Sécurité des systèmes Linux

Mécanismes de prévention et de cloisonnement d'attaques sous Linux

Mathieu BLANC
mathieu.blanc@cea.fr

Commissariat à l'énergie atomique et aux énergies alternatives
Direction des applications militaires

7 décembre 2011

- EPIC (Public, Industriel et Commercial) ;
- 15700 chercheurs, ingénieurs, techniciens et administratifs ;
- Pôles :
  - Energie nucléaire ;
  - Recherche Technologique ;
  - Sciences de la matière, sciences du vivant ;
  - **Défense** : Direction des Application Militaires
    - Dissuasion : armes nucléaires, simulation ;
    - Propulsion nucléaire ;
    - Lutte contre la prolifération et sécurité globale.

- Détournement du flux d'exécution du CPU sous Linux
  - Écrasement de la sauvegarde d'adresse de retour (***stack buffer overflow***)
  - Exploitation du chaînage des blocs du tas (***heap buffer overflow***)
  - Abus de l'interprétation de *format string*
  - Exploitation du dépassement de capacité d'une variable entière (***Integer overflow***)
  - Écrasement de pointeur de fonction
- Injection d'un *payload*
  - *Shellcode*, éventuellement en plusieurs étapes
  - Appels à des fonctions de bibliothèques dynamiques (*return to libc*)
  - *Return-Oriented Programming*
- Fonctions du *shellcode*
  - Ouverture de port TCP en écoute, connexion vers l'attaquant (*connect back*, *reverse shell*)
  - Modification du système (*rootkit*)
  - *Shellcode* évolué (CANVAS, Meterpreter)

energie atomique • energies alternatives

- Première stratégie : prévenir les attaques
  → Comment ?
  - Produire des applications sans vulnérabilité ?
    → -> Difficile à garantir, problème de recherche toujours actif
  - Bloquer les attaques avant qu'elles n'arrivent sur la machine ?
    → Mais on ne sait pas toujours les repérer
  - **Renforcer les applications à la compilation**
    → Et si on ne dispose pas du code source ?
  - **Implémenter des protections dans l'environnement d'exécution**
- Et pour des schémas d'attaque encore inconnus ?

- Seconde stratégie : **limiter l'impact d'une attaque réussie**
  - En exécutant une application à risque de façon isolée dans le système d'exploitation
  - En limitant les droits d'accès de l'application sur le système au strict nécessaire
  - En exécutant un système d'exploitation isolé pour cette application

energie atomique • energies alternatives

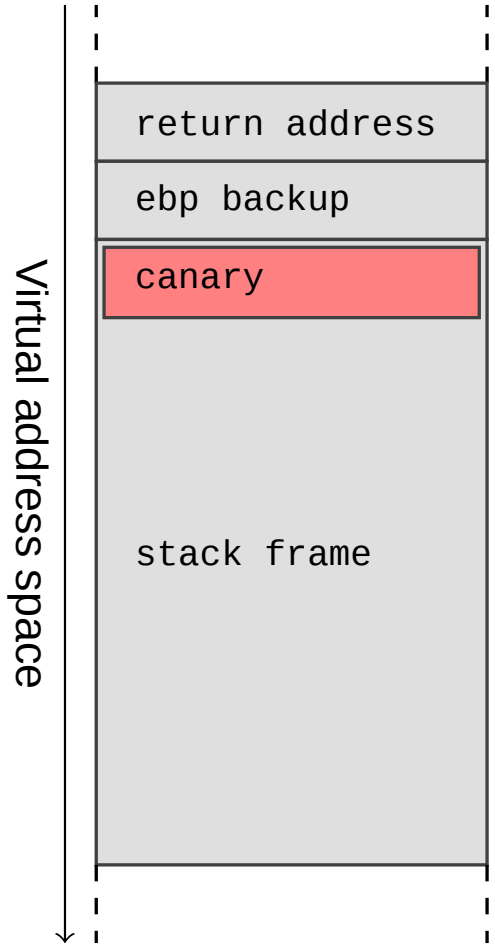
energie atomique • energies alternatives

- Idée : détecter des débordements de tampon mémoire dans la pile
  - et arrêter le programme lorsqu'un tel événement est détecté
- Principe de fonctionnement :
  - Le compilateur repère les fonctions allouant des buffers de plus de quelques octets ;
  - il insère dynamiquement des instructions supplémentaires :
    - au début de la fonction, ajout d'un *canary* entre les variables locales et l'adresse de retour ;
    - avant le retour de la fonction, lecture du *canary* et vérification de la non-modification.
  - Le programme quitte par l'appel d'une fonction spécifique en cas de modification.

```
1 *** stack smashing detected ***: /sbin/vuln terminated
2 ======= Backtrace: =========
3 /lib32/libc.so.6(__fortify_fail+0x50)[0xf7677aa0]
4 /lib32/libc.so.6(+0xe4a4a)[0xf7677a4a]
5 /sbin/vuln[0x8048628]
6 [0x41414141]
7 ======= Memory map: ========
```

energie atomique • energies alternatives

```
080485f4 <main>:
 80485f4:       push   %ebp
 80485f5:       mov    %esp,%ebp
 80485f7:       and    $0xfffffff0,%esp
 80485fa:       add    $0xffffff80,%esp
 80485fd:       mov    %gs:0x14,%eax
 8048603:       mov    %eax,0x7c(%esp)
 8048607:       xor    %eax,%eax
```



