Sécurité des systèmes d'exploitation

Focus sur les systèmes à base de noyau Linux

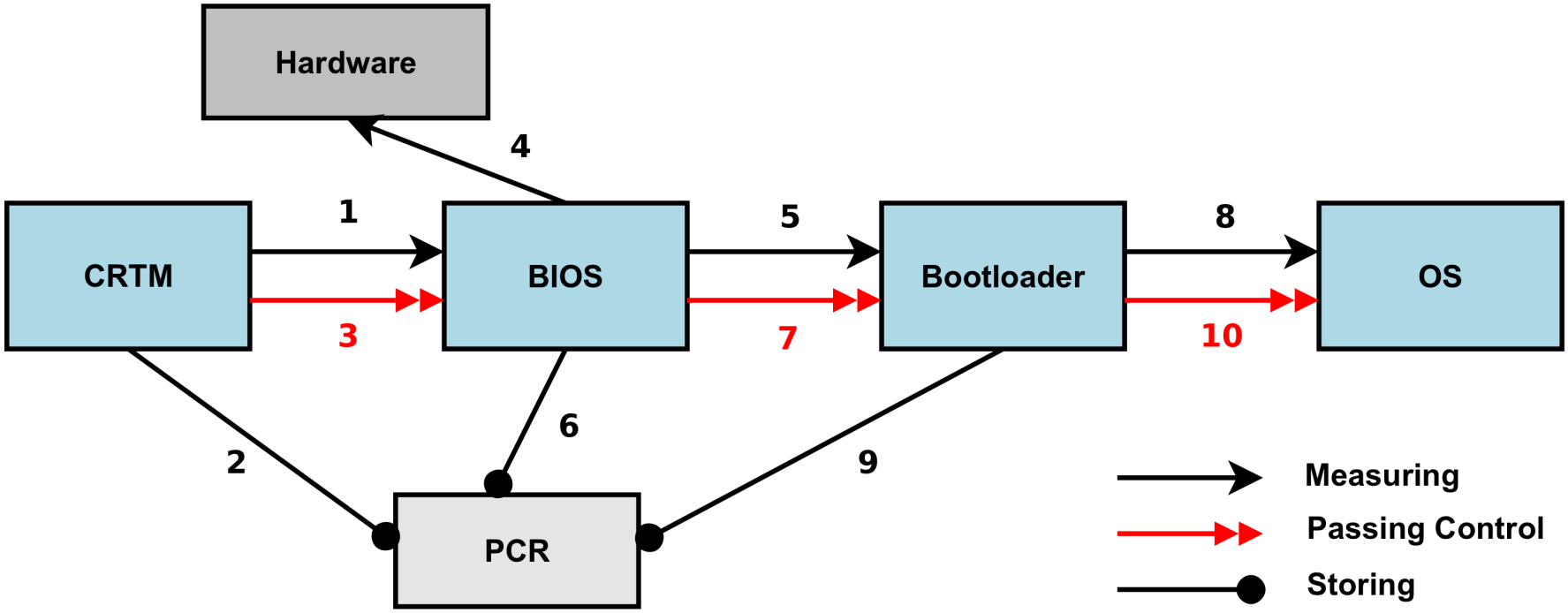
Éric Lacombe

eric.lacombe@security-labs.org

2019-2020

La sécurité dans un monde idéal sans vulnérabilités

Chaîne de confiance (prônée par le Trusted Computing Group)



CRTM : Core Root of Trust for Measurement
PCR : Platform Configuration Registers

Mais le monde n'est pas idéal, car trop complexe...

Des vulnérabilités peuvent être introduites :

- Dans les différents composants logiciels/matériels
- Lors de l'intégration de ces composants
- Et même dans les mécanismes de sécurité tels que ceux qui mettent en œuvre le RTM...

Faille dans Intel TxT (Trusted eXecution Technology)

SINIT Buffer Overflow Vulnerability (5 décembre 2011) :
http://security-center.intel.com/advisory.aspx?intelid=INTEL-SA-00030&languageid=en-fr

- Buffer overflow exploitable trouvé dans le code SINIT ACM (Authenticated Code Module) utilisé par Intel TxT
- Ce code (stocké avec le BIOS) est exécuté par le processeur (si sa signature est valide) pour placer une machine pourvue de la technologie Intel TxT dans un environnement de confiance
- Compromission possible de la chaîne de confiance

Application des correctifs **à la discrétion des OEM** (Original Equipment Manufacturer) et des utilisateurs :

- Intel a publié une mise à jour des SINIT ACMs
- Mais qu'en est-il de la mise à jour des BIOS par les OEM ?
- Palliatif temporaire : Intel publie également une mise à jour du microcode de ces processeurs pour empêcher l'utilisation de SINIT ACMs vulnérables

Sécurité dans un monde réel rempli de vulnérabilités

Protection en profondeur

- Mise en œuvre d'une RTM $\rightarrow$ Bien, mais pas suffisant
- Différentes mesures de sécurité complémentaires sont nécessaires pour un haut niveau de sécurité
  - Mesures fonctionnelles
  - Mesures architecturales
  - Mesures d'assurance

Objectifs du cours

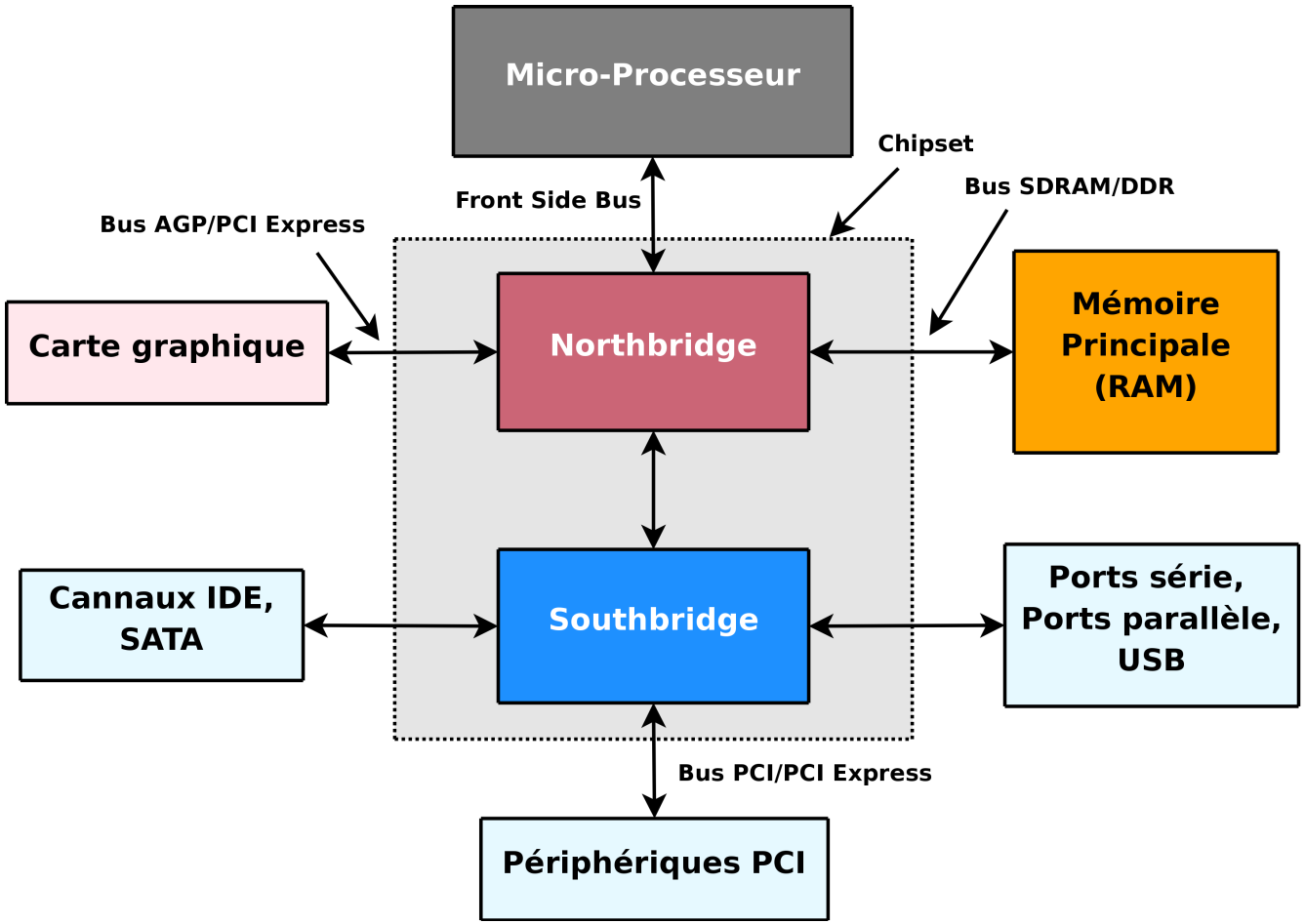
Donner des éléments de réponse aux questions suivantes :

- Comment protéger l'espace utilisateur depuis le noyau ?
- Comment protéger le noyau ?

Plan

Plan

Vue d'ensemble de l'architecture matérielle



Environnement basique d'exécution x86

→ Permet l'exécution d'un ensemble d'instructions génériques (arithmétique, contrôle du flux d'exécution, adressage mémoire, etc.).

Éléments principaux de l'environnement :

- Espace d'adressage en mémoire
- Registres basiques d'exécution d'un programme :
  - registres généraux (EAX, EBX, ECX, etc.) : stockage temporaire de données
  - registre d'état (EFLAGS) : contient différents flags d'état et de contrôle de l'exécution d'une tâche
  - registre de compteur d'instructions (EIP)
- Pile d'exécution et ressources de gestion de pile (registres ESP, EBP) : supporte l'appel aux sous-routines et le passage de paramètres

Ressources supplémentaires pour l'implémentation d'OS

- Ports d'E/S : permet de transférer des données avec les composants matériels via des canaux de communications (via instructions `in`, `out`)
- Registres de contrôle :
  - CR0 : détermine le mode d'opération du processeur, mode d'adressage mémoire, etc.
  - CR2 et CR3 : pour la gestion de la mémoire paginée
  - CR4 : activation d'extensions architecturales (ex : hugepage)
- Registres de gestion mémoire et de gestion de l'exécution :
  - GDTR, LDTR : référence les structures de contrôle de l'unité de segmentation
  - IDTR : référence la table des interruptions en mémoire
  - TR : référence le TSS (Task State Segment) qui contient l'état d'exécution de la tâche courante (état des registres, ...)
- MTRR (Memory Type Range Registers) : ensemble de registres permettant de spécifier le comportement de la mémoire avec le cache de la CPU (write-through, write-back, etc.)
- MSR (Model-Specific Registers) : pour le contrôle de spécificités fonctionnelles ou architecturales des processeurs

Plan

Modes d'opération du processeur

Le processeur s'exécute suivant différents modes d'opération (contrôlés par le registre CR0) :

- Mode protégé : mode nominal du processeur, supporte la protection de la mémoire et la ségrégation des instructions par niveaux de privilèges
- Mode réel : les processeurs x86 démarre dans ce mode de compatibilité (mode d'opération originel des Intel 8086 mais étendu)
- Mode de gestion système (SMM) : le processeur passe dans ce mode pour la gestion d'énergie, gestion des erreurs physiques de la RAM, etc.
  $\rightarrow$ Accès sans restrictions aux ressources matérielles
- Mode 8086 virtuel : permet au processeur depuis le *mode protégé* l'exécution de logiciels anciens prévus pour du 8086 (ces logiciels étant exécutés dans un environnement isolé)
- Mode IA-32e (uniquement sur Intel 64) : sous-mode de compatibilité 32 bits ou sous-mode 64 bits

Niveaux de privilèges du processeur

Anneaux de privilèges (ou "*rings*") :

