~~CONFIDENTIEL DÉFENSE~~ ~~SPECIAL FRANCE~~

Liberté • Égalité • Fraternité
RÉPUBLIQUE FRANÇAISE

PREMIER MINISTRE

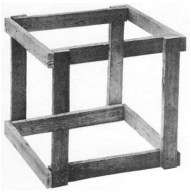
Secrétariat général de la défense et de la sécurité nationale

Agence nationale de la sécurité des systèmes d'information

DÉCLASSIFIÉ
par décision n°15699/ANSSI/SDE/ST/LAM
du 18 juillet 2018

Documentation CLIP
1201

CLIP-LSM



ANSSI, 51 boulevard de la Tour Maubourg, 75700 Paris 07 SP.

~~CONFIDENTIEL DÉFENSE~~

HISTORIQUE

Révision	Date	Auteur	Commentaire
2.2	08/12/2008	Vincent Strubel	Ajout des *xid* aux *struct clsm_file_sec*. A jour pour *clip-kernel-2.6.22.24-r19* (CLIP v03.00.25)
2.1	14/10/2008	Vincent Strubel	Ajout des privilèges *CLSM_PRIV_XFRM**, à jour pour *clip-kernel-2.6.22.24-r14* (CLIP v03.00.21).
2.0	10/10/2008	Vincent Strubel	Description du nouveau code "modularisable", ajout de la section 5.
1.6.1	30/07/2008	Vincent Strubel	Convention plus lisible pour les références. Corrections de style pour mettre en harmonie avec les documents plus récents.
1.6	28/04/2008	Vincent Strubel	Ajout du contrôle d'accès sur *pivot_root* et les montages *MS_MOVE*, et des options par défaut pour le contrôle des nouveaux montages. Retrait de *CLSM_CHROOT_PTRACE* de la configuration par défaut.
1.5	13/02/2008	Vincent Strubel	Ajout de l'option de montage *MS_NOSYMFOLLOW* et du drapeau *CLSM_FLAG_MIGRATED*, modification de la prise en compte de *CLSM_PRIV_IMMORTAL*. Conversion au format OpenOffice Open Document. A jour pour *clsm-2.6.22.17*.
1.4	18/09/2007	Vincent Strubel	Ajout de l'option de montage *MS_NOLOCK*. A jour pour *clsm-2.6.19.7-r15*.
1.3	14/09/2007	Vincent Strubel	Ajout des mécanismes spécifiques de gestion du *swap*. A jour pour *clsm-2.6.19.7-r14*.
1.2	22/08/2007	Vincent Strubel	Ajout du sous-système *devctl* et de l'utilitaire du même nom. Rétablissement du contrôle des nouveaux montages, basé sur *devctl*.
1.1	26/07/2007	Vincent Strubel	Mise à jour pour *clsm-2.6.21.6-r1 / 2.6.19.7-r10* : suppression du contrôle des nouveaux montages, ajout des options *CLSM_NET_FULL* et *CLSM_NOSUID_ROOT*, interdiction des accès en écriture aux fichiers projetés en exécution, mécanisme de cache pour *veriexec*.
1.0	18/06/2007	Vincent Strubel	Version initiale, à jour pour le patch *clsm-2.6.19.7-r9* et *verictl-1.1.8*.

Table des matières

Introduction 5

1 Rappels : capacités POSIX et Linux, *hooks* LSM 6
1.1 Modèle de capacités POSIX et Linux . . . 6
1.2 Interface Linux Security Modules . . . 7

2 LSM CLIP 7
2.1 Etiquettes de sécurité . . . 7
2.1.1 Etiquettes sur les tâches et les exécutables . . . 7
2.1.2 Etiquettes sur les fichiers . . . 10
2.1.3 Etiquettes sur les *inodes* . . . 10
2.2 Gestion des capacités et privilèges CLSM . . . 11
2.2.1 Gestion des capacités par exécutables . . . 11
2.2.2 Privilèges CLSM complémentaires . . . 12
2.2.3 Gestion de niveaux de privilèges hétérogènes . . . 15
2.3 Contrôle d'accès réseau . . . 18
2.3.1 Contrôle des opérations sur les *sockets* non locales . . . 18
2.3.2 Contrôle des opérations sur les *sockets Netlink* . . . 20
2.3.3 Contrôle du paramétrage IPsec . . . 20
2.4 Gestion des montages VFS . . . 20
2.5 Durcissement des cages *chroot* . . . 23
2.6 Interface utilisateur et contrôle d'accès . . . 24
2.6.1 Options de compilation . . . 24
2.6.2 Interface *sysctl* . . . 26
2.6.3 Interface *proc* . . . 26
2.6.4 Journalisation . . . 27

3 Sous-système *devctl* 28
3.1 Base d'entrées *devctl* . . . 28
3.2 Intégration au contrôle d'accès . . . 29
3.3 Interfaces utilisateur . . . 29
3.3.1 Options de compilation . . . 30
3.3.2 Fichier spécial */dev/devctl* . . . 30
3.3.3 Interface *proc* . . . 30

4 Sous-système *veriexec* 30
4.1 Base d'entrées *veriexec* et vérifications dynamiques . . . 30
4.1.1 Entrées *veriexec* . . . 30
4.1.2 Traitement *veriexec* lors d'un appel *exec* . . . 31
4.1.3 Autres appels au sous-système *veriexec* . . . 33
4.1.4 Mise en cache des vérifications *veriexec* . . . 33
4.2 Intégration à la gestion de privilèges . . . 36
4.3 Partitionnement *veriexec* . . . 37
4.4 Interfaces utilisateur et contrôle d'accès . . . 42
4.4.1 Options de compilation . . . 42
4.4.2 Fichier spécial */dev/veriexec* . . . 42
4.4.3 Interface *proc* . . . 44

4.4.4 Journalisation . . . 45

5 Insertion du LSM CLIP dans le noyau 45
5.1 *Hooks* ajoutés à l'interface LSM . . . 46
5.1.1 *Hooks* sur les *inodes* . . . 46
5.1.2 *Hooks* sur les fichiers . . . 46
5.1.3 *Hooks* sur les tâches . . . 47
5.1.4 *Hooks* IPsec . . . 48
5.1.5 Autres *hooks* . . . 49
5.2 Autres modifications des sources du noyau . . . 49
5.2.1 Options de montage . . . 49
5.2.2 Interdiction des accès en écriture aux projections exécutables . . . 49
5.2.3 Blocage temporaire des *fork()* . . . 50
5.2.4 Export de symboles supplémentaires . . . 50
5.2.5 Adaptation de *grsecurity* . . . 50
5.3 Initialisation du LSM CLIP . . . 50
5.3.1 Cas de la compilation statique . . . 51
5.3.2 Cas de la compilation modulaire . . . 51

6 Utilitaires *verictl* et *devctl* 51
6.1 Utilitaire *verictl* . . . 52
6.1.1 Ajout / Suppression d'entrées . . . 52
6.1.2 Lecture / Modification de niveau . . . 52
6.1.3 Ajout / Suppression / Modification d'un contexte . . . 53
6.1.4 Autres opérations . . . 54
6.2 Utilitaire *devctl* . . . 55

Références 56

Résumé

Le patch CLIP-LSM (aussi abrégé en "CLSM") est un module de sécurité pour le noyau Linux développé spécifiquement dans le cadre du projet CLIP. Son implémentation en couche noyau se décompose en deux sous-modules principaux. Le module de sécurité (LSM) proprement dit, d'une part, définit un certain nombre de points de contrôle associés à des opérations sensibles, afin de mettre en œuvre une gestion plus fine et plus restrictive des privilèges noyau traditionnels, et de contrôler certaines opérations qui ne sont normalement pas considérées comme privilégiées dans un système Linux, par exemple l'ouverture d'une connexion réseau. Ce module LSM est complété, d'autre part, par deux sous-systèmes : *devctl*, qui permet de configurer un contrôle d'accès mandataire aux périphériques de type bloc, et *veriexec*, qui permet d'associer des privilèges spécifiques à des exécutables individuels, en conditionnant l'attribution de ces privilèges à la vérification de l'empreinte cryptographique de l'exécutable. Certains privilèges qui sont dans un système Linux classique attribués automatiquement, soit à tous les utilisateurs (cas de la possibilité d'ouvrir une connexion réseau), soit au seul utilisateur *root* (cas notamment des capacités POSIX), deviennent du fait de restrictions supplémentaires imposées par le LSM CLIP, accessibles uniquement à travers des exécutables " privilégiés " au sens de *veriexec*. De manière complémentaire, le module LSM introduit des mécanismes spécifiques permettant de mieux protéger les processus exécutant ces fichiers " privilégiés ", afin d'éviter la récupération de leurs privilèges par d'autres processus (à travers par exemple l'envoi de signaux arbitraires, ou un attachement *ptrace*). Afin de limiter la charge de maintenance de ce développement spécifique, les différents mécanismes de contrôle d'accès implémentés par le LSM CLIP s'appuient autant que possible sur l'interface *Linux Security Modules* offerte par le noyau Linux à de tels modules de sécurité. Cette interface doit néanmoins être complétée de quelques points de contrôle spécifiques.

Le présent document, après un résumé de quelques principes de base de la gestion de privilèges sous Linux, détaille la conception du LSM CLIP et des deux sous-systèmes noyau associés, ainsi que celle des utilitaires *verictl* et *devctl*, utilisés pour configurer les sous-systèmes *veriexec* et *devctl* respectivement, depuis la couche utilisateur.

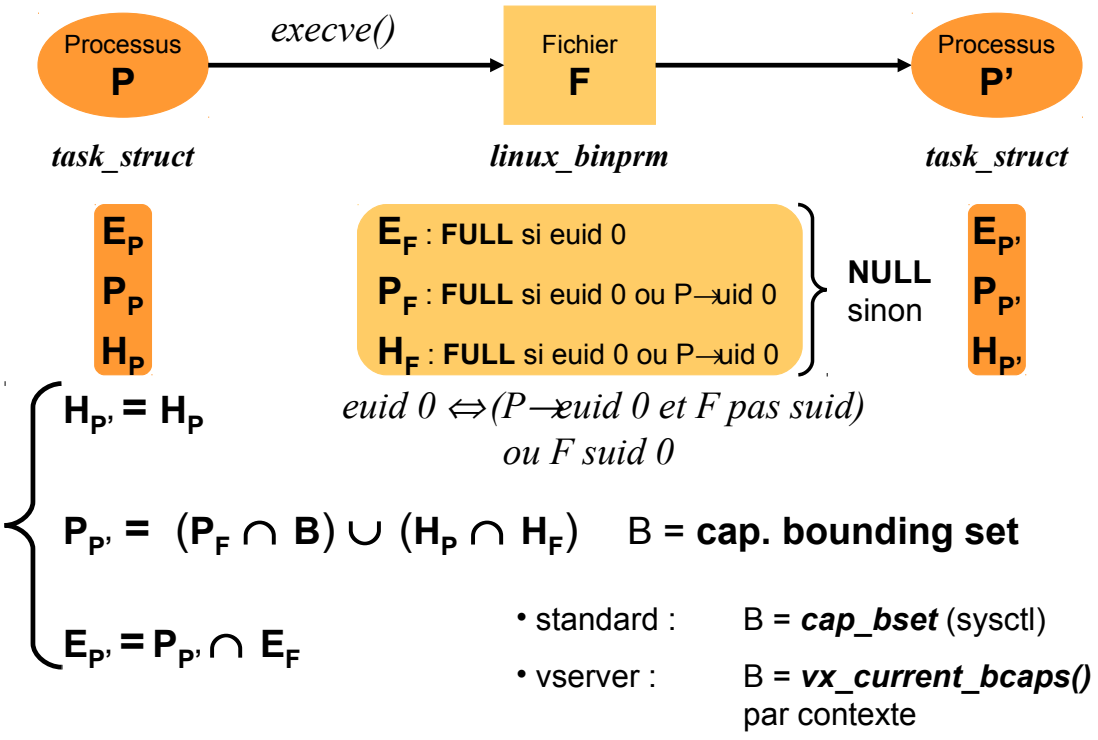


FIGURE 1 – Modèle de capacité POSIX, sous Linux standard et Linux patché *Vserver*.

1 Rappels : capacités POSIX et Linux, *hooks* LSM

1.1 Modèle de capacités POSIX et Linux

Le mécanisme de capacités POSIX, sur lequel est basé le contrôle d'accès au sein du noyau Linux, résulte d'une implémentation partielle du *draft* POSIX 1003.1e. Les privilèges habituellement attribués à *root* sont répartis en plusieurs [1] "capacités". Un masque de capacités, correspondant à un sous-ensemble de ces privilèges, est normalement représenté par un *bitmap* sur 32 bits. Trois tels *bitmaps* sont associés à chaque processus (*struct task_struct* de Linux) du système, représentant respectivement les masques **effectif** (*cap_effective*), **permis** (*cap_permitted*), et **héritable** (*cap_inheritable*) du processus. Lors d'un appel de type *exec*, trois masques similaires sont associés au fichier exécutable (*struct linux_binprm* sous Linux), et combinés aux masques du processus appelant afin de déterminer les masques du processus résultant de l'appel. Le calcul associé est résumé dans la figure 1. D'autres transformations sont associées par exemple aux appels *setuid()/setgid()*, et implémentées dans le fichier *security/commoncap.c* au sein de l'arborescence source de Linux.

L'implémentation du *draft* POSIX 1003.1e n'est cependant que partielle sous Linux, dans la mesure où elle n'est pas associée à un mécanisme d'attribution de capacités à des fichiers individuels. A défaut d'un tel mécanisme, Linux fonctionne selon un mode de compatibilité, qui accorde l'ensemble des capacités (à l'exception de *CAP_SETPCAP*, qui n'aurait de sens que dans une implémentation complète) à tout processus *root* (y compris suite à l'exécution d'un binaire *suid*), et aucune capacité aux processus non-*root*. On retrouve ainsi le modèle UNIX classique d'un *root* "surpuissant". La gestion plus fine des privilèges permise par le modèle

1. XXXX31 dans les versions récentes de Linux, auxquelles le patch *vserver* ajoute une trente-deuxième capacité, *CAP_CONTEXT*. Voir aussi [CLIP 1204].

de capacités n'est mise en œuvre qu'à travers le mécanisme de *cap_bound*, permettant de limiter globalement et de manière irréversible les privilèges de *root* sur le système. On notera que le masque de capacités héritable, qui permet de contourner cette limite, n'est pas mis en œuvre par défaut sous Linux : ce masque reste nul pour tous les processus.

