/share/honeyd/config.sample  system_u:object_r:honeyd_conf_t

# Les scripts lancés par honeyd
/usr/local/share/honeyd/scripts(/.*)?
system_u:object_r:honeyd_script_exec_t

# Les logs des scripts test.sh et web.sh vont dans /tmp/log
/tmp/log     system_u:object_r:honeyd_script_log_t
# ...
```

Une fois ces ressources labellisées, il faut déterminer l'ensemble des privilèges et possibilités de honeyd. Si un bogue de sécurité existait dans honeyd ou dans un des sous-programmes simulant un service et lancé par honeyd, les pirates seraient bloqués et ne pourraient pas par exemple contrôler la machine visée (honeyd tournant pourtant sous root !).

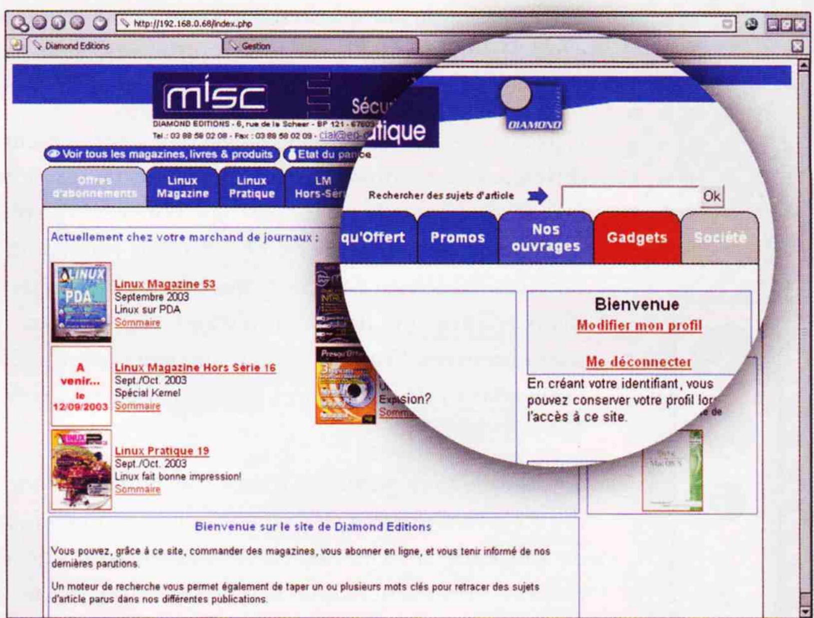
Extrait du paramétrage du Type Enforcement pour honeyd (honeyd.te) :

```
# [attention on rappelle que ceci est montré pour illustration]
# Declarons honeyd comme daemon réseau
daemon_domain(honeyd)
can_network(honeyd_t)

# Type declarations pour honeyd
type honeyd_conf_t, file_type, sysadmfile;
type honeyd_script_exec_t, file_type, sysadmfile, exec_type;
type honeyd_script_log_t, file_type, sysadmfile;
type honeyd_script_t, domain, privlog;

# Honeyd peut être exécuté par init ou l'admin
role system_r types honeyd_t;
role sysadm_r types honeyd_t;

# Quand sysadm lance honeyd, il passe dans le domaine honeyd_t
domain_auto_trans(sysadm_t, honeyd_exec_t, honeyd_t)
# Autorisation à honeyd de lire les fichiers de configuration
# Meme si ces fichiers étaient en écriture (rw-), honeyd ne pourrait
# les modifier, à cause du MAC assuré grâce à ces lignes de politiques
allow honeyd_t honeyd_conf_t:dir { search };
allow honeyd_t honeyd_conf_t:file { getattr read };

# /dev/urandom utilisé par honeyd pour générer des nombres aléatoires
allow honeyd_t random_device_t:chr_file { read };

# Honeyd a besoin du réseau
# La capability net_raw est nécessaire (sniffer réseau à base de libpcap)
allow honeyd_t honeyd_t:capability { net_raw };
# reste du réseau :
allow honeyd_t honeyd_t:packet_socket { bind create getopt ioctl read
setopt write };
allow honeyd_t honeyd_t:rawip_socket { sendto write create getopt setopt };
# ...
```

Si honeyd a été choisi ici comme exemple, c'est par originalité (pas de politique existante à l'heure actuelle) et pour la richesse des actions effectuées par ce démon (réseau, système, etc.). Le fait de créer une politique pour ce programme nous a permis de mieux comprendre la configuration de SELinux. Nos fichiers de tests sont disponibles sur [RSTACK].

DIAMOND EDITIONS

Vous présente son nouveau site !



Nouveau ! Un moteur de recherche ultra pratique et rapide pour vous permettre de cibler dans l'ensemble de nos titres et hors-séries les articles dont les sujets vous interressent.