- Mécanisme principal de sécurité fourni par les processeurs x86
- Disponible depuis le mode protégé ou le mode IA-32e
- 4 anneaux sont définis pour 4 niveaux de privilèges croissants
- Permet de restreindre l'accès aux ressources matérielles et à l'utilisation du jeu d'instructions du processeur

Ring 0 (mode d'exécution du noyau de l'OS – mode noyau) :

- Aucune restriction sur l'accès aux ressources matérielles
- Tout le jeu d'instructions peut être utilisé
- Mode d'exécution nécessaire à un noyau d'OS

Ring 3 (mode d'exécution des applications – mode de l'espace utilisateur) :

- Niveau de privilège minimal
- Certaines ressources ne sont pas accessibles (la plupart des registres de configuration du processeur, mémoire noyau, etc.)
- Les instructions jugées critiques pour la sécurité ne peuvent être exécutées

Interruptions et exceptions

Différents types d'interruptions :

- Interruptions "matérielles" : permettent à un composant matériel de notifier l'occurrence d'évènements au processeur (réception d'un paquet réseau, appui sur une touche du clavier, etc.)
- Exceptions : levées par le processeur pour lui-même en cas de problème en cours d'exécution (division par zéro, faute de pages, etc.)
- Interruptions "logicielles" : levées par du code (via l'instruction `int`)
  - Pour l'exécution d'appels système (sur les architectures d'aujourd'hui, `sysenter` et `syscall` sont préférées)
  - Pour le *debugging* de programme (instruction `int3` pour déclencher l'appel au gestionnaire de l'exception de debug)

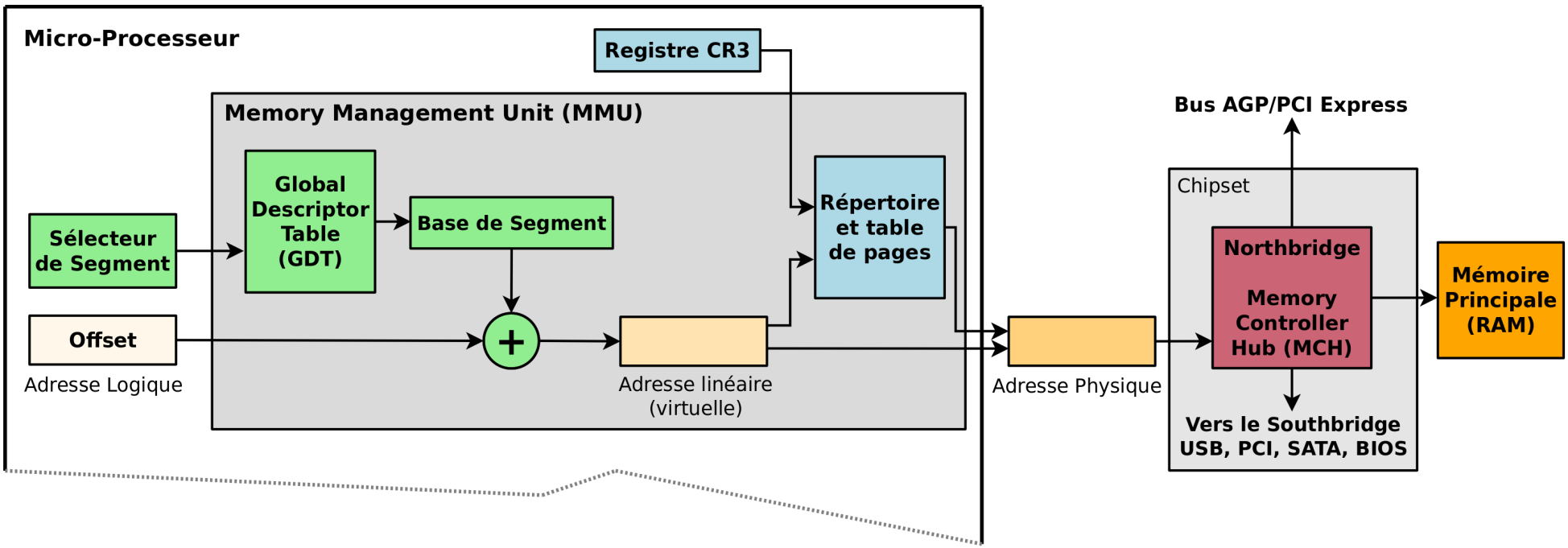
→ Lorsque une interruption survient, le processeur passe en ring 0 et donne la main à une routine de traitement configurée par le noyau.

L'IDT (Interrupt Descriptor Table) : table des routines de traitement des interruptions (gestionnaires d'interruptions)

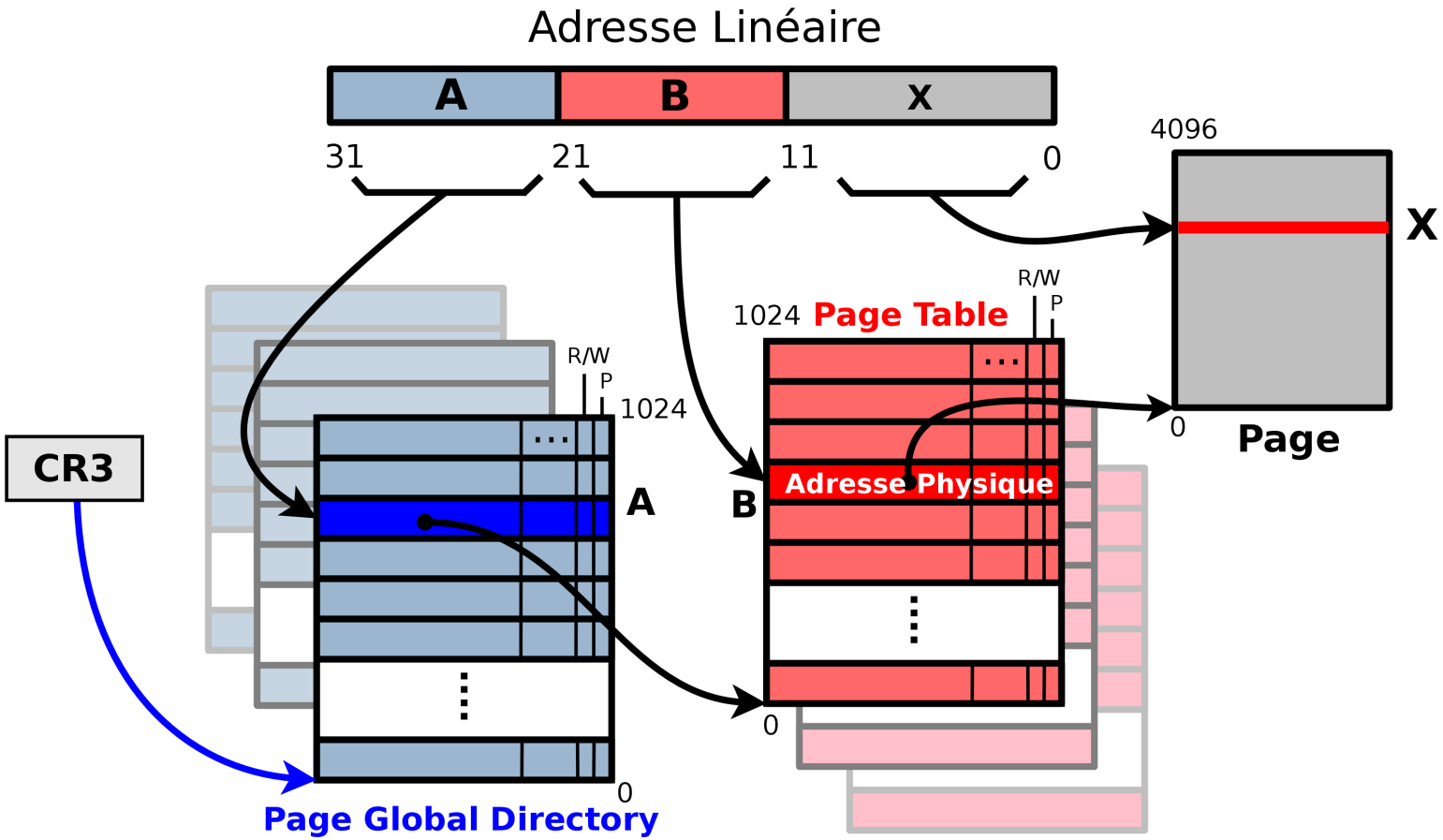
- 1 ligne correspond à 1 interruption (de 0 à 255) et contient l'adresse du gestionnaire de l'interruption
- Remplie par le noyau et ensuite référencée dans le processeur via `lidt` (qui charge IDTR avec l'adresse de la table)

Plan

MMU (Memory Management Unit) : Segmentation et Pagination



Mécanisme de pagination



Plan

Accès aux périphériques

Configuration des périphériques et accès aux fonctions du chipset se fait par l'écriture dans des registres de configuration :

- Registres de configuration accessibles en MMIO (Memory-Mapped I/O) :
  - Projetés en mémoire principale par le chipset à une adresse donnée
  - Accessibles en lecture et écriture via l'instruction `mov`
- Registres de configuration PIO (Programmed I/O) :
  - Projetés dans un espace d'adressage 16 bits indépendant
  - Accessibles en lecture et écriture via les instructions `in` et `out`
- Registres de configuration PCI :
  - Situés dans un 3^{eme} espace d'adressage
  - Accès depuis le mécanisme de configuration PIO :
    - On écrit dans le registre PIO `0xcf8`, l'adresse du registre PCI auquel on souhaite accéder
    - Le chipset met à jour automatiquement le registre PIO `0xcfc` auquel on peut accéder en lecture ou écriture

Transferts de données entre périphériques et mémoire principale

DMA (Direct Memory Access) :

- Mécanisme où les transferts de données entre la mémoire principale et les périphériques sont effectués sans intervention directe du processeur
- Deux catégories d'accès DMA :
  - Accès à l'initiative d'un périphérique capable d'adresser lui-même des données sur le bus système (DMA bus-master)
  - Accès commandé par la CPU (cas par exemple d'un driver qui commande un transfert de donnée sur une clé USB)
- La conclusion du transfert peut être signalée par interruption
- IOMMU : permet de contrôler les accès des périphériques à la mémoire principale (rôle analogue à la MMU)

Polling :