1.2 Interface Linux Security Modules

L'interface LSM (*Linux Security Modules*) constitue un ajout plus récent au noyau Linux. Elle offre un certain nombre de points de contrôle prédéfinis (*hooks*), associés aux principales opérations privilégiées au sein du noyau, ainsi qu'aux transitions d'état (*exec(), setuid()*, etc.). Ces points de contrôle (définis dans l'en-tête *include/linux/security.h*) peuvent être utilisés par un module externe afin de définir une politique de contrôle d'accès obligatoire. Les fonctions invoquées à cette occasion peuvent par ailleurs s'appuyer sur des informations portées par des étiquettes de sécurité (ou *security blobs*) qui peuvent être attachées aux principales structures du noyau (*task_struct*, *file*, *socket*, *linux_binprm*) grâce à l'inclusion de pointeurs *void *security* (ou *f_security*) dans ces structures, et à la définition de *hooks* d'allocation/désallocation (**_alloc_security() / *_free_security()*) qui sont automatiquement invoqués lors de l'allocation et de la désallocation de telles structures. Le type exact de ces *blobs*, et des informations qu'ils contiennent, sont laissés au choix des développeurs de chaque module LSM.

Le modèle par défaut des capacités POSIX est repris dans un module LSM inclus dans les sources du noyau (option "*Default Linux Capabilities*", sources dans *security/capability.c*), mais les mêmes *hooks* peuvent être utilisés pour un contrôle d'accès plus fin, comme c'est le cas par exemple avec *SELinux*. On notera en revanche que *grsecurity* ne s'appuie pas sur l'interface LSM, mais plutôt sur ses propres *hooks*, ce qui conduit à un patch beaucoup plus intrusif. Afin de faciliter la maintenance et le suivi des nouvelles versions noyau, le patch CLIP-LSM utilise au contraire autant que faire se peut les points de contrôle prédéfinis de l'interface LSM.

2 LSM CLIP

Le LSM CLIP est implémenté dans plusieurs fichiers sources spécifiques, rassemblés dans un nouveau répertoire *security/clsm* au sein de l'arborescence source de Linux. Son interface est décrite dans un fichier d'en-tête unique, *include/linux/clip_lsm.h*. Par ailleurs, sa mise en œuvre nécessite aussi l'application d'un *patch* à certains fichiers sources préexistants du noyau Linux, afin d'y insérer notamment les points de contrôle supplémentaires, qui ne font pas l'objet d'un *hook* dans l'API LSM standard. Ce support additionnel au sein du noyau est décrit plus en détail en section 5.

2.1 Etiquettes de sécurité

2.1.1 Etiquettes sur les tâches et les exécutables

Le LSM CLIP attribue une étiquette de sécurité à toutes les tâches du système. La structure *clsm_task_sec* correspondante est allouée par le *hook task_alloc_security()* lors de la création d'une tâche (par *fork* ou *clone*), et libérée par le *hook task_free_security()* lors de la terminaison de celle-ci. De même, une étiquette *struct clsm_binprm_sec* est associée à chaque structure *linux_binprm* représentant un exécutable chargé ou en cours de chargement. Les étiquettes *clsm_binprm_sec* reprennent les mêmes champs que les étiquettes *clsm_task_sec*. Elles sont utilisées comme "pivot" lors d'un appel *exec*, afin de propager certaines des informations de la tâche réalisant l'appel à travers le traitement de cet appel, et jusqu'à la tâche résultante. Les champs de ces structures sont :

- Un *bitmask* de statut, de type *clsm_flags_t*, composé des drapeaux décrits dans le Error : Reference source not found. Ces drapeaux permettent de tracer des particularités d'une tâche, par exemple son

niveau de privilège plus élevé, ou son enfermement dans une cage *chroot*.

- Un *bitmask* de privilèges CLSM, de type *clsm_privs_t*, composé des drapeaux décrit dans le Error : Reference source not found. Ces drapeaux correspondent à des privilèges spécifiques au LSM CLIP -- et en particulier distincts des capacités POSIX ou Linux -- qui sont attribués à la tâche porteuse de l'étiquette.
- Un *bitmask* de privilèges CLSM sauvegardés, de type *clsm_privs_t*, utilisés uniquement afin de permettre à un processus *root* d'abandonner temporairement certains de ses privilèges, par un *seteuid()*.
- Un *bitmask* d'options *veriexec*, de type *veriexec_flags_t*, telles que décrites en section 4.1.1 Lorsque la tâche porteuse de l'étiquette résulte de l'exécution d'un fichier auquel est associée une entrée *veriexec* (4.1.2), ce champ est une copie du champ *ve_flags* de l'entrée, dont certains drapeaux conditionnent le traitement ultérieur de la tâche par le sous-système *veriexec* (par exemple les drapeaux *NEEDLIB* ou *SCRIPT*). Dans le cas contraire (pas d'entrée *veriexec*), ce masque est laissé nul.

Par exemple, le type de l'étiquette de tâche est le suivant :

```
struct clsm_task_sec {
  clsm_flags_t t_flags;/* statut de la tâche */
  clsm_privs_t t_privs ;/* privilèges CLSM de la tâche */
  clsm_privs_t t_sprivs ;/* privilèges sauvegardés */
  veriexec_flags_t t_vflags ;/* options veriexec de la tâche */
};
```

Ces deux types de structures, *clsm_task_sec* et *clsm_binprm_sec*, sont allouées dans un cache *kmem* dédié, afin d'optimiser les opérations d'allocation/désallocation, et d'éviter le gaspillage et la fragmentation de l'allocateur *slab* principal. En contrepartie, ces deux structures doivent être exactement de la même taille.

Lors d'un appel *exec*, certains drapeaux des masques de statut et de privilèges de la tâche appelante sont copiés dans l'étiquette de sécurité de la *struct linux_binprm* associée au fichier exécuté, mais les options *veriexec* ne sont pas propagées. Lors d'un appel *fork* ou *clone*, les masques de statut et de privilèges sont aussi partiellement propagés, tandis que les options *veriexec* sont cette fois entièrement recopiées. Par ailleurs, l'étiquette d'une tâche est modifiée lors d'une perte de privilèges, soit par *setuid*, soit par changement de contexte *vserver*. En résumé, ces transformations sont les suivantes :

- Lors d'un *exec* (*clsm_binprm_sec* en fonction du *clsm_task_sec* de l'appelant - l'étiquette *clsm_task_sec* de la tâche en sortie d'*exec* est une copie directe de l'étiquette *clsm_binprm_sec*, mais après que celle-ci ait été modifiée par le sous-système *veriexec*.) :

```
b_flags = t_flags & CLSM_FLAGS_COPY_EXEC ;
b_privs = t_privs & CLSM_PRIVS_COPY_EXEC ;
b_vflags = 0 ;
```

- Lors d'un *fork/clone* :

```
t_fils_flags = t_pere_flags & CLSM_FLAGS_COPY_CLONE ;
t_fils_privs = t_pere_privs & CLSM_PRIVS_COPY_CLONE ;
t_fils_vflags = t_pere_vflags ;
```

- Lors d'une perte de privilège par *setuid* (c'est-à-dire lors d'un changement d'identité quelconque entre utilisateurs non privilégiés, ou lors d'un changement d'identité de *root* vers un utilisateur non privilégié, avec abandon de toutes les identités *root* : *euid, uid* et *suid*. Ces cas correspondent également aux changements d'identité entraînant la perte de toutes les capacités POSIX et Linux.) :

```
t_flags &= ~CLSM_FLAGS_DROP_SETUID ;/* + réévaluation privs */
t_privs &= ~CLSM_PRIVS_DROP_SETUID ;
/* t_vflags inchangé */}
```

- Lors d'un changement de contexte *vserver* :

```
t_flags \&= ~CLSM_FLAGS_DROP_CHCTX ;/* + réévaluation privs */
t_privs \&= ~CLSM_PRIVS_DROP_CHCTX ;
/* t_vflags inchangé */
```

Par ailleurs, lors d'un changement d'identité de *root* vers un *uid* effectif non-*root* quelconque, sans perte de toutes les identités *root* (*uid* ou *suid* reste nul), certains privilèges CLSM -- définis par le masque *CLSM_PRIVS_ROOT_MASK* -- sont sauvegardés dans le *bitmask* de sauvegarde *sprivs*, et effacés du *bitmask* effectif *privs*. Lors d'un rétablissement de l'identité effective *root*, ces mêmes privilèges sont rétablis dans le masque effectif. Ce fonctionnement est largement comparable à celui déjà en place pour les capacités : lors d'un changement vers un *euid* non *root*, le masque de capacités effectif est mis à zéro, tandis que lors d'une transition vers un *euid root*, le masque de capacités permis est recopié dans le masque effectif.

Drapeau	Signification
CLSM_FLAG_RAISED	La tâche dispose de capacités permises ou effectives supérieures à celles attribuées par défaut à son identité.
CLSM_FLAG_BUMPED	La tâche dispose de privilèges CLSM normalement réservés à *root*, c'est-à-dire compris dans le masque *CLSM_PRIVS_ROOT_MASK*. Ce drapeau reste présent lorsque la tâche ne possède ces privilèges que dans son masque de privilèges sauvegardés.
CLSM_FLAG_INHERITED	La tâche dispose d'une capacité héritable supérieure à celle attribuée par défaut à son identité, suite à une affectation forcée lors de son exécution, ou à la transmission du masque héritable de son père.
CLSM_FLAG_CHROOTED	La tâche est enfermée dans une cage *chroot*.
CLSM_FLAG_MIGRATED	La tâche a effectué un changement de contexte *vserver*, et n'a pas réalisé d'appel *exec()* depuis cette migration. Ce drapeau est typiquement porté par *vsctl* lors d'une opération *vsctl setup* ou *vsctl stop* (cf. [CLIP 1202]).

Table 1 – Drapeaux de statut CLSM

2.1.2 Etiquettes sur les fichiers

Le LSM CLIP fait aussi appel à des étiquettes de sécurité associées aux fichiers (au sens *struct file*), de type *struct clsm_file_sec*. A la différence des étiquettes associées aux tâches et exécutables, ces étiquettes de fichiers ne sont pas systématiquement associées à tout fichier manipulé par le noyau. A ce stade, une telle structure n'est en effet allouée que lors d'un appel *fcntl(... F_SETOWN ...)*, ou *fcntl(... F_SETSIG ...)*, sur un fichier, permettant à une tâche de désigner, respectivement, le *pid* du processus devant recevoir les signaux d'entrées/sorties asynchrones liés à ce fichier, et le numéro de signal envoyé lors de cette signalisation asynchrone. L'allocation est donc faite soit par le *hook file_set_fowner()*, soit par le *hook* -- spécifique au LSM CLIP car non inclus dans l'API LSM standard -- *file_set_fsignum()*, plutôt que par *file_alloc_security()*. La désallocation se fait quant à elle de manière classique dans le *hook file_free_security()*. Ces opérations font appel à un cache *kmem* dédié à ces structures *clsm_file_sec*.

La structure *clsm_file_sec* comprend les champs suivants :

```
struct clsm_file_sec {
    clsm_flags_t f_flags; /* statut CLSM */
    kernel_cap_t f_eff; /* masque de capacités effectif */
    kernel_cap_t f_perm; /* masque de capacités permis */
    kernel_cap_t f_inh ; /* masque de capacités héritable */
    unsigned long  f_xid ; /* xid vserver */
};
```

Une telle structure permet ainsi de sauvegarder les privilèges [2] et les éventuelles propriétés CLSM et *vserver* (cf. [CLIP 1202]) propres à la tâche ayant réalisé, pour un fichier donné, le premier appel *fcntl* de définition d'un *FOWNER* ou du numéro de signal associé. Tout appel ultérieur, destiné à modifier ces informations, nécessite pour la tâche appelante des privilèges supérieurs ou égaux à ceux sauvegardés [3]. Lorsqu'un signal d'entrée/sortie asynchrone est délivré au processus *FOWNER* d'un fichier, les informations sauvegardées dans l'étiquette de sécurité du fichier sont utilisées pour décider si l'envoi de signal est autorisé, selon des conditions similaires à l'envoi direct de signal d'une tâche à une autre (cf. 2.2).

2.1.3 Etiquettes sur les *inodes*

Des étiquettes de sécurité peuvent par ailleurs être ajoutées aux *struct inode*. La seule utilisation de ces étiquettes à ce stade correspond à la mise en cache des vérifications *veriexec*, activée par l'option de configuration *CONFIG_VERIEXEC_CACHE* (cf. 4.1.4). A l'instar des étiquettes sur les fichiers, celles-ci ne sont allouées que lorsqu'elles sont nécessaire, c'est-à-dire à la suite d'une vérification *veriexec* réussie. La désallocation est en revanche réalisée de manière classique par le *hook* LSM *inode_free_security()*, qui ne réalise naturellement cette désallocation que dans le cas où l'étiquette a été préalablement allouée.

La structure étiquette est limitée à un champ *bitmap* unique, comprenant à ce stade un unique drapeau significatif : *CLSM_IFLAG_CACHED*, qui indique une vérification *veriexec* mise en cache. La définition de cette structure est la suivante :

```
struct clsm_inode_sec {
    int i_flags;
};
```

2. Les privilèges CLSM ne sont pas sauvegardés à ce stade, dans la mesure où aucun privilège CLSM spécifique n'est associé à la gestion des signaux d'entrée/sortie asynchrone. Seul le statut *CLSM_FLAGS_BUMPED* d'une tâche est pris en compte.

3. Plus précisément, les privilèges nécessaires sont :
Soit posséder la capacité CAP_KILL et le drapeau CLSM_FLAGS_RAISED
Soit posséder la capacité CAP_KILL lorsque le statut CLSM sauvegardé ne comporte par CLSM_FLAGS_RAISED
Soit posséder un masque de capacités effectif supérieur ou égal (bit à bit) au masque effectif sauvegardé, et un masque héritable supérieur ou égal au masque héritable sauvegardé.

2.2 Gestion des capacités et privilèges CLSM

2.2.1 Gestion des capacités par exécutables

Le LSM CLIP permet la mise en œuvre d'une politique de contrôle d'accès basée sur les capacités POSIX et Linux mais plus fine que la politique par défaut décrite en 1.1, en particulier en ce qui concerne les privilèges accordés à l'utilisateur *root*. Lorsque l'option de compilation *CONFIG_CLSM_ROOTCAPS* a été sélectionnée, il devient en effet possible de réduire le masque de capacité accordé par défaut à *root* lors de l'exécution d'un fichier ("*FULL*" dans la figure 1), et de ne plus attribuer par défaut que certaines capacités (voire éventuellement aucune) à *root*. Ce masque de capacité devient alors une variable, *rootcap*, modifiable par *sysctl*. Afin de permettre l'utilisation du système dans ces conditions plus restrictive, le LSM s'appuie par ailleurs sur le sous-système *veriexec* pour accorder des capacités supplémentaires à certains exécutables uniquement. Ces capacités supplémentaires peuvent être accordées selon deux principes :

- D'une part, un appel à la fonction *veriexec_getcreds()* depuis le *hook bprm_set()* permet au sous-système *veriexec* d'ajouter des capacités aux trois masques associés à un fichier exécutable, et ce avant l'application de la formule standard de transfert des capacités lors d'un *exec()*, telle qu'elle est décrite en figure 1. Ce mécanisme s'apparente à la composante "capacités des fichiers" prévue par le *draft POSIX*, à ceci prêt que les capacités ne constituent pas directement un attribut des fichiers, mais font l'objet d'une description distincte dans la base *veriexec*. Il est ainsi possible de définir des exécutables privilégiés, qui donnent à *root* accès aux capacités masquées dans le *rootcap*, ou bien qui accordent à un autre utilisateur certaines capacités uniquement, en remplacement d'un bit *setuid* placé sur le même exécutable. Cette attribution de capacité peut être directe, ou faire intervenir le mécanisme d'héritage (cf. 4.2).
- D'autre part, le *hook bprm_apply_creds()*, qui réalise le calcul des capacités du processus résultant d'un *exec()*, et intervient de ce fait après le *hook bprm_set()* évoqué ci-dessus, fait lui aussi appel au sous-système *veriexec*, par la fonction *veriexec_tasl_raisecaps()*. Cet appel intervient après le transfert au processus des capacités associées à l'exécutable, selon la formule POSIX standard. Il permet de "forcer" le masque héritable (cf. 4.1.2) de la tâche appelante, masque qui n'est normalement jamais augmenté au cours de la durée de vie d'un processus. Cette copie forcée intervient après le calcul standard des nouveaux masques effectif et permis du processus, et n'affecte donc pas ce calcul. Elle est complétée d'un test spécifique, destiné à éviter le dépassement du *cap-bset*[4] par cette copie directe de masque héritable.