www.ed-diamond.com

En conclusion par rapport à la sécurité apportée par SELinux, on peut dire que cela permet de contenir des agressions grâce à la mise en place de politiques de sécurité bas niveau appliquées à toutes les interactions système (sorte de "pare-feu" système). Citons par exemple que les attaques classiques d'écriture dans `/dev/kmem` seront donc facilement bloquées (quels processus root ont réellement besoin du droit d'écrire dedans sur une machine sécurisée ?), les *tags* de site Web pourront aussi être interdits (le noyau bloquant l'écriture dans la configuration et dans les pages Web d'un serveur), et votre serveur FTP (`wu-ftpd` au hasard ;-)) ne pourra plus être utilisé comme outil d'administration à distance par des attaquants, etc.

Limites

Complexité d'administration

En regardant en détail le fonctionnement de SELinux, vous aurez sûrement compris qu'une des premières difficultés rencontrées concerne l'exploitation de cette solution.

En effet, si la mise en place est envisageable sur une machine jouant un rôle particulier (serveur Web et DNS, etc.), le déploiement de SELinux sur un réseau de plusieurs machines peut demander un travail important (par exemple sur un parc de postes de travail, etc.).

Notons par exemple qu'il n'existe pas encore de moyens pour gérer le déploiement et la configuration des règles de sécurité à distance. Cela ne serait de toutes façons pas suffisant, car la complexité des règles et fichiers à gérer pour SELinux est telle qu'une compréhension de ce qui est installé sur tout un parc informatique semble illusoire.

Certains outils encore pauvres ou en développement par différents sites (console Webmin, etc.) permettront peut-être de pallier cela, à moins que l'on ne voit plutôt apparaître des solutions clef en main (distributions blindées basées sur SELinux) qui nécessiteraient peu de maintenance et de paramétrage de la part des administrateurs. D'autres outils manquent encore, comme l'intégration du *backup* des contextes de sécurité des fichiers (*labels*).

Notons de plus qu'aucun retour d'expérience véritable et reconnu n'existe à ce jour sur la mise en production de SELinux, la NSA elle-même se gardant un droit de réserve compréhensible sur la question (*Never Say Anything*).

Enfin, dans le cadre de l'ajout d'une application qui n'aurait pas été prévue initialement, donc sans politique de sécurité, l'écriture de cette dernière peut s'avérer très difficile si l'on veut être certain d'appliquer le principe de moindre privilège. Il semble en effet nécessaire de bien connaître à l'avance le fonctionnement de l'application à intégrer dans SELinux. Pour générer des politiques de sécurité, on peut utiliser l'outil `newrules.pl` analysant les logs systèmes et couplé avec le mode `permissive` de SELinux : cela fournit les lignes de politique nécessaires au fonctionnement d'une application (ce qui manquerait pour qu'elle fonctionne si SELinux tournait en mode réel), mais ces dernières sont souvent trop englobantes (on ne peut pas automatiser la définition de type pour les ressources concernées, donc ces lignes sont souvent avec des types génériques, ce qui peut au final poser un problème de sécurité).

Ce travail est donc compliqué et demande encore une forme d'expertise (imaginez les problèmes posés par une mise à jour de nombreuses applications.). *A contrario*, le blindage de `honeyd` avec Grsecurity ou Systrace [MISC8] fut pour nous très rapide (quelques minutes) par rapport au même travail sous SELinux. Mais il est tout à fait normal que la définition d'une politique de sécurité soit une démarche fastidieuse, étant donné que l'approche retenue par SELinux est de déclarer l'ensemble des actions autorisées.

Robustesse

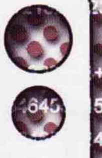
A priori, à ce jour, aucune faille majeure sur le projet SELinux n'a été annoncée publiquement. Par contre, s'il peut transformer le noyau Linux avantageusement, il ne faut pas oublier que la sécurité qu'il apporte sera basée notamment sur la qualité des politiques de sécurité (et leur maintenance, car les applications évoluent rapidement, surtout dans le logiciel libre, et les politiques de sécurité doivent donc suivre).

Aussi, nous avons pu constater que certaines proposées par des contributeurs SELinux ne sont pas peaufinées pour limiter fortement un pirate (les démons peuvent souvent lire tout `/etc`, à l'exception de fichiers particuliers comme `shadow` et la configuration SELinux).

On peut donc s'attendre à un travail de fond sur les politiques de sécurité, notamment sur les applications nécessitant beaucoup de droits.

En effet, si un programme requiert des règles assez souples, un pirate ne pourra peut-être pas obtenir un shell, mais il pourra lancer d'autres actions malveillantes entrant dans le cadre des actions légitimes du programme violé.