- Problème des attaques par *heap buffer overflow* :
  - Manque de contrôle lors de la désallocation d'un bloc alloué ;
  - Un *overflow* donne à l'attaquant le contrôle sur les pointeurs de blocs précédents et suivants ;
  - Possibilité pour un attaquant d'écrire une valeur choisie à une adresse choisie ;
- Principe de protection : renforcer le contrôle des pointeurs lors de la désallocation.
- Première implémentation dans le projet StackShield, puis intégration à `gcc` et la `glibc`.
- Pour activer les différentes protection, quelques options de `gcc` :
  - `-fstack-protector[-all]`
  - `-D_FORTIFY_SOURCE=2`

energie atomique • energies alternatives

Historiquement, l'architecture x86 ne peut empêcher l'exécution de code dans une page mémoire particulière

- Problème : certaines zones mémoire sont marquées non-exécutables pour des raisons légitimes (pile, zones de données) ;
- Dommage collatéral : pendant longtemps les développeurs ont programmé avec le postulat que toute la mémoire est exécutable.

Pour protéger contre les attaques par débordement de buffer, la première protection est d'empêcher l'exécution dans les zones de données :

- Première implémentation logicielle avec le projet PaX (via une définition revue des permissions sur les segments de mémoire) ;
  - Aujourd'hui intégrée au noyau Linux standard ;
- A partir de 2004, intégration du support de pages mémoire non-exécutables dans les processeurs AMD (NX) et Intel (XD).

energie atomique • energies alternatives

Aujourd'hui le réglage par défaut est l'utilisation de mémoire non-exécutable ;

- Cependant de nombreux programmes ont nécessité quelques changements dans leur code pour fonctionner ainsi (notamment le serveur X) ;
- En particulier, pour démarrer un programme avec une pile exécutable, il faut modifier une valeur de contrôle dans l'en-tête ELF (outil : `execstack`).

```
1 % objdump -p vuln1
2
3 vuln1:     file format elf32-i386
4
5 Program Header:
6 ...
7    STACK off    0x00000000 vaddr 0x00000000 paddr 0x00000000 align 2**2
8           filesz 0x00000000 memsz 0x00000000 flags rwx
9 ...
```

Suite à la mise en place de la fonction NX, une nouvelle technique d'exécution de code a été mise au point : *return-to-libc*

- Plutôt que d'injecter un shellcode dans la pile ou le tas, on va provoquer un appel à une fonction de la `libc`.
- Par exemple, pour exécuter un shell :
  - empilement des paramètres nécessaire pour faire un `execve("/bin/sh", 0, 0)`;
  - écrasement de l'adresse de retour avec l'adresse de la fonction `execve` dans la bibliothèque standard.

Pour contrer ce nouveau type de *payload*, une technique appelée ASLR a été inventée.

- Principe : placer la pile et charger les bibliothèques dynamiques à des adresses aléatoires pour qu'un attaquant ne puisse pas prédire leurs adresses.
  - Première implémentation dans le projet PaX ;
  - Toute la sécurité repose sur l'entropie de variation des adresses, bien plus grande sur architecture 64 bits.
  - Les premières structures "randomisées" sont donc la pile, et les structures allouées par `mmap()` soit le tas et les bibliothèques dynamiques.

Dans la veine de l'attaque *return-to-libc*, d'autres techniques d'exécution de code ont été conçues, notamment regroupées sous le terme ROP (*Return-Oriented Programming*, ∼2009) :

- L'attaquant va remplacer les instructions constituant un shellcode classique par l'exécution d'une série de *gadgets* ;
- Chaque *gadget* est une petite série d'instructions suivies d'un `ret`, contenues dans le programme vulnérable ;
- Ainsi l'attaquant n'a plus besoin d'injecter de code exécutable, il peut directement exécuter celui qui est contenu dans des parties du programme sans ASLR.

Pour contrer ces nouvelles techniques, l'ASLR a été étendu :

- au code du programme, qui doit être compilé en mode PIE (*Position-Independant Executable*) ;
- à la zone brk (suit immédiatement la zone de données du programme) ;
- et enfin à la zone VDSO (*Virtual Dynamic Shared Object*), qui contient le code binaire pour effectuer un appel système.

S'il est capable d'exploiter une faille d'un service, un pirate va chercher à exécuter d'autres commandes sur la machine.