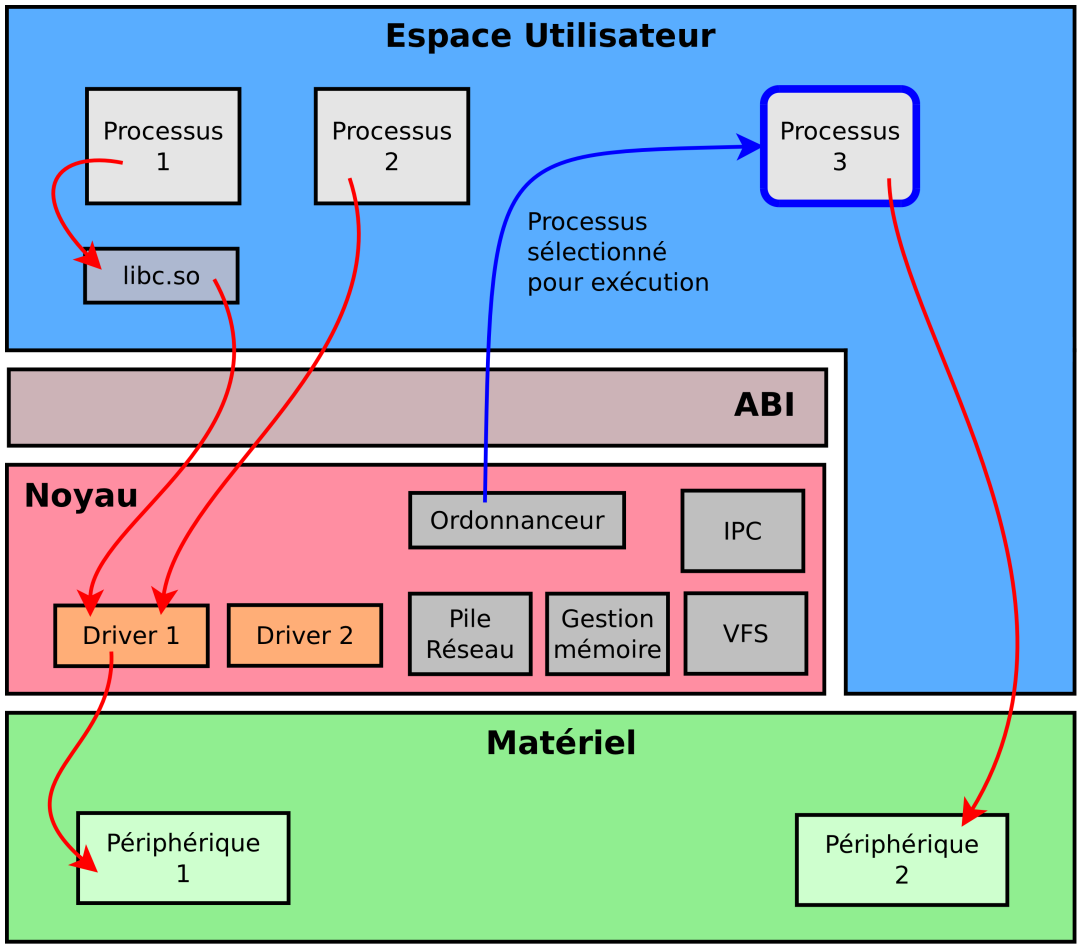
- Le processeur attend chaque donnée en scrutant périodiquement les registres des périphériques
- Permet de se passer des interruptions (utile pour l'implémentation de systèmes temps réel dur)

Plan

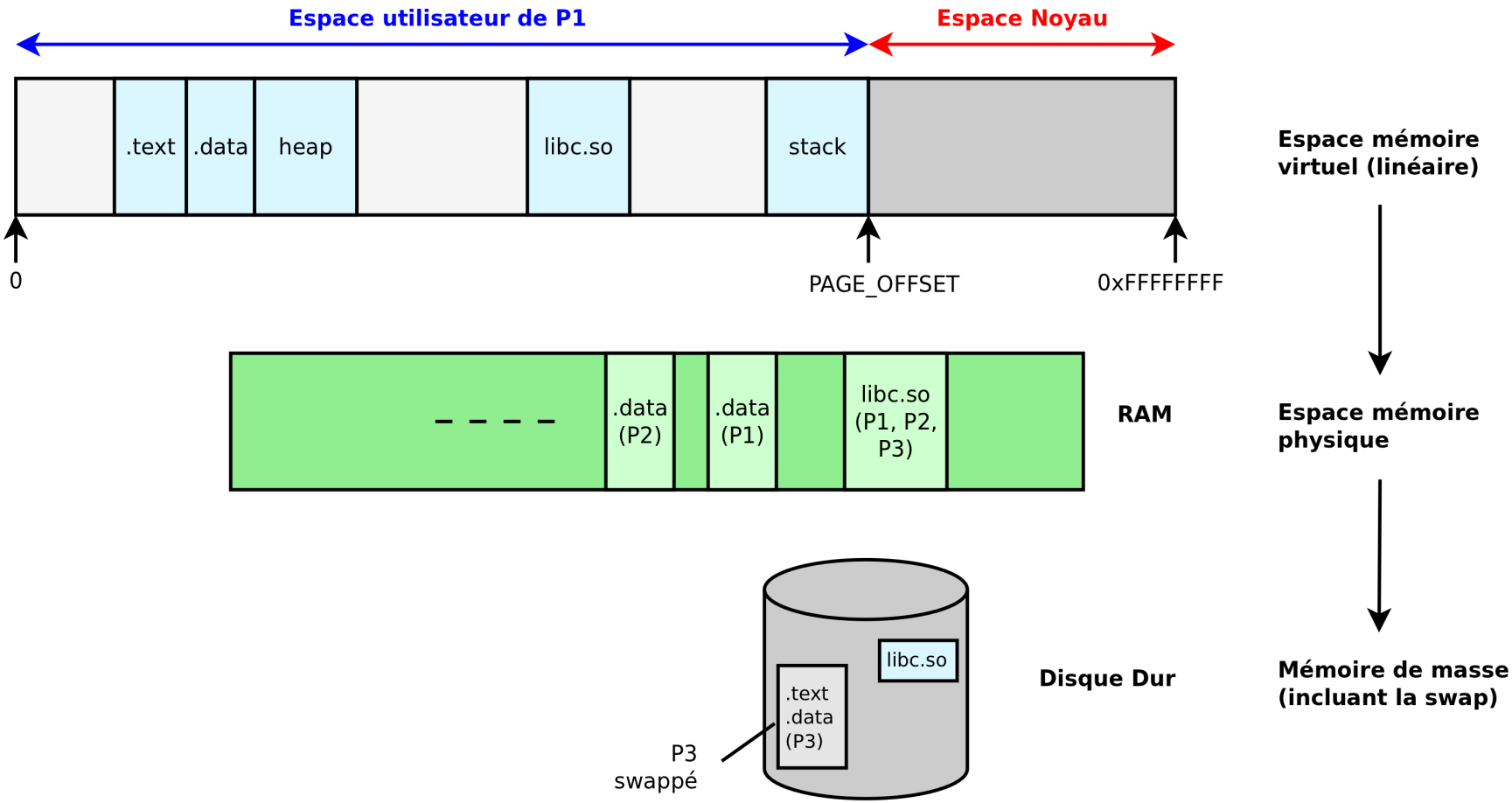
Les fonctions d'un système d'exploitation

- Assurer la configuration et le bon fonctionnement du matériel :
  - Traitement des interruptions et exceptions
  - Configuration de la CPU et du chipset
  - Configuration de la mémoire (segmentation et pagination)
  - Configuration des périphériques
- Fournir des abstractions qui facilitent la programmation des applications :
  - Définition de services pour l'utilisateur (les appels-système)
  - Mise en oeuvre d'une interface d'accès à ces services (`int 0x80`, `sysenter`, vDSO, MSR)
- Assurer le partage des ressources matérielles pour toutes les applications qui s'exécutent :
  - Gestion des ressources d'exécution (CPU, GPU, ...)
  - Gestion des ressources de mémoire (RAM, disque dur, ...)
  - Gestion des ressources de communication (affichage, réseau, ...)

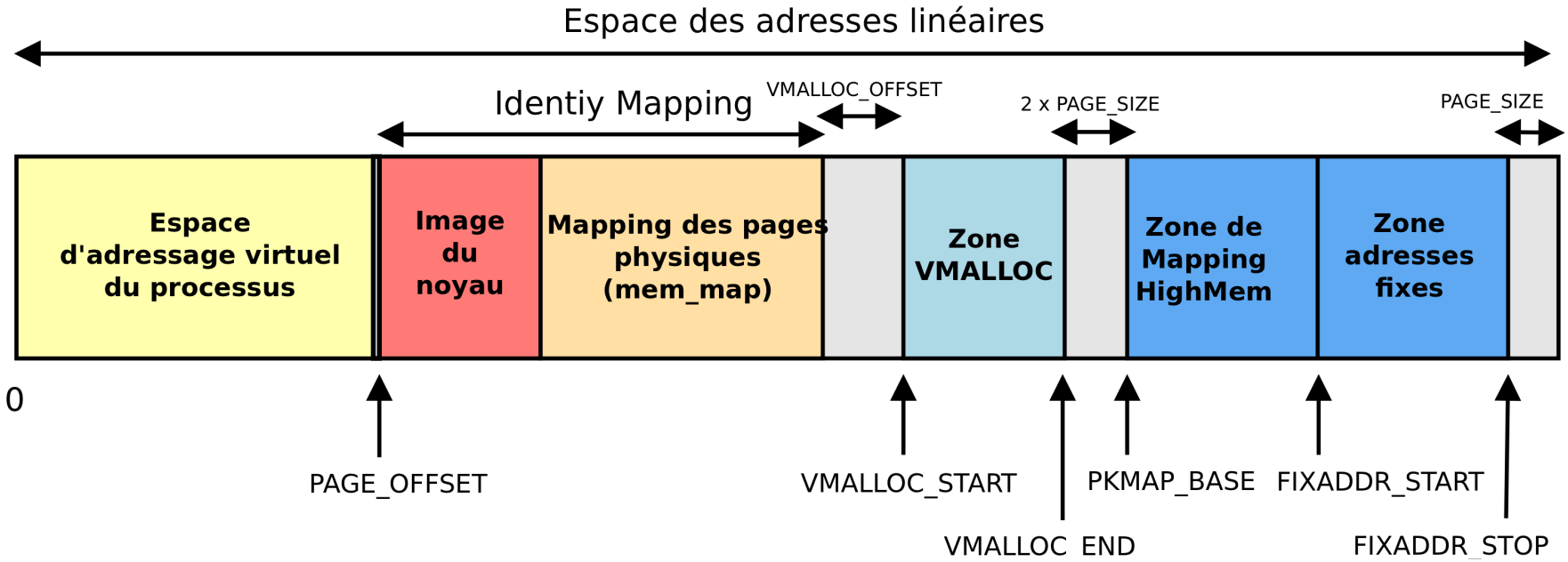
Vue d'ensemble de l'architecture logicielle monolithique



Espace d'adressage (simplifié) d'un processus sous Linux



Espace d'adressage du noyau Linux



Plan

Vue d'ensemble

Objectif 1 : Limiter les conséquences suite à des abus de privilèges

- Contrôle d'accès obligatoire :
  - Mise en œuvre du principe du moindre privilège
  - Séparation des pouvoirs

Objectif 2 : Limiter les conséquences ou augmenter la difficulté de l'exploitation de vulnérabilités dans les applications

- Environnement d'exécution contraint pour les applications :
  - Protection de la mémoire des applications
  - Restrictions d'accès aux ressources
- Agencement sécuritaire de l'espace mémoire :
  - Limiter les conséquences des exploitations de type "return-to-libc", ou plus généralement du ROP (Return-Oriented Programming)

Plan

Contrôle d'accès en bref

- Les politiques DAC (droits Unix, ACL, ...) ne protègent pas contre l'abus de pouvoir
- Une politique MAC est un premier pas pour mettre en œuvre le principe du moindre privilège et pour rendre possible la séparation des pouvoirs
  - Les utilisateurs du système sont contraints par la politique d'autorisation mise en œuvre $\rightarrow$ DAC toujours possible dans ce contexte mais contraint
  - Si la politique est bien conçue alors l'abus de pouvoir devient plus difficile
  - Exemples : SELinux, SMACK
- Les politiques MAC sont difficiles à écrire : il faut connaître précisément le comportement des différentes applications du système
- Les politiques RBAC sont plus naturelles, et permettent une gestion des droits allégées

Limites du contrôle d'accès

- Difficultés de vérifier la cohérence de la politique (Est-il possible de violer la politique en suivant les règles ?)
- Difficultés de contrôler tous les vecteurs d'accès
- Quid des vulnérabilités dans les mécanismes de contrôle d'accès ?
  - Politiques trop peu restrictives
  - Défauts dans l'implémentation

LSM : Linux Security Module

Infrastructure intégrée au noyau Linux afin de rendre possible l'implémentation de diverses politiques de contrôle d'accès au travers de module de sécurité (LSM).