Par exemple, prenons le cas d'un serveur FTP qui a besoin de pouvoir ouvrir des sockets TCP vers des ports supérieurs à 1024 n'importe où sur Internet (notamment



pour envoyer les fichiers demandés par un client légitime en mode FTP actif). Si, par exemple, un buffer overflow modifie son comportement, le pirate pourra peut-être essayer d'utiliser le serveur FTP comme rebond vers d'autres services sur Internet.

Autre exemple, que se passe-t-il si un serveur Web se met à ne plus utiliser ses privilèges pour lire les pages Web demandées par des clients, mais qu'il décide (via un buffer overflow) de répondre toujours la même chose pour chaque requête : "ce site Web a été piraté lamentablement" ?

On peut ainsi penser que de nouvelles formes de *shellcodes* et attaques verront le jour, non pas pour contourner les politiques de sécurité mais pour abuser des possibilités offertes dans le cadre des actions légitimes d'une application.

On touche ici la limite de ce genre de techniques de containment, et d'autres solutions comme Grsecurity, qui répondent différemment à ces problèmes liés aux vulnérabilités logicielles, et méritent d'être étudiées.

Un dernier point concernant la sécurité est celui de la surveillance. La lecture des traces de sécurité de SELinux et son interprétation n'est pas encore chose aisée (aucun outil associé) en cas de gros volumes de traces.

De plus, le système *antiflood* de logs fait qu'une limite de vitesse est imposée aux logs de SELinux : passé cette frontière, des logs seront perdus plutôt que de prendre le risque de trop remplir le disque ou perdre en performances (serait-ce utilisable pour cacher la fin d'une séquence d'attaques ?).

Si vous souhaitez regarder rapidement à quoi ressemble un SELinux et sa robustesse, n'hésitez pas à vous connecter via SSH sur les machines suivantes laissées en libre service avec le compte `root` (vous verrez que ce dernier est bloqué et ne peut rien modifier/lire de concret à cause des politiques utilisées par le Security Server de SELinux) :

```
Gentoo SELinux, root en libre service, mot de passe : "gentoo"
$ ssh -a -x root@selinux.dev.gentoo.org
Debian SELinux, root en libre service, mot de passe : "azerty"
$ ssh -a -x root@cose.coker.com.au
```

Pendant le FOSDEM 2003 à Bruxelles, une machine Debian SE Linux était installée de la même manière pour tous les attaquants potentiels désirant tester la fiabilité de SE Linux (concours monté sur place et non annoncé sur le site du FOSDEM) : personne ne perça la machine.

Conclusion et perspectives

Si nous devions coller un seul adjectif à SELinux, nous choisirions sans hésiter le suivant : **ambitieux**.

En effet, les possibilités quant à la sécurité sont très complètes et la NSA a ainsi offert au logiciel libre un bel exemple et élan en termes de recherche et de développement sur la façon de construire un système d'exploitation robuste, fiable et stable.La contre-partie de ce caractère ambitieux réside dans la difficulté de définir des politiques de sécurité fines pour l'ensemble des applications utilisées par un système d'exploitation standard. Cela nécessite un investissement important pour tenir à la fois compte des spécificités des applications, de la grammaire de SELinux et des contextes de déploiement. Par ailleurs, il faut garder à l'esprit que ce genre de solution ne couvre que l'aspect "modélisation" du problème de containment, et à ce titre, d'autres solutions plus simples à déployer, comme [GrSecurity], intégrant notamment des protections contre l'abus des failles logicielles (PaX), semblent complémentaires et méritent d'être prises en compte.

Quel futur attend donc ce projet libre très avancé techniquement ? On voit déjà poindre à l'horizon des solutions de sécurité utilisant SELinux (plateformes robustes dédiées à certaines fonctions, etc.). Par ailleurs, certaines parties de SELinux sont encore en développement, comme `selopt` qui pourrait permettre de transporter les SID au travers d'un réseau sur un environnement homogène composé de SE Linux (un parc informatique, un cluster de calcul...). Le site de SELinux sur [SourceForge] représente une excellente source d'information.

Avec la NSA, force de renseignement américaine colossale derrière un tel projet, ainsi que l'avènement des LSM et du noyau 2.6, gageons que cette solution fera parler d'elle, à moins que des restrictions politiques ne s'y opposent (certains mouvements de pensée aux USA ont annoncé leur étonnement de voir qu'une solution de sécurité était développée grâce aux impôts américains et distribuée gratuitement (sous licence GPL) à tout le monde, y compris leurs "ennemis"...).

