Backdoors et rootkits avancés

Nicolas Dubée
Sté Secway
ndubee@secway.com

Résumé
La confiance envers ses systèmes de surveillance est un composant essentiel d'une sécurité maîtrisée. Hollywood est grand consommateur de la notion, avec l'ingrédient essentiel de tout bon film d'action – le gardien et sa caméra de surveillance trompés par une photographie plantée juste devant. Les concepteurs des logiciels actuels semblent pourtant avoir oublié ces préceptes simples ; derrière la complexité des systèmes d'information et des moyens à mettre en œuvre pour les protéger se cache le vice des fondations. A qui se référer lorsque aucune des composantes du système n'a été prévue pour assurer cette base de confiance dont les applications auraient tant besoin ? Cet exposé tentera d'illustrer la notion de base de confiance et son application aux systèmes d'exploitation actuels. Au travers de l'exemple de « rootkits kernel » sous Solaris, nous démontrerons comment les meilleurs logiciels de sécurité sont mis en défaut par la compromission du système d'exploitation sous-jacent.

Abstract
Trust in your monitoring systems is a key component in a well-designed security. Hollywood movies are great consumers of this idea, with the essential ingredient of every good robbery movie – the security guard and his security console being fooled by a picture put just in front of the camera… However, the computer industry seems to have forgotten those simple concepts; behind the complexity of today's information systems and the means aimed at securing them stands flawed foundations. Who can we refer to when none of the components of the system has been designed to ensure the trust applications need? This document will show the notion of trust base and its application to modern operating systems. Through the example of so-called "kernel backdoors", we will explain how the best security software can be defeated by the compromising of the underlying operating system.

1. Introduction

La confiance envers ses systèmes de surveillance est un composant essentiel d'une sécurité maîtrisée. Hollywood est grand consommateur de la notion, avec l'ingrédient essentiel de tout bon film d'action – le gardien et sa caméra de surveillance trompés par une photographie plantée juste devant. L'informatique semble pourtant avoir oublié ces préceptes simples ; derrière la complexité des systèmes d'information actuels et des moyens à mettre en œuvre pour les protéger se cache le vice des fondations. A qui se référer lorsque aucune des composantes du système n'a été prévue pour assurer cette base de confiance dont les applications auraient tant besoin ? Cet exposé tentera d'illustrer la notion de base de confiance et son application aux systèmes d'exploitation actuels. Au travers de l'exemple de « backdoors kernel » sous Solaris, nous démontrerons comment les meilleurs logiciels de sécurité sont mis en défaut par la compromission du système d'exploitation sous-jacent.

2. Les systèmes d'exploitation modernes

2.1 Définitions

Une **backdoor** est une porte dérobée logicielle permettant à un pirate de revenir ou d'élever ses privilèges plus facilement sur le système.

Un **rootkit** est une modification non autorisée d'un ou plusieurs composants légitimes existants du système, compromettant son intégrité et changeant son fonctionnement normal. Un rootkit peut par exemple désactiver les fonctions de journalisation des évènements, ou insérer des backdoors dans les programmes critiques du système.
De ce fait, un rootkit est souvent un pack implantant sur un système un ensemble de backdoors. Même s'ils ne se réfèrent pas strictement à la même notion, nous utiliserons les deux termes rootkits et backdoors de façon interchangeable dans la suite du présent document.

2.2 Modèles actuels

Au cœur de tout système informatique, le système d'exploitation est devenu bien plus que la simple interface matériels-logiciels traditionnelle et est maintenant, derrière une façade simple, un logiciel très complexe.

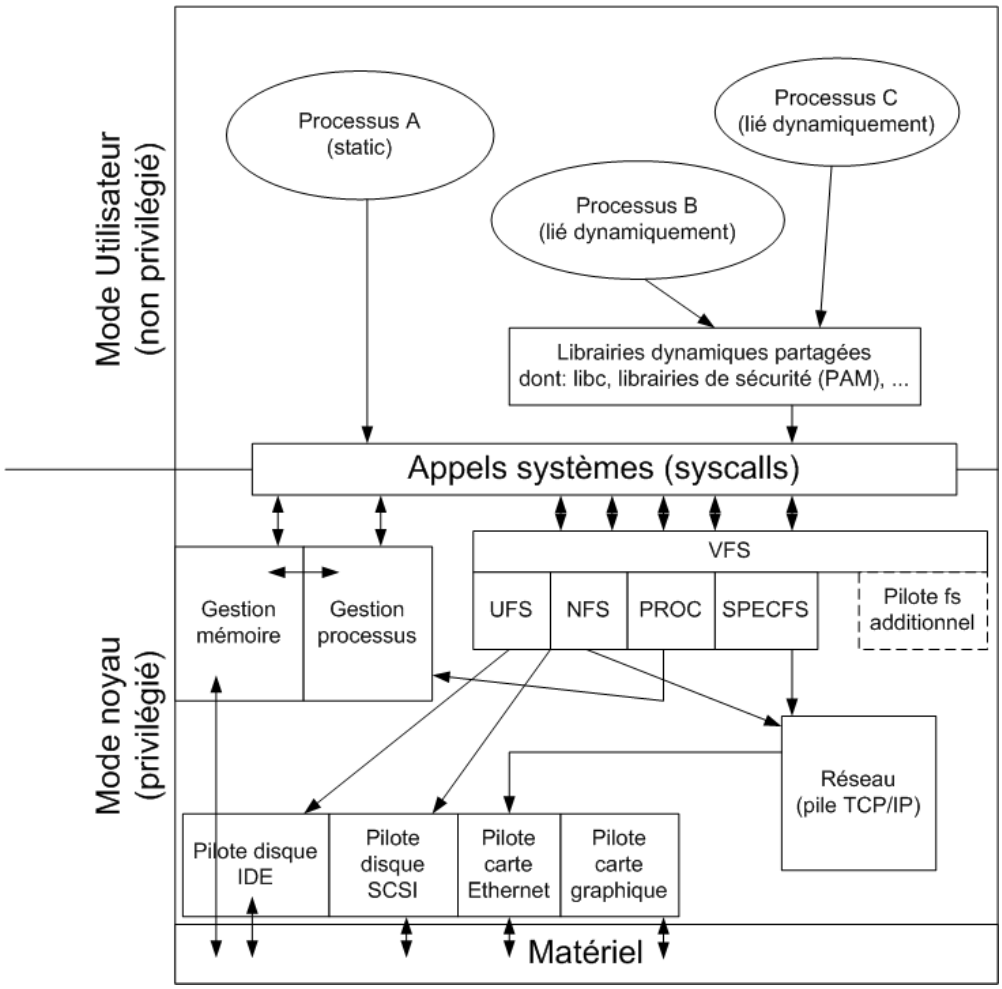

Figure 1: Modélisation d'un système d'exploitation

Le modèle de référence est constitué de deux niveaux ou modes de fonctionnement :

- **Un mode noyau**, mode privilégié dans lequel tourne le cœur même du système d'exploitation, souvent appelé *kernel* et découpé en de nombreux pôles de fonctionnalités.