- Des points de contrôle (*hooks*) sont placés aux endroits critiques dans le noyau (ouverture de fichiers, ...)
- Chaque point de contrôle fait appel à une fonction précise de la structure globale `security_ops`, laquelle autorise ou non un accès en fonction du contexte
- Ces fonctions sont définies au niveau des LSM et implémentent la logique de contrôle d'accès
- La plupart des mécanismes de contrôle d'accès sous Linux reposent sur les LSM : Capabilities, SELinux, SMACK, etc.

SMACK : Simplified Mandatory Access Control Kernel

SMACK : LSM qui sert à mettre en œuvre une politique de contrôle d'accès obligatoire sur un système Linux

- La politique se fonde sur les concepts de sujets (processus, thread) et d'objets (fichier, socket, etc.)
- Les sujets et objets sont étiquetés lors de la configuration du système (besoin d'un système de fichier supportant les attributs étendus)
- Une fois l'étiquetage effectué, une politique doit être écrite de façon à ce que seul les accès légitimes entre sujets et objets soient autorisés
- La politique doit ensuite être chargée dans `/smack/load` pour que le noyau l'applique

Note : Une étiquette peut servir à identifier un objet précisément, mais aussi un type d'objet. Ainsi une même étiquette peut être utilisée sur plusieurs fichiers (tout dépend de la granularité voulue).

Autres LSM

- SELinux :
  - Similaire à SMACK, mais permet la mise en œuvre de politiques variées (MAC, RBAC, TE, ...) et assure un contrôle plus fin et plus complet
  - Architecture complexe (basée sur FLASK), initialement développée par la NSA
  - SELinux est capable de contrôler les échanges réseaux, dispose de fonctions d'audit, ...
- Grsecurity RBAC (Role-Based Access Control) :
  - `gradm` (outil d'administration) est capable d'analyser la politique pour corriger certains types de fautes (accès à la configuration de la politique depuis le rôle par défaut, ...)
  - Mode d'apprentissage qui permet de créer une politique de moindre privilège en fonction des données collectées
- AppArmor :
  - Définition de la politique centrée sur les tâches
  - Des profils de comportements valides doivent être définis pour tous les programmes du système
  - Pas d'étiquetage, utilisation des chemins d'accès du système de fichiers
- TOMOYO : Similaire à AppArmor

Plan

Empêcher l'injection de code : droits d'accès à la mémoire

Respect du principe `W^X` lors de la création d'un processus et tout au long de l'exécution :

- Une région de code (exécutable) ne doit pas être modifiable
- Une région de données modifiables ne doit pas être exécutable
- Mais aussi : une région de données constantes ne doit pas être accessible en exécution ou en écriture

La solution de sécurité PaX pour Linux met en œuvre ce principe via :

- PAGEEXEC : Agit lors de la création en mémoire des différentes régions d'un processus (il respecte le principe en positionnant correctement les attributs des pages)
- MPROTECT :
  - Interdit tout changement de mode d'accès aux régions qui contrevient au principe (restriction dans `mprotect()`)
  - Empêche la création de nouvelles régions en cours d'exécution ne respectant pas le principe (restriction dans `mmap()`)

L'objectif est d'empêcher l'introduction de code exécutable malveillant dans l'espace d'adressage d'une tâche.

Empêcher l'injection de code : Contrôle de ptrace() 1/2

Contexte

L'appel système `ptrace()` permet à un utilisateur d'examiner et de modifier la mémoire et l'état d'exécution de tous ses processus ($\rightarrow$ utilisé pour le debugging notamment)

Comment limiter le risque qu'un attaquant ayant pris le contrôle d'un processus vulnérable d'un utilisateur donné, corrompe les autres processus de cet utilisateur ?

- `prctl(PR_SET_DUMPABLE, 0, 0, 0, 0)` : empêche la création de *core dump* (lors d'un *segfault* par exemple), et rend impossible pour un autre processus de s'attacher via `ptrace()`
- Pas suffisant car dépend des choix d'implémentation de l'application uniquement

$\rightarrow$ Besoin de configurer le comportement de `ptrace()` de façon globale

Empêcher l'injection de code : Contrôle de ptrace() 2/2

Solution Grsecurity / Yama

Le comportement de `ptrace()` est défini en positonnant la valeur de `/proc/sys/kernel/yama/ptrace_scope` :

- `ptrace_scope = 0` : Le comportement classique est appliqué → un processus peut s'attacher à un autre du même utilisateur tant que celui-ci est dumpable.
- `ptrace_scope = 1` : L'appel à `ptrace()` est restreint de la façon suivante. Pour qu'un processus s'attache à un autre, il doit avoir une relation prédéfini avec lui (via `prctl(PR_SET_PTRACER, PID_traceur, 0, 0, 0)`). Par défaut, la relation autorisée est que le processus doit être un des descendants du processus qui souhaite s'attacher.
- `ptrace_scope = 2` : Seul les processus avec la capability `CAP_SYS_PTRACE` peuvent utiliser `ptrace()` dans sa fonction `PTRACE_ATTACH`.
- `ptrace_scope = 3` : Aucun processus ne peut utiliser ptrace() avec la capacité `PTRACE_ATTACH`. De plus lorsque ce mode a été choisi il n'est plus possible de le dégrader à un niveau inférieur sans redémarrer le système.

Détecter les corruptions de la pile/tas : les canaris

Détecter les corruptions de la pile (Stackguard 1998) :

- Une donnée (appelée "canari") est ajoutée dans chaque *stack frame*, entre les données de contrôle (sauvegarde de `eip` et `ebp`) et les variables locales
- La valeur d'un canari est vérifiée avant le retour à la fonction appelante correspondante
- Si le canari est modifié alors un débordement a eu lieu

Différents types de canari :

- Random : le canari est généré aléatoirement
- Terminator : le canari contient les valeurs NULL, CR, LF et -1 (empêche les débordements de chaînes de caractères uniquement)
- XOR Random : le canari est un `XOR` entre l'adresse de retour et une valeur aléatoire (empêche une écriture ciblée dans la pile, après un débordement ayant altéré un pointeur par exemple)

Le même principe est utilisée pour la protection du tas.

Protection du flot de contrôle : support matériel

Comment empêcher la corruption des données de contrôle dans la pile ?

→ Intel CET – Shadow Stacks : Création d'une nouvelle pile pour les opérations de transfert de contrôle

Comment empêcher la corruption d'appels indirects de fonction ?

→ Intel CET – Indirect Branch Tracking : Ajout de l'instruction ENDBRANCH au début des cibles légitimes d'un programe (comme les fonctions)

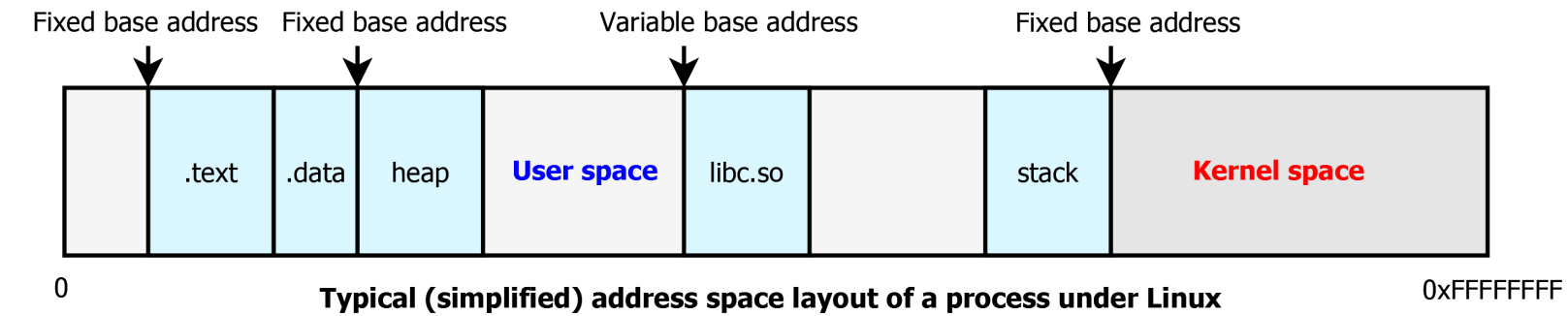
Agencement sécuritaire de l'espace mémoire

Objectif : Rendre difficile / improbable certains types d'attaques évoluées (détournement du flot de contrôle vers du code existant – ret2libc, ROP) sur les applications.

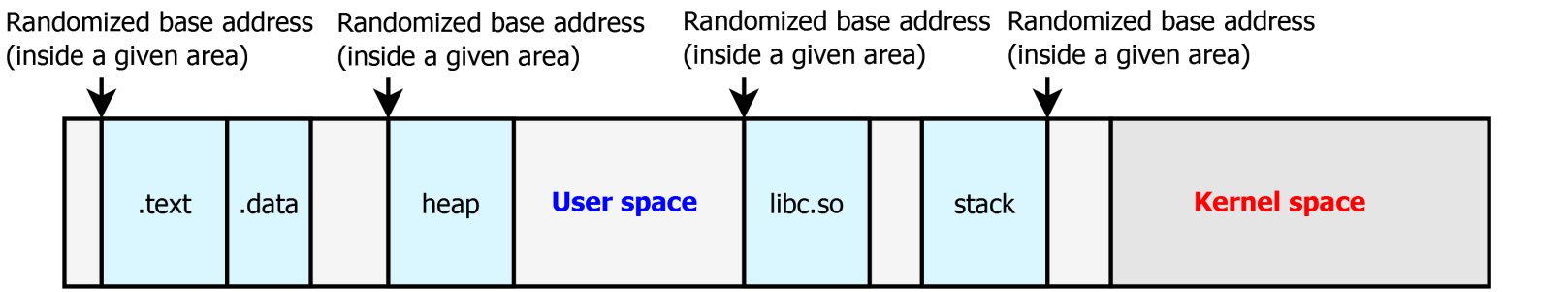
Solutions :

- Randomisation de l'agencement de l'espace mémoire (en anglais : *Address Space Layout Randomization* – ASLR)
- Mapping du code au début de l'espace d'adressage

Randomisation de l'agencement de l'espace mémoire



Typical (simplified) address space layout of a process under Linux



(simplified) Address space layout of a process under Linux with randomization (ASLR)

Mapping du code au début de l'espace d'adressage

L'idée est que toute adresse d'un bout de code quelconque de l'application (ou des bibliothèques chargées) contiennent des octets NULL.

Solution :

- Les régions de code du programme et des bibliothèques sont chargées au début de l'espace d'adressage
- Ainsi, un débordement de tampon (via les fonctions de la libc manipulant des chaînes de caractères) visant à détourner l'exécution vers une fonction existante va échouer (à l'exception des fonctions sans argument)
  - Arrêt du débordement au premier caractère NULL rencontré
  - Rends impossible la copie dans la pile des arguments à passer à la fonction

Exemple sur Ubuntu

```
$ cat /proc/self/maps
00400000-0040b000 r-xp 00000000 08:02 1180118  /bin/cat
0060a000-0060b000 r--p 0000a000 08:02 1180118  /bin/cat
0060b000-0060c000 rw-p 0000b000 08:02 1180118  /bin/cat
```

Plan

Restriction de l'environnement d'exécution (sandboxing) – Vue d'ensemble

Les processus/threads à contrôler subissent des restrictions d'accès aux ressources du système.

Différentes stratégies :

- Contrôle des appels système (Linux : seccomp-bpf)
- Contrôle des ressources (Linux : ulimit, cgroups, Landlock – OpenBSD : pledge, unveil)
- Isolation des resources (Linux : namespaces)
- ou Combinaison des précendentes stratégies (FreeBSD : Capsicum)

Mode d'établissement des restrictions :

- statiques ou dynamiques
- configurables en mode privilégié ou non-privilégié

L'objectif est de limiter les conséquences des attaques sur le système

Contrôle des appels système : SECCOMP (mode basique)

Objectif originel : rendre possible l'exécution de code "étranger" (potentiellement hostile) depuis un processus de confiance.