Mathieu Blanc
moutane@rstack.org
Doctorant en Informatique au Commissariat à l'Energie Atomique et au Laboratoire d'Informatique Fondamentale d'Orléans (LIFO)

Laurent Oudot
oudot@rstack.org
Ingénieur Chercheur au Commissariat à l'Energie Atomique

Références

[SELinux] *Security-Enhanced Linux.* http://www.nsa.gov/selinux/

[Assemblée2000] Rapport d'information numéro 2623 de l'Assemblée Nationale française du 11 octobre 2000 intitulé "Les systèmes de surveillance et d'interception électroniques pouvant mettre en cause la sécurité nationale". Voir le chapitre I-A nommé "Une organisation vraisemblablement détournée de sa finalité militaire initiale".

[Spencer99] Ray Spencer, Stephen Smalley, Peter Loscocco, Mike Hibler, David Andersen and Jay Lepreau. *The Flask Security Architecture: System Support for Diverse Security Policies*. In *Proceedings of the 8th USENIX Security Symposium*, 1999.

[Smalley01] Stephen Smalley, Chris Vance and Wayne Salamon. *Implementing SELinux as a Linux Security Module.* 2001. http://www.nsa.gov/selinux/module-abs.html

[Bell73] D. E. Bell and L. J. LaPadula. *Secure Computer Systems: Mathematical Foundations and Model.* Technical Report, The MITRE Corp., Number M74-244, May 1973.

[Amoroso94] Edward Amoroso. *Fundamentals of Computer Security Technology.* 1994. ISBN 0-13-108929-3.

[PolicyConf] Stephen Smalley. *Configuring the SELinux Policy.* Last revised: January 2003. http://www.nsa.gov/selinux/policy2-abs.html

[SYSADMIN03] Kerry Thomson. *SELinux.* In *SysAdmin Magazine*, March 2003, Volume 12, Number 3

[MISC8] Arnaud Guignard, Laurent Oudot, V. G. *Honeyd.* In *MISC Magazine* n°8, Juillet-Août 2003.

[RSTACK] Configuration de SELinux pour honeyd (pour test) http://www.rstack.org

[FAQ] *The UnOfficial SELinux FAQ.* http://www.crypt.gen.nz/selinux/faq.html

[Coker] *Security Enhanced Linux information.* http://www.coker.com.au/selinux/

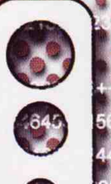
[SourceForge] *SELinux Distribution Integration News.* http://selinux.sourceforge.net/

[HOWTO] *Getting Started with SE Linux HOWTO.* http://sourceforge.net/docman/display_doc.php?docid=15285&group_id=21266

[RedHat] *Red Hat on Security: SELinux.* http://www.redhat.com/solutions/security/SELinux.html

[GrSecurity] http://www.grsecurity.org/

[LMHS16] Frédéric Raynal, Philippe Bondie. Linux Security Modules. In *Linux Magazine Hors-Série* n°16



Abonnez-vous !

L'abonnement d'un an à Linux Magazine
11 numéros

A renvoyer **(original ou photocopie)** avec votre règlement à
Diamond Editions, service des abonnements / Commandes -
6, rue de la Scheer - 67603 Sélestat Cedex

Oui, je souhaite m'abonner à Linux Magazine

Je coche le type d'abonnement choisi :

Durée de L'abonnement	1 An (11 N°s) France	☐ 1 An (11 N°s) Etranger et DOM-TOM
Mode de Paiement	☐ Chèque ☐ Carte Bancaire	☐ CB ☐ Mandat Postal International
Linux Magazine	☐ 53 Euros	☐ 83 Euros
OFFRES DE COUPLAGE		
11 N°s de Linux Magazine + 6 N°s Hors série Linux Magazine	☐ 79 Euros	☐ 128 Euros
11 N°s de Linux Magazine + 6 N°s de Misc	☐ 83 Euros	☐ 128 Euros
11 N°s de Linux Magazine +6 N°s de Misc + 6 N°s Hors série Linux Magazine	☐ 105 Euros	☐ 173 Euros

Nom ....
Prénom ....
Adresse ....
....
....
CODE POSTAL ....
VILLE ....

☐ Je règle par chèque bancaire ou postal à l'ordre de Diamond Editions

☐ Paiement C.B.

N° Carte |_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|

Expire le |_|_|_|_| Date et signature obligatoires :

OFFRES DE COUPLAGE

CHOISISSEZ VOS NUMÉROS DANS CE TABEAU ici

N°	Titre	
~~N°2~~	~~Un système fiable, économique et multiplates-formes~~	épuisé
N°3	Linux : l'enfant terrible de l'informatique	
~~N°4~~	~~Yes, it really happened Linux~~	épuisé
~~N°5~~	~~Linux : C'est comme un peu de couleur, ça change tout~~	épuisé
N°6	GNOME - The Gimp	
N°7	Dopez Linux	
N°8	Le futur résolument objet	
N°9	Prêt pour le jeu !	