$\rightarrow$ Pour réduire l'impact d'une exploitation réussie, on peut limiter l'environnement de ce service.

energie atomique • energies alternatives

Le principe de la cage

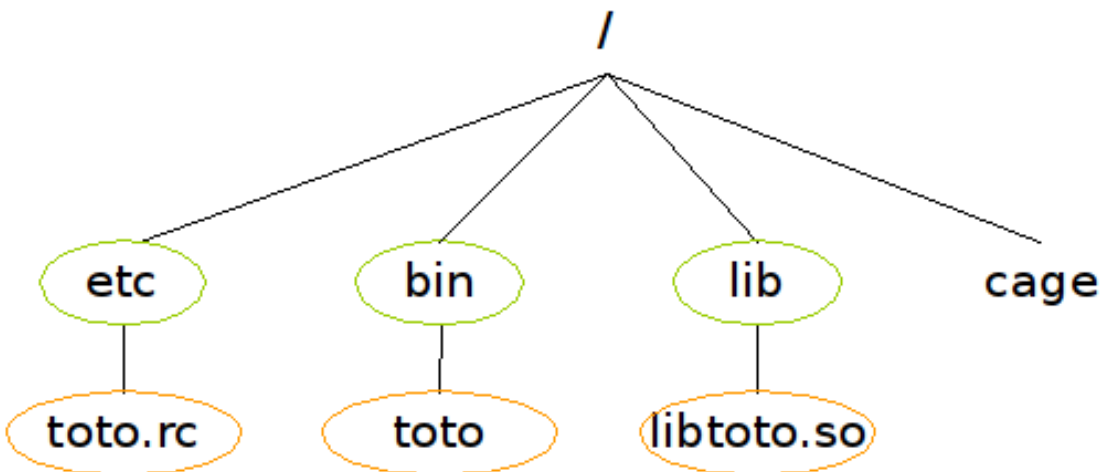
- dissimuler une partie du système au programme
- ne lui laisser voir que ce dont il a besoin pour s'exécuter correctement
  - fichiers de configuration
  - bibliothèques
  - autres exécutables, scripts…
  - fichiers spéciaux (*pipes*, *sockets*, *devices*…)
  - journaux d'evénements

En pratique

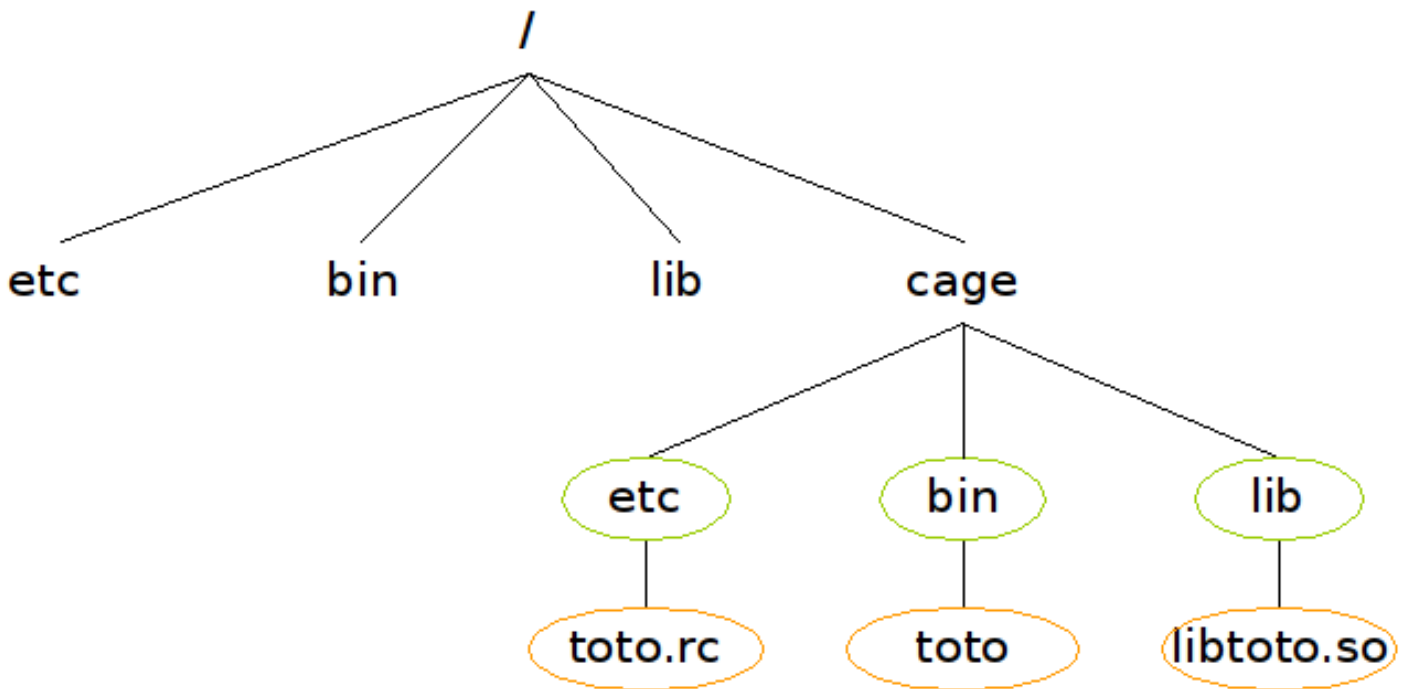
On fait passer un répertoire du système pour la racine (/) de celui-ci, l'application ne pouvant (en théorie) rien voir en dehors de ce répertoire.