- Un processus en mode SECCOMP n'a accès qu'aux appels système : `read()`, `write()`, `exit()`, et `sigreturn()`
- Toute tentative d'invocation d'un autre appel système met fin à l'exécution du processus
- `prctl(PR_SET_SECCOMP, 1)` active le mode SECCOMP pour le processus appelant

Contrôle des appels système : SECCOMP (mode filtre) 1/7

Objectif : Augmenter la flexibilité de SECCOMP

- *SECCOMP filter* permet de filtrer les appels systèmes autorisés dans l'environnement d'exécution restreint
- Mais également les paramètres qui leurs sont passés
- Filtrage défini par des "bouts de code" spécialisés, appelés programmes BPF (Berkeley Packet Filter)

Utilisation :

1. Définition du programme BPF `filtre`
2. Appel de `prctl(PR_SET_NO_NEW_PRIVS, 1, 0, 0, 0)` pour rendre impossible l'obtention de nouveaux privilèges par le processus appelant
   - lors d'un `execve()` : *setuid*, *file capacities* sont ignorés, contraintes LSM conservées
   - propriété persistante après un `fork()`/`clone()`
3. Appel de `prctl(PR_SET_SECCOMP, 2, &filtre)`
   → L'exécution de tout appel système est assujetti à ce filtre qui renvoie une décision autorisant ou non son exécution

Contrôle des appels système : SECCOMP (mode filtre) 2/7

Les programmes BPF :

- Les programmes BPF ont été initialement pensés pour des raisons de performances dans un contexte de filtrage de paquets réseau (ex : tcpdump)
- Un programme BPF est prévu pour s'exécuter sur une pseudo-machine BPF, constituée de :
  - accumulateur
  - registre d'index
  - mémoire de travail
  - un compteur d'instructions implicite
- L'évolution de l'état de la machine passe par l'utilisation d'un jeu d'instructions spécifique remplissant certaines propriétés
- A la charge du noyau d'interpréter alors ces instructions, et donc d'agir comme une machine BPF

Contrôle des appels système : SECCOMP (mode filtre) 3/7

Adaptation des programes BPF à SECCOMP :

- Au lieu d'agir sur des paquets réseau, les programmes BPF de SECCOMP agissent sur les appels système et leurs paramètres
- Plus précisément ils agissent sur une `struct seccomp_data` qui contient les informations relatives à l'appel système

```
struct seccomp_data {
  int nr;
  __u32 arch;
  __u64 instruction_pointer;
  __u64 args[6];
};
```

Contrôle des appels système : SECCOMP (mode filtre) 4/7

Une instruction BPF

```
struct bpf_insn {
  u_short opcode;
  u_char  jt;
  u_char  jf;
  u_long k;
};
```

- `opcode` correspond à l'identifiant de l'instruction
- `jt` (jump true) et `jf` (jump false) à des offsets pour les instructions de branchement
- `k` est utilisée de différentes façons suivant les instructions

→ Un programme BPF est une succession d'instructions en mémoire, c'est-à-dire un tableau de `struct bpf_insn`

→ Les offsets des instructions de branchement correspondent à des offsets dans le tableau à partir de la position courante

Contrôle des appels système : SECCOMP (mode filtre) 5/7

Il existe huit classes d'instructions, où une instruction est la somme de codes de modificateurs variés. Exemples :

`BPF_LD` : Copie une valeur dans l'accumulateur. Le type de la source est spécifié par un mode d'adressage :

- `BPF_IMM` pour une constante (champ k)
- `BPF_ABS` pour une donnée dans la structure `seccomp_data` à un offset fixe (champ k)
- `BPF_IND` pour une donnée dans cette structure à un offset variable (champ "valeur d'index + k")
- `BPF_LEN` pour la longueur de la donnée
- `BPF_MEM` pour un mot dans la mémoire de travail
- Pour les modes `BPF_ABS` et `BPF_IND`, la taille de la donnée doit être spécifiée et peut être un mot (`BPF_W`), ou un demi-mot (`BPF_H`), ou un octet (`BPF_B`)

→ Ex : `BPF_LD+BPF_W+BPF_ABS` charge dans l'accumulateur le mot de donnée de la structure `seccomp_data` situé à l'offset k

Contrôle des appels système : SECCOMP (mode filtre) 6/7

`BPF_JMP` : Modifie le flot d'exécution. Les sauts conditionnels compare l'accumulateur avec la constante k ou avec l'index. Si le résultat est "vrai" la branche `jt` est prise, sinon c'est la branche `jf` qui est prise

→ Ex : `BPF_JMP+BPF_JGT+BPF_K` saute à l'offset `jt` si l'accumulateur est strictement supérieur à k, sinon elle saute à `jf`

`BPF_RET` : Termine l'exécution du filtre et réalise une action spécifique suivant la valeur de k (`BPF_RET+BPF_K`). SECCOMP définit actuellement cinq actions qui sont représentées par les valeurs de k :

- `SECCOMP_RET_KILL` : Provoque la terminaison de la tâche
- `SECCOMP_RET_ALLOW` : Exécute l'appel système
- ...

L'interface BPF fournie enfin les deux macros suivantes pour faciliter l'écriture des filtres :

- `BPF_STMT(opcode, k)`
- `BPF_JUMP(opcode, k, jt, jf)`

Contrôle des appels système : SECCOMP (mode filtre) 7/7

Exemple de filtre BPF contrôlant notamment les appels système `read()` et `write()` :

```
#define syscall_arg(n) (offsetof(struct seccomp_data, args[n]))
#define syscall_nr (offsetof(struct seccomp_data, nr))

struct sock_filter filter[] = {
  /* Grab the system call number */
  BPF_STMT(BPF_LD+BPF_W+BPF_ABS, syscall_nr),
  /* Jump table for the allowed syscalls */
     [...]
  BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, __NR_exit, 0, 1),
  BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_ALLOW),
  BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, __NR_read, 1, 0),
  BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, __NR_write, 3, 2),
  /* Check that read is only using stdin */
  BPF_STMT(BPF_LD+BPF_W+BPF_ABS, syscall_arg(0)),
  BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, STDIN_FILENO, 3, 0),
  BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_KILL),
  /* Check that write is only using stdout */
  BPF_STMT(BPF_LD+BPF_W+BPF_ABS, syscall_arg(0)),
  BPF_JUMP(BPF_JMP+BPF_JEQ+BPF_K, STDOUT_FILENO, 0, 1),
  BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_ALLOW),
  BPF_STMT(BPF_RET+BPF_K, SECCOMP_RET_KILL),
}
```

Linux Control Group – Principe

Les control groups permettent de partitionner l'ensemble des processus/threads (tâches) du système en des ensembles, appelés "groupes", sur lesquels peuvent s'appliquer différents contrôles.

- Des sous-systèmes de contrôle permettent d'imposer des politiques d'utilisation de ressources pour différents groupes de tâches
- Le partitionnement de l'ensemble des tâches est à la discrétion de l'administrateur du système
  - Réalisé au travers d'un système de fichiers virtuel où les dossiers représentent les groupes, sous-groupes, etc.
  - Peut être spécifique à chaque sous-système de contrôle enregistré dans le noyau
  - Et modifié en cours d'exécution

Limites actuelles concernant la sécurité (refonte en cours des *control groups*) :

- Absence de contrôle de la profondeur de l'arborescence $\rightarrow$ DoS possible par surconsommation de la mémoire noyau
- Une sous arborescence déléguée à un utilisateur non privilégié peut être configurée de façon à perturber les processus d'autres groupes frères (même plus privilégiés)

Linux Control Group – Exemples

Différents contrôleurs :

- cpuset : Fournit un moyen pour assigner des ensembles de CPU et de noeuds de mémoire distincts à différents groupes de tâches
- blkio : Restriction des débits maximaux d'écriture et de lecture sur les différents périphériques bloc suivant les groupes de tâches
- devices : Restrictions d'accès aux périphériques du système suivant les groupes de tâches
- memcg : Contrôle de l'utilisation de mémoire (limite max pour un groupe donné, notification à l'espace utilisateur lors d'un dépassement, ...)

Auto-restriction des ressources d'une application : Landlock

Inspiré de SECCOMP-bpf mais complémentaire

- Permet à un programme de limiter son accès aux ressources lors de son exécution
- des programmes eBPF implémentent la logique d'accès et sont insérés dans le noyau via l'appel système `bpf` puis activé via l'appel système `seccomp` (opération `SECCOMP_PREPEND_LANDLOCK_HOOK`)
  - Ces programmes eBPF sont appelés au niveau des hooks LSM
  - Un tableau associatif spécifique (*BPF map*) est utilisée pour identifier les ressources à contrôler

  $\rightarrow$ permet de créer des logiques d'accès très flexible

Notes :

- Les programmes eBPF peuvent être associés à des `cgroup`
- Toujours en cours de développement
- Limitations actuelles : utilisation possible uniquement avec `CAP_SYS_ADMIN` et contrôle du FS uniquement

Auto-restriction du mode opératoire d'un processus sur OpenBSD

L'appel système pledge() permet de restreindre le mode opératoire du processus appelant par contrôle des appels-système

```
int pledge(const char *promises, const char *execpromises);
```

- Le processus fournit une liste d'engagements (*promises*) sous forme de chaîne de caractères (ex : `"stdio rpath"`)
- Les engagements sont définis comme des familles d'appels système, et pour certains des restrictions d'accès s'appliquent. Par exemple :
  - rpath : appels système liés au FS et ne provoquant pas d'accès en écriture (dans le contexte d'appel) comme `open(..., O_RDONLY)`
  - inet : appels système réseau dont la portée s'étend aux domaines `AF_INET(6)`
  - unix : idem mais pour le domaine `AF_UNIX`
- Tout appel ultérieur à `pledge` ne peut que restreindre encore plus le mode opératoire du processus (pas de possibilité de récupérer des privilèges)
- `execpromises` est la liste d'engagements hérités par les fils du processus appelant (en supposant qu'il dispose au moins de `"proc"` et `"exec"`)

Auto-restriction de l'accès au système de fichiers sur OpenBSD

L'appel système unveil() permet de restreindre l'accès au FS

```
int unveil(const char *path, const char *permissions);
```

- Le premier appel rend inaccessible le FS à l'exception du chemin fourni en argument et avec les droits spécifiés (combinaison de `'r'`, `'w'`, `'x'` et/ou `'c'`)
- Les appels suivant permettent de "dévoiler" (*i.e.* rendre accessible) d'autres parties du FS
- Appeler `unveil()` avec ses deux arguments à NULL empêche tout nouvel appel $\rightarrow$ fige la vision du FS pour le processus appelant

Espace de noms – Principe

Les espaces de noms sont une des solutions pour mettre en œuvre la virtualisation par *container*.

- Vise à construire un système dans lequel il est possible de créer des groupes de processus ayant des vues différentes des ressources contrôlées par le noyau (réseau, processus, IPC, etc.)
- Ces vues, appelées espace de noms, sont créées au travers de l'appel système `clone()` via les flags `CLONE_NEW*`
- Un nouvel espace n'est visible que par le processus appelant et sa descendance
- Un espace particulier `CLONE_NEWUSER` (depuis noyau 3.8) permet de créer différents espaces d'UID et GID
  - À la différence des autres, peut être créé sans être root
  - Rend accessible les fonctionnalités, normalement réservées à root, à tous les utilisateurs du système
    → **Uniquement au sein du container créé !**

Espace de noms – Exemples

`CLONE_NEWNET` :

- Création d'une nouvelle vue réseau associée au processus créé
- Il s'agit d'une vue particulière de la pile réseau, qui consiste en des instances différentes :
  - des interfaces réseau
  - des piles protocolaires IPv4 et IPv6
  - des tables de routage IP, des règles de pare-feu
  - des arborescences `/proc/net` et `/sys/class/net`
  - des sockets, etc.