N°10	The HURD : 100% GNU	
N°11	Exclusif : l'avenir de G.N.O.M.E	
N°12	NT et Linux : Guerre ou complément?	
N°13	Cryptage : la clé de la sécurité	
N°14	XFree 4.0 : le futur à notre portée	
N°15	Passez à la vitesse supérieure	
N°16	OpenSources : Est-ce suffisant?	
N°17	Linux : Système embarqué	
N°18	Spécial interview : l'avenir de Linux	
N°19	Dossier spécial : Postgre SQL 7.0	
N°20	Un langage puissant pour le web : php4	
N°21	Le protocole Internet du 21e siècle : IPv6	
N°22	Le multi-hreading : Une manière moderne de programmer le Multitâche	
N°23	Débugger sous Linux	
N°24	Palm et Linux	
N°25	Kernel 2.4.0	
N°26	<Dossier> XML </Dossier>	
N°27	Les systèmes de fichiers journalisés	
N°28	Scripting : la force d'Unix	
N°29	L.F.S. Linux From Scratch	
N°30	Le chiffrement des données	
N°31	VPN et tunneling	
N°32	Changez de coquille	
N°33	Open SSH	
N°34	XSL - FO : TeX Killer?	
N°35	QoS et iproute : optimisation et contrôle du trafic IP	
N°36	Linux embarqué : Le projet µGlinux	
N°37	L'impression sous Linux	
N°38	Le desktop Shell : Enlightenment	
N°39	Sécurité : Patchez votre noyau !	
N°40	MySQL : la base de donnée OpenSource	
N°41	Steganographieou l'art de la dissimulation de données	
N°42	Développez vos pilotes de périphérique	
N°43	Administrez facolement votre réseau SNMP	
N°44	Comprenez NetBios pour Maîtriser l'interopérabilité windows -GNU Linux	
N°45	Cohabitation : UnDNS Bind dans un reseau Windows 2000	

POWER Pack X5 — Exemplaires de Linux Magazine** — au lieu de 29.75 €* 15€

☐ **Oui,** je désire acquérir un POWER PACK X5

1. Linux Magazine n°______
2. Linux Magazine n°______
3. Linux Magazine n°______
4. Linux Magazine n°______
5. Linux Magazine n°______

POWER Pack X10 — Exemplaires de Linux Magazine** — au lieu de 59.50 €* 25€

☐ **Oui,** je désire acquérir un POWER PACK X10

1. Linux Magazine n°______
2. Linux Magazine n°______
3. Linux Magazine n°______
4. Linux Magazine n°______
5. Linux Magazine n°______
6. Linux Magazine n°______
7. Linux Magazine n°______
8. Linux Magazine n°______
9. Linux Magazine n°______
10. Linux Magazine n°______

Nom ....

Prénom ....

Adresse ....

....

CODE POSTAL ....

VILLE ....

Magazine	Prix	✓	Total
Power Pack X10			
1 Power Pack X10	25 Euro	☐	____
2 Power Pack X10	50 Euro	☐	____
3 Power Pack X10	75 Euro	☐	____
Power Pack X5			
1 Power Pack X5	15 Euro	☐	____
2 Power Pack X5	30 Euro	☐	____
3 Power Pack X5	45 Euro	☐	____
4 Power Pack X5	60 Euro	☐	____
5 Power Pack X5	75 Euro	☐	____
6 Power Pack X5	90 Euro	☐	____
	Total		
	Frais de port	+	3.81
	Total		

Montant TOTAL 15 Euro + 3.81 de frais de port Le TOTAL s'élève à 18.81 Euro pour l'achat d'un POWER Pack x5

* Si numéros à 5.95 Euro.

** Uniquement du magazine n°2 au n°43, les hors-séries et numéros spéciaux sont exclus des POWER PACK. Seulement en France Métropolitaine !

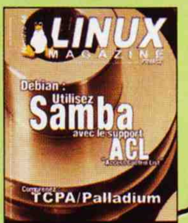
☐ **Je règle par chèque bancaire ou postal à l'ordre de Diamond Editions**

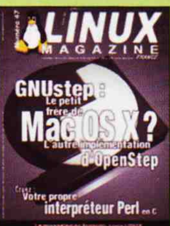
☐ **Paiement C.B.**

N° Carte |_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|

Expire le |_|_|_|_| **Date et signature obligatoires :**

Bon de commande des anciens numéros





Magazine Linux Magazine	Prix N°	Q.	Total
LM N°46 Debian : Utilisez Samba avec le support ACL "Access Contrôl List"	5,95€		
LM N°47 GNUstep : le petit frère de Mac OSX ? L'autre implémentation d'Openstep	5,95€		
LM N°48 Caudium, votre prochain serveur Web !	5,95€		
LM N°49 Après MySQL & PostgreSQL SAP DB La base de données libre & puisante	5,95€		
LM N°50 Créez un album Photo avec PHP ...et sans MySQL	5,95€		
LM N°51 Booster votre site web avec XML grâce à XSLT, CSS & >XPath	5,95€		
LM N°52 Linux Temps réel où en est-on aujourd'hui?	5,95€		
Linux Pratique			
LP N°15 Le Mail de A à Z	5,95€		
LP N°16 DOSSIER MUSIC mp3, cd audio, ogg vorbis, flac	5,95€		
LP N°17 La vidéo sous Linux, DVD DivX, mpeg	5,95€		
Misc : 100% Sécurité informatique			
MISC N°1 Les vulnérabilités du Web !	5,95€		
MISC N°2 Windows et la sécurité	7,45€		
MISC N°3 IDS : La detestion d'intrusions	7,45€		
MISC N°4 Internet, un château construit sur du sable ...ou les protocole réseaux en question?	7,45€		
MISC N°5 Virus mythes et réalités	7,45€		
MISC N°6 Insécurité du wireless? Les differentes normes, fonctionnement, attaques, sécurité	7,45€		
MISC N°7 La guerre de l'information	7,45€		
Linux Magazine Hors-Série			
LM Spécial Débian	5,95€		
LM HS8 Introduction à la crypto	5,95€		
LM HS9 Installer son serveur Web à la maison	5,95€		
LM HS10 Complétez l'installation de votre serveur Internet	5,95€		
LM HS11 Maîtrisez THE GIMP par la pratique	5,95€		
LM HS12 Le firewall votre meilleur ennemi Acte 1	5,95€		
LM HS13 Le firewall votre meilleur ennemi Acte 2	5,95€		
LM HS14 Maîtrisez blender	5,95€		
LM Spécial DVD 3	8,99€		
Frais de port : France métropolitaine 3,81 Euros; U.E. plus Suisse, Liechtenchtein, Maroc, Tunisie, Algérie 5,34 Euros — Total :			
Frais de port :			
Total de la commande :			

NOM .... PRÉNOM ....

ADRESSE .... CODE POSTAL ....

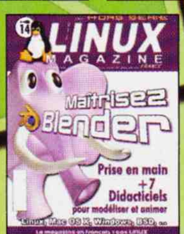
VILLE ....

Mode de règlement

☐ Carte bancaire	Numéro : ______/______/______/______
☐ Chèque bancaire	Date d'expiration ____/____
☐ Chèque postal	Signature :

Frais de port Dom-Tom et autres pays nous contacter au 03 88 58 02 08

A RENVOYER AVEC VOTRE RÈGLEMENT À :
DIAMOND EDITIONS - Service Commandes - B.P. 121 - 67603 Sélestat Cedex



NETWORLD+INTEROP 2003

Exposition · Hall 1 · Paris expo · Porte de Versailles · Paris · France

beinterop

Haut Débit
Sécurité
Wireless
Stockage
Informatique à la demande
Logiciel Libre

19 - 21 novembre 2003

votre *Fast*PASS maintenant sur

www.interop.fr/fastpass

PREMIER ÉVÉNEMENT EUROPÉEN · **INTERNET** · **RÉSEAUX** · **TÉLÉCOMS**

AN INTEROP EVENT

Sous le haut patronage du Président du Sénat, M. Christian Poncelet

Sac à dos pour portable

Ce sac à dos **ergonomique, léger et résistant** comporte deux sangles réglables pour une répartition de poids optimum. Il est composé d'un compartiment principale avec une poche ajustable pour s'adapter à tout type de portable et différentes petites poches de rangement facile d'accès dont une pour un lecteur externe plat (graveur, disquette, DVD,...). Sur le devant vous retrouverez deux poches supplémentaires plus une petite sur le côté pour un téléphone portable. Dimensions : 330 x 420 x 80mm ► Toile résistante de haute qualité

Réf. KS201
Prix : 39,90€TTC/261,73F

Montre Mémoire USB

Joignez l'utile à l'utile! Encore plus fort qu'une clé USB, cette fine montre à mémoire "mass storage" (connectique USB 1.1) vous permet de sauvegarder vos plus précieuses données et de les conserver à votre poignet: clés GPG, SSH, alias, bookmarks, mails importants... et pourquoi pas une Flonix (nouvelle distribution spéciale mémoire USB)! En plus, vous saurez toujours l'heure, même si vous devez vous éloigner de votre PC! La prise USB se range dans le bracelet, une rallonge de 120 cm est incluse à l'envoi! Logiciel de protection par mot de passe fourni. Résiste aux projections d'eau, aux chocs, anti-statique. Une LED indique l'activité (connexion, transfert). ► Dimensions du cadran: 50 x 40 x 12 mm ► Poids total: 45 g ► Alimentation: 1 pile bouton SR626SW (incluse)

Capacité 128 Mo Réf.PE7089 Prix : 79,90€TTC/524,11F
Capacité 256 Mo Réf.PE7091 Prix : 129,90€TTC/852,09F

Câbles USB Lumineux

Ces câbles USB 1 et 2 intègrent des LED qui clignotent lorsque des données circulent

Flash Câble bleu

Longueur 180cm Réf.PE8592 Prix : 6,90€TTC/45,26F
Longueur 300cm Réf.PE8593 Prix : 9,90€TTC/64,94F

Flash Câble rouge

Longueur 180cm Réf.PE8594 Prix : 6,90€TTC/45,26F
Longueur 300cm Réf.PE8595 Prix : 9,90€TTC/64,94F

Flash Câble blanc

Longueur 180cm Réf.PE8596 Prix : 6,90€TTC/45,26F
Longueur 300cm Réf.PE8597 Prix : 9,90€TTC/64,94F

CD-R carte de visite 50Mb/5mn

CD-R par 5 Réf. PE99 Prix : 7,47 €TTC/ 49,-F
CD-R par 10 Réf. PE5093 Prix : 13,95 €TTC/91,51F
CD-R par 20 Réf. PE100 Prix : 22,95 €TTC/150,54F

Mini CD-R 8cm 185 Mb/21mn

CD-R par 5 Réf. PE5013 Prix : 6,90€TTC/45,26F
CD-R par 20 Réf. PE5014 Prix : 23,78€ TTC/156,-F

Montre - réveil radio pilotée

Ce réveil élégant est doté d'une fonction très pratique : **l'heure se règle toute seule**, même lors du passage de l'heure d'hiver à l'heure d'été (et vice et versa). Un large écran LCD vous indiquera l'heure actuelle, le jour de la semaine en cours ainsi que la date ► Ecran rétro éclairé ► Affichage du jour en 5 langues différentes ► Fonctionne dans tous les pays d'Europe excepté la Grande Bretagne ► Alimentation : 2 piles AA (non incluses) ► Dimensions : 100x86x45mm

Réf.PE4620 Prix : 4,90€TTC/32,14F

Lampe de poche à LED sans pile

Totalement écologique et pratique, cette lampe de poche fonctionnera en tout temps sans piles! Il suffit de la secouer de 15 à 30 secondes pour qu'elle éclaire pendant environ 5 minutes (selon le principe non breveté d'induction magnétique de Faraday). L'éclairage est assuré par une LED dont la durée de vie est quasi éternelle (11 ans en continu). Avec cette lampe, vous êtes sûr d'avoir toujours une source de lumière quoiqu'il arrive! Très utile pour trouver les vis qui ont roulé par terre! Munie d'un interrupteur marche/arrêt. ► Longueur: 240 mm ► Diamètre: 50 mm ► Poids: 270 g

Réf.PE4665 Prix : 9,90€TTC/64,94F

NAPA MDC381 FM

Un mini lecteur au design très soigné qui accepte aussi bien les CD audio que les CD MP3 et les CD WMA ! Pour rajouter à ces fonctionnalités, il est également muni d'un tuner FM. Quoiqu'il arrive, vous êtes sûr de pouvoir toujours écouter de la musique. ► Buffer anti-choc de 960 / 480 / 160 secondes (WMA / MP3 / CD audio) ► Télécommande ► Affichage des informations ID3

Réf. PE710 Prix : 109,90€TTC/720,90F

Etui pour 24 Mini CD 8cm

Transportez jusqu'à 24 mini CD avec un encombrement minimum grâce à cet étui. Les pochettes transparentes vous permettent d'identifier les CD d'un simple coup d'oeil. Réf. PE8994
Prix : 6,90€TTC/45,26F

Lampe USB pour Notebook

Livrée sans portable

Pour coder la nuit, n'importe où, même au lit, cette mini-lampe se branche sur le port USB et éclaire votre clavier d'une lumière blanche et agréable grâce à une LED ultra puissante et peu gourmande en énergie (30 mA). Montée sur un câble de type spirale, étirable sur environ 40 cm, cette lampe complètement orientable se fixe au bord de l'écran d'un portable et ce, sans danger d'éraflures car des mousses recouvrent la pince de fixation.

Réf.PE5800 Prix : 13,90€TTC/91,18F

Rétroviseur à écran

Ne vous faites plus surprendre inopinément pendant une session de Quake, d'IRC, ou pire encore... Ce miroir autocollant et orientable, que vous fixerez à votre moniteur, vous permettra de voir derrière vous et de réagir rapidement sur l'envoi de SIGSTOP ou SIGKILL! Une pince est même intégrée au dispositif : vous pouvez y attacher quelques documents d'importance vitale, ou qui font allusion au travail !

► **Diamètre du miroir :** 47 mm
Réf.PE4893 Prix : 2,90€TTC/19,02F

www.pearl.fr

Demandez gratuitement votre Catalogue 124 pages

PEARL Diffusion 6, rue de la Scheer
Z.I. Nord - B.P. 121 - 67603 SELESTAT Cedex

0,12 €/mn
N° Indigo 0 820 822 823

NOMS DE DOMAINE • EMAILS PERSONNALISES • HEBERGEMENT • CREATION DE SITE • E-COMMERCE • REFERENCEMENT

Hébergez votre site à partir de 1 € ! HT / MOIS

NOM DE DOMAINE

NOUVEAU ! En option : hébergement **1 €** HT/MOIS par tranche de 10 Mo

Pack Web Nom........................ à partir de **1 €** HT/MOIS
Votre nom de domaine .com, .net, .org, .fr, .info, .biz… + service DNS + redirection web…
Propriétaire de votre nom de domaine, vous maîtrisez intégralement la gestion de votre nom grâce à nos outils gratuits d'administration en ligne (ex.: modification de DNS, changement de registrar…)

EMAILS PERSONNALISES

NOUVEAU ! En option : hébergement **1 €** HT/MOIS par tranche de 10 Mo

Pack Web Adress........................ à partir de **2 €** HT/MOIS
Votre nom de domaine .com, .net, .org, .fr, .info, .biz… + redirection web transparente + redirection illimitée de vos emails…

Pack Web Mail........................ à partir de **3 €** HT/MOIS
Votre nom de domaine .com, .net, .org, .fr, .info, .biz… + redirection web transparente + 10 comptes POP + redirection illimitée de vos emails + alias illimités + webmail + répondeurs + listes de diffusion…

HEBERGEMENT

Pack Web Pro........................ à partir de **7,5 €** HT/MOIS
Votre nom de domaine .com, .net, .org, .fr, .info, .biz…+ redirection web transparente +10 comptes POP (ext. à 100) + alias illimités + webmail + répondeurs + listes de diffusion + hébergement dynamique 100 Mo (ext à 1 Go), PHP4, Perl 5.0, 2 bases MySQL, FTP/CGI privés, FrontPage 2002, statistiques, trafic illimité…

Pack Serveur Privé Linux ou Windows........................ à partir de **19 €** HT/MOIS
Les fonctionnalités d'un serveur dédié au prix d'un mutualisé !
300 Mo (ext. à 1,5 Go), trafic illimité, 40 applications pré-installées (Python, Tomcat…)…

OUTIL DE CREATION DE SITE EN LIGNE

Testez **gratuitement** Web Site Creator

Web Site Creator > **Hébergement offert !**........................ à partir de **0,5 €** HT/MOIS
7 étapes pour construire vous-même votre site web de qualité professionnelle. Avec Web Site Creator, profitez d'un hébergement offert par Amen ou optez pour celui de votre choix ! Via une interface web, notre outil de création **Web Site Creator** vous permet de créer, d'éditer et de mettre à jour votre site en toute autonomie, à partir de milliers de combinaisons possibles !

INNOVATION 2003

CATALOGUE ET BOUTIQUE EN LIGNE

Power Boutique Start > **Hébergement offert !**........................ **62 €** HT/MOIS
Publiez et actualisez votre catalogue en ligne en français et en anglais sans intermédiaire et en un temps record ! Gestion d'articles illimités, graphisme 100 % personnalisable…

Power Boutique Pro > **Hébergement offert !**........................ **97 €** HT/MOIS
Créez et gérez votre boutique en ligne en toute autonomie !…

REFERENCEMENT MANUEL AVEC GARANTIE DE RESULTATS

Référencement professionnel (10 mots clés)........................ à partir de **49 €** HT/MOIS

Avec plus de **40 000 sites hébergés** et **140 000 noms de domaine gérés**, Amen est **le leader français** sur le marché des services liés à la présence sur Internet.

Grâce à **une innovation permanente**, **un rapport qualité /prix** inégalé, **une relation Clients personnalisée**, **un support technique 24h / 24, 7j / 7…** Amen vous apporte les solutions adaptées à tous vos besoins.

Depuis 1999, plus de 90 000 clients nous font confiance. Et vous ?

NOS ENGAGEMENTS * :

0892 55 66 77

0,34 € TTC/mn depuis la France 9H - 19H

www.amen.fr

Paiement sécurisé