Remarque : Bon entraînement avant de passer à SELinux ou grsecurity !

- Exemple de mise en cage avec `chroot`
  - Fichier exécutable : `/bin/toto`
  - Configuration : `/etc/toto.rc`
  - Bibliothèque : `/lib/libtoto.so`
  - Nouvelle racine : `/cage`
- On reproduit l'arborescence sous `/cage`
  - `mkdir /cage/etc`
  - `mkdir /cage/bin`
  - `mkdir /cage/lib`



- On déplace les fichiers du service :
  - `cp /bin/toto /cage/bin/toto`
  - `cp /etc/toto.rc /cage/etc/toto.rc`
  - `cp /lib/toto.so /cage/lib/libtoto.so`



On démarre enfin le programme comme suit :

```
chroot /cage /bin/toto [options]
```

Comment déterminer les fichiers nécessaires au programme et devant se trouver dans la cage ?

On peut distinguer deux cas :

- les bibliothèques dynamiques
  - liste avec `ldd`
  - on les copie dans l'arborescence de la cage
  - ou on compile le programme avec l'option `-static`
- le reste
  - les divers fichiers accédés par le programme lors de son exécution peuvent être déterminés avec un outil tel que `strace`

Certains programmes ont des options d'exécution pour faire appel à `chroot()`.

On peut difficilement considérer `chroot()` comme un outil de sécurité à part entière

- Plusieurs possibilités d'évasion ;
- Ne pas faire tourner de code *root* à l'intérieur, ce n'est pas assez résistant ;

$\rightarrow$ Toutefois, ce mécanisme peut être renforcé par des améliorations dans grsecurity.

Certains programmes sont particulièrement vulnérables parce qu'ils traitent des données complexes, comme les moteurs de rendu HTML ou PDF.
→ Ce type de code peut être isolé avec le mode `seccomp`.

- Principe : après avoir été placé dans le mode `seccomp`, un processus est limité à quatre appels système, soit `read`, `write`, `exit` et `sigreturn`.
- En résumé, ce processus peut uniquement utiliser des *file descriptors* déjà ouverts ou se terminer.
- Ce mode est pensé au départ pour des codes complexes tels que des machines virtuelles Java ou Flash, ou des codes de calcul intensif.

Pour utiliser ce mode

- Le processus doit d'abord ouvrir tous les fichiers nécessaires ;
- Le changement de mode se fait par `prctl(PR_SET_SECCOMP, 1);` ;
- Si le processus utilise un appel système non autorisé, il reçoit immédiatement un `SIGKILL` et se termine.

La notion de contexte d'exécution isolé est apparue avec `jail` sous FreeBSD. Plusieurs implémentations existent aussi pour Linux.

Principe

- Les contextes d'exécution partagent le même noyau que le système hôte ;
- Ils ne peuvent pas voir les processus en cours d'exécution dans d'autres contextes ou sur l'hôte ;
- Ils sont restreints pour l'accès au système de fichiers et ont une adresse IP propre.

Implémentations :

- FreeBSD : **jail** ;
- Linux : **vServer**, OpenVZ, **cgroups** ;
- OpenSolaris : Zones.

Exécution compartimentée sur un même noyau FreeBSD :

- Confinement d'une seule application ;
- Confinement d'un système FreeBSD complet ;

Limitations sur l'environnement isolé :

- Racine spécifique pour le système de fichiers (comme `chroot()`) ;
- Adresse IP et nom d'hôte spécifiques ;
- Limitation sur l'accès aux périphériques matériels ;
- Isolation des processus (visibles depuis le contexte hôte) ;
- Possibilité de positionner un *securelevel* plus élevé que celui de la machine hôte.

```
jail <path> <hostname> <ip-number> <command>
```

Exécution compartimentée sur un même noyau Linux :

- Plutôt orientée isolation de systèmes Linux complets ;
- Avec partage du noyau ;
- Uniquement sous forme de patch.

Spécificités par rapport à `jail` :

- Le système de fichiers est contenu dans un fichier opaque (disque virtuel) plutôt que d'être un dossier à part ;
- Il peut être créé par *copy-on-write* à partir d'un disque *template*.