`CLONE_NEWPID` :

- Création d'une vue vierge de tout processus existant dans laquelle le processus créé devient le PID 1
- Fonctionnement :
  - Tout nouveau processus dans cet espace de noms qui se retrouve orphelin est placé sous le giron du processus ayant initié le nouvel espace de PID
  - Les processus du nouvel espace de noms sont visibles depuis l'espace parent (l'inverse étant faux)

Mécanisme de sandboxing sur FreeBSD : Capsicum 1/7

Vue d'ensemble

- Sécurité basée *capability* (via sandboxing)
  - Attention : rien à voir avec les *capabilities* POSIX des systèmes *nix
  - Une *capability* = token d'autorité non reproductible ($\sim$ descripteur de fichier + autorisation)
- Approche non *full capability* mais hybride (on a toujours le principe d'autorité ambiante)
  - Différent des approches DAC, MAC, RBAC, ...
  - Mais complémentaire $\rightarrow$ permet d'aller plus loin en donnant les moyens à n'importe quelle application de se "sandboxer" elle-même (par exemple pour un navigateur : une sandbox par onglet, par plugin, etc.).
  - Permet de mettre en place une politique de sécurité basée sur l'application elle-même et non sur l'utilisateur (d'où la complémentarité avec des politiques de type MAC)
  - Permet de construire des applications respectant le principe du moindre privilège
- Protection fournie par l'OS via différents services
- Ne nécessite pas de privilèges OS pour créer les sandbox ou de droits administrateur
- Interface simple, bibliothèque existante pour faciliter son utilisation par les applications (ex : pour convertir chromium à capsicum sous FreeBSD $\rightarrow$ 100 lignes de code ajoutées)

Mécanisme de sandboxing sur FreeBSD : Capsicum 2/7

Bibliothèque `libcapsicum`

`libcapsicum` implémente 2 API qui permettent à une application de créer, gérer, et d'intéragir avec du logiciel "sandboxé" s'exécutant en mode "capability" :

- API `libcapsicum host` API pour les processus hôte. Permet de :
  - Lancer des composants logiciels à l'intérieur de sandbox
  - Éventuellement de communiquer avec les composants sandboxés (socket I/O, ou RPC)
- API `libcapsicum sandbox` pour les processus sandboxés qui s'exécute en mode "*capability*", et ne peuvent utiliser que les ressources qui ont été assignées à leur sandbox durant la création, ou bien a posteriori, mais fournies explicitement

Mécanisme de sandboxing sur FreeBSD : Capsicum 3/7

Analyse de GZIP en version Capsicum – Opération de compression "sandboxé"

```
insize = gz_compress_wrapper(in, out, &size, basename(file), (uint32_t)isb.st_mtime);

    [...]

off_t gz_compress_wrapper(int in, int out, off_t *gsizep, const char *origname, uint32_t mtime)
{
  gzsandbox_initialize();

  if (gzsandbox_enabled){
    return (gz_compress_insandbox(in, out, gsizep, origname,
    mtime));
  }else{
    return (gz_compress(in, out, gsizep, origname, mtime));
  }
}
```

- Rouge : API libcapsicum
- Vert : "fil conducteur"
- Bleu : descripteur de sandbox

Mécanisme de sandboxing sur FreeBSD : Capsicum 4/7

Analyse de GZIP en version Capsicum – gzsandbox_initialize

```
static void gzsandbox_initialize(void)
{
  if (gzsandbox_initialized){
    return;
  }
  gzsandbox_enabled = lch_autosandbox_isenabled("gzip");
  gzsandbox_initialized = 1;
  if (!gzsandbox_enabled){
    return;
  }

  if (lch_start("/usr/bin/gzip", lc_sandbox_argv, LCH_PERMIT_STDERR, NULL, &lcsp) < 0){
    err(-1, "lch_start %s", "/usr/bin/gzip");
  }
}
```

- `lch_*` : font partie de l'API host de la libcapsicum
- `lch_start()` : lance gzip dans une sandbox (à suivre ...)
- `lcsp` : référence sur la sandbox

Mécanisme de sandboxing sur FreeBSD : Capsicum 5/7

Analyse de GZIP en version Capsicum – gz_compress_insandbox

```
static off_t gz_compress_insandbox(int in, int out, off_t *gsizep, const char *origname, uint32_t mtime)
{
  ...

  fdarray[0] = cap_new(in, CAP_FSTAT | CAP_READ | CAP_SEEK);
  fdarray[1] = cap_new(out, CAP_FSTAT | CAP_WRITE | CAP_SEEK);
  ...

  if (lch_rpc_rights(lcsp, PROXIED_GZ_COMPRESS, &iov_req, 1,fdarray, 2, &iov_rep, 1, &len, NULL, NULL) < 0) {
    err(-1, "lch_rpc_rights");
  }

  ...
}
```

- `cap_new` : création des capabilities (descripteurs de ressources + droits d'accès)
- `lch_rpc_rights` : envoi du RPC `PROXIED_GZ_COMPRESS` à la sandbox avec les capabilities, et le nom du fichier à ouvrir (`iov_req`)

Mécanisme de sandboxing sur FreeBSD : Capsicum 6/7

Analyse de GZIP en version Capsicum – Traitement des RPC par la sandbox

```
int gzsandbox(void)
{
  ...
  if (lcs_get(&lchp) < 0) {
    errx(-1, "libcapsicum sandbox binary");
  }
  while (1) {
    fdcount = 2;
    if (lcs_recvrpc_rights(lchp, &opno, &seqno, &buffer, &len,
    fdarray, &fdcount) < 0) {
      ...
    }
    switch (opno) {
      case PROXIED_GZ_COMPRESS:
      ...
      sandbox_gz_compress_buffer(lchp, opno, seqno, buffer, len, fdarray[0], fdarray[1]);
      ...
      break;
```

- `gzsandbox` : correspond au `main()` du programme `gzip` lorsque le mode capsicum est activé.
  → `lch_start()` s'occupe de :
  - la création du processus qui accueille la sandbox
  - et du passage en mode capsicum via `cap_enter()` (indirectement via `lch_sandbox()`)
- `lcs_get` : récupération du descripteur de l'instance de la sandbox dans l'hôte (pour recevoir les RPC).
  → lorsque la sandbox est créée depuis l'hôte par `lch_start()`, un descripteur y est associée (pour la communication avec la sandbox notamment)

Mécanisme de sandboxing sur FreeBSD : Capsicum 7/7

Analyse de GZIP en version Capsicum – `sandbox_gz_compress_buffer`

```
static void sandbox_gz_compress_buffer(struct lc_host *lchp, uint32_t opno, uint32_t seqno, char *buffer,
      size_t len, int fd_in, int fd_out)
{
  struct host_gz_compress_req req;
  struct host_gz_compress_rep rep;
  struct iovec iov;

  if (len != sizeof(req)){
    err(-1, "sandbox_gz_compress_buffer: len %zu", len);
  }

  bcopy(buffer, &req, sizeof(req));
  bzero(&rep, sizeof(rep));
  numflag = req.hgc_req_numflag;
  rep.hgc_rep_retval = gz_compress(fd_in, fd_out, &rep.hgc_rep_gsize,
  req.hgc_req_origname, req.hgc_req_mtime);
  iov.iov_base = &rep;
  iov.iov_len = sizeof(rep);
  if (lcs_sendrpc(lchp, opno, seqno, &iov, 1) < 0)
  err(-1, "lcs_sendrpc");
}
```

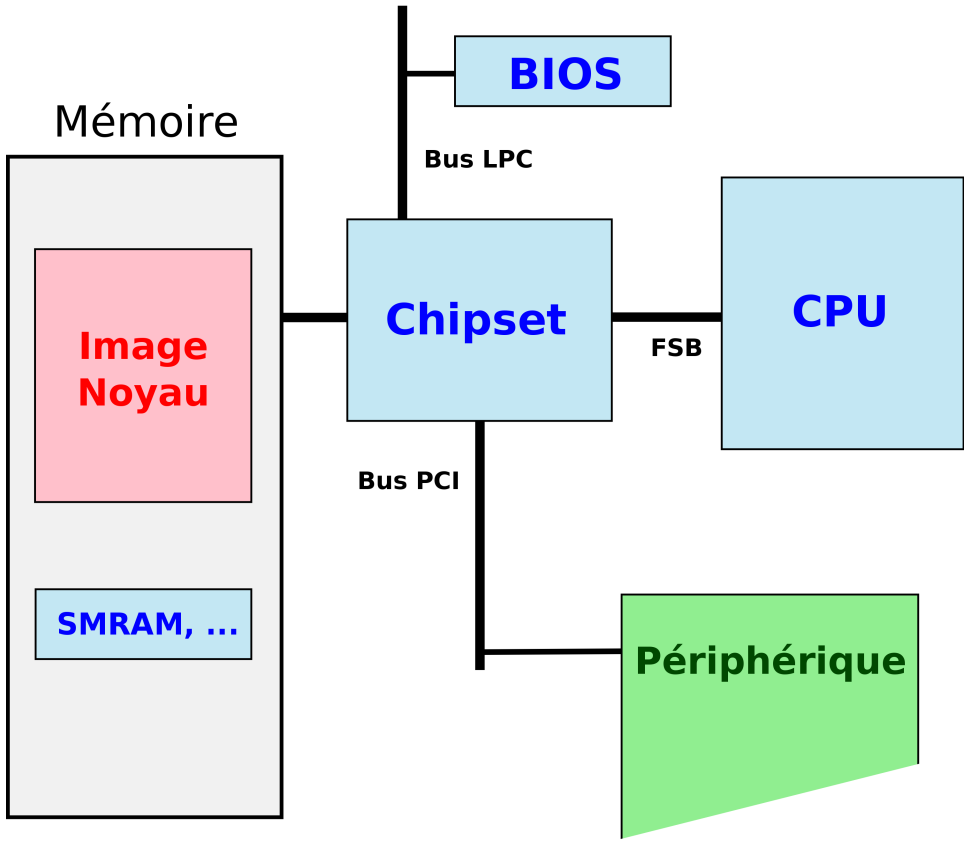
→ On retrouve ici l'appel à la primitive de compression du programme `gzip` original

Plan

Perte d'intégrité d'un noyau

Provient de diverses altérations :

- Image du noyau en mémoire
  - Code du noyau
  - Données du noyau
- Support de contrôle
  - Registres/Mémoire de la CPU
  - Registres du Chipset
  - BIOS
- Périphériques avec lesquels communique le noyau



Focus sur l'altération de l'image du noyau en mémoire

Comment accéder à la mémoire du noyau ?

- Depuis la CPU
  - Détournement de fonctionnalités du système
    $\rightarrow$ Abus de privilèges
  - Exploitation de failles de sécurité du système
- Depuis des périphériques aptes à effectuer du DMA
  - À l'initiative d'un périphérique (Firewire, ...)
  - Actions commandées par la CPU

Plan

Abus de privilèges

Des fonctionnalités du système fournissent directement le moyen de modifier n'importe quelle région de l'espace mémoire du noyau :

- Fonctionnalités logicielles : chargeur de modules noyau, accès aux périphériques virtuels `/dev/kmem`, `/dev/mem` et `/dev/ports`
- Fonctionnalités matérielles : mode SMM (*System Management Mode*) de la CPU, ACPI, ...

Injection de malware via le chargeur de modules noyau

Nécessite d'être `root` ou d'avoir la capacité `CAP_SYS_MODULE`.

Deux possibilités :

- Injection directe par l'attaquant d'un module malveillant après avoir obtenu les privilèges nécessaires
- Injection par un tiers après corruption par l'attaquant d'un module existant

Injection de malware via /dev/kmem

Nécessite d'être root ou d'avoir la capacité CAP_SYS_RAWIO (ne fonctionne plus sur les distributions Linux actuelles).