Ce mode est dit privilégié car il dispose d'un accès et un contrôle complet à toutes les ressources matérielles ou logicielles du système ; de ce fait le système d'exploitation est souvent considéré comme l'interface entre les applications et le matériel.

Chaque pôle peut mettre ses services directement à disposition des processus grâce à l'interface des appels systèmes. Les appels systèmes constituent un mécanisme permettant aux processus d'appeler des fonctions données du kernel. Ces fonctions, environ 200 sur les systèmes Unix traditionnels, sont les primitives de base de toutes les opérations d'entrée-sortie, de gestion de la mémoire, des processus, …. Ainsi, l'ouverture et la lecture dans un fichier se font par les appels système *open()* puis *read()*, mises à disposition par le pôle gestion des fichiers (VFS) du kernel.

Le pôle gestion mémoire s'occupe de gérer la vision mémoire de chaque processus en relation avec la mémoire physique. Cette correspondance est faite par l'intermédiaire de morceaux de mémoire appelés « pages ». D'autre part, le pôle gestion des processus travaille pour organiser les processus, de leur création jusqu'à leur mort en exportant des fonctions de création de processus (fork()), de réglage des priorités, de gestion des signaux.

Le système propose aussi un accès transparent par un mécanisme dit VFS à de nombreux types de systèmes de fichiers, aussi bien des systèmes de fichiers « classiques » servant à gérer les arborescences sur des disques physiques que des systèmes de fichiers virtuels complexes comme *procfs*, *specfs*. *Procfs* permet, via la hiérarchie dynamique */proc*, un accès aux informations d'état sur chaque processus du système (un répertoire par processus actif). Sur de nombreux systèmes Unix, la commande de liste des processus en cours d'exécution *ps* est implémentée en parcourant l'arborescence /proc. *Specfs* propose quant à lui un accès de type fichiers aux ressources matérielles grâce à l'arborescence */dev*.

```
bash-2.04#
bash-2.04# ls -l /proc/116
total 3766
-rw-------   1 root     root     1904640 Feb 24 16:42 as
-r--------   1 root     root         152 Feb 24 16:42 auxv
-r--------   1 root     root          32 Feb 24 16:42 cred
--w-------   1 root     root           0 Feb 24 16:42 ctl
lr-x------   1 root     root           0 Feb 24 16:42 cwd ->
dr-x------   2 root     root        4144 Feb 24 16:42 fd
```

Figure 2: Aperçu de la hiérarchie proc pour le processus 116

La gestion des fichiers et périphériques est implémentée en utilisant largement des *dispatchs*, (aiguillages) : un ensemble de fonctions de base est défini, chaque pilote système de fichiers ou périphérique est tenu d'implémenter ces fonctions de base. Celles-ci incluent par exemple *open()*, *read()*, *write()* et *close()*. Chaque pilote exporte alors au kernel la liste des fonctions qui implémentent ces appels standards, une table des correspondances est ainsi créée :

fat_open()	implémente *open()* pour le système de fichiers FAT
fat_read()	implémente *read()* pour le système de fichiers FAT
fat_write()	…

Quand un programme ouvre un fichier, il appelle la fonction générique *open()*. Le kernel détecte alors quel est le type de périphérique ou de système de fichiers derrière le fichier demandé en ouverture,et appelle grâce à la table des correspondances la « bonne » fonction *open()*.

- **un mode utilisateur**, dans lequel tournent des applicatifs appelés processus, formes « vivantes » des programmes stockés sur disque.

Chaque processus s'exécute en tant qu'un utilisateur donné, dans un contexte qui lui est propre et dans un espace mémoire virtuel géré par le système d'exploitation. Il n'a aucune vue directe des autres processus, de leur mémoire, ou même des ressources matérielles. Il doit obligatoirement se référer au système d'exploitation par l'intermédiaire des appels systèmes pour accéder aux ressources systèmes ou communiquer avec les autres processus.

Ces processus sont traditionnellement créés par un appel système appelé *fork()*, et le fils ainsi créé par un appel à *fork()* est une copie conforme mais complètement indépendante du parent, qui continue normalement son exécution. Ainsi, chaque processus créé hérite des paramètres

du père, notamment de son contexte de sécurité, c'est-à-dire de l'utilisateur sous lequel il s'exécutait.

2.3 Fonctionnalités de sécurité des systèmes d'exploitation

Les systèmes d'exploitation modernes fournissent de nombreuses fonctionnalités de sécurité.

Cependant, l'implémentation de telles fonctionnalités n'est la plupart du temps pas réalisée au cœur même du système d'exploitation (*kernel*), mais largement dans des applicatifs mode utilisateur fournis avec le système. Ainsi, l'invite de login Unix n'est rien d'autre qu'un programme aucunement particulier, sinon qu'il est lancé au démarrage par un autre processus (*init*) ou par les gestionnaires de liaisons réseaux (*telnetd*). D'un point de vue machine, rien ne différencie */bin/login* d'autres programmes, le coeur du système d'exploitation ignore complètement le rôle si crucial de login.

```
bash-2.04# ls -l /bin/login /bin/ls
-r-xr-xr-x   1 root     bin        29512 Mar 14  2000 /bin/login
-r-xr-xr-x   1 bin      bin        17440 Jul 16  1997 /bin/ls
bash-2.04#
```

Figure 3: Programmes /bin/ls et /bin/login sur Sun Solaris 2.6

On peut alors se demander comment sont réalisées les opérations de sécurité nécessitées par */bin/login*, telles que la vérification des mots de passe et le chiffrement de données. Ces fonctionnalités n'étant pas considérées comme des opérations canoniques, elles ne sont pas implémentées dans le kernel du système, mais dans un ensemble de librairies dynamiques liées aux applicatifs eux-mêmes, et s'exécutant donc dans leur contexte.