On notera bien que dans les deux cas, et exception faite de la modification peu standard du masque héritable d'un processus dans la deuxième approche, le calcul des capacités du processus en fonction des capacités de l'exécutable respecte la formule POSIX "canonique", telle qu'elle est décrite en figure 1. Ce calcul est réimplémenté spécifiquement dans le *hook bprm_apply_creds()* du LSM CLIP, d'une manière similaire à qui est classiquement réalisé par la fonction *cap_bprm_apply_creds()* (*security/commoncap.c*).

Par ailleurs, ces deux mécanismes d'augmentation des capacités sont l'un comme l'autre inhibés dès lors que la tâche réalisant l'appel *exec* est tracée (par *ptrace*, avec ou sans *CAP_SYS_PTRACE*) ou partage des éléments de son *task_struct* (par exemple descripteurs de fichiers ouverts, suite à un *clone()*). Dans ce cas, aucune capacité héritable ne peut être transmise, et les capacités attribuées au fichier exécutable sont limitées au masque par défaut pour l'identité réalisant l'appel (*rootcap* pour les identités *root* -- *euid* et/ou *uid* selon le type de masque -- et 0 pour les autres identités). Une alarme est enregistrée au journal noyau dans le cas où ces conditions particulières empêcheraient la transmission d'une capacité supplémentaire.

La perte de capacités se fait-elle aussi de manière classique, avec une perte du masque de capacité effectif pour tout changement d'identité effective (sauf retour vers un *euid 0*), et perte du masque permis pour tout changement d'identité quelconque entre deux utilisateurs non-*root*, ou pour un changement d'identité depuis *root*, en abandonnant toutes les identités *root*. Ce mécanisme est complété par une remise à zéro du masque de capacité héritable dans les mêmes conditions que celle du masque permis, ainsi que lors d'un changement

4. Dans le cas d'un noyau par ailleurs patché *vserver*, ce test est réalisé vis-à-vis du *cap-bset* du contexte de sécurité *vserver* courant.

de contexte de sécurité *vserver*.

Les capacités attribués par le biais de vérifications *veriexec* peuvent notamment être substituées au positionnement de *bits `s' (setuid)* sur des exécutables possédés par *root* (cf. 4.2). Cette substitution peut être rendue obligatoire par une option de compilation du LSM CLIP, *CONFIG_CLSM_NOSUID_ROOT*. Lorsque cette option a été activée à la compilation, le noyau ignore silencieusement les *bits `s'* sur des exécutables possédés par *root* (pas d'attribution de capacités, pas de changement d'identité). En revanche, le traitement normal est maintenu lorsque le propriétaire de l'exécutable est autre que *root*, ainsi que pour les *bits `S' (setgid)* indépendamment du propriétaire.

2.2.2 Privilèges CLSM complémentaires

Le modèle de sécurité basé sur les capacités POSIX est par ailleurs complété par un certain nombre de privilèges spécifiques, listés dans le , et qui correspondent selon les cas :

- A des opérations qui ne sont pas considérées comme des privilèges *root*, comme la connexion au réseau (hors ports privilégiés).
- A des opérations spécifiques au LSM CLIP, comme par exemple l'administration de la base *veriexec*.
- A un sous-ensemble des opérations contrôlées par une capacité POSIX existante, qu'il peut être souhaitable d'attribuer à une tâche sans lui attribuer l'ensemble des privilèges accordés par cette capacité.

Ces différents privilèges sont rassemblés dans un masque unique, de type *clsm_privs_t*, stocké dans l'étiquette de sécurité de chaque tâche. Ils sont attribués uniquement lors d'un appel *exec*, par le sous-système *veriexec*, selon le même principe que pour l'attribution des capacités POSIX : le *hook bprm_set()* du LSM fait appel au sous-système *veriexec* par la fonction *veriexec_getcreds()*. Cet appel permet à *veriexec* d'inscrire des privilèges spécifiques dans le champ correspondant de la *struct clsm_binprm_sec* associée à l'exécutable en cours de chargement. Par la suite, ce champ est recopié dans celui associé au processus résultant, par le *hook bprm_apply_creds()*.

Lorsque la tâche réalisant l'appel *exec()* est tracée, ou partage des éléments de son *task_struct*, ce transfert de privilèges est limité aux privilèges non compris dans le masque *CLSM_PRIVS_ROOT_MASK*, c'est à dire aux privilèges qui ne correspondent pas à des opérations réservées à *root* dans un noyau Linux classique. Une alerte est enregistrée au journal noyau lorsqu'un privilège "*root*" n'est pas transmis à une tâche du fait de telles conditions.

La perte des privilèges CLSM se fait selon des conditions similaires à celle des capacités : masquage de certains privilèges lors d'un changement d'identité effective (cf. 2.1.1), et révocation définitive de certains [5] privilèges lors d'un changement complet d'identité ou d'un changement de contexte *vserver*.

5. Définis par les masques *CLSM_PRIVS_DROP_SETUID* et *CLSM_PRIVS_DROP_CHCTX*, respectivement.

Privilège	Signification
CLSM_PRIV_CHROOT	Permet le contournement des vérifications imposées par *CONFIG_CLSM_CHROOT*.
CLSM_PRIV_VERICTL	Nécessaire pour réaliser des opérations d'administration de *veriexec* depuis un contexte dans lequel il est actif. *NB :* la capacité POSIX *CAP_SYS_ADMIN* est aussi nécessaire dans le cas d'opérations réalisées depuis le contexte ADMIN.
CLSM_PRIV_NETCLIENT	Permet la création de sockets non-locales et l'appel *connect()* sur une telle socket. Aussi nécessaire pour réaliser un appel *sendmsg()* sur une socket non-locale dans un état non connecté (cas des clients UDP non connectés).
CLSM_PRIV_NETSERVER	Permet la création de sockets non-locales et les appels *bind()*, *listen()* et *accept()* sur une telle socket.
CLSM_PRIV_NETOTHER	Permet la création de sockets non-locales, sans autoriser pour autant les différents appels nécessaires à l'établissement d'une connexion distante. Ce privilège permet ainsi uniquement des opérations de type *setsockopt()* ou *ioctl()*.
CLSM_PRIV_PROCFD	Permet l'accès aux descripteurs de fichiers ouverts d'une tâche exécutée sous une autre identité ou avec un niveau de privilège différent, à travers le répertoire */proc/<pid>/fd* et le fichier */proc/<pid>/maps*, sans disposer de la capacité *CAP_SYS_PTRACE*. Utilisé principalement par l'utilitaire *fuser*.
CLSM_PRIV_SIGUSR	Permet à une tâche d'un contexte *vserver* non-ADMIN d'envoyer un signal *SIGUSR1* ou *SIGUSR2* à une tâche du contexte ADMIN. Ce privilège est accordé à *Xorg*, afin de lui permettre de dialoguer avec *XDM*.
CLSM_PRIV_RECVSIG	Permet à une tâche du contexte AUDIT de recevoir les signaux *SIGUSR1 et SIGUSR2* envoyés par une tâche d'un contexte non-ADMIN possédant le privilège *CLSM_PRIV_SIGUSR*.
CLSM_PRIV_NETLINK	Permet la création de sockets de la famille *AF_NETLINK*. On notera que les autres opérations sur une telle socket ne font l'objet d'aucun contrôle spécifique par le LSM CLIP.
CLSM_PRIV_KSYSLOG	Donne accès au journaux noyau sans *CAP_SYS_ADMIN* et sans nécessairement appartenir au contexte *vserver* ADMIN.

TABLE 2 – Privilèges CLSM

Privilège	Signification
CLSM_PRIV_IMMORTAL	Protège contre la réception de signaux envoyés par toute tâche qui ne possède pas *CAP_SYS_BOOT* dans son masque héritable, et réduit sensiblement les risques d'être tué par le *OOM (Out Of Memory) killer* du noyau.
CLSM_PRIV_KEEPPRIV	Permet de conserver ses privilèges CLSM (mais pas ses capacités, qui font l'objet d'un traitement spécifique par *prctl*) lors d'une perte de l'identité *root* ou d'un changement de contexte *veriexec*.
CLSM_PRIV_XFRMSP	Permet de créer, ou supprimer des politiques de sécurité IPsec (SP).
CLSM_PRIV_XFRMSA	Permet de créer, mettre à jour ou supprimer des associations de sécurité IPsec (SA).

TABLE 3 – Privilèges CLSM (suite)

2.2.3 Gestion de niveaux de privilèges hétérogènes

De manière complémentaire aux mécanismes décrits plus haut d'attribution de privilèges exécutable par exécutable, le LSM CLIP redéfinit l'essentiel des *hooks* de contrôle d'accès sur lequel se base le modèle "canonique" des capacités POSIX et Linux, et les complète par des *hooks* spécifiques. Ces adaptations sont nécessaires pour tenir compte des spécificités du modèle de sécurité associé aux exécutables privilégiés mis en œuvre dans CLIP. En effet, le modèle de sécurité Linux classique repose essentiellement sur la distinction entre identités : les identités *root* disposent de privilèges *root* (effectifs et/ou autorisés, selon qu'il s'agisse d'une identité effective, réelle ou sauvegardée), généralement sous la forme d'un masque de capacité "FULL", tandis que les identités non-*root* ne disposent d'aucun privilège. En particulier, un unique niveau de privilège peut être associé à chaque identité, et les seules possibilités d'escalade de privilèges sont liées aux identités mixtes (identités réelle et effective différentes, du fait typiquement de la présence d'un bit `s' sur un exécutable). De ce fait, et malgré les possibilités plus fines offertes par le modèle de capacités, plusieurs fonctions de contrôle d'accès au sein d'un système Linux standard se réfèrent uniquement aux identités pour leur prise de décision.

Contrairement à ce modèle simple, le LSM CLIP introduit -- tout comme dans un modèle de capacités par fichiers plus "canonique" - une hétérogénéité dans les niveaux de privilèges mis à la disposition d'une identité donnée, selon l'exécutable qu'elle invoque. Un utilisateur non privilégié peut ainsi mettre en œuvre certaines capacités sans bénéficier d'un changement d'identité effective, tandis que certains processus *root* sont seuls à disposer des capacités les plus "lourdes", qui ne sont par défaut pas attribuées même à cet utilisateur privilégié. Il est donc nécessaire de contrôler plus finement les interactions entre processus s'exécutant sous une même identité mais avec des privilèges différents, en généralisant et en étendant les protections classiquement offertes aux exécutables *setuid*. Ces protections s'appuient sur des comparaisons de masques de capacités, ainsi que sur deux "attributs" supplémentaire des tâches :

- Une tâche est dite *"raised"* dès lors qu'elle dispose de privilèges supplémentaires par rapport à son identité, c'est-à-dire des masques de capacités effectif ou permis augmentés (drapeau de tâche *CLSM_FLAG_RAISED* -- qui ne peut résulter que d'un traitement *veriexec* lors de l'*exec* à l'origine de la tâche), ou une capacité héritable augmentée (*CLSM_FLAG_INHERITED* -- cette capacité peut avoir été donnée directement par *veriexec*, ou avoir été transmise par le père de la tâche), ou encore un privilège appartenant au masque *CLSM_PRIVS_ROOT_MASK* (drapeau *CLSM_FLAG_BUMPED*).
- Une tâche est dite *"capraised"* si ses masques de capacités effectif ou permis ont été augmentés. Cette propriété traduit un niveau de privilège généralement plus fort que *"raised"*, en particulier dans la mesure où elle ne peut pas avoir été simplement héritée du père de la tâche (ou de la tâche ayant réalisé l'appel *exec()*).

De manière générale, le LSM CLIP exige d'une tâche qu'elle soit *"capraised"* pour pouvoir affecter l'état d'une autre tâche *"raised"*, ou, alternativement, qu'elle dispose de capacités permises et héritables au moins égales à celle de la tâche cible.

Envoi de signaux

Ce principe est appliqué en particulier à l'envoi de signaux. Outre les restrictions imposées par le noyau Linux standard [6], le *hook* LSM *task_kill()* impose pour l'envoi d'un signal quelconque d'une tâche P à une tâche P' la satisfaction d'au moins une des conditions suivantes :

(a) P dispose de *CAP_KILL* et P' n'est pas « *raised* »,

(b) P dispose de *CAP_KILL* et est « *capraised* »,

(c) Les masques de capacités effectif et héritable de P dominent les masques permis et héritable, respectivement, de P.

6. Le noyau Linux impose à une tâche de posséder la capacité *CAP_KILL* pour envoyer un signal à une autre tâche dès lors que ni l'*euid* ni l'*uid* de la tâche d'origine ne sont égaux à l'*uid* ou au *suid* de la tâche cible (une seule égalité suffit), à l'exception de l'envoi du signal *SIGCONT*, qui est autorisé vers toute tâche de la même session. Ces restrictions sont appliquées en amont de l'appel au *hook* LSM, dans *kernel/signal.c :check_kill_permission()*.

L'envoi de signal est de plus naturellement conditionné par l'appartenance à un contexte de sécurité donné. Dans un noyau "patché" CLIP LSM, cette appartenance est testée dans le *hook* LSM lui-même plutôt que directement dans la fonction *kernel/signal.c :check_kill_permission()* qui fait appel à ce *hook*, afin de permettre la prise en compte du privilège *CLSM_PRIV_SIGUSR*, qui permet l'envoi de signaux *SIGUSR1/2* vers une tâche du contexte *vserver* ADMIN depuis un contexte non-ADMIN.

Ce contrôle des envois de signaux est complété par un contrôle des envois indirects en utilisant la signalisation des entrées-sorties asynchrones sur un fichier. Lorsqu'une tâche spécifie le numéro de signal à envoyer suite à un événement asynchrone sur un fichier (typiquement, possibilité d'une lecture ou écriture sur le fichier), ou le *pid* de la tâche à qui envoyer ce signal, les capacités, le statut CLSM et le *xid vserver* de la tâche appelante sont stockées dans l'étiquette de sécurité du fichier (cf. 2.1.2). Toute modification ultérieure de ces éléments nécessite des privilèges au moins équivalent à ceux utilisés lors de la première modification, et ne peut être réalisée que depuis le contexte *vserver* correspondant au *xid* stocké dans l'étiquette de sécurité. Lors de l'envoi effectif du signal suite à un événement sur le fichier, les mêmes trois conditions sont évaluées que pour un envoi de signal direct de tâche à tâche, en prenant pour P' la tâche destinataire du signal, si elle existe, et pour P la tâche à l'origine de la première modification du numéro ou du *pid* destinataire du signal (qui est par construction la tâche la moins privilégiée ayant pu modifier ces éléments). De plus, les numéros de contexte *vserver* sont aussi évalués avant l'envoi du signal, de manière à éviter un possible contournement du cloisonnement des signaux par *vserver*. Cette prise en compte des signaux d'entrée/sortie sur les fichiers repose sur les *hooks* LSM *file_set_fowner()* (enregistrement du *pid* destinataire), *file_send_sigiotask()* (envoi du signal), et sur un *hook* spécifique, *file_fsignum()*, pour la prise en compte des modifications du numéro de signal à envoyer.

Modifications de priorité

Un principe similaire est adopté pour contrôler les modifications de priorité de tâche à tâche. Les trois conditions évaluées sont équivalentes à celles mentionnées pour l'envoi de signaux, à ceci près que *CAP_SYS_NICE* est substituée à *CAP_KILL*. Ces restrictions sont imposées indifféremment pour le changement de priorité statique d'ordonnancement (*"nice"*, avec le *hook* LSM *task_setnice()*), pour le changement d'ordonnanceur (*task_setscheduler()*) ou pour le changement de priorité d'entrée/sortie (*task_setioprio()*).

Traçage par *ptrace*

Le traçage des processus par *ptrace* est en revanche plus strictement contrôlé, dans la mesure où il permet un contrôle direct de la tâche tracée. De ce fait, le LSM CLIP interdit toute acquisition de capacités supplémentaires ou de privilèges CLSM "root" par *ptrace*, en ne testant pas l'équivalent de la condition (b) testée pour les envois de signaux, et en ajoutant un test de privilèges à la condition (c). Ainsi, pour tracer une tâche P', une tâche P doit satisfaire l'une des deux conditions suivantes :

(i) P dispose de *CAP_SYS_PTRACE*, et P' n'est pas *" raised"*

(ii) Les masques de capacités effectif et héritable de P dominent respectivement les masques permis et héritable de P', et P possède tous les privilèges CLSM de P' appartenant au masque *CLSM_PRIVS_ROOT_MASK*.

Ces conditions sont évaluées par le *hook* LSM *ptrace()*, qui se charge aussi éventuellement d'interdire toute activité *ptrace* à un processus enfermé dans une cage *chroot* (cf. 2.5).

Cette interdiction dynamique de l'attachement *ptrace* est par ailleurs complétée par l'interdiction pour une tâche déjà tracée (y compris par une tâche disposant de CAP_SYS_PTRACE) d'obtenir lors d'un exec() des capacités ou des privilèges supplémentaires par veriexec, ou de conserver des capacités héritables supplémentaires à travers un appel *exec()*. On notera que ces restrictions, mises en œuvre par le *hook bprm_apply_creds()*, s'appliquent aussi aux tâches partageant des parties de leur *task_struct* suite à un *clone()* partiel.

Possibilité de *"dumper"* une tâche

Le LSM CLIP interdit de plus la génération d'un *core dump*[7] pour toute tâche *"raised"*. Cet interdiction est réalisée, d'une part, lors d'un appel *exec()*, par le positionnement de *setuid_dumpable* dans le champ *dumpable* de la tâche courante (*hook bprm_apply_creds()*), et d'autre part, par le blocage des appels *prctl(PR_SET_DUMPABLE)* pour toute tâche *"raised"* (*hook task_prctl()*).

Chargement de bibliothèques

Afin de limiter les possibilités d'escalade de privilèges par injection de code lors de l'édition de liens au chargement d'un exécutable privilégié, le *hook bprm_securexec()* déclenche le positionnement de la variable *AT_SECURE* dans le vecteur auxiliaire passé par le noyau à l'éditeur de liens ELF, dès lors que l'exécutable chargé (représenté par sa *struct linux_binprm*) dispose de capacités augmentées (*CLSM_FLAGS_RAISED* ou *CLSM_FLAGS_INHERITED*) après son traitement par le sous-système *veriexec*. La variable *AT_SECURE* entraîne la mise en œuvre du mode *__libc_enable_secure* par l'interpréteur ELF de la *glibc*. Dans ce mode, un certain nombre de variables d'environnement spécifiques à la *glibc* ne sont pas prises en compte par l'interpréteur, en particulier les variables *LD_PRELOAD, LD_LIBRARY_PATH* et *LD_LIBRARY_ORIGIN* qui permettent de détourner les mécanismes de chargement de bibliothèques dynamiques.

En complément de ce durcissement de l'interpréteur ELF, le sous-système *veriexec* peut spécifiquement positionner l'un des drapeaux *VRX_FLAG_NEEDLIB* ou *VRX_FLAG_CHECKLIB* (cf. tableau 6) lors du chargement d'un exécutable donné. Ce drapeau est transmis de l'étiquette de sécurité attachée à la *struct linux_binprm* associée à l'exécutable vers l'étiquette de la tâche résultante. Sa présence entraine le rappel par le module LSM du sous-système *veriexec*, cette fois-ci par la fonction *veriexec_updatecreds()*, dans les conditions suivantes :

- Chargement de l'interpréteur associé au format de l'exécutable, en passant en argument de l'appel *veriexec* le *struct file* correspondant à l'interpréteur (par exemple */lib/ld-linux.so*). Cet appel repose sur un *hook* spécifique au LSM CLIP, *file_interpreter()*.
- Projection en exécution d'un fichier dans la mémoire du processus (typiquement, lors d'un chargement de bibliothèque dynamique), en passant en argument le fichier projeté. Cet appel repose sur un *hook* spécifique au LSM CLIP[8] *file_mmap_exec()*.
- Modification des droits associés à une projection en mémoire, pour conférer des droits en exécution à la projection, en passant en argument le fichier projeté (s'il existe). Cet appel est réalisé par un *hook* spécifique, *file_mprotect_exec()*.

Ces différents appels permettent de forcer la vérification par *veriexec* de tout code exécutable chargé dans la mémoire d'un processus particulièrement privilégié. Le sous-système *veriexec* se charge en cas d'échec de la vérification de supprimer les privilèges spéciaux accordés au processus[9], tandis que le module LSM interdit la projection en mémoire. La différence entre les deux drapeaux, *NEEDLIB* et *CHECKLIB*, tient à ce que le premier entraîne une simple vérification de présence dans la base d'entrées *veriexec*, tandis que le second y associe une vérification de l'empreinte cryptographique du fichier projeté, avec un fort impact sur les performances car le principe de chargement en mémoire à la demande, généralement mis en œuvre par le noyau Linux, perd dans ce cas toute son utilité.

Accès en écriture aux bibliothèques actives

7. Il est rappelé que cette interdiction relève uniquement de la défense en profondeur dans un système CLIP, dans la mesure où la génération de *core dump* n'est elle-même pas incluse dans le noyau CLIP.

8. Cet appel aurait pu être réalisé à l'aide du *hook* LSM standard, *file_mmap()*, mais celui-ci exposait à une *race condition* potentielle dans la mesure où il est appelé avant l'interdiction des accès en écriture à un fichier projeté en exécution. Ainsi, un attaquant aurait éventuellement pu modifier une bibliothèque entre sa vérification par le sous-système *veriexec* et sa projection effective dans la mémoire d'un processus privilégié.

9. Un simple échec du chargement pourrait être jugé suffisant, mais la réalisation d'une telle tentative de chargement illicite est interprétée par le LSM comme le signe que le processus en question fait l'objet d'une tentative d'attaque, et qu'il est donc préférable de révoquer ses privilèges immédiatement. Cependant, cette révocation ne prend pas en compte les autres types de "capacités", par exemple descripteurs de fichiers ouverts, qui auraient pu être obtenus précédemment par le processus.

Le noyau Linux interdit par défaut tout accès en écriture [10] aux fichiers exécutables actifs (en cours d'exécution), dans la mesure où ces fichiers servent de *swap* pour les sections en lecture seule, en particulier *.text*, des espaces mémoires des processus en train de les exécuter. En revanche, aucune protection similaire n'est assurée pour les fichiers simplement projetés en exécution dans la mémoire d'un processus existant, par un appel *mmap(… PROT_EXEC…)* ou *mprotect(… PROT_EXEC…)*. Le noyau Linux ignore silencieusement [11] le drapeau *MAP_DENYWRITE* passé par l'éditeur de lien dynamique lors de la projection d'une section de code de bibliothèque. En effet, dans la mesure où aucun contrôle d'accès spécifique n'est associé à ce drapeau, sa prise en compte offrirait un fort potentiel de déni de service, car elle permettrait d'interdire tout écriture sur un fichier (par exemple un fichier de journaux, ou */var/log/wtmp*) à la seule condition de disposer de droits discrétionnaires en lecture sur ce fichier.

Cette absence de contrôle d'accès en écriture permet théoriquement d'injecter du code arbitraire dans un processus en cours d'exécution. Il suffit pour cela de modifier sur le disque une des bibliothèques utilisées par ce processus, en s'assurant au préalable que la ou les pages mémoire correspondant à la zone modifiée ne sont pas présentes en mémoire physique (soit parce qu'elles n'ont pas encore été chargées, soit parce qu'elles ont été *swappées*). Après cette modification, le prochain accès par le processus aux pages mémoires concernées déclenchera le chargement depuis le disque et l'exécution de la version modifiée du code. Une telle incohérence n'est pas réellement critique dans le modèle de sécurité habituel de Linux (si l'attaquant dispose des droits en écriture sur les bibliothèques, il peut tout aussi facilement, dans la plupart des cas, les modifier avant leur chargement par le processus visé). En revanche, elle doit être prise en compte de manière spécifique dans le cadre du LSM-CLIP, dans la mesure où certains privilèges ne sont accordés qu'après vérification de l'intégrité des exécutables et bibliothèques. Cette vérification d'intégrité n'ayant lieu que lors de la projection initiale des fichiers en mémoire, elle pourrait être contournée par une modification de ces fichiers après leur vérification.

De ce fait, le patch CLIP-LSM attribue automatiquement le drapeau *MAP_DENYWRITE* à toute projection exécutable, et réalise explicitement l'appel *deny_write_access()* associé lorsque cela est nécessaire [12]. Afin d'éviter les dénis de service, un contrôle d'accès supplémentaire est réalisé, exigeant, pour toute projection exécutable, que l'appelant dispose de droits discrétionnaires en exécution sur le fichier projeté. La projection échoue, en revoyant l'erreur *--ETXTBUSY*, dès lors que les accès en écriture ne peuvent pas être interdits du fait d'écritures en cours. Les accès en écriture et en exécution sont synchronisés par le *spin_lock i_lock* de chaque *inode*, qui protège les accès au compteur d'écrivains *i_writecount*. Les écritures sont automatiquement autorisées à nouveau dès que la dernière projection exécutable d'un fichier est supprimée. On notera de plus que les fonctions de vérification *veriexec* sont appelées systématiquement après l'interdiction des accès en écriture aux fichiers vérifiés, afin d'éviter les *race conditions*.

2.3 Contrôle d'accès réseau

2.3.1 Contrôle des opérations sur les *sockets* non locales

Le LSM CLIP offre un mécanisme de contrôle des accès au réseau. Celui-ci est activable par l'option de compilation *CONFIG_CLSM_NET*, qui sélectionne automatiquement les *hooks* LSM réseau sur lesquels s'appuie le contrôle. Lorsque cette option est activée, les principales opérations sur les sockets font l'objet d'un contrôle spécifique, associé à plusieurs privilèges LSM. Les sockets de la famille AF_UNIX, qui sont purement locales, ne

10. "Accès en écriture" étant entendu comme un accès en écriture à l'*inode* correspond, tel qu'il pourrait être obtenu par un *open(...O_RDWR…)*. Il est en revanche toujours possible de réaliser un *unlink* du fichier, puis de créer un nouveau fichier du même nom. Une telle action ne perturbe pas les processus en train d'exécuter l'*inode* d'origine, qui conservent une référence vers ce dernier. Elle permet par ailleurs la mise à jour des exécutables sans conditions particulières sur les processus en cours d'exécution.

11. Plus précisément, ce drapeau est supprimé des options de l'appel système *sys_mmap()*. Il est en revanche pris en compte par les fonctions internes du noyau, en particulier *mm/mmap.c/do_mmap_pgoff()*, ce qui permet au noyau de protéger en écriture les sections de code qu'il projette lui-même dans l'espace mémoire d'un processus, en particulier la section de code de l'interpréteur associé à ce processus.

12. C'est-à-dire dans le code de *mprotect()*, mais pas dans celui de *mmap()*, qui prend automatiquement en compte le drapeau *MAP_DENYWRITE* dès lors qu'il est passé par le noyau et non par un processus utilisateur.

sont pas contrôlées. De même, les opérations sur les socket *netlink*, elles-aussi locales, ne font pas l'objet d'un contrôle spécifique, sauf lors de la création de socket, à laquelle un privilège CLSM spécifique est associé : *CLSM_PRIV_NETLINK*. Les opérations sur les autres familles de sockets sont regroupées dans trois profils d'emploi, associés chacun à un privilège CLSM.

Profil "client" (*CLSM_PRIV_NETCLIENT*)

Ce profil permet des opérations de type client réseau. Le privilège associé est exigé pour les opérations suivantes :

- *connect()* d'une socket (client en mode *STREAM*)
- *sendmsg()* sur une socket en état non-connecté (client en mode *DATAGRAM*). On notera que même pour le mode *DATAGRAM*, un serveur correspond à une socket dans l'état connecté (*SS_CONNECTED*), suite au *bind()* de la socket.

Par ailleurs, le privilège *CLSM_PRIV_NETCLIENT* autorise, au même titre que les deux autres profils, la création de sockets.

Profil "serveur" (*CLSM_PRIV_NETSERVER*)

Ce profil correspond aux opérations de type serveur réseau. Le privilège associé est exigé pour les opérations suivantes :

- *bind()* d'une socket
- *listen()* sur une socket
- *accept()* sur une socket

Par ailleurs, le privilège *CLSM_PRIV_NETSERVER* autorise, au même titre que les deux autres profils, la création de sockets.

Profil "autre" (*CLSM_PRIV_NETOTHER*)

Ce profil correspond à l'utilisation d'une socket non-locale sans dialogue réel sur le réseau. Il permet la création d'une telle socket, mais pas son utilisation, ni en mode client, ni en mode serveur. A défaut des deux autres privilèges réseaux, *CLSM_PRIV_NETOTHER* permet la création de socket, ainsi que toutes les opérations qui ne sont pas contrôlées explicitement par le LSM, en particulier les *get/setsockopt()* et les *ioctl()*.

Ces différents contrôles sont activés par défaut au démarrage. L'option de compilation *CONFIG_CLSM_NET_DEVEL* active le mode "développement" de ce mécanisme, dans lequel les vérifications de privilège (y compris celle portant sur les sockets *netlink*) peuvent être désactivées par *sysctl* (cf. 2.6.2).

Une option supplémentaire, *CONFIG_CLSM_NET_FULL*, permet de rendre ce contrôle plus strict, en l'étendant aux opérations élémentaires d'envoi et de réception de paquets. Lorsque cette option est activée, les *hooks* LSM *socket_sendmsg()* et *socket_recvmsg()* n'autorisent ces opérations sur des sockets non locales qu'aux tâches disposant de l'un des trois privilèges d'accès au réseau, *CLSM_PRIV_NETOTHER*, *CLSM_PRIV_NETCLIENT* ou *CLSM_PRIV_NETSERVER*. Aucune distinction n'est opérée à ce niveau entre ces trois privilèges, sauf pour le cas d'un *sendmsg()* sur une socket non connectée, qui reste réservé au profil client. Le principal apport de ces contrôles supplémentaires et d'interdire la transmission d'un accès au réseau au travers d'un appel *exec()*, par transmission d'un descripteur de fichier sur une socket connectée [13](cas typique d'un *remote shell* exécuté après compromission du flot de contrôle d'un exécutable connecté au réseau).

13. Ce qui correspond en particulier au cas typique d'un *remote shell* exécuté après compromission du flot de contrôle d'un exécutable connecté au réseau. Dans ce cas, *CONFIG_CLSM_NET_FULL* interdit le lancement (par exemple suite à une attaque de type *return to libc*) de l'exécutable */bin/sh* en conservant pour ses entrées/sorties une socket créée par le processus corrompu. La création d'un *remote shell* demeure possible en injectant directement le code de ce dernier dans l'espace mémoire du processus corrompu (sans appel *exec*), mais une telle approche est généralement interdite par *PaX*.

2.3.2 Contrôle des opérations sur les *sockets Netlink*

2.3.3 Contrôle du paramétrage IPsec

Par ailleurs, le LSM CLIP réalise un contrôle spécifique des accès en écriture aux bases de politiques et d'associations de sécurité IPsec (SPD et SAD, respectivement). Outre la capacité *CAP_NET_ADMIN*, qui est exigée pour tout accès en écriture à ces bases dans le noyau Linux standard, le LSM CLIP exige un privilège supplémentaire dans chaque cas :

- *CLSM_PRIV_XFRMSP* est exigé pour tout ajout ou suppression de politique dans la SPD.
- *CLSM_PRIV_XFRMSA* est exigé pour tout ajout, mise à jour ou suppression d'association dans la SAD.

Il devient ainsi possible de limiter fortement l'accès à ces bases critiques pour la sécurité, en l'interdisant en particulier aux exécutables qui disposent de *CAP_NET_ADMIN* sans besoin légitime d'accès à la configuration IPsec, par exemple à des fins de modification des adresses IP (cas d'un client *dhcp*) ou de création de cages *vserver*. L'utilisation de deux privilèges distincts permet de plus de distinguer les exécutables capables de modifier la SPD de ceux modifiant la SAD. Dans le cas d'emploi typique d'un système CLIP, le démon *iked* se voit ainsi privé de la possibilité de modifier directement les politiques de sécurité.

Ce contrôle d'accès à la configuration IPsec est actif dès l'initialisation du LSM, dès lors que celui-ci a été compilé avec l'option *CONFIG_SECURITY_NETWORK_XFRM*. Il repose sur l'utilisation de deux *hooks* standards de l'interface LSM, *xfrm_policy_delete_security()* et *xfrm_state_delete_security()* pour le contrôle des suppressions, et sur deux *hooks* spécifiques, *xfm_policy_add()* et *xfrm_state_add()*, pour le contrôle des ajouts et mises à jour.

2.4 Gestion des montages VFS

Le LSM CLIP offre aussi des mécanismes de contrôle des montages *VFS*, qui s'ajoutent au contrôle standard du niveau de privilège (capacité *CAP_SYS_ADMIN*), et sont activés par l'option de compilation *CONFIG_CLSM_MOUNT*. Ces mécanismes n'introduisent pas de nouveaux privilèges, mais imposent des restrictions supplémentaires aux opérations de montage, restrictions qui s'appliquent indépendamment du niveau de privilège de l'appelant. Trois types d'opérations sont contrôlées à ce stade : les nouveaux montages, les montages de type *bind*, et les remontages de montages existants. Les vérifications sont désactivées au démarrage, afin de ne pas interférer avec le montage initial des partitions de base du système ; elles peuvent ensuite être activées par un *sysctl* (cf. 2.6.2). La désactivation ultérieure des vérifications n'est en revanche possible que si le mode "développement" a été activé à la compilation du noyau, par l'option *CONFIG_CLSM_MOUNT_DEVEL*. Ces différents points de contrôle s'appuient sur le sous-système *devctl* (cf. 3) lorsque celui-ci est inclus dans le noyau, ou dans le cas contraire sur des mécanismes par défaut plus simples.

Les différentes fonctions de contrôle de montages *VFS* sont toutes appelées depuis le *hook* LSM *sb_mount()*, qui reporte la prise de décision sur la fonction appropriée en fonction du type de montage. Ces fonctions ne sont par ailleurs actives qu'une fois que la variable *sysctl kernel.clip.mount* a été mise à zéro (cf. 2.6.2). En particulier, elles ne sont pas actives au démarrage du système, ce qui permet de réaliser les différents montages spécifiés par */etc/fstab* sans contrainte.

Option de montage *MS_NOLOCK*

Le patch CLIP-LSM ajoute le support d'une option de montage supplémentaire, *MNT_NOLOCK*. Lorsque ce drapeau est présent dans le champ *mnt_flags* d'un montage *VFS* (*struct vfsmount*), aucun verrou (POSIX, BSD, ou *lease*) ne peut être posé sur les fichiers de ce montage. Cette restriction permet de bloquer un canal caché de communication entre contextes *vserver* partageant un même montage (a priori en lecture seule). De plus, la présence de ce drapeau sur un montage interdit l'ajout d'une "veille" *inotify* (appel système *inotify_add_watch()* supporté uniquement lorsque l'option *CONFIG_INOTIFY* est activée dans le noyau) sur les fichiers de ce montage. De telles veilles *inotify,* qui ne sont pas prises en compte par *vserver* à ce stade, offriraient en effet un autre canal de communication entre contextes partageant un montage.

L'option *MNT_NOLOCK* est attribuée à un montage en incluant *MS_NOLOCK* dans les options passées à l'appel système *sys_mount*. Les utilitaires déployés dans CLIP qui sont susceptibles d'être utilisés pour réaliser des opérations de montage [14] sont modifiés de manière à supporter dans leurs arguments des options *nolock* et *lock*, correspondant respectivement au drapeau *MS_NOLOCK* et à son absence (cas par défaut).

Option de montage *MS_NOSYMFOLLOW*

De même, le patch CLIP-LSM ajoute une option de montage *MNT_NOSYMFOLLOW*, inspirée par l'option du même nom disponible sous certains systèmes BSD. Lorsque ce drapeau est présent dans les *mnt_flags* d'un montage *VFS*, aucun lien symbolique ne peut être suivi au sein de ce montage. Cette restriction permet de contrer les possibilités d'attaques de type *"confused deputy"*, dans lesquelles l'ouverture légitime par un processus privilégié d'un fichier modifiable par des utilisateurs non privilégiés serait redirigée vers un autre fichier par l'intermédiaire d'un lien symbolique, de manière par exemple à exploiter les privilèges du processus réalisant l'ouverture pour modifier un fichier système. L'option *MNT_NOSYMFOLLOW* est attribuée à un montage en incluant *MS_NOSYMFOLLOW* dans les options passées à l'appel *sys_mount*. Les utilitaires susceptibles de réaliser un montage au sein de CLIP sont modifiés de manière à supporter des options *nosymfollow* et *symfollow* (par défaut) dans leurs arguments, permettant de positionner ou non ce drapeau.

Contrôle des options de remontage

Le LSM CLIP interdit toute opération de remontage qui conduirait à retirer certaines options restrictives associées à un montage existant. Plus précisément, il interdit :

- de remonter en lecture-écriture un montage en lecture seule (qu'il s'agisse de la propriété standard du *superblock* associé ou de la propriété *vserver* des montages *VFS*, par exemple de type *bind*)
- de passer l'option *exec* à un montage *noexec*
- de passer l'option *suid* à un montage *nosuid*
- de passer l'option *dev* à un montage *nodev*
- de passer l'option *atime* à un montage *noatime*
- de passer l'option *diratime* à un montage *nodiratime*
- de passer l'option *lock* à un montage *nolock*

Contrôle des options de montage bind

De manière similaire, le LSM vérifie, lors d'un montage de type *bind*, les options associées au montage *VFS* auquel appartient le répertoire source du montage, et interdit le masquage d'options restrictives de ce montage source dans le nouveau montage, selon les mêmes règles que pour un remontage.

On notera cependant qu'à ce stade une exception est maintenue pour les montages récursifs (drapeaux *MS_BIND | MS_REC*) de la racine n'un *namespace* VFS (c'est-à-dire lorsque le point de montage est *"/"*). Une telle opération, qui est typiquement réalisée lors de la création d'une cage *vserver* (cf. 6.2), n'est soumise à aucun contrôle des options de montage associées, en l'attente d'une correction des interfaces de configuration associées.

Contrôle des nouveaux montages

Les options associées aux nouveaux montages (autre que *bind*) sont vérifiées uniquement lorsque le sous-système *devctl* est inclus dans le noyau. Dans ce cas, les nouveaux montages réalisés depuis un *device* local (c'est-à-dire ceux auxquels le noyau associe une structure *struct file_system_type* dont le champ *fs_flags* porte le drapeau *FS_REQUIRES_DEV*) sont soumis à un contrôle de leurs options de montage vis-à-vis des restrictions d'accès éventuellement imposées par *devctl* au *device* source. Les options contrôlées à ce niveau sont uniquement les drapeaux *MS_NOSUID*, *MS_NOEXEC* et *MS_NODEV*, le contrôle des droits d'accès en lecture-écriture étant réalisé par ailleurs lors de l'ouverture du périphérique bloc proprement dit, comme dé-

14. Soit l'utilitaire *mount* du paquetage *sys-apps/util-linux*, et les utilitaires *vsctl* et *nsmount* du paquetage *app-clip/vsctl*.

crit ci-dessous. Par défaut, lorsque la source d'un nouveau montage n'est pas référencée dans la base *devctl*, le système n'autorise les montages qu'avec les trois options *MS_NOSUID, MS_NOEXEC* et *MS_NODEV,* les tentatives de réaliser un montage non référencé par *devctl* et sans une ou plusieurs de ces trois options sont systématiquement rejetées (dès lors que *kernel.clip.mount* vaut 0).

Contrôle des accès directs aux périphériques de type bloc

Ces vérifications réalisées lors des opérations de montage sont complétées d'un mécanisme spécifique de contrôle d'accès aux périphériques de type bloc, de manière à interdire notamment le contournement des restrictions en écriture sur les montages VFS par des écritures directes sur les fichiers */dev/hdaX* ou */dev/sdaX* correspondant au disque racine. Ce contrôle d'accès est réalisé par le *hook* LSM *inode_permission()* d'une part, et par un *hook* spécifique *inode_blkdev_open()*, appelé par la fonction interne au noyau, *blkdev_open()*, d'autre part. Ce contrôle interne est nécessaire dans la mesure où certains mécanismes pilotables depuis la couche utilisateur, en particulier le *device-mapper*, accèdent directement aux *devices* par leur numéro plutôt que par le fichier associé, et ne sont donc pas soumis normalement au contrôle d'accès sur les fichiers de */dev*. Ainsi, sans cette interception des fonctions internes du noyau, il resterait possible de contourner le contrôle d'accès au disque en projetant une partition à l'aide du *device mapper* et en accédant directement en écriture (sans montage) au *device* projeté.

Le type de contrôle d'accès réalisé lors de ces opérations varie selon que le sous-système *devctl* est inclus dans le noyau ou non.

- Dans le premier cas, le type d'accès demandé (lecture-seule ou lecture-écriture) est comparé aux permissions que *devctl* associe au *device* concerné : *CLSM_DEVICE_RO* ou *CLSM_DEVICE_RW*.
- Dans le second cas, seuls les accès en écriture sont contrôlés, et interdits dès lors que le numéro de majeur du *device* correspond à la variable *sysctl kernel.clip.ro_major* et que son numéro de mineur est compris entre les variables *kernel.clip.ro_minor_low* et *kernel.clip.ro_minor_high*. Ces trois variables doivent être configurées par les scripts d'initialisation au démarrage du système, et leur modification ultérieure doit être ensuite interdite, typiquement par le masquage de *CAP_SYS_MODULE* par le *cap_bset* global du système (cf. 2.6.2).

Le deuxième mécanisme, plus simple que la mise en œuvre de *devctl*, ne permet que d'interdire le contournement de l'option *ro* associée au montage de la racine d'un système, et ce uniquement lorsque cette racine est montée directement, sans RAID logiciel par exemple. Il ne permet en revanche pas de protéger le *device* associé au *swap* lorsque celui-ci est chiffré [15], ce qui ouvre potentiellement un vecteur d'escalade de privilèges entre processus *root* de niveau de privilèges différents [16]. La mise en œuvre de *devctl* permet au contraire de protéger séparément le disque racine, et le fichier du *device-mapper* associé au *swap* clair, et autorise la prise en compte d'un mécanisme de RAID pour le disque racine, dans lequel il est nécessaire de protéger à la fois les partitions logiques RAID et les disques physiques sous-jacents.

Contrôle des appels *pivot_root* et des montages *MS_MOVE*

Une fois la variable *kernel.clip.mount* mise à zéro, le noyau restreint l'accès à l'appel *pivot_root*(), ainsi qu'aux montages de type *MS_MOVE* (qui peuvent être utilisés pour réaliser un substitut à *pivot_root()*) au seul processus *init* du système.

Contrôle de l'activation de *swap*

15. Dans ce cas, le *device* chiffré peut typiquement être protégé contre les accès en écriture, dans la mesure où il est généralement porté par le même disque (donc le même numéro de majeur) que la racine du système, mais cette protection ne peut pas être étendue au *device* clair, qui correspond à une projection *device-mapper* ou *loop*, avec un numéro de majeur différent de celui associé au disque racine.

16. Un processus *root* sans privilèges particuliers, mais disposant des droits d'accès discrétionnaires à */dev/mapper/swap0*, pourrait par exemple modifier la pile d'un processus *root* disposant de privilèges spécifiques, lorsque les pages correspondant à cette pile sont écrites sur le *swap*.

Enfin, afin de compléter un éventuel contrôle du périphérique de type bloc associé au *swap* (comme décrit ci-dessus), l'activation d'un nouveau *swap* par l'appel *sys_swapon* est interdite dès lors que la variable *sysctl kernel.clip.mount* est mise à zéro. Il est ainsi impossible de contourner les restrictions d'accès au périphérique *swap* légitime du système en introduisant un *swap* supplémentaire porté par un fichier (typiquement un fichier régulier, plutôt qu'un *device*) auquel un attaquant pourrait accéder plus librement.

2.5 Durcissement des cages *chroot*

Le LSM CLIP intègre enfin plusieurs mécanismes spécifiques de durcissement des cages *chroot*, qui offrent des contre-mesures contre plusieurs attaques classiques permettant à *root* de sortir d'une telle prison, sans pour autant pouvoir être vus comme exhaustifs à ce stade. Ces mécanismes, ainsi que le mode "développement" permettant leur désactivation par un *sysctl*, sont activables individuellement par des options de compilation (cf. 2.6.1).

- Un contrôle des descripteurs de fichiers ouverts détenus par une tâche lors d'un appel *chroot*. L'appel *sys_chroot()* renvoie une erreur --EPERM si la tâche appelante possède des descripteurs de fichiers ouverts correspondant à des répertoires. Ce mécanisme s'appuie sur un *hook* spécifique *task_chroot()*, appelé par *sys_chroot()*. Il est par ailleurs contournable par le privilège CLSM *CLSM_PRIV_CHROOT*[17].
- Un contrôle des descripteurs de fichiers ouverts transmis à une tâche enfermée dans un *chroot*. Ce contrôle se base sur le *hook* LSM *file_receive()*. Il interdit la transmission à une tâche *"chrootée"* d'un descripteur de fichier ouvert situé en dessous de la racine de la tâche. Cette restriction est contournable par le privilège *CLSM_PRIV_CHROOT* (qui doit être possédé par la tâche réceptrice).
- Un contrôle des appels *ptrace*, interdisant toute opération de trace depuis un processus enfermé dans une prison *chroot*. Ce mécanisme se base sur le drapeau *CLSM_FLAG_CHROOTED* de l'étiquette CLSM de la tâche appelante pour déterminer que celle-ci est enfermée dans un *chroot*. On notera qu'une restriction aussi large interdit aussi l'accès au */proc/<pid>/fd* et */proc/<pid>/environ* de toute autre tâche du système, et est de ce fait généralement incompatible avec la mise en oeuvre d'exécutables dont le fonctionnement requiert de tels accès, comme la machine virtuelle *java*. L'option *CLSM_FLAG_CHROOTED* est ainsi désactivée sur la plupart des configurations CLIP [18].
- Un mécanisme permettant de tester ce même drapeau *CLSM_FLAG_CHROOTED* dans les contrôles spécifiques aux tâches *chrootées* du patch *Grsecurity*, au lieu du test *proc_is_chrooted()* normalement employé dans ces contrôles. En effet, *proc_is_chrooted()* se base sur une comparaison du système de fichier racine de la tâche testée à celui de la "faucheuse " de processus (*init*) pour déterminer si une tâche est dans une cage *chroot*, ce qui présente l'inconvénient de considérer tout processus d'une instance *vserver* comme ainsi *chrooté*. A contrario, le drapeau *CLSM_FLAG_CHROOTED* est remis à zéro lors d'un changement de contexte, ce qui signifie qu'un processus d'une instance *vserver* non-ADMIN n'est considéré comme *chrooté* que s'il a fait un appel *chroot* après son changement de contexte. Par ailleurs, le mécanisme de remontée vers la racine (implémenté dans le fichier *grsecurity/grsec_chroot.c* au sein de l'arborescence de sources Linux) mis en œuvre dans la fonctionnalité *GRKERNSEC_CHROOT_FCHDIR* afin de bloquer les *fchdir()* sur un répertoire situé hors du *chroot* est lui-aussi modifié par l'option *CLSM_CHROOT_GRSEC*, afin de prendre en compte le changement de *namespace VFS* et de racine réelle au sein d'une instance *vserver*.

17. Ce contournement est nécessaire en particulier au fonctionnement de l'utilitaire *secure-mount* inclus dans les outils *util-vserver*.
18. On notera cependant que l'option *GRSEC_CHROOT_FINDTASK*, combinée avec *CLSM_CHROOT_GRSEC*, permet dans tous les cas d'interdire la mise en oeuvre d'appels *ptrace* pour sortir d'une prison *chroot*.

2.6 Interface utilisateur et contrôle d'accès

2.6.1 Options de compilation

Les principales fonctionnalités du LSM CLIP sont configurables lors de la compilation du noyau, par l'interface *kconfig* standard, un sous-menu spécifique lui étant consacré dans la catégorie *Security* du menu de configuration noyau. Ces options de configuration, récapitulées dans le 4, comportent aussi des options de développement, qui relaxent certains mécanismes de contrôle d'accès aux paramètres de configuration du module, afin d'en faciliter le développement ou l'intégration. On notera bien que ces options introduisent de sérieuses failles dans le modèle de sécurité du module, et ne doivent donc en aucun cas être retenue sur un système en production.

Option	Signification
CONFIG_CLSM_ROOTCAPS	Rend le masque de capacités attribué par défaut à *root* configurable par la variable *sysctl kernel.clip.rootcap*, cf. 2.2.
CONFIG_CLSM_ROOTCAPS_DEVEL	Mode ``développement'' de la gestion de *kernel.clip.rootcap*, cf. 2.6.2. Dépendant de CONFIG_CLSM_ROOTCAPS
CONFIG_CLSM_NOSUID_ROOT	Interdit la prise en compte des bits *suid root*.
CONFIG_CLSM_DEBUG	Option de débogage qui génère des traces *printk* supplémentaires lors des traitements CLSM.
CONFIG_VERIEXEC	Active le support de *veriexec*, et donne accès aux options de configuration de ce sous-système.
CONFIG_CLSM_MOUNT	Active les contrôles supplémentaires lors des opérations de montage VFS, cf. 2.4.
CONFIG_DEVCTL	Active le support de *devctl*, et donne accès aux options de configuration de ce sous-système.
CONFIG_CLSM_MOUNT_DEVEL	Mode ``développement'' de l'option précédente, permettant une manipulation moins contrainte de la variable *sysctl kernel.clip.mount*.
CONFIG_CLSM_NET	Active le contrôle d'accès au réseau, cf. 2.3.
CONFIG_CLSM_NET_FULL	Etend le contrôle d'accès réseau aux opérations élémentaires d'envoi et de réception de paquets.
CONFIG_CLSM_NEL_DEVEL	Mode ``développement'' de l'option précédente, permettant la désactivation de ces contrôles par le *sysctl kernel.clip.net*.
CONFIG_CLSM_CHROOT	Permet d'activer les traitements spécifiques de durcissement des cages *chroot*, cf. 2.5.
CONFIG_CLSM_CHROOT_DEVEL	Mode ``développement'' de l'option précédente, permettant la désactivation de ces contrôles par le *sysctl kernel.clip.chroot*.
CONFIG_CLSM_CHROOT_OPENDIRS	Active les contrôles portant sur les répertoires ouverts lors de l'appel à *chroot()*.
CONFIG_CLSM_CHROOT_SOCKFD	Active les contrôles portant sur les descripteurs de fichiers transmis par socket UNIX.
CONFIG_CLSM_CHROOT_PTRACE	Interdit toute activité *ptrace* aux processus enfermés dans un *chroot*.
CONFIG_CLSM_CHROOT_GRSEC	Active la réutilisation des informations CLSM dans les contrôles *grsecurity* spécifiques aux processus enfermés dans des *chroot*.
CONFIG_CLSM_PROC_PID	Affiche les informations de l'étiquette de sécurité CLSM d'une tâche dans */proc/pid/status*.

TABLE 4 – Option de compilation CLIP-LSM

2.6.2 Interface *sysctl*

Le LSM CLIP est configurable par l'intermédiaire de plusieurs variables *sysctl*, rassemblées dans la catégorie *kernel.clip.**. Ces variables constituent la seule interface utilisateur accessible en écriture du module, exception faite des interfaces propres aux sous-systèmes *veriexec* (cf. 4.4) et *devctl* (cf. 3.3) .

- ***kernel.clip.rootcap*** *:* cette variable correspond au masque de capacités attribué par défaut à l'utilisateur *root*. Sa valeur initiale correspond à la constante *CAP_INIT_EFF_SET*, c'est-à-dire toutes les capacités sauf *CAP_SETPCAP*. Cette variable est normalement accessible selon les mêmes modalités que *kernel.cap-bound* : tout accès (en lecture aussi bien qu'en écriture) nécessite la capacité *CAP_SYS_MODULE*, et les accès en écriture ne peuvent que réduire la variable (une intersection bit à bit est réalisée systématiquement entre le nouveau masque et l'ancien). Cette restriction est cependant levée lorsque l'option *CONFIG_CLSM_ROOTCAPS_DEVEL* a été sélectionnée à la compilation, auquel cas la variable est modifiable sans autre contrôle que les droits discrétionnaires classiques.
- ***kernel.clip.mount*** *:* cette variable permet d'activer ou désactiver les contrôles CLSM portant sur les montages VFS (cf. 2.4). On notera que cette variable est "inversée" : sa mise à zéro active les contrôles de montages, qui ne sont pas réalisés tant que la variable est non-nulle. Cette inversion permet la réutilisation des mêmes routines de modification que pour la variable *kernel.cap-bound*. Ainsi, tout accès à *kernel.clip.mount* nécessite la capacité *CAP_SYS_MODULE*, et ne peut que réduire la valeur de la variable. Celle-ci étant initialisée à 1 au démarrage, le seul accès possible en écriture consiste à activer les contrôles de montage, qui ne peuvent alors plus être désactivés. Ces restrictions d'accès peuvent néanmoins être supprimées en sélectionnant l'option *CONFIG_CLSM_MOUNT_DEVEL* lors de la compilation du noyau. Dans ce cas, l'accès à la variable est libre, aux droits discrétionnaires près.
- ***kernel.clip.networking*** *:* cette variable permet d'activer (variable à 1) ou de désactiver (variable à 0) les contrôles d'accès au réseau implémentés par le LSM (cf. 2.3). Elle ne fait l'objet d'aucun contrôle d'accès autre que les droits discrétionnaires sur les fichiers du */proc*. Elle n'est par ailleurs supportée que si l'option *CONFIG_CLSM_NET_DEVEL* a été sélectionnée à la compilation.
- ***kernel.clip.chroot*** *:* cette variable permet d'activer (1) ou de désactiver (0) les contrôles de durcissement des prisons *chroot* (cf. 2.5). Elle ne fait l'objet d'aucun contrôle d'accès spécifique, et n'est supportée que si l'option *CONFIG_CLSM_CHROOT_DEVEL* a été sélectionnée à la compilation. On notera que cette option ne désactive pas les vérifications réalisées à l'aide des *hooks grsecurity*, qui peuvent être individuellement désactivées par les *sysctl kernel.grsecurity.chroot**.
- ***kernel.clip.ro_major, kernel.clip.ro_minor_low, kernel.clip.ro_minor_high*** *:* ces trois variables correspondent respectivement au numéro de majeur et aux plus petit et plus grand numéros de mineur associés au disque racine du système, à protéger absolument contre toute écriture. Elles ne sont définies que lorsque l'option *CONFIG_CLSM_MOUNT* est activée, sans inclusion du sous-système *devctl*. Leur modification se fait par la même routine que la variable *kernel.cap-bound* : seule la réduction de la variable est permise, et ce uniquement par une tâche disposant de la capacité *CAP_SYS_MODULE*. Leur valeur initiale est *0* (plus grand entier non signé), ce qui les rend inopérantes par défaut (le test "numéro de mineur supérieur ou égal à *ro_minor_low*" n'est jamais vérifié).

2.6.3 Interface *proc*

Le LSM offre par ailleurs une interface utilisateur, accessible en lecture uniquement, dès lors que l'option *CONFIG_PROC_PID_STATUS* a été sélectionnée à la compilation. Dans ce cas, les champs *t_privs*, *t_flags* et *t_vflags* de chaque *struct clsm_task_sec*, c'est à dire les privilèges CLSM, le statut CLSM et les options *veriexec* de chaque tâche du système sont consultables dans le fichier */proc/<pid>/status* associé, avec *<pid>* le *pid* de la tâche.

2.6.4 Journalisation

Le LSM CLIP journalise son activité par l'interface standard des *printk*, consultable par un utilitaire *syslog* en couche utilisateur. Tout refus d'accès par un *hook* du LSM fait l'objet d'une entrée au journal, avec la priorité *KERN_WARNING*, et préfixée du mot clé *CLSM :* suivi du nom de la fonction dans lequel le refus est constaté.

En revanche, les accès autorisés par le module ne font normalement l'objet d'aucune journalisation. Certaines opérations peuvent cependant générer des traces si le module a été compilé avec l'option *CONFIG_CLSM_DEBUG*.

3 Sous-système *devctl*

3.1 Base d'entrées *devctl*

Le sous-système *devctl* s'articule autour d'une base d'entrées *devctl*, de type *struct devctl_device*, stockées en mémoire noyau (et allouées dans un *kmem_cache* dédié). Chacune de ces entrées comporte les champs suivants :

- un numéro de majeur *d_major* (entier strictement positif) ;
- un numéro de mineur *d_minor* (entier positif ou nul) ;
- une "largeur" *d_range* (entier positif ou nul) ;
- une priorité *d_priority* (entier strictement positif) ;
- un masque de permissions d'accès *d_perm* ;

Une telle entrée décrit l'ensemble des périphériques de type bloc de numéro de majeur *d_major*, et de mineur compris entre *d_minor* et *d_minor + d_range*, extrémités incluses. Elle associe à ces périphériques un certains nombre de droits d'accès, représentés par les différents bits du masque *d_perm*, décrits dans le . La description par segments de numéros de mineur autorise des recouvrements ou des intersections non-nulles, dans lesquelles un ou plusieurs périphériques sont décrits par plusieurs entrées *devctl*. Dans ce cas, les droits d'accès applicables à ces périphériques sont ceux inclus dans l'entrée qui possède la plus haute priorité. Le cas où plusieurs entrées de même priorité décriraient un même périphérique est à éviter, car l'entrée utilisée dans ce cas est déterminée par un choix d'implémentation qui est susceptible d'évoluer.

Drapeau	Droits associés
DEVCTL_PERM_NONE	Aucun
DEVCTL_PERM _RO	Montage avec les options *ro, nodev, noexec, nosuid.* Accès en lecture seule au fichier spécial de type bloc.
DEVCTL_PERM _RW	Montage sans l'option *ro.* Accès en lecture-écriture au fichier spécial de type bloc.
DEVCTL_PERM _EXEC	Montage sans l'option *noexec.*
DEVCTL_PERM _SUID	Montage sans l'option *nosuid.*
DEVCTL_PERM _DEV	Montage sans l'option *nodev.*

Table 5 – Drapeaux de contrôle d'accès aux périphériques bloc

Les différentes entrées constituant la base sont chaînées et insérées dans une table de hachage afin de faciliter leur recherche. Le schéma de hachage retenu garanti que deux entrées de même numéro de majeur (en particulier, deux entrées ayant une intersection non nulle) sont hachées dans un même *hash bucket.*

La base d'entrée *devctl* est initialisée par la fonction *devctl_init()*, qui est appelée lors de l'initialisation du LSM CLIP. Elle est vide à l'origine. Des entrées peuvent être ajoutées ou supprimées par la couche utilisateur à l'aide d'appels *ioctl()* sur un *device* spécifique (cf. 3.3.2), et ce uniquement tant que les contrôles d'accès aux montages VFS du LSM CLIP n'ont pas été activés par le *sysctl kernel.clip.mount* (cf. 2.6.2). Les accès à cette base sont synchronisés par des sections critiques *RCU*[19] pour les lectures, et par un *spin_lock* global pour les écritures, ce qui autorise (cf. [LDD], chapitre 5) un nombre arbitraire de lecteurs et au maximum un écrivain à un moment donné. Aucun comptage de référence n'est nécessaire pour les entrées *devctl*, qui ne sont pas

19. On pourra noter que les sections critiques RCU protégeant les accès en lecture ne sont pas strictement nécessaires, dans la mesure où les accès en lecture et en écriture ne sont, par construction, pas simultanés (les premiers ne sont réalisés que lorsque *kernel.clip.mount* est nul, et les seconds ne sont autorisés que lorsque cette variable est non-nulle). Elle est cependant laissée en place, vu son faible impact, afin de gérer les états transitoires (lors de la mise à zéro de *kernel.clip.mount*) ainsi que d'éventuels développements futurs.

utilisées hors des sessions critiques et peuvent donc être simplement désallouées, après un "*quiescent state*" RCU, dès leur suppression de la base.

3.2 Intégration au contrôle d'accès

Outre *devctl_init()*, le sous-système *devctl* exporte au reste du noyau une unique fonction, *devctl_check()*. Cette fonction prend pour arguments un numéro de *device* (*dev_t*) et un mode d'accès souhaité, sous la forme d'un masque de drapeaux similaires à ceux décrits dans le . Cette fonction recherche dans la base *devctl* une entrée applicable au *device* passé en argument. Lorsqu'une telle entrée est trouvée, son masque de permissions est comparé au mode passé en argument. Un `1' est renvoyé lorsque l'accès est autorisé (lorsque le mode demandé est inclus dans le masque de permissions), et un `0' dans le cas contraire. Le comportement lorsqu'aucune entrée applicable au *device* n'est trouvée dans la base varie selon que l'option *CONFIG_DEVCTL_STRICT* a été sélectionnée ou non lors de la compilation du noyau. Dans le premier cas, l'accès est interdit, tandis qu'il est autorisé dans le second.

Cette fonction est appelée par deux fonctions du LSM CLIP :

- Lors d'un montage local associé à un *device* : le mode passé en argument est une combinaison des permissions *DEVCTL_PERM_SUID, DEVCTL_PERM_EXEC* et *DEVCTL_ PERM_DEV*, selon celles des options *nosuid, noexec* et *nodev* qui ne sont pas passées en argument du montage. Les accès en lecture-écriture ne sont pas pris en compte à ce niveau.
- Lors de l'ouverture d'un périphérique de type bloc, soit par *sys_open()* sur le fichier spécial correspondant, soit par la fonction interne au noyau *blkdev_open()* : le mode passé en argument est soit *DEVCTL_PERM_RO*, soit *DEVCTL_PERM_RW* selon que l'ouverture se fait en lecture seule ou non. Une telle ouverture est réalisée en particulier lors d'un montage, ce qui permet de prendre en compte à ce niveau les options *ro* ou *rw* passées à ce montage.

On notera que les appels à cette fonction ne sont réalisés par le LSM que lorsque les contrôles d'accès aux montages VFS (cf. 2.4) sont activés, par mise à zéro de la variable *sysctl kernel.clip.mount*. En particulier, aucune vérification n'est demandée à *devctl* au démarrage du système, ce qui permet de configurer entièrement la base d'entrées avant d'y faire appel.

Ces appels sont complétés par une modification de l'appel système *sys_swapoff*, permettant de désactiver un *swap* porté par un périphérique de type bloc auquel est associé un masque de permissions *devctl* nul [20]. Lorsque l'option *CONFIG_DEVCTL* est activée à la compilation du noyau, la fonction implémentant *sys_swapoff* ouvre le fichier qui lui est passé en paramètre à l'aide de fonctions internes aux noyau (*open_namei()*) plutôt que par l'interface exportée *filp_open*, ce qui permet de spécifier une ouverture sans permissions (compatible avec un masque *devctl* nul), plutôt qu'avec les permissions *O_RDWR* normalement utilisées.

3.3 Interfaces utilisateur

Le sous-système *devctl* supporte un nombre limité d'options de configuration statiques. Il est dynamiquement administrable depuis la couche utilisateur à travers deux fichiers spécifiques. On notera que *devctl* ne journalise pas directement son activité. Les accès rejetés (lors d'un montage ou de l'ouverture d'un *block device*) sont en revanche journalisé par les fonctions de contrôle d'accès du LSM CLIP.

20. Ce qui constitue a priori la norme : il n'est pas souhaitable de donner un quelconque accès, aussi bien en lecture qu'en écriture, au périphérique bloc associé à un *swap*, à deux exceptions près : l'appel *swapon* d'activation de ce *swap*, et l'appel *swapoff* permettant de le désactiver. Le premier cas peut être obtenu simplement en réalisant l'appel *swapon* avant d'activer le contrôle d'accès aux périphériques blocs. Le deuxième nécessite un traitement spécifique, dans la mesure où l'appel *swapoff* est typiquement réalisé lors de l'arrêt du système, alors que ce contrôle d'accès est actif.

3.3.1 Options de compilation

Le sous-système *devctl* supporte deux options de compilation :
- *CONFIG_DEVCTL_STRICT* : lorsque cette option est activée, *devctl* interdit tout accès à un *device* auquel aucune entrée *devctl* n'est associée. Dans le cas contraire, tous les accès à un tel *device* sont autorisés. Cette option n'est pas activée dans CLIP à ce stade.
- *CONFIG_DEVCTL_PROC* : cette option active l'interface */proc* décrite en 3.3.3. Elle est sélectionnée dans CLIP à ce stade.

3.3.2 Fichier spécial */dev/devctl*

Ce fichier spécial de type caractère, de majeur 1 et de mineur 15, constitue l'interface de configuration de la base d'entrées *devctl*. Il supporte à cette fin deux commandes *ioctl* :
- *DEVCTL_IO_LOAD* pour le chargement d'une entrée dans la base ;
- *DEVCTL_IO_UNLOAD* pour la suppression d'une entrée de la base.

Les deux commandes nécessitent un argument de type *struct devctl_arg*, qui comprend les cinq champs significatifs d'une entrée *devctl* : majeur, mineur, largeur, priorité et permissions. Seuls les trois premiers champs sont utilisés lors d'une suppression. De manière corrélée, la fonction d'ajout interdit l'insertion dans la base d'une entrée comportant les mêmes majeur, mineur et largeur qu'une entrée existante. En revanche, aucune mesure spécifique n'est mise en œuvre pour interdire la création de recouvrements entre entrées de priorités égales.

Le chargement et la suppression d'entrées *devctl* ne sont possibles que tant que les contrôles d'accès aux montages VFS du LSM CLIP ne sont pas engagés. Dans la mesure où l'activation de ces contrôles est irréversible (sauf option *CONFIG_CLSM_MOUNT_DEVEL*, cf. 2.6.1), l'activation de *devctl* s'accompagne du verrouillage de la base. Les *ioctl* sur */dev/devctl* nécessitent de surcroit la capacité *CAP_SYS_ADMIN*, ainsi que les droits d'accès discrétionnaires en écriture sur le fichier lui-même.

3.3.3 Interface *proc*

Cette interface n'est disponible que lorsque l'option *CONFIG_DEVCTL_PROC* a été sélectionnée à la compilation du noyau. Dans ce cas, le fichier */proc/devctl* offre une représentation en lecture seule de la base d'entrées *devctl*. Ce fichier est par défaut lisible par tous.

4 Sous-système *veriexec*

4.1 Base d'entrées *veriexec* et vérifications dynamiques

4.1.1 Entrées *veriexec*

Le sous-système *veriexec* offre des fonctionnalités similaires à une implémentation des *File POSIX Capabilities*[21]. Il permet d'attribuer des capacités spécifiques à un exécutable, mais sans faire de ces capacités un attribut du fichier lui-même. Les capacités supplémentaires associées à un fichier exécutable sont stockées dans une représentation spécifique (*veriexec store*), en mémoire noyau. Chaque entrée dans cette représentation associe :
- un identifiant unique de fichier : numéro de *device* identifiant de manière univoque le système de fichiers, et numéro d'*inode* identifiant de manière univoque le fichier au sein de ce système de fichiers ;

21. Implémentation qui ajouterait typiquement des masques de capacités spécifiques comme attribut de chaque fichier, de manière similaire aux attributs de sécurité -- ou *file tags* -- mis en œuvre par SELinux. Erreur : source de la référence non trouvée donne une implémentation possible d'un tel mécanisme pour Linux.

- une empreinte cryptographique (*md5, sha1* ou *sha256* ou *ccsd* - cf. [CLIP 1205] - le support des autres fonctions de hachage fournies par la couche *crypto_tfm* du noyau pourrait être trivialement ajouté) du contenu du fichier ;
- un identifiant (type *enum veriexec_digest*) de fonction de hachage, correspondant à la fonction utilisée pour calculer l'empreinte ci-dessus ;
- trois masques de capacités POSIX (effectif, permis et héritable) attribuées au fichier ;
- un *bitmap* de privilèges CLSM (cf. 2.2.2) ;
- un *bitmap* d'options *veriexec*, conditionnant de manière plus spécifique le traitement de cette entrée (cf. tableau 6) ;
- un compteur de référence.

Les entrées *veriexec* sont allouées dans un cache noyau (*kmem_cache_t*) dédié, de manière à minimiser la consommation [22] et la fragmentation mémoire. Après allocation, les entrées sont insérées dans une table de hachage (par hachage du numéro d'*inode* à l'aide de *hash_long()*), afin de permettre une recherche rapide (cf. 5). Ce référencement, de même que chaque accès spécifique à une entrée, s'accompagne d'une incrémentation de son compteur de référence. Une entrée est automatiquement désallouée lorsque son compteur de référence est décrémenté à zéro.

Une interface utilisateur "ancrée" sur le fichier spécial en mode caractère */dev/veriexec* (cf. 4.4.2) permet, par des *ioctl()*, l'ajout ou le retrait d'entrées *veriexec* à la base stockée en mémoire noyau.

4.1.2 Traitement *veriexec* lors d'un appel *exec*

Lors d'un appel *exec()*, le sous-système LSM fait appel -- systématiquement sauf dans le cas de l'exécution d'un script référencé par *veriexec*, comme décrit plus bas -- au sous-système *veriexec* par la fonction *veriexec_getcreds()*, invoquée par le *hook* LSM *bprm_set()*, qui lui passe en argument la *struct linux_binprm* correspondant au fichier en cours d'exécution. La fonction *veriexec_getcreds()* recherche dans la base *veriexec* une entrée correspondant à ce fichier et portant le drapeau *VRX_FLAG_EXE*. Si aucune telle entrée n'est trouvée, ou encore si une entrée est trouvée mais que les conditions d'éligibilité [23] ne sont pas satisfaites, la fonction retourne immédiatement, et le chargement de l'exécutable se poursuit de manière standard, avec en particulier les masques de capacités attribués par défaut à son identité et un masque de privilèges CLSM nul. Lorsqu'une entrée éligible est trouvée, l'empreinte cryptographique du fichier en cours de chargement est calculée à l'aide de la fonction de hachage spécifiée par l'entrée. En cas d'échec de la comparaison, la fonction renvoie un code d'erreur non nul, qui est propagé par la couche LSM et met fin au chargement du fichier. En cas de succès, les masques de capacités et de privilèges CLSM inclus dans l'entrée sont ajoutés bit à bit aux masques présents dans la *struct binprm* (capacités) ou dans son étiquette de sécurité (privilèges CLSM, cf. 2.2.2), après quoi les drapeaux CLSM de cette même étiquette de sécurité sont mis à jour [24], et les options *veriexec* de l'entrée sont recopiées dans le champ correspondant de l'étiquette de sécurité. Le sous-système *veriexec* rend ensuite la main à la couche LSM, et le chargement se poursuit, les privilèges ajoutés par *veriexec* à la *struct linux_binprm* étant transférés à la tâche résultante selon la formule POSIX standard, à deux exceptions près :

- Les privilèges supplémentaires accordés par *veriexec* sont annulés par le *hook bprm_apply_creds()* si la tâche appelante est en train d'être tracée ou partage des éléments de son *task_struct*. Dans ce cas, les privilèges CLSM "*root*" de la *struct linux_binprm* sont effacés, et ses masques de capacités sont ramenés à la valeur par défaut pour l'identité associée (cf. 2.2.3). Par ailleurs, le sous-système *veriexec* est rappelé

22. On notera que la taille d'une entrée *veriexec* est fonction de la taille du plus gros *fingerprint* des différentes fonctions de hachage cryptographique (*VERIEXEC_DIG_**) configurées dans le noyau. Ainsi, sélectionner l'option *VERIEXEC_DIG_SHA256* alors que seules des empreintes *md5* ou *sha1* sont utilisées entraîne un gaspillage de mémoire.

23. Ces conditions correspondent à des propriétés du fichier (par exemple, exécution depuis un montage *nosuid* lorsque *CONFIG_VERIEXEC_MNTSUID* a été activée à la compilation), ou de la tâche appelante (*euid* et *uid* de la tâche, lorsque l'entrée possède le drapeau *veriexec VRX_FLAG_NEEDROOT*).

24. Ajout des drapeaux *CLSM_FLAG_RAISED, CLSM_FLAG_INHERITED* ou *CLSM_FLAG_BUMPED*, selon les masques et options de l'entrée.

Option	Signification
VRX_FLAG_EXE	Entrée associée à un exécutable, invalide pour une projection en mémoire de type bibliothèque ou interpréteur (utilisable par *veriexec_getcreds()*, mais pas par *veriexec_updatecreds()*)
VRX_FLAG_LIB	Entrée associée uniquement à une bibliothèque, ne peut pas conférer de capacités lors d'une exécution directe (mais reste exécutable, par ex. */lib/ld-linux.so*). Une telle entrée est utilisable par *veriexec_updatecreds()*, mais pas par *veriexec_getcreds()*.
VRX_FLAG_NEEDROOT	Entrée qui n'est prise en compte que pour des tâches d'*uid et euid 0*.
VRX_FLAG_NEEDLIB	Entrée dont l'exécution entraîne la vérification systématique du référencement dans la base *veriexec* de tout fichier projeté en exécution dans l'espace mémoire de la tâche.
VRX_FLAG_CHECKLIB	Effet similaire à *VRX_FLAG_NEEDLIB, à ceci prêt qu'une vérification d'empreinte cryptographique est aussi réalisée pour les fichiers projetés en exécution.*
VRX_FLAG_INHERIT	Entrée qui transfère directement son masque de capacités héritables au processus qui l'exécute (capacité héritable "forcée").
VRX_FLAG_SCRIPT	Entrée correspondant à un exécutable au format script. Sans cette option, les caractéristiques *veriexec* du script sont écrasées par celles de son interpréteur (*sh, perl*, etc…) NB : cette option introduit en l'état une vulnérabilité importante, dans la mesure où il est trivial de modifier l'environnement d'exécution d'un script. Elle est cependant utile en phase de développement.

TABLE 6 – Options des entrées *veriexec*

dans ce cas par la fonction *veriexec_bprm_resetopts()*, afin de "purger" les options *veriexec* qui pourraient influer sur le calcul ultérieur des privilèges, c'est-à-dire à ce stade les drapeaux *VRX_FLAG_INHERIT* et *VRX_FLAG_SCRIPT*. Ces différentes opérations ont lieu avant le transfert de privilèges de la *struct linux_binprm* à la tâche appelante.

- Le sous-système *veriexec* est systématiquement rappelé après le calcul des capacités de la tâche par la formule standard. La fonction appelée dans ce cas, *veriexec_task_raisecaps()*, réalise à ce stade une seule opération : si le drapeau *VRX_FLAG_INHERIT* est présent parmi les options *veriexec* de la *struct linux_binprm*, le masque de capacité héritable de cette dernière est transféré à la tâche courante (dont le masque héritable n'a pas été modifié par le calcul POSIX standard). On notera que ce masque héritable est dans ce cas réduit au masque spécifié dans l'entrée, et ne comporte pas la composante attribuée par défaut à l'identité de l'appelant.

Le calcul des empreintes cryptographique est réalisé à l'aide de l'interface *sendfile()* des structures *struct file_operations*, en utilisant un *read_actor_t* spécifique. Les fonctions de hachage utilisées sont celles de la couche *crypto_tfm* standard du noyau Linux. A ce stade, les fonctions *md5, sha1* et *sha256* sont les seules supportées par *veriexec*, mais le support des autres fonctions de hachage offertes par *crypto_tfm* (*sha384/512, tiger, whirlpool*) pourrait être trivialement ajouté.

Lorsque l'option *CONFIG_VERIEXEC_ALLOWSCRIPT* a été sélectionnée à la compilation du noyau, un traitement particulier est réalisé pour une entrée portant le drapeau *VRX_FLAG_SCRIPT*[25], typiquement as-

25. Ce drapeau est simplement ignoré lorsque l'option *CONFIG_VERIEXEC_ALLOWSCRIPT* n'a pas été activée à la compilation du noyau.

socié à un exécutable de type script (tout fichier texte exécutable commençant par `#!'). Pour un tel fichier, le *hook bprm_set()* est appelé, comme pour tout autre type d'exécutable, sur le fichier script lui-même, mais est ensuite aussi rappelé aussi sur le fichier correspondant à l'interpréteur (puis à l'interpréteur de l'interpréteur, si le premier interpréteur est lui-même un script, et ainsi de suite jusqu'au premier exécutable dans un format binaire). Le *hook bprm_set()* examine en premier lieu les options *veriexec* de la *struct linux_binprm()* par un appel à la fonction *veriexec_binprm_scriptmode()*. Si cette fonction, qui teste la présence du drapeau *VRX_FLAG_SCRIPT* dans l'étiquette de sécurité, retourne une valeur non-nulle, le *hook* retourne immédiatement, sans procéder à l'attribution de capacités par défaut associées à l'identité de l'appelant, ni appeler *veriexec_getcreds()*. Ainsi, dans le cas d'un script associé à une entrée *veriexec* possédant le drapeau "script", le premier appel au *hook* invoquera *veriexec_getcreds()*, qui, après vérification de l'empreinte du script, positionnera ce même drapeau dans l'étiquette de sécurité de la *struct linux_binprm*. Toute invocation ultérieure de ce même *hook* sur la même structure mais avec un fichier différent correspondant à l'interpréteur "court-circuitera" l'affectation des privilèges à cette structure, évitant ainsi l'écrasement des privilèges associés au script d'origine par ceux (vraisemblablement nuls) associés à l'interpréteur.

On notera que ce dernier mécanisme, qui permet d'attribuer des privilèges *root* spécifiques à des scripts, s'apparente à la prise en compte d'un *bit s* sur un script (qui n'est normalement pas pris en compte par Linux), et que son utilisation introduit potentiellement une **vulnérabilité grave**, étant donné qu'il est très facile de détourner l'exécution d'un script, notamment par modification de son environnement. La présence d'une telle fonctionnalité dans *veriexec*, sans restriction supplémentaire [26], se veut réservée aux phases de développement uniquement, afin de faciliter le prototypage par des scripts, étant entendu que ceux-ci devront être remplacés par des exécutables compilés avant leur déploiement.

4.1.3 Autres appels au sous-système *veriexec*

Le sous-système *veriexec* peut par ailleurs être rappelé ultérieurement par la couche LSM, lorsque l'étiquette de sécurité d'une tâche possède, dans son *bitmask* d'options *veriexec*, l'un des drapeaux *VRX_FLAG_NEEDLIB* ou *VRX_FLAG_CHECKLIB*. Dans ce cas, la fonction *veriexec_updatecreds()* est appelée lors de chaque projection d'un fichier dans la mémoire de la tâche avec des droits en exécution, c'est-à-dire lors du chargement de l'interpréteur, et lors de tout appel *mmap()* ou *mprotect()* avec l'option *PROT_EXEC* (cf. 2.2.3). La fonction *veriexec_updatecreds()* reçoit en argument un pointeur vers le fichier projeté, pour lequel elle recherche une entrée *veriexec* portant le drapeau *VRX_FLAG_LIB*. Si une telle entrée est trouvée, la fonction retourne immédiatement sans erreur (cas du drapeau *VRX_FLAG_NEEDLIB*) ou procède à la vérification de l'empreinte cryptographique associée au fichier (drapeau *VRX_FLAG_CHECKLIB*). En cas de succès de la vérification, la fonction retourne sans erreur, et le chargement se poursuit. En cas d'échec de la vérification, ou si aucune entrée correspondant au fichier n'est trouvée (en particulier dans le cas d'une projection auquel aucun fichier n'est associé), la fonction supprime tous les privilèges de la tâche appelante, par remise à zéro des trois masques de capacités, des masques effectifs et sauvegardés de privilèges CLSM, et du masque d'options *veriexec*, puis renvoie un code d'erreur à la couche CLSM, qui interrompt le chargement, mais pas nécessairement l'exécution de la tâche sauf dans le cas du chargement de l'interpréteur.

4.1.4 Mise en cache des vérifications *veriexec*

Les vérifications d'intégrité *veriexec* peuvent avoir un impact significatif sur les temps de chargement des exécutables. Afin de mitiger cet impact, l'option de compilation *CONFIG_VERIEXEC_CACHE* active un cache de vérification *veriexec*. Ce cache s'appuie sur le cache d'*inodes* du noyau Linux standard, enrichi des informations supplémentaires portées par les étiquettes de sécurités ajoutées à ces dernières (cf. 2.1.2). La mise en cache

26. Par exemple, limitation à certains privilèges et au seul utilisateur *root*, accompagné éventuellement d'un nettoyage de l'environnement.

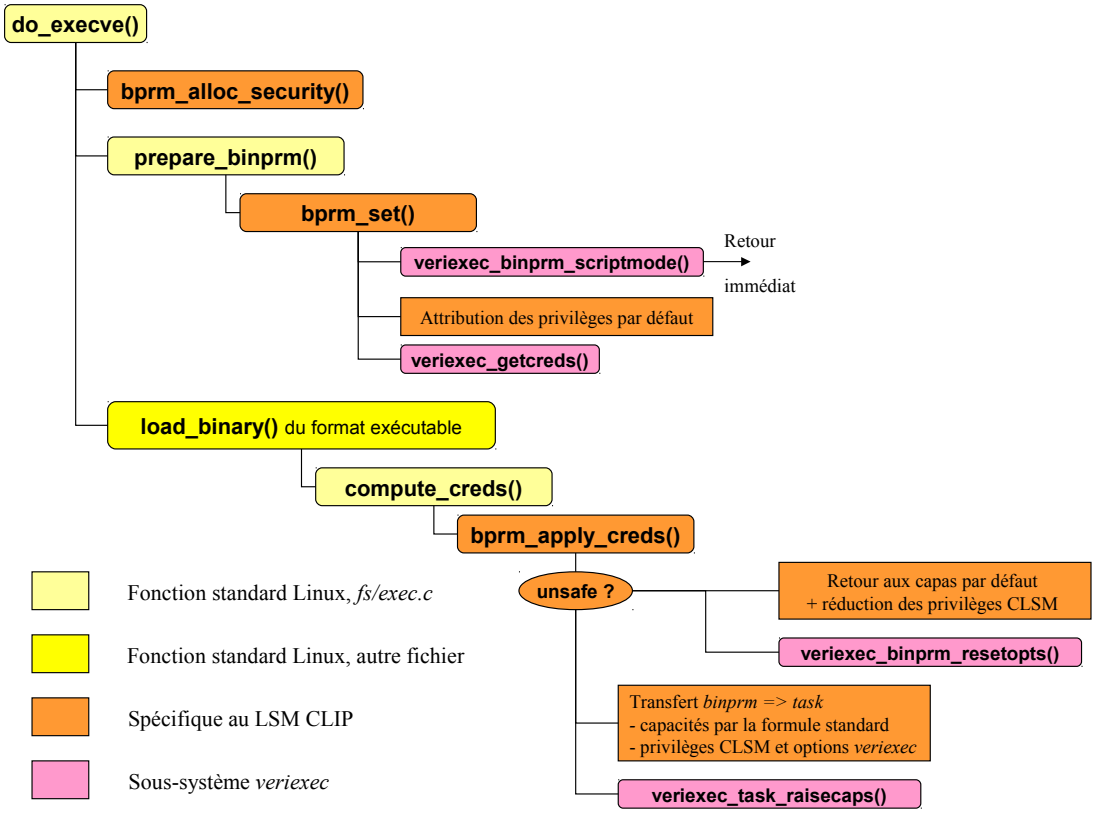


FIGURE 2 – Appels LSM et *veriexec* pour le traitement d'un appel *exec()*

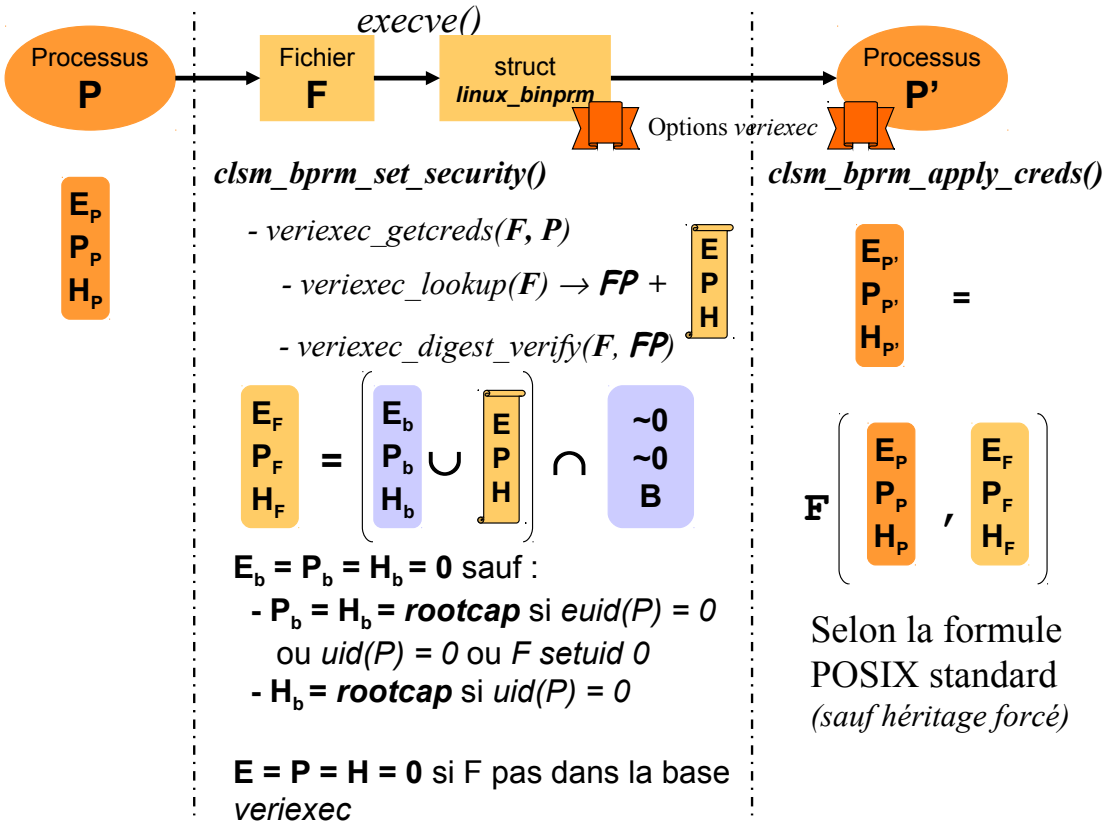


FIGURE 3 – Calcul des capacités lors d'un appel *exec* sur un fichier référencé dans la base *veriexec*

d'une vérification se fait par l'ajout du drapeau *CLSM_IFLAG_CACHED* dans le champ *i_flags* de l'étiquette de sécurité de l'*inode* vérifié. Les vérifications *veriexec* sont systématiquement précédées d'un test de présence de ce drapeau. Lorsque le drapeau est présent, la vérification renvoie immédiatement un code de succès, sans procéder au calcul d'empreinte cryptographique. Dans le cas contraire, une vérification complète est réalisée. En cas de succès de cette dernière, le drapeau est ajouté. Ce drapeau est par ailleurs révoqué explicitement dans deux cas :

- Lorsqu'un processus obtient un accès en écriture à l'*inode*. La révocation est réalisée lors de l'appel à *get_write_access()*, sous protection du *spin_lock i_lock* de l'*inode*, immédiatement après qu'il ait été vérifié que les accès en écriture ne sont pas interdits sur l'*inode*. L'absence de *race-condition* avec la mise en cache tient au fait que celle-ci ne peut être réalisée qu'alors que les accès en écriture à l'*inode* sont interdits, et ne sont rétablis qu'une fois terminée l'exécution de l'*inode*.
- Lorsque l'entrée *veriexec* associée à la vérification initiale est supprimée. Dans ce cas, les numéros de *device* et d'*inode* associés à l'entrée sont recherchés dans le cache d'*inode* du noyau, et le drapeau est retiré à l'*inode* correspondant.

Enfin, la mise en cache est naturellement révoquée lorsque l'*inode* lui-même est retiré du cache noyau.

4.2 Intégration à la gestion de privilèges

Veriexec apporte en soi deux fonctionnalités nouvelles pour la gestion de privilèges :

- Une alternative