Extrait de code simplifié pour écrire dans /dev/kmem

```
unsigned int write_kmem(off64_t offset, void *buff,
                        unsigned int len)
{
   kernel_desc = open("/dev/kmem", O_RDWR|O_LARGEFILE);
   lseek64(kernel_desc, offset, SEEK_SET);
   write(kernel_desc, buff, len);

   return len;
}
```

Plan

Focus sur une vulnérabilités noyau particulière

Déréférencement de pointeurs de l'espace utilisateur depuis le noyau (cas particulier : déréférencement de pointeurs nuls).

Principe :

- Un attaquant crée dans l'espace d'adressage d'un processus qu'il contrôle une région mémoire malveillante (code ou données) à une adresse X
- Il exploite une vulnérabilité d'un service du noyau (généralement un appel système), lui permettant d'effectuer un débordement sur un pointeur, et d'y inscrire l'adresse X
- Lorsque le noyau déréférence le pointeur :
  - Si pointeur de fonction, alors exécution du code malveillant situé en espace utilisateur, en mode noyau (*ring 0*)
  - Si pointeur de données, alors utilisation de données choisies par l'attaquant et pouvant aboutir à une modification malveillante du comportement du noyau

Exemple : vulnérabilités ayant affecté vmsplice() [2008] 1/4

vmsplice() est un appel système sous Linux qui sert à connecter un descripteur de fichier à une région de mémoire de l'espace utilisateur.

```
ssize_t vmsplice(int fd, const struct iovec *iov,
                 unsigned long nr_segs, unsigned int flags);
```

- Implémentation initialement vulnérable dans les versions 2.6.17 à 2.6.24 (CVE-2008-0009 et CVE-2008-0010)
- Conséquences : exécution arbitraire en mode noyau depuis un compte utilisateur sans privilège

Exemple : vulnérabilités ayant affecté vmsplice() [2008] 2/4

Principe d'exploitation :

1. Mise en place de régions mémoire (via `mmap()`)
   - en 0 et 4096, des `struct page` de type *compound* sont écrites
   - une région pour stocker un shellcode noyau
   - une autre région mémoire M
2. Création d'un *pipe* qui sera utilisé comme destination dans `vmplice()`, et fermeture de sa partie en lecture
3. Appel de `vmsplice()` avec le `pipe` et une `struct iovec` pointant sur la région M mais ayant une longueur `ULONG_MAX`

```
iov.iov_base = map_addr;
iov.iov_len  = ULONG_MAX;
vmsplice(pi[1], &iov, 1, 0);
```

Exemple : vulnérabilités ayant affecté vmsplice() [2008] 3/4

4. Débordement d'entier dans `get_user_pages()` utilisée par `vmsplice()` (décrémentation de la valeur 0)
   1. Entraîne le débordement du tableau `partial[PIPE_BUFFERS == 16]` placé en mémoire sous le tableau `pages[]`.
      - les valeurs écrites sont alternativement 0 et 4096 (correspondant au champ `offset` et `len` des `struct partial_page`)
      - les pointeurs dans `pages[]` sont donc remplacés par les valeurs 0 et 4096
   2. [Effet de bord] Entraîne le débordement du tableau `pages[PIPE_BUFFERS]` qui contient des pointeurs vers des descripteurs de pages (la région de mémoire en espace utilisateur)

- Ces débordements s'arrêtent avant d'avoir atteint les données de contrôle dans la pile.
  $\rightarrow$ `get_user_pages()` retourne normalement

Le cas de vmsplice() - Exécution arbitraire en mode noyau 4/4

5. La routine `splice_to_pipe()` est alors exécutée
   - Retourne presque immédiatement (car le `pipe` a été fermé en lecture)
   - Entame une procédure de nettoyage qui s'effectue sur la région mémoire fournie en paramètre à `vmsplice()`
   - Note : le nettoyage est effectué en consultant les `struct page` associées, référencées dans le tableau `pages[]`
6. Le shellcode noyau est exécuté lors du nettoyage des pages dont les descripteurs sont lus aux adresses 0 et 4096
   - Ces descripteurs de pages *compound* contiennent un destructeur ($\rightarrow$ un pointeur de fonction) qui pointe vers le shellcode.
   - Extrait du descripteur :

```
pages[0]->flags    = 1 << PG_compound;
pages[0]->private  = (unsigned long) pages[0];
pages[0]->count    = 1;
pages[1]->lru.next = (long) kernel_code;
```

   - Le destructeur (pointé par `->lru.next`) contenant le code de l'attaquant en espace utilisateur est exécuté par le noyau en *ring 0*

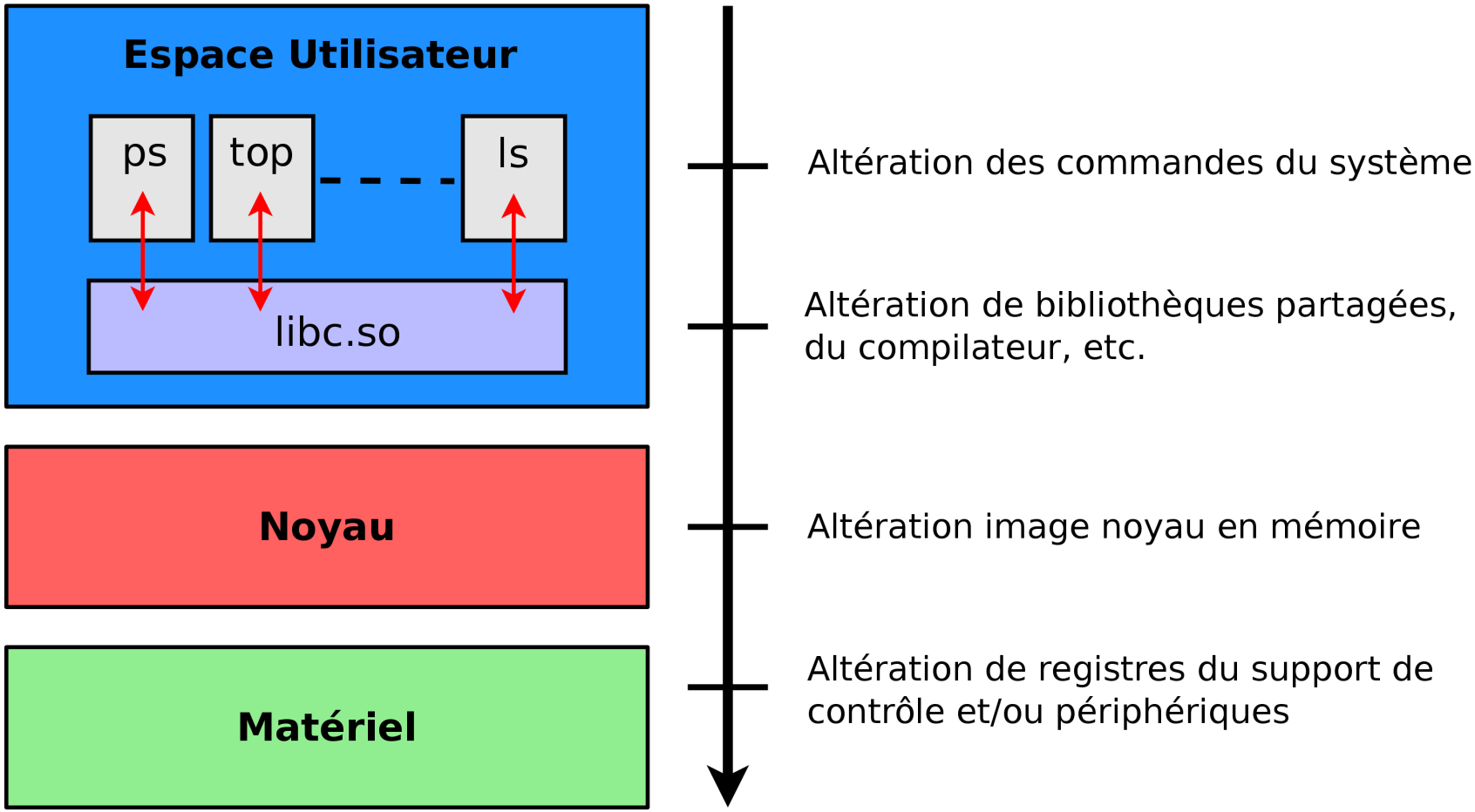
Plan

Définition de *rootkit*

- Ensemble de modifications sur un système informatique
- Maintien dans le temps
- Contrôle frauduleux du système

Attributs essentiels d'un *rootkit*

- Invisibilité : exprime la difficulté de la détection du *rootkit*
- Robustesse : exprime la difficulté de retirer le *rootkit* du système
- Pouvoir de nuisance : exprime la "gravité" de l'action du *rootkit*


Espace Utilisateur
ps
top
ls
libc.so
Noyau
Matériel
Altération des commandes du système
Altération de bibliothèques partagées, du compilateur, etc.
Altération image noyau en mémoire
Altération de registres du support de contrôle et/ou périphériques

Plan

Plan

Restrictions sur le chargeur de module et sur `/dev/kmem`

PaX propose différents mécanismes visant à limiter l'insertion de modules noyau malveillants :

- Autoriser le chargement automatique de modules uniquement pour l'utilisateur `root`
  - Pour éviter qu'un utilisateur sans privilège puisse déclencher le chargement de modules vulnérables
- Chargement des modules limité au démarrage
  - Une fois les modules chargés (par l'`initramfs` par exemple) il suffit de positionner le sysctl `disable_modules` à 1
    (`echo 1 > /proc/sys/kernel/grsecurity/disable_modules`)
  - Il n'est alors plus possible de charger un nouveau module noyau avant le prochain redémarrage du système

L'accès aux périphériques virtuels `/dev/kmem` et `/dev/mem` a également été désactivé dans les noyaux récents, excepté pour les accès en MMIO (drivers graphiques sous X notamment)
$\rightarrow$ Régions circonscrites en mémoire

Signature des modules noyau : patch de David Howells

- Paire de clé publique/privée générée initialement par l'utilisateur, ou la distribution Linux, ...
  - Clé publique utilisée par le noyau pour vérifier l'authenticité des modules à charger
  - Clé privée employée par l'utilisateur pour signer les modules noyau
- La clé publique est inscrite après génération dans le fichier `crypto/signature/key.h`
- Une nouvelle section ELF `.module_sig` est ajoutée au format des modules noyau pour y inscrire la signature

Références :

- `Documentation/module-signing.txt`
- https://lwn.net/Articles/470906/

Chargement de modules depuis un système de fichier de confiance : LoadPIN

Le LSM LoadPIN restreint le chargement de modules noyau

- `CONFIG_SECURITY_LOADPIN`
- Lors du chargement du premier module, LoadPIN retient le FS d'où il provient et restreint tout futur chargement à ce FS
- si le FS n'est pas en lecture seule, cette information est écrite dans le journal système

Plan

Le projet Kernel Self-Protection

Le projets a pour objectif la conception et l'implémentation de solutions visant à protéger le noyau de ses propres faiblesses.