Création de groupes de processus (*Control Groups*) pour isolation et contrôle de l'usage des ressources :

- Fonctionnalité incluse dans le noyau Linux depuis 2007 ;
- Limitation d'usage de la mémoire, prioritisation de l'accès au CPU et aux disques ;
- *Namespaces* séparés pour les processus, les interfaces réseau, les points de montage de systèmes de fichiers ;
- Contrôle de l'exécution : pause, *checkpoint / restart*.

Différentes interfaces de contrôle :

- Commandes en ligne : `cgcreate`, `cgexec`, `cgclassify` ;
- Interface de contrôle LXC.

Les avantages :

- Performances équivalentes à l’exécution sans isolation ;
- Plusieurs solutions déjà disponibles dans les distributions Linux usuelles ;
- Configuration généralement simple.

Les écueils :

- Si l’on souhaite isoler uniquement un programme, il faut déterminer toutes les ressources dont il a besoin ;
- Le plus simple est d’isoler un système complet, mais ce n’est pas toujours nécessaire ;
- La sécurité repose essentiellement sur l’implémentation du mécanisme d’isolation, qui n’a pas toujours été pensé pour la sécurité…
  - …et peut laisser des possibilités d’évasion vers le système hôte.

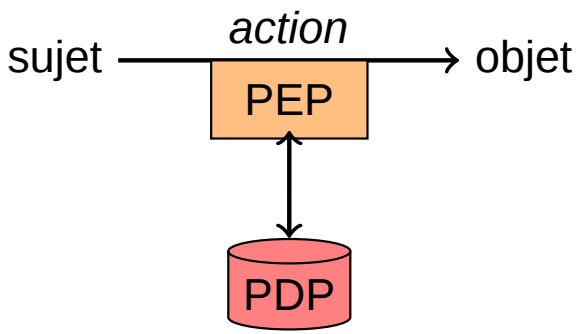
energie atomique • energies alternatives

Le contrôle d'accès est le mécanisme en charge de décider si un *sujet* peut effectuer une *action* sur un *objet*.

- Sujet : entité active (processus, utilisateur) ;
- Objet : entité passive, ressource (fichier, socket) ou autre sujet ;
- Action : les appels système.

Le contrôle d'accès est effectué par deux entités :

- PEP (Policy Enforcement Point) : intercepte l'action et applique la décision d'accès ;
- PDP (Policy Decision Point) : rend une décision d'accès en fonction du triplet $(\text{sujet}, \text{objet}, \text{action})$.



Discretionary Access Control

- Les permissions sont déterminées par le propriétaire de la ressource ;
- L'utilisateur *root* outrepasse le contrôle DAC (sauf si retrait de la *capability* `DAC_OVERRIDE`) ;
- Par conséquent, il est difficile de restreindre les droits d'un processus exécuté avec le compte *root*.

Mandatory Access Control

- Les permissions d'accès sont définies dans une configuration centrale non modifiable par les utilisateurs (politique de contrôle d'accès) ;
- La politique est conservée dans un composant appelé le Moniteur de référence ;
- Tous les processus du système sont soumis à ce contrôle, même ceux appartenant à *root* ;
- Il devient possible de limiter précisément l'accès d'un processus aux seules ressources dont il a besoin pour s'exécuter.

energie atomique • energies alternatives

Le modèle Bell-LaPadula (she's my baby)

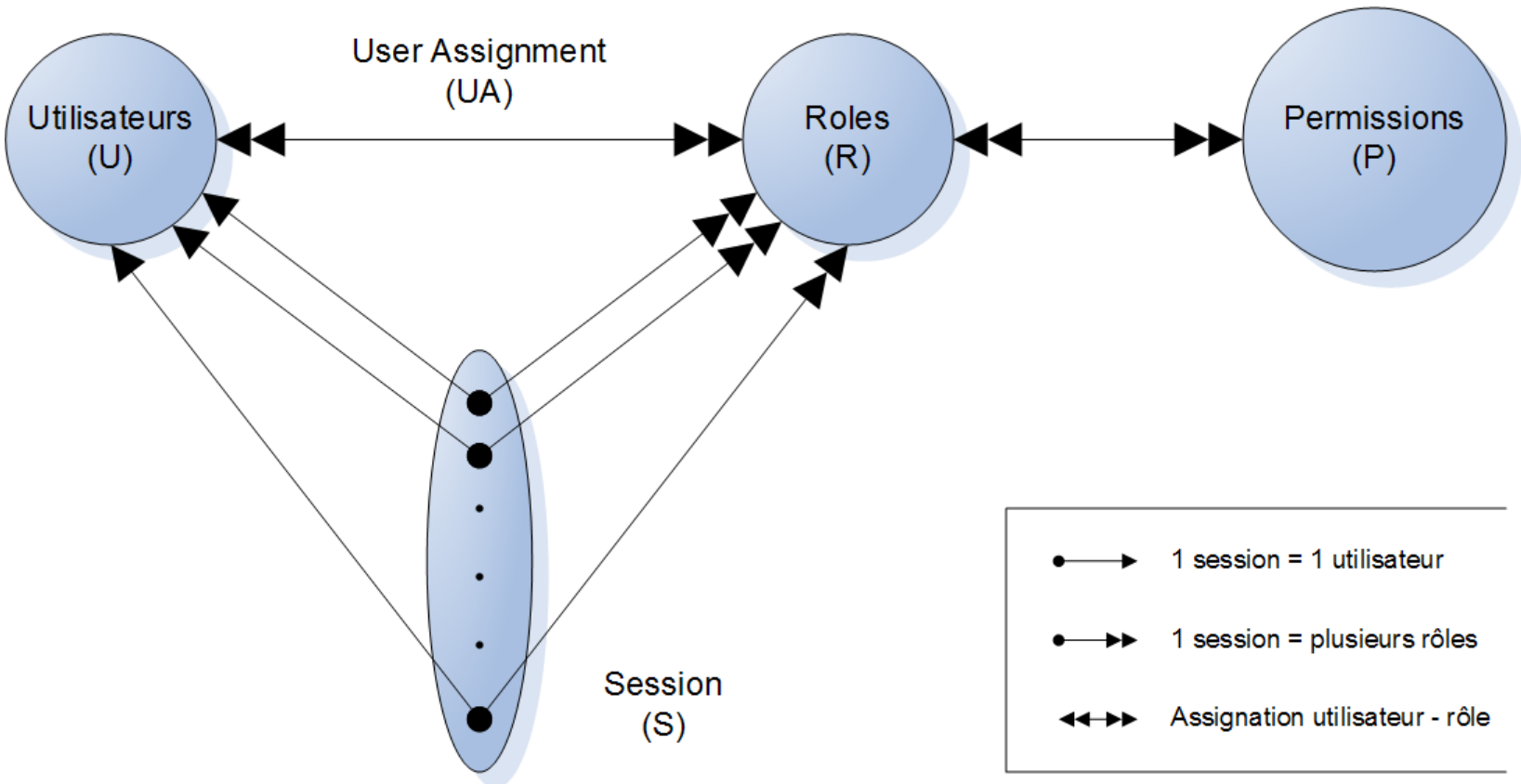
- Modèle de confidentialité pour l'accès à l'information dans la Défense ;
- Notion de label de sécurité : habilitation pour les sujets, classification pour les objets ;
- Notion de catégorie : répartition de l'information par type d'activité ;
- Deux règles de sécurité :
  - No Read Up
  - No Write Down

Le modèle Biba ('p-a-lula)

- Modèle de préservation de l'intégrité utilisé dans des systèmes industriels critiques ;
- Notion de niveau d'intégrité pour les sujets et objets ;
- Deux règles de sécurité :
  - No Read Down
  - No Write Up

energie atomique • energies alternatives

Role-Based Access Control



Le projet *Linux Security Modules* (LSM) a pour objectif de fournir une API générique l'écriture de modules de sécurité.

- Support de tout type de mécanisme de sécurité ;
- API stable (enlève la nécessité de faire évoluer un patch) ;

Security Hooks

- Intégration de points de contrôle dans le noyau ;
- Au niveau des entrées vers les différents "sous-systèmes" du noyau Linux :
  - Fichiers (VFS) ;
  - Réseau ;
  - Gestion des processus ;
  - Mémoire partagée, IPC ;

Security Attributes

- Création d'un *namespace* dans les attributs étendus de fichiers, dédié aux informations de sécurité ;
- Nom et contenu dépendant du module de sécurité utilisé.

Modules actuellement intégrés : SELinux, AppArmor, Tomoyo, SMACK.

Security-Enhanced Linux

- Développement initial par la NSA (∼1999) ;
- Intégration au noyau Linux vers 2001, donnant naissance au *framework* LSM ;
- Principes de conception
  - Séparation du composant de décision (*Security Server*) et du composant d'application (*hooks* LSM) ;
  - Labellisation persistante des fichiers ;
  - Règles d'accès indépendantes du système de fichiers ;
- Modèles de sécurité implémentés
  - RBAC, *Type Enforcement*
  - SELinux UID
  - Bell-LaPadula : *Multi-Level Security* (MLS), *Multi-Category Security* (MCS)
- Format des contextes de sécurité
  `user:role:type:levels.categories`

- Interface avec les appels système via LSM ;
- *Security Server* (SS) en espace noyau
  - Contient la configuration compilée (`policy.<version>`) ;
  - La configuration est modulaire ;
- Système de cache intégré : *Access Vector Cache* (AVC)
  - Les décisions d'accès sont calculées sous forme de vecteur de permissions ;
  - Dès qu'un vecteur est calculé par le SS, il est placé dans le cache ;
  - Les nouvelles entrées remplacent les plus anciennes ;
  - Nombre d'entrée dans le cache par défaut de 512 sur les systèmes RedHat ;
- Intégration dans les commandes standard
  - Code commun dans la `libselinux` ;
  - `ls -Z`, `ps -Z`, `id -a`...
- Commandes d'administration spécifiques
  - Etat : `sestatus`, `seinfo`, `semodule -l`
  - Compilation : `checkmodule`, `checkpolicy`, `semodule -i`
  - Labels : `setfiles`, `chcon`, `restorecon`

Les modules de configuration se composent de trois parties :

- Définition de contextes (fichiers `.fc`) ;
- Macros publiques (fichiers `.if`) ;
- Règles d'accès (fichiers `.te`).

La définition des contextes à appliquer sur les fichiers se fait à l'aide d'expression régulières sur les chemins.

Par exemple :

```
HOME_DIR/\.mozilla(/.*)?  gen_context(system_u:object_r:mozilla_home_t,s0)
/usr/bin/mozilla  --  gen_context(system_u:object_r:mozilla_exec_t,s0)
/usr/lib(64)?/[^/]*firefox[^/]*/firefox-bin -- gen_context(system_u:
     object_r:mozilla_exec_t,s0)
```

Après avoir chargé cette politique, on peut appliquer les contextes :

- Commande `setfiles` appliquée sur les différents systèmes de fichiers concernés ;
- Ou `touch /.autorelabel` et redémarrage du système.

Langage de définition de règles d'accès

- Les règles d'accès comportent un sujet, un objet, une classe d'objet et un ensemble d'actions autorisées ;
- Ce qui n'est pas explicitement autorisé est interdit, et un message de *log* est généré ;
- On peut également définir des règles d'audit sur les actions autorisées ;
- Enfin on peut donner des contraintes à vérifier (mot-clé `neverallow`).

L'écriture des règles est (relativement) facilité par l'existence de nombreuses macros. Par exemple :

```
type mozilla_t;
type mozilla_exec_t;
type mozilla_home_t;
application_domain(mozilla_t, mozilla_exec_t)
kernel_read_system_state(mozilla_t)
allow mozilla_t self:capability { sys_nice setgid setuid };
allow mozilla_t self:process { sigkill signal setsched getsched };
allow mozilla_t self:fifo_file rw_fifo_file_perms;
allow mozilla_t self:socket create_socket_perms;
allow mozilla_t mozilla_conf_t:file read_file_perms;
```

- SELinux est intégré dans la plupart des distribution Linux, notamment :
  - Fedora, RedHat → activé par défaut
  - Suse
  - Ubuntu, Debian

- On trouve en général quatre configurations :
  - la *Reference Policy* disponible sur le site de MITRE ;
  - les déclinaisons *strict* et *targeted* créées par RedHat ;
  - enfin la version MLS, seule version avec le support des niveaux.

- La configuration est placée dans `/etc/selinux/<variante>`.

energie atomique • energies alternatives

- AppArmor
  - Patch noyau développé par Immunix puis Novell pour Suse Linux (∼1998)
  - Récemment intégré au noyau Linux comme LSM, maintenu en particulier par Canonical (∼2009)
- Langage de configuration simple
  - Définit des profils de confinement
  - Un sujet est représenté par un chemin vers un exécutable
  - Contrôle l'accès aux fichiers, au réseau, aux *capabilities*
  - Peut désigner les fichiers dont le propriétaire est le même que celui du processus
  - Définition de variables globales
- Chacun des profils peut être chargé en mode *enforcing* ou en mode *complain*
- Dossier de configuration : `/etc/apparmor` et `/etc/apparmor.d`

Extrait de la configuration pour Firefox :

```
#include <tunables/global>
/usr/lib/firefox-8.0/firefox{,*[^s][^h]} {
  #include <abstractions/ubuntu-browsers.d/firefox>
  /etc/firefox*/ r,
  /etc/firefox*/** r,
  deny /usr/lib/firefox-8.0/** w,
  deny /usr/lib/firefox-addons/** w,
  owner @{HOME}/Downloads/ r,
  owner @{HOME}/Downloads/* rw,
  owner @{HOME}/.{firefox,mozilla}/ rw,
  owner @{HOME}/.{firefox,mozilla}/** rw,
  owner @{HOME}/.{firefox,mozilla}/**/*.{db,parentlock,sqlite}* k,
```

energie atomique • energies alternatives

- Développement de grsecurity
  - Coïncide avec celui de PaX
  - Ajoute un mécanisme de confinement et renforce la sécurité de fonctions standard du noyau
  - Toujours distribué sous forme de patch
- Fonctions de PaX
  - Pages mémoire non exécutables
  - Implémentation spécifique de l'ASLR
- Fonctions de confinement
  - *Process-Based ACLs*
  - Gestion de rôles type RBAC
- Fonctions additionnelles
  - Restrictions sur `chroot()`
  - Audit
  - Restrictions sur `/dev/mem`, sur `/proc`…
- L'activation de ces fonctions se fait par le menu de configuration du noyau

Pas de configuration complète disponible comme pour SELinux : le mode d'apprentissage est un passage obligatoire !

- Mode d'apprentissage complet
  - Etape 1 : examen de l'activité système
    `gradm -F -L learning.logs`
  - Etape 2 : génération du fichier de configuration
    `gradm -F -L learning.logs -O policy.new`
- Mode restreint aux rôles marqués `I` dans la configuration
  - `gradm -E -L learning.logs`
  - `gradm -L learning.logs -O policy.new`

Définition des rôles

```
role admin sA
[...]
role default G
role_transitions admin
[...]
```

La syntaxe vue pour AppArmor est assez proche de celle de grsecurity

- Résolution du rôle :
  - login → group → default
- Un sujet est un chemin vers un exécutable
- Les objets sont désignés par des expressions régulières
- Contrôle sur les fichiers, sur le réseau, sur les *capabilities*

Quelques-unes des différences en faveur de grsecurity :

- Gestion des rôles
  - Chargement de règles différentes suivant le rôle
  - Possibilité de s'authentifier pour passer au rôle admin
- Restrictions sur l'administration
  - Seul un rôle marqué `G` peut arrêter ou démarrer grsecurity, ou changer la configuration
  - Il faut donc rentrer le mot de passe de ce rôle pour l'administrer
- Possibilité de cacher des fichiers pour un processus
  - Les fichiers avec la permission h ne sont pas visibles pour le sujet


```
subject /usr/sbin/sshd dpo
/ h
/bin/bash x
/dev h
/dev/log rw
/dev/random r
/dev/urandom r
/dev/null rw
/dev/ptmx rw
/dev/pts rw
/dev/tty rw
/dev/tty? rw
/etc r
/etc/grsec h
/home
/lib rx
/root
/proc r
/proc/kcore h
/proc/sys h
/usr/lib rx
/usr/share/zoneinfo r
/var/log
/var/mail
/var/log/lastlog rw
/var/log/wtmp w
/var/run/sshd
/var/run/utmp rw

-CAP_ALL
+CAP_CHOWN
+CAP_SETGID
+CAP_SETUID
+CAP_SYS_CHROOT
+CAP_SYS_RESOURCE
+CAP_SYS_TTY_CONFIG
```

Virtualisation matérielle -> le matériel est virtualisé (plus ou moins)

energie atomique • energies alternatives

Différents systèmes de virtualisation

- Hyperviseur de type 1
  - Hyperviseur léger exécuté directement sur la machine (*bare metal*)
  - Tous les systèmes d'exploitation s'exécutent en environnement virtualisé
  - Avec accélération matérielle (VMware ESX, Hyper-V, Xen) ou sans (Xen en mode paravirtualisation)
- Hyperviseur de type 2
  - Hyperviseur exécuté au sein d'un système d'exploitation
  - Avec ou sans accélération matérielle (VMware Workstation, Virtualbox, Virtual PC…)
- Emulateur x86
  - Contrairement aux hyperviseurs, aucune instruction de l'environnement virtualisé n'est exécutée sur le processeur, tout est émulé
  - Qemu, Bochs

energie atomique • energies alternatives

On peut créer des conteneurs d'exécution dédiés pour des services ou applications

- Une attaque ne compromet que la machine virtuelle
- On peut la replacer dans son état initial facilement (mais cela ne corrige pas la faille que l'attaquant a utilisé)
- Cela reste intéressant pour des environnements "jetables" comme des PC en accès libre

Mais les hyperviseurs ne sont pas réellement conçus pour la sécurité

- L'hyperviseur rajoute une couche à surveiller et auditer
  - En général une vulnérabilité dans l'hyperviseur signifie que l'attaquant peut prendre le contrôle du système hôte
- Les *guest tools* installent des portes dérobées pour communiquer avec l'hyperviseur ou le matériel
- Seul les émulateurs garantissent que l'environnement virtuel s'exécute sans effet de bord sur le système hôte, mais au prix de la performance

Nous avons vu un certain nombre de mécanismes

- Certains bloquent des attaques
- D'autres limitent les dégâts en cas de compromission effective d'une application

Des limites subsistent

- Vulnérabilité dans le code de protection ou d'isolation
  - Exemple : vulnérabilités fréquentes dans Xen, VMware, VirtualBox
  - Longue mise au point de mécanismes comme SELinux ou grsecurity
- Difficile de contrôler si la configuration élimine un maximum de chemins d'attaque
  - Notamment dans les cas SELinux, grsecurity, AppArmor
  - Vrai aussi pour les systèmes de virtualisation ou d'isolation
- De nouvelles techniques d'attaque sont découvertes
  - Les mécanismes actuels peuvent être inefficaces
- Enfin, il reste le risque de vulnérabilité dans le noyau
  - Des protections en espace noyau commencent à être intégrées, comme l'insertion de *canaries* dans la pile
  - Futur : améliorer le contrôle de l'intégrité du noyau

C’est fini ! (Vous pouvez réveiller votre voisin)

Et pour la route…