Ainsi, le programme */bin/login* vérifie le couple (login, mot de passe) fourni par l'utilisateur qui souhaite s'authentifier. Cette vérification est faite en interrogeant la base des utilisateurs Unix du fichier */etc/passwd*. Les fonctions *getpwent()*,*getpwnam()*,*getpwuid()* implémentées dans la libc s'occupent de cette interrogation ; elles vont elles-mêmes appeler les appels systèmes d'ouverture de fichier (*open()*) et de lecture (*read()*) afin de lire séquentiellement les lignes de la base des mots de passe */etc/passwd* et vérifier la correspondance avec les informations d'authentification entrées par l'utilisateur. Une fois l'utilisateur trouvé dans la base, */bin/login* construit un nouvel environnement pour exécuter le shell qui sera présenté à l'utilisateur. Ce nouvel environnement devant s'exécuter sous l'identifiant (*uid*) de l'utilisateur en question, */bin/login* effectue alors un appel système *setuid()* permettant de positionner l'identifiant utilisateur du processus appelant (c'est-à-dire du nouvel environnement). Le seul contrôle effectué par le kernel avant d'autoriser l'opération *setuid()* consiste à vérifier que le processus appelant s'exécute jusqu'alors en tant que ***root***, le super-utilisateur Unix.

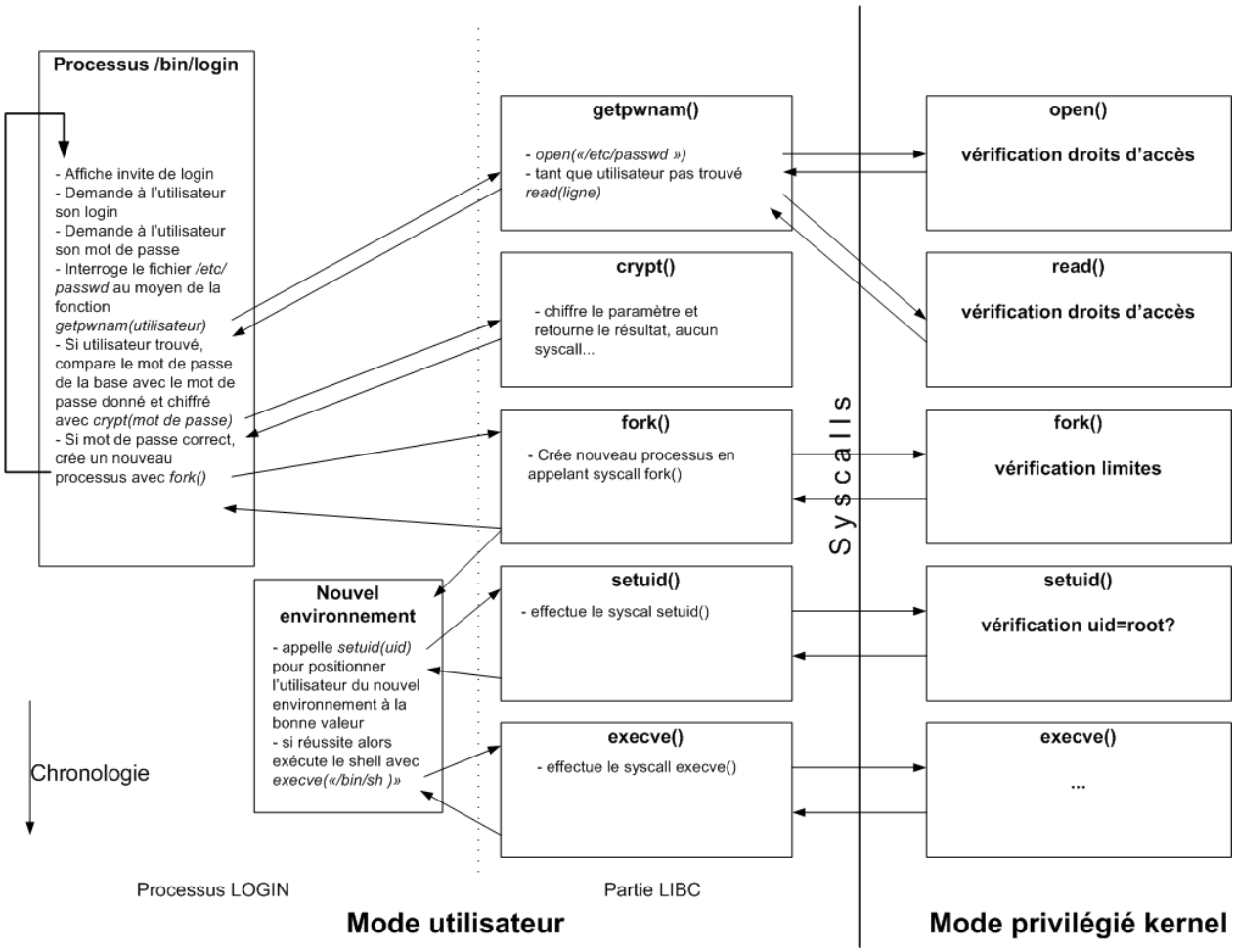

Figure 4: déroulement d'une session de login

Comme nous pouvons voir du diagramme précédent, l'entrée sur le système d'un utilisateur (c'est-à-dire, la vérification de ses informations d'authentification et la création de son environnement) est complètement à la charge de processus s'exécutant en mode utilisateurs, avec une aide minime du kernel.

En effet, les fonctionnalités de sécurité offertes par la plupart des kernels actuels ne concernent que deux aspects :

- **la protection des processus entre eux, notamment au niveau de la gestion mémoire**. De ce fait, deux processus voulant communiquer sont obligés de le faire par l'intermédiaire de fonctionnalités offertes par le kernel, aucune communication directe purement mode utilisateur n'est possible. La protection des processus est un élément extrêmement important des systèmes d'exploitation, notamment en terme de fiabilité et de disponibilité du système car c'est ce mécanisme qui assure aussi qu'un processus n'entraîne pas dans sa chute d'autres processus. En terme de sécurité, la communication entre processus peut être suivie de manière fine par le système d'exploitation puisque toute communication impliquera des appels systèmes.
- **la sécurité du système de fichiers**. C'est le kernel, au niveau de primitives syscall telles que *open()*, *read()*, *write()*, *close(),* qui gère les permissions sur les objets du

système de fichiers au sens large (dont socket réseaux, …). La notion d'identifiants utilisateurs et groupes est largement associée à la sécurité du système de fichiers.
- **la gestion des identifiants d'utilisateurs (uid) et de groupes (gid)**, au moyen d'algorithmes simples (possibilité de passer de l'uid 0 à une uid quelconque sans restriction, opération inverse possible dans certains cas seulement).

Ainsi un outil de détection d'intrusion locale comme Tripwire®[1] met chronologiquement en jeu les fonctionnalités de sécurité kernel suivantes pour assurer son bon fonctionnement :
- Avant son exécution, **sécurité du système de fichiers**, afin d'éviter que le binaire lui-même de Tripwire ne soit remplacé par un cheval de Troie ignorant les modifications commises au système par le pirate.
- Durant son exécution
  - **Protection des processus**, pour empêcher la modification dynamique de la mémoire (code, données) du processus Tripwire par un autre processus pirate situé sur la machine. Ce processus pirate pourrait par exemple prendre au vol le contrôle de la mémoire du processus Tripwire et modifier le code de vérification des fichiers pour qu'il saute les fichiers qui ont été modifiés par le pirate, ceci sans avoir besoin de toucher le binaire Tripwire présent sur disque.
  - **Sécurité du système de fichiers**, garantissant que les fichiers qui sont présentés à Tripwire lorsqu'il les ouvre pour en effectuer des vérifications sont bien ceux stockés sur disque et utilisés par le système. Tripwire repose ainsi sur le fait que le kernel offre la même vision des fichiers pour tous les processus.

2.4 Le paradigme de l'utilisateur root

Le **super utilisateur « root »** est sans doute une des caractéristiques les plus connues des systèmes Unix. Issus d'un héritage historique fort, précédant l'arrivée massive des réseaux, les utilisateurs Unix sont gérés sous le modèle du tout ou rien :
- d'une part, des utilisateurs non privilégiés, identifiés par leur login/mot de passe mais surtout de manière interne par un uid entier strictement supérieur à 0 sont restreints dans leurs actions sur les processus/fichiers/…,
- d'autre part un utilisateur spécial, dit « root » correspondant en interne à l'uid 0 dispose de tous les pouvoirs sur tous les objets du système : aussi bien fichiers que ressources et processus.

Si pour l'administrateur, qui doit accéder à des fonctionnalités critiques, ajouter des utilisateurs ou des services, l'utilisateur root trouve toute son application, au niveau système lui-même, les attributions de root sont pour le moins obscures. On pourrait en effet résumer par le simple fait que l'utilisateur root est, du point de vue kernel, un utilisateur « fourre-tout » n'ayant aucune restriction :
- manipulation complète de tous les objets du système de fichiers
- accès sans restrictions aux périphériques
- accès complet à la mémoire
  - mémoire utilisateur : prise de contrôle d'autres processus, consultation de leur mémoire et même modification (modification dynamique du code ou des données)

[1] Logiciel de vérification de l'intégrité des fichiers, http://www.tripwire.com/

- mémoire kernel : consultation pour surveillance de l'exploitation et modification au vol des paramètres kernel (*tuning* ou ajout de fonctionnalités, pilotes périphériques, …)

De ce fait, tout programme lancé avec les privilèges *root* n'aura sur son chemin aucune barrière de sécurité.

Malheureusement, le modèle précédent induit des effets transitoires forts désagréables. Beaucoup de fonctions systèmes étant légitimement restreintes à l'utilisateur root, nombre de programmes cruciaux nécessitent d'avoir au moins un bout de leur exécution effectuée en tant que root. Tout processus devant à un moment donné accéder à une fonction privilégiée devra être root au minimum de son début jusqu'à l'appel lui-même de la fonction en question.

C'est par exemple le cas des processus serveurs réseaux tels que les serveurs http ou dns, devant obligatoirement être démarrés en tant que *root* pour pouvoir s'attacher au port TCP/UDP d'écoute correspondant[2]. Ainsi, pour environ 10 lignes de code privilégié, des serveurs réseaux complexes nécessitent des privilèges élevés qu'ils gardent bien souvent par la suite.

La suite a très certainement déjà été vécue par la plupart des administrateurs systèmes : vulnérabilité sur tel processus serveur (DNS par exemple) permettant une prise de contrôle du processus, de son chemin d'exécution et donc possibilité pour le pirate d'exécuter à distance n'importe quel code, directement en tant que root.

Dans la suite du document, nous nous intéresserons aux possibilités qu'a un pirate, passé *root* sur un système Unix, pour éviter d'être détecté et s'y maintenir de façon aisée. En particulier, nous verrons comment un agresseur peut rendre inefficients tous les systèmes de sécurité et de détection d'intrusion dès lors qu'il est root, en modifiant la base de confiance qu'est normalement le système d'exploitation.

[2] Ceci car les fonctions de mise en écoute TCP/UDP interdisent aux utilisateurs normaux de se mettre en écoute sur les ports inférieurs à 1024 (dits ports privilégiés, ceci pour éviter que les utilisateurs ne simulent des serveurs légitimes en se mettant en écoute sur des ports privilégiés standards).

3. Compromission totale d'un système par modification du kernel

3.1 Mise en évidence

Nous avons pu ainsi constater que, par manque de fonctions sécurité critiques, la plupart des systèmes actuels sont complètement vulnérables à une faille localisée sur un de leurs composants logiciel. Afin d'illustrer cette affirmation, nous allons maintenant étudier un exemple des possibilités qu'a un pirate ayant obtenu l'accès root sur un serveur pour le compromettre complètement, jusqu'à rendre inopérants tous les moyens de sécurité de la machine en question.

Une des fonctionnalités spécifiées précédemment dans les possibilités de l'utilisateur root était le paramétrage et la modification du kernel.

Alors que ces opérations sont impossibles pour un utilisateur normal, root dispose de plusieurs façons pour modifier un kernel Unix :

- **modification des fichiers kernels eux-mêmes**

Ces fichiers (*/platform/platform-name/kernel/unix* et */kernel/genunix* sous Solaris) constituent les programmes du kernel stockés sur disque, exactement comme tout autre logiciel dispose d'au moins un fichier exécutable sur disque. Ils sont chargés au démarrage de la machine par un mini programme dit *bootstrap*, et leur accès, lors de l'exécution, est limitée par permissions comme tout autre fichier.

La modification des fichiers du kernel sera effective au prochain démarrage du système. Avant cela, les modifications faites aux fichiers kernel sont facilement identifiables puisque le code de mise en échec des logiciels de surveillance n'a pas encore été chargé. Cette manipulation est donc peu pratique pour un pirate qui ne peut risquer à se faire détecter en redémarrant le système. D'autre part, bien souvent (cas des systèmes commerciaux) les sources du kernel ne sont pas disponibles ; les modifications sont donc extrêmement délicates à réaliser. Cependant, cette méthode assure que les modifications effectuées survivront aux redémarrages de la machine et seront en place longtemps, le kernel étant rarement mis à jour. Elle est donc utilisée assez couramment en conjonction avec d'autres méthodes.

- **Ajout au vol de pilotes périphériques ou de modules**

Une des fonctionnalités attendues d'un système d'exploitation actuel est qu'il soit modulaire : que l'ajout de périphériques et donc des pilotes correspondants se fasse le plus facilement possible. Dans cette optique, de nombreux systèmes Unix proposent l'ajout dynamique par root, sans redémarrage, de bouts de code dits « **modules** » ou « **LKM** », tels que des pilotes périphériques. Ces modules peuvent être chargés dans le système d'exploitation à la volée

grâce à des commandes simples. Ils sont alors « digérés » par le kernel et s'exécutent en mode privilégié dans ce dernier, ayant un accès complet et sans restriction à l'ensemble du système.

L'idée d'utiliser cette fonctionnalité légitime des systèmes d'exploitation modernes[3] a été introduite dès 1995 sur les systèmes libres et popularisée par plusieurs articles[4] techniques en 1997, agrémentés des premiers codes publics (Linux et FreeBSD). De nombreux kits sont maintenant disponibles pour modifier de la sorte des kernels Solaris ou Linux, FreeBSD, ….

- **Modification au vol de la mémoire du kernel**

La mémoire étant un périphérique comme un autre, un fichier spécial */dev/mem* lui est souvent associé qui permet la lecture et la modification de la mémoire physique. L'écriture à une position donnée dans ce fichier entraîne la modification de la zone mémoire correspondante. De même, le fichier */dev/kmem* est associé à la mémoire virtuelle du kernel. Ces deux méthodes permettent un accès commode à la mémoire, vue comme un fichier.

Si un de ces fichiers est accessible en écriture, un agresseur peut injecter du code en écrivant dans la mémoire comme dans un fichier, et remplacer par exemple certains appels systèmes sensibles par ses propres versions, pourvues de portes dérobées.

Cette technique, assez peu rencontrée en pratique car très délicate techniquement (la moindre erreur d'un octet peut détruire le système) a cependant été démontrée comme réalisable, et des outils d'injection de la sorte seraient disponibles dans certains cercles pirates. La méthode de prédilection reste celle des modules ; plusieurs kits sont disponibles dans le domaine public utilisant les modules pour injecter du code dans le noyau et en modifier le comportement.

Un pirate ayant obtenu l'accès root à une machine peut donc, en utilisant des fonctionnalités légitimes du système, modifier le système d'exploitation à son gré afin d'être invisible aux systèmes de surveillance locaux et à l'administrateur. Nous allons maintenant voir quelles sont les possibilités et les techniques communément implémentées.

3.2 Techniques usuelles

En modifiant le cœur même du système d'exploitation, le pirate contrôle tous les applicatifs qui tournent sur la machine. Il peut leur présenter une vue biaisée de l'activité du système, et n'a donc plus, comme avec les backdoors traditionnelles, besoin de modifier tous les programmes de surveillance un à un. Les buts recherchés sont le maintien aisé sur le système (en particulier, pouvoir revenir facilement, sans avoir à dérouler l'intrusion une nouvelle fois) et la dissimulation de ses fichiers, de ses processus, de son activité en général. Cette dissimulation doit tromper aussi bien l'administrateur que les outils automatiques de détection.

[3] Quasiment aucun système Unix moderne n'est dépourvu de cette fonctionnalité.
[4] Halflife, *Bypassing integrity checking systems*, Phrack 51/9
Plaguez, *Weakening the Linux kernel*, Phrack 52/18

3.2.1 Dissimulation de processus

Dissimuler des processus est une des premières préoccupations du pirate. En effet, la commande de listing des processus actifs – ps – est sans doute, avec ls, la commande la plus utilisée sur un système Unix. En dissimulant ses propres processus, le pirate peut laisser sans crainte sniffers[5], portshell[6], ….

Plusieurs méthodes existent pour lister les processus actifs sur un système. Toutes ces méthodes reposent sur une interrogation du système d'exploitation, qui est le seul à même de fournir cette liste.
Très souvent, la liste des processus est obtenue en parcourant le répertoire virtuel /proc, qui contient un sous-répertoire par processus actif. Afin de mettre en défaut les programmes reposant sur /proc, la backdoor kernel modifie la partie VFS du kernel, et en particulier les fonctions d'ouverture et de lecture des répertoires. En remplaçant les fonctions originales par ses propres fonctions qui omettent de créer des sous-répertoires pour les processus à cacher, le pirate dissimule les processus qu'il souhaite.

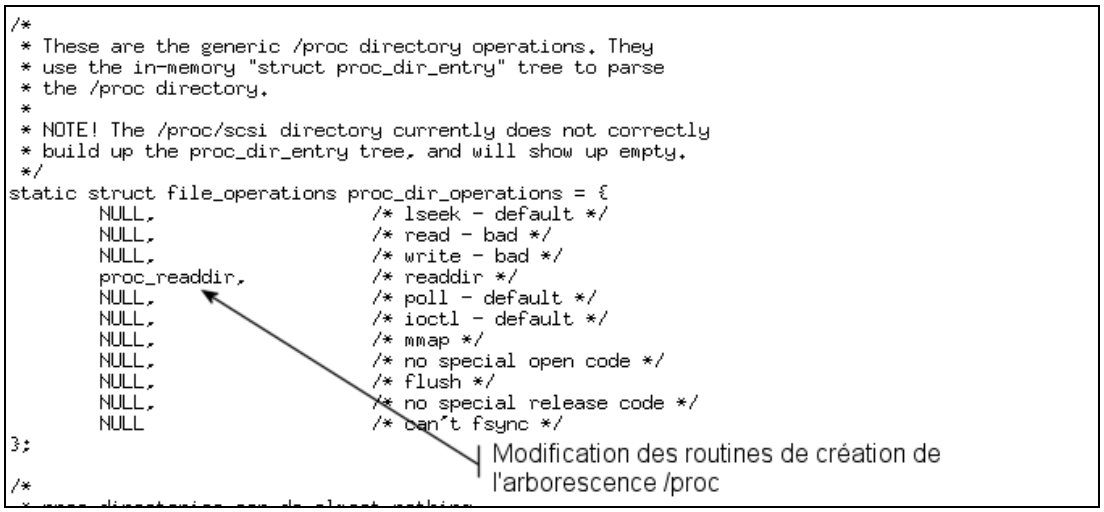

Figure 5: Remplacement par un module de la fonction système proc_readdir par une fonction dissimulant les processus pirates. Lorsqu'un processus cherche à lister le contenu du répertoire /proc, le kernel regarde quelle fonction réalise cette opération dans un tableau des opérations interne au kernel. Il appelle alors la fonction en question. Cet aiguillage peut être détourné par un module ayant accès à la mémoire kernel.

Certains codes publics utilisent même des mécanismes assez poussés à base de signaux pour indiquer dynamiquement au code pirate dans le kernel quels processus il doit cacher, sans avoir besoin de reprogrammer le module. Le module intercepte le signal de type spécial envoyé à un processus par le pirate au moyen de la commande *kill*, et cache le processus en question.

Sous certains systèmes comme Solaris, le listing des processus est aussi obtenu par lecture de la mémoire du kernel au travers du périphérique virtuel */dev/kmem*. Comme précédemment, un aiguillage indiquant au kernel quelle est la fonction à appeler pour ouvrir et lire dans le

[5] Programmes d'écoute de réseaux enregistrant les mots de passe qui y transitent.
[6] Type de programmes très souvent utilisés par les pirates et donnant accès à un *shell* sur un port TCP ou UDP donné.

périphérique virtuel /dev/kmem. Le pirate remplace alors ces fonctions par les siennes, qui « nettoient » les valeurs renvoyées au processus demandeur.

3.2.2 Dissimulation de fichiers

Les fichiers sont dissimulables exactement comme précédemment, en modifiant les aiguillages du pilote système de fichiers correspondant dans le noyau. Ainsi, si l'intrus cherche à cacher des fichiers stockés sur un disque dur formaté en UFS, il modifie la routine UFS de listing des répertoires afin de cacher les fichiers qu'il souhaite. Il veut cependant toujours pouvoir y accéder en lecture ou écriture (consultation d'enregistrements de sniffers par exemple), il ne modifie donc pas les autres fonctions.

```
bash-2.04#
bash-2.04# uname -a
SunOS sol 5.6 Generic_105181-28 sun4m sparc SUNW,SPARCstation-10
bash-2.04#
bash-2.04# ls -al
total 4
drwxrwxr-x   2 bin      bin          512 Mar 13  2000 .
drwxr-xr-x  25 root     sys          512 May 22  2000 ..
bash-2.04#
bash-2.04#
bash-2.04# cat monfichier
ceci est le contenu d'un fichier cache.
fin.
bash-2.04#
```

Figure 6: Dissimulation d'un fichier sous Solaris par modification kernel, pour cet exemple le programme ls n'a pas été modifié. Le fichier « monfichier » n'est pas visible dans le listing du répertoire mais est toujours accessible et modifiable si l'on connaît son nom.

Nous avons pu aussi étudier des modules kernel Solaris pirates très perfectionnés qui introduisaient une fonctionnalité « freeze » dans toute la partie gestion des fichiers du noyau. Cette fonctionnalité permettait de modifier le système de fichiers (accéder en lecture/écriture aux fichiers) sans qu'aucune date d'accès ou de modification ne soit changée . Le code était activé en envoyant un signal spécial au processus sur lequel on voulait installer la fonctionnalité, et tous les descendants du processus en héritaient. Le pirate envoyait donc très probablement le signal à son propre shell afin que toutes les consultations/modifications de fichiers qu'il effectuait à partir de ce shell soient invisibles.

3.2.3 Exécution détournée

La modification de binaires systèmes tels que /bin/login ou sshd est une technique favorite des pirates. Ils peuvent en effet facilement introduire des chevaux de Troie à la place des programmes de login standard pour pouvoir se connecter sans avoir besoin de login / mot de passe autorisé. Une fois que le cheval de Troie (ou plutôt, backdoor « porte dérobée » dans ce cas) a détecté que son maître souhaite se connecter, il lui accorde l'accès comme à un utilisateur légitime mais n'enregistre nulle par de traces de cette connexion.

Cependant, la pose de backdoors à la place de programmes standards nécessite leur modification, et des outils performants tels que Tripwire existent pour détecter de telles modifications aux fichiers sensibles. Ces outils de vérification peuvent quand même être mis en

échec par des modifications au kernel. N'oublions pas en effet que pour lire les fichiers et en faire des signatures, les outils de vérification se fondent sur les informations fournies par le kernel (ils utilisent les services du kernel pour ouvrir et lire les fichiers, et ne peuvent pas faire autrement).
Cette constatation a été levée par plusieurs articles de la revue Phrack afin de mettre en défaut les outils de vérification comme Tripwire. L'idée est simple : si nous pouvons contrôler le noyau, pourquoi ne pas fournir aux outils de vérification la vue qu'ils veulent des fichiers, alors que ceux-ci ont pu être modifiés ?

Prenons le cas du remplacement du programme de connexion sécurisée SSH, */usr/local/sbin/sshd*, par un cheval de Troie. Une technique pour ce faire est la suivante :

- le fichier en lui-même */usr/local/sbin/sshd* n'est pas modifié. Ainsi, les programmes de vérification ne trouveront rien à redire.
- Le programme ssh de remplacement (cheval de Troie) est mis sur un autre endroit du système de fichiers, par exemple */var/tmp/backdoorsshd*. Son existence peut être masquée en le cachant comme vu précédemment.
- Le module kernel pirate modifie l'appel système d'exécution de programmes, *execve()*, pour détecter si on essaie d'exécuter /bin/login. Si tel est le cas, on exécute à la place le cheval de Troie, */var/tmp/backdoorlogin*.

De ce fait, lorsque le programme sshd sera lancé au démarrage de la machine, la backdoor sera lancée à la place, tandis que si un programme de vérification ouvre le programme *sshd* pour le vérifier, elle verra le fichier original.

3.2.4 Backdoors réseaux

Une des dernières avancées en matière de modification des kernels est la modification des piles réseaux elles-mêmes afin d'y introduire des portes dérobées.

Dès janvier 1998, des expérimentations sont publiées dans la revue Phrack concernant des modules kernel détectant les paquets UDP contenant un mot clef donné et exécutant un programme. Ces premiers exemples de déclenchements de backdoors à distance par modification des piles réseaux ont été largement améliorés par la suite.
Ainsi, un article[7] de la même revue en septembre 1999 détaille un module permettant entre autres d'inverser les bits SYN et FIN dans une connexion TCP, créant des connexions TCP non reconnues par les systèmes non équipés du module, et obscures à l'analyseur de trafic. Les IDS, n'étant pas capables d'interpréter de tels flux, sont aussi mis en défaut par ces modifications des couches réseaux, idéales pour réaliser des *covert chanels*[8].

La modification des couches réseaux peut aussi avoir une incidence sur le système local lui-même. Un module[9] récupéré par nos soins sur une machine Linux compromise modifiait par exemple la couche SOCK_RAW du kernel afin de masquer aux analyseurs ou sondes réseaux situées sur la machine certains types de paquets. Normalement, la fonctionnalité SOCK_RAW permet de remonter aux applications toutes les trames au format brut ; c'est ainsi que sont développés les IDS réseaux ou les analyseurs comme tcpdump. Le système équipé du module

[7] Kossak, Lifeline, *Building into the Linux network layer*, Phrack 55/12
http://www.phrack.com/show.php?p=55&a=12
[8] Flux dissimulés permettant par exemple de faire sortir secrètement d'un réseau local des informations sensibles, en les mélangeant à du trafic normal.
[9] Module apparemment non connu publiquement.

en question ne signalait pas à l'IDS les paquets ayant un port source particulier ou contenant un mot clef donné. Le pirate utilisait ce port source pour effectuer toutes ses connexions vers ou à partir de la machine compromise. Il était donc impossible pour la sonde IDS qui était installée sur la machine de détecter le trafic réseau généré par l'activité du pirate.

3.3 Importance du problème

Bien que le problème des modifications kernel soit connu de longue date (il a même fait l'objet d'un avis du CERT[10]), il n'est pas encore maîtrisé

En particulier, il est souvent dit pour minimiser l'impact de tels outils pirates que ceux-ci nécessitent une technicité élevée. Cependant, ce serait oublier que nombre de modules publics facilement utilisables (notamment sous Linux) sont disponibles, et pas plus délicats d'utilisation que la plupart des programmes d'attaques. La littérature est de plus abondante et les exemples d'utilisation disponibles à foison :

- **Adore** v0.34 (Linux) par Stealth[11]
  Module kernel Linux avec programme d'installation et d'administration aisée permettant de gagner root par une fonction cachée, dissimuler des processus, des fichiers des connexions TCP (contre netstat) et du mode Promisc.

- **Kernmod** 0.2 (Solaris) par Job de Haas - ITSX[12]
  Démonstration d'un module kernel Solaris réalisant toutes les opérations de base (dissimulation de fichiers, processus, connexions TCP, …).

- **Rootkit** 2000[13] (Windows NT 4.0, Windows 2000) par Greg Hoglund & collectif
  Rootkit kernel sous Microsoft Windows.

Comme le montre l'exemple précédent de kernmod et rootkit 2000, le fait que les sources du kernel Solaris ou de Windows NT ne soient pas disponibles n'a pas dissuadé la communauté de développer des rootkits Solaris ou NT. En effet, les fabricants de systèmes d'exploitation ont tout intérêt à documenter les méthodes de programmation du kernel afin de favoriser le développement de pilotes périphériques pour leurs systèmes d'exploitations ; les méthodes utilisées pour développer les pilotes périphériques sont alors reprises pour la réalisation de modules pirates.

Aujourd'hui, tous les systèmes d'exploitation sont modulaires et documentés, pour permettre un enrichissement fonctionnel par la communauté des développeurs et fabricants de matériels. Malheureusement, aucun de ces systèmes n'intègre en parallèle des mécanisme de sécurité permettant de limiter l'impact d'une compromission privilégiée (root). Les possibilités sans limites qu'a un agresseur pouvant modifier un système d'exploitation montrent clairement les limites du concept actuel de tout ou rien au niveau des privilèges. Quand le kernel, normalement base de

[10] http://www.cert.org/vul_notes/VN-98.02.kernel_mod.html
[11] http://online.securityfocus.com/data/tools/adore-0.34.tgz
[12] http://www.itsx.com/
[13] http://www.rootkit.com/

confiance, a été compromis, force est de constater qu'il n'y a plus aucun salut possible pour le système.

3.4 Solutions

3.4.1 Détection post intrusion

La perte de la base de confiance qu'est le kernel rend aléatoire toute solution a posteriori pour détecter et corriger une machine affectée. Le fait que les systèmes de détection se fondent sur des informations fournies par un kernel potentiellement compromis rend la détection très délicate.

Cependant, on peut arrive à détecter quelques-uns des rootkits kernel classiques. Les méthodes utilisées sont à base de signatures et même très largement des bugs ou oublis des rootkits.

Une méthode couramment utilisée consiste à parcourir l'arborescence /proc, en essayant d'aller dans les répertoires /proc/N/ avec N allant de 1 à 65535. Si l'opération réussie mais que le répertoire n'était pas dans la liste obtenue par un ls, le processus N est caché par le système d'exploitation et une modification du kernel a sans doute été effectuée. En effet, la plupart des modules, par soucis de facilité ou même par mauvaise connaissance du fonctionnement interne des systèmes, ne changent que certaines fonctions du kernel, laissant une fois qu'ils sont installés de nombreux bugs ou effets indésirables induits par l'introduction des modifications. Cette méthode, peu fiable, détecte uniquement les LKM simples ou mal programmés. Elle reste quand même très utile pour des compromissions automatiques ou agresseurs bas de gamme, et est surtout facilement mise en œuvre par des outils comme **chkrootkit** (http://www.chkrootkit.org/) ou même certains antivirus sous Microsoft Windows.

```
bash-2.04#
bash-2.04# cd /proc
bash-2.04# ls
0    116  137  153  172  185  2    219  243  283  286  289  295  3    374  377  396
1    118  148  155  176  191  201  229  277  284  288  294  297  309  376  382
bash-2.04# cd 202
bash-2.04#
bash-2.04# ls
as        cred      cwd       lpsinfo   lusage    map       pagedata  rmap      sigact
auxv      ctl       fd        lstatus   lwp       object    psinfo    root      status
bash-2.04#
bash-2.04#
bash-2.04#
```

Figure 7: Compromission possible par LKM, le processus 202 n'apparaît pas dans la liste pourtant il semble exister (bug de la backdoor).

Malheureusement, les rootkits kernel bien conçus peuvent être extrêmement difficiles à détecter, et même virtuellement impossibles s'il n'y a pas de grandes suspicions de compromission. Rappelons que les outils de détection reposent sur des informations qui leurs sont transmises par un système potentiellement complètement détourné.

La seule méthode fiable pour détecter de tels rootkits est l'arrêt complet du système et l'examen de ses disques à partir d'une machine sûre. Si une compromission est détectée, la réinstallation totale est de rigueur.

3.4.2 Prévention

3.4.2.1 Prévention des attaques root

L'utilisateur root étant le seul à pouvoir modifier les fichiers sensibles et le noyau du système, la parade ultime consisterait donc à éviter à tout pris qu'un agresseur ne puisse devenir root.

- désinstallation de tous les programmes non utilisés
- installation régulière des patchs
- filtrage des services réseaux non nécessaires de l'extérieur
- restriction de l'utilisateur root aux seuls endroits où cela est complètement nécessaire.

Malheureusement, cette dernière fonctionnalité, qui devrait être de rigueur sur tout système moderne se prétendant sécurisé, est par défaut absente et souvent bien difficile à réaliser soi-même. Elle est même presque impossible à effectuer sans modifications drastiques au kernel. Comme nous l'avons fait remarquer dans la première partie du document, faute des fonctionnalités adéquates, la compartimentation des rôles et des processus n'est réalisable en pratique que sur les systèmes ayant été spécialement écrits pour[14], c'est-à-dire jamais dans les usuels. Il est vrai que de telles fonctionnalités ajouteraient en complexité aux systèmes d'exploitation, qui préfèrent à tort ou à raison offrir aux programmeurs et utilisateurs une convivialité plus importante (au niveau de l'ajout de drivers, de l'administration, …).
Cependant, les systèmes Microsoft Windows NT 4.0, Microsoft Windows 2000 et successeurs proposent une gestion des rôles et permissions assez élaborée permettant de déployer un plan de sécurité[15] limitant le recours à l'utilisateur Administrateur. Il est ainsi possible de déléguer à certains groupes non privilégiés des attributions de sécurité telles que le backup complet, évitant ainsi le recours systématique à l'utilisateur Administrateur dès qu'un privilège élevé est nécessaire. Remarquons quoiqu'il en soit que de nombreux services réseaux doivent toujours disposer de privilèges systèmes, c'est-à-dire les plus élevés ; la gestion d'attributions spécifiques par processus (et non par utilisateur) n'étant pas encore supportée.

3.4.2.2 Limitation des pouvoirs de root

Considérons maintenant le cas où il n'est pas possible de se prémunir de l'utilisation de root, et qu'une compromission de cet utilisateur n'est pas exclue. On peut vouloir minimiser ses attributions et lui faire perdre un peu du modèle classique tout ou rien.

Nous avons vu que la perte complète de la confiance en un système était due à la compromission de son système d'exploitation. Si le système d'exploitation reste sain, il pourra toujours fournir aux applications de détection les moyens de repérer des modifications de fichiers par exemple. Une première technique consiste à interdire toute modification du kernel, même par root.

Si le système est destiné à être largement fixe (cas des systèmes de production), sans changements de configuration, les systèmes BSD implémentent un tel mécanisme connu sous

[14] Le noyau Linux devrait être doté d'une interface de réglages fins des permissions dans ce but. D'autre part, des systèmes commerciaux comme ceux de la société Argus peuvent venir se greffer sur les systèmes d'exploitation du marché pour leur ajouter de nombreuses fonctionnalités de sécurité.
[15] Le site Web de la société Edelweb possède une section sécurité Windows fort à propos (http://www.edelweb.fr/EdelStuff/EdelPages/) comprenant notamment des articles par Maxime de Jabrun, Nicolas Ruff et Patrick Chambet sur la sécurisation avancée de machines Microsoft Windows NT/2000.

le nom des *securelevels*. Des permissions étendues sur les fichiers et un identifiant spécial sont introduits. L'identifiant spécial, un entier appelé *securelevel*, peut uniquement être incrémenté par des administrateurs. A l'état 0, le système est dit non sûr et le comportement est celui d'un système classique sans *securelevel*. Les permissions étendues peuvent être changées et le kernel modifié par */dev/kmem* ou */dev/mem*. Aux états « sûrs » 1 et 2, les fichiers virtuels */dev/kmem* et */dev/mem* ainsi qu'entre autres les permissions spéciales des fichiers ne sont plus modifiables.

Sans ces mécanismes interdisant les modules (malheureusement, enlevant de ce fait une fonctionnalité bien pratique), l'écriture dans les périphériques mémoire et la modification des fichiers kernel eux-mêmes, il est quasiment impossible de protéger un système où root a été compromis d'une modification du kernel. Certaines tentatives ont été faites pour développer des outils vérifiant en temps réel la table des appels systèmes, cependant il existe de nombreuses parties d'un système d'exploitation à modifier à part les appels systèmes, et ces outils ne feraient que rebuter les pirates moins techniques.

4. Conclusion

Nous avons pu voir comment, en détournant des possibilités légitimes, un agresseur peut manipuler à sa guise un système entier au point qu'aucune information ne puisse plus être validée.

Les fonctionnalités de convivialité et de souplesse ont longtemps été privilégiées par rapport à celles de sécurité, comme en témoignent les installations par défaut des plus grands systèmes d'exploitation, d'office vulnérables à de nombreuses attaques. Cependant, nous avons pu assister ces derniers mois, en raison notamment d'évènements majeurs tels que l'arrivée de vers et virus à diffusion exceptionnelle, d'un retournement de la tendance. Même si nous ne pouvons qu'approuver ces nouvelles orientations, le problème de la modification des noyaux nous montre bien que des changements drastiques seront nécessaires aux cœurs même des systèmes d'exploitation, afin que la confiance perdue de longue date puisse être restaurée.