Catégories des solutions :

- Réduction de la surface d'attaque du noyau (ex : W^X, KPTI – *Kernel Page Table Isolation*)
- Intégrité de la mémoire (protection contre stack buffer/depth overflow, counter overflow, sanity checks sur les méta-données du tas...)
- Défenses statistiques (ex : KASLR, canaries)
- Prévention des fuites d'information (adresses et contenu)

Protection `mmap_min_addr`

- Empêche le mapping de régions mémoire en dessous de l'adresse spécifiée dans le sysctl `/proc/sys/vm/mmap_min_addr`
- Permet d'éviter l'exploitation de vulnérabilité de type "déréférencement de pointeur NULL" ou avoisinant
- Les débordements de tampon écrasant un seul octet d'un pointeur null (off-by-one) peuvent également être couvert par ce mécanisme
  $\rightarrow$ en positionnant `mmap_min_addr` à 4096
- Non suffisant pour se protéger contre tout déréférencement de pointeur en espace utilisateur
- Note : KPTI rend ce mécanisme obsolète pour se protéger d'une attaque visant le noyau

Protection contre le déréférencement de pointeur en espace utilisateur 1/2

PaX UDEREF :

- Sur x86 32 bits : UDEREF repose sur l'utilisation de la segmentation
  - Les segments de l'espace utilisateur et du noyau sont distincts
  - Si le noyau accède à l'espace utilisateur, une exception (GPF) est levée par le processeur

- Sur x86 64 bits : La segmentation ne garantie plus rien, d'où l'implémentation d'un mécanisme différent
  - Lors d'une transition du mode utilisateur vers le mode noyau, la projection de l'espace utilisateur est supprimée, puis projetée à un autre endroit en utilisant l'attribut NX

Protection contre le déréférencement de pointeur en espace utilisateur 2/2

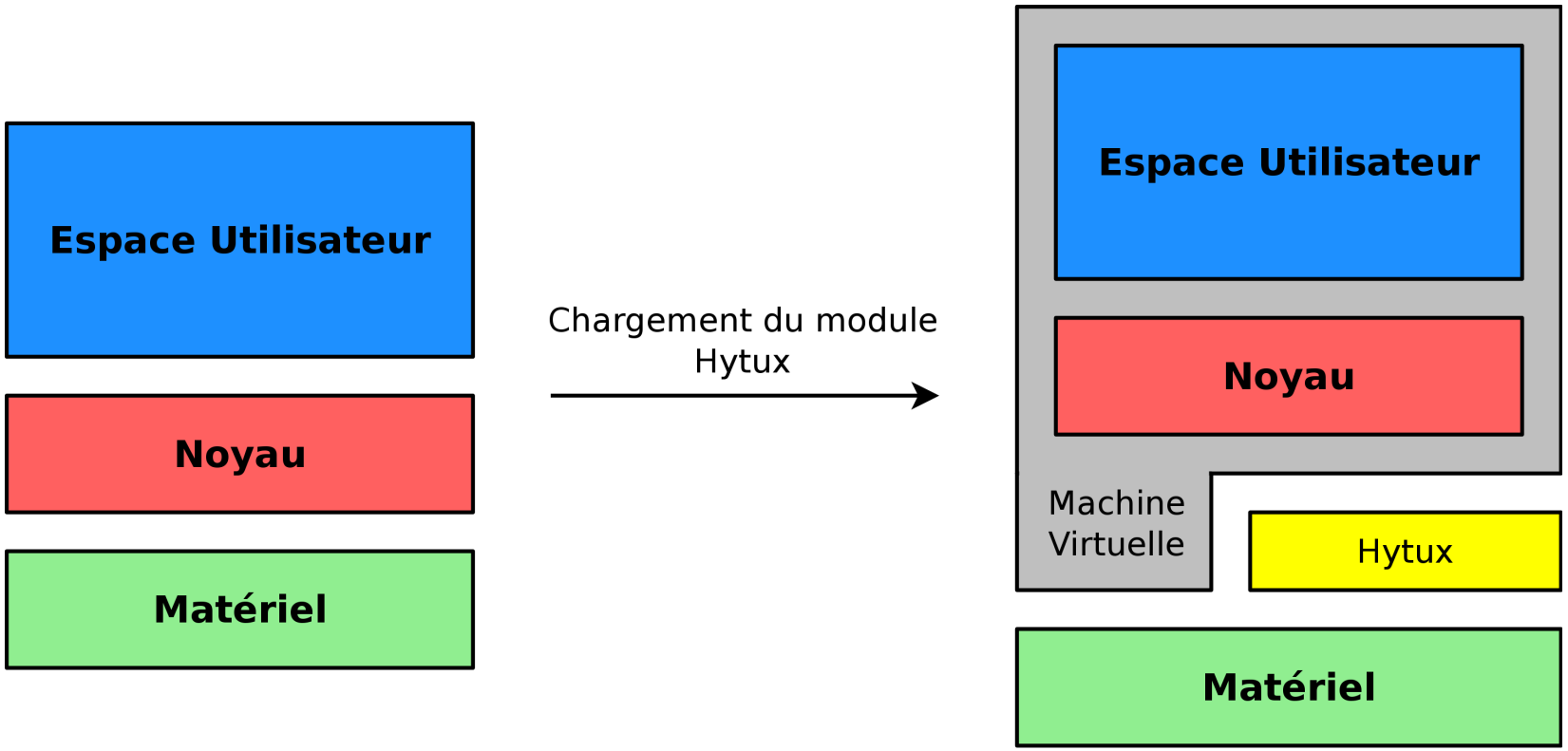
Protection Hardware (sur Intel) :

- bit SMEP (Supervisor Mode Execution Prevention) dans le `CR4`
  - Quand positionné à 1 : l'exécution depuis le mode noyau de code en espace utilisateur provoque une faute de page
- bit SMAP (Supervisor Mode Access Prevention) dans le `CR4`
  - Quand positionné à 1 : les accès R/W à l'espace utilisateur depuis le mode noyau provoque une faute de page
  - débrayage du mécanisme via instruction CTAC qui positione le bit EFLAGS.AC et réactivation via STAC
- Protection Keys (`CR4.PKE = 1`)
  - Permet d'empêcher l'accès à des régions de mémoire identifiées par une clé
  - clé : valeur 1 à 15 encodée dans les bits `[62:59]` des différentes entrées des tables de pages
  - registre PKRU : permet de spécifier les droits d'accès pour chaque clé

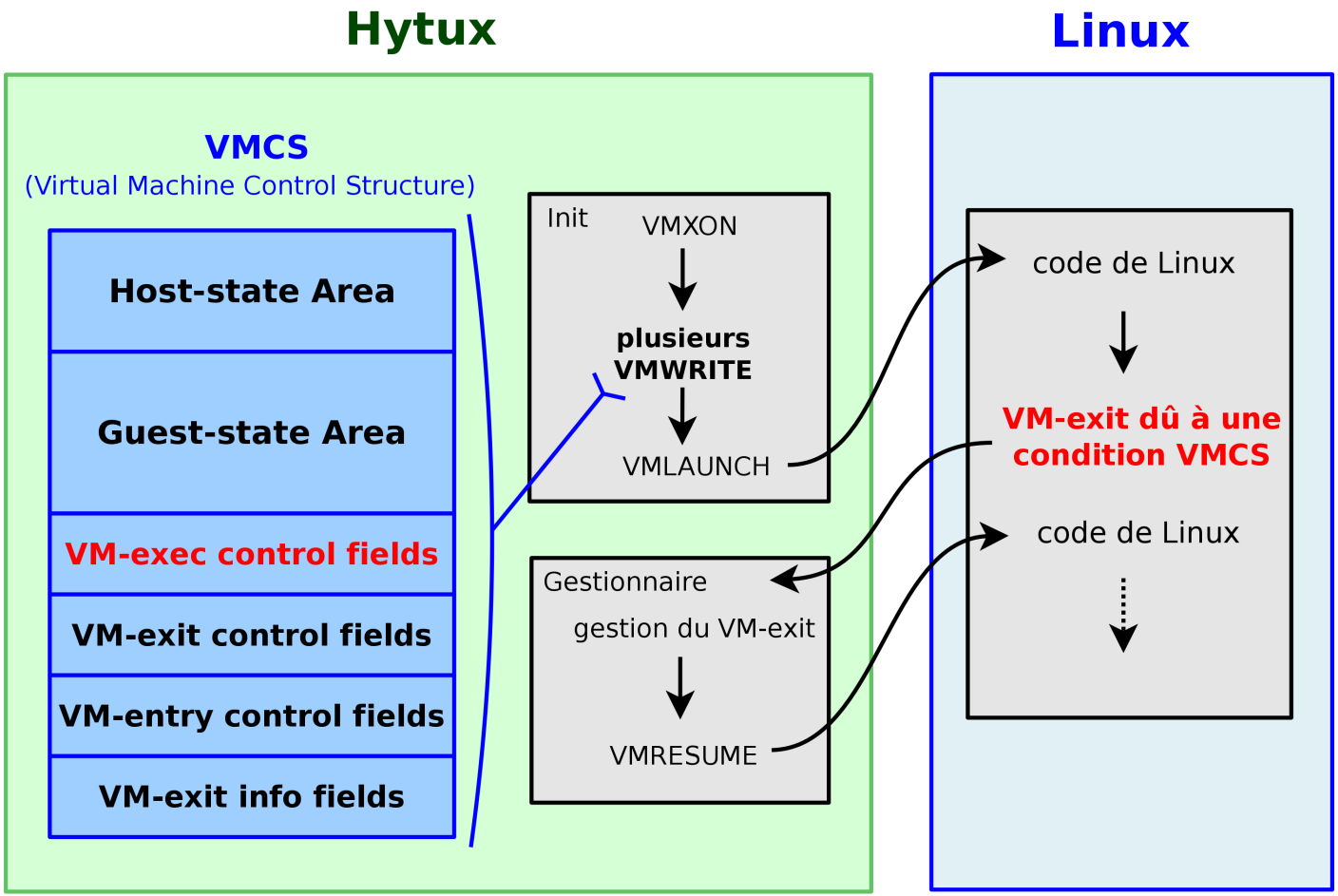
Contrôle du noyau par une entité plus privilégiée : hyperviseur léger

- Contrôle des actions du noyau en le plaçant dans une machine virtuelle
- Utilisation des technologies matérielles d'aide à la virtualisation (Intel VT, AMD SVM) pour :
  1. intercepter tout type d'actions que souhaitent effectuer le noyau, mais avant réalisation
  2. contrôler (via l'hyperviseur de sécurité) le contexte dans lequel l'action s'apprête à être réalisée
     - soit, action jugée correcte et exécution
     - soit, action jugée incorrecte et réaction (émulation de l'action, notification d'incident, mécanisme de résilience – reconfiguration, suppression de la menace – tuer le processus hostile, etc.)
- Faible taille $\rightarrow$ plus facile à auditer / prouver
- Pratique pour le contrôle d'OS COTS

Hytux - 1/2


Espace Utilisateur
Noyau
Matériel
Chargement du module Hytux
Espace Utilisateur
Noyau
Machine Virtuelle
Hytux
Matériel

Hytux - 2/2


Hytux
Linux
VMCS
(Virtual Machine Control Structure)
Host-state Area
Guest-state Area
VM-exec control fields
VM-exit control fields
VM-entry control fields
VM-exit info fields
Init
VMXON
plusieurs
VMWRITE
VMLAUNCH
Gestionnaire
gestion du VM-exit
VMRESUME
code de Linux
VM-exit dû à une
condition VMCS
code de Linux

Plan

Autres thématiques sur la sécurité système

- Nous avons vu des mécanismes pour améliorer la sécurité des OS. Mais d'autres protections sont nécessaires pour assurer l'intégrité du noyau et plus généralement l'intégrité du système :
  - Protection de la mémoire principale face aux attaques des périphériques ou autres contrôleur d'E/S (besoin d'une I/O MMU)
  - Protection du support de contrôle (CPU, chipset, BIOS) et des périphériques contre des attaques issues de la CPU (code malveillant) ou de périphériques (attaques DMA)
- Nous avons abordé la sécurité des noyaux monolithiques (comme Linux, Windows, ...), mais il existe des architectures de noyau pensée pour la sécurité :
  - Micro-noyau
  - Separation kernel / Architecture MILS

  Note : L'infrastructure VFIO (*Virtual Function I/O*) de Linux permet l'implémentation de *drivers* en espace utilisateur $\rightarrow$ approche micro-noyau limitée aux drivers

Merci de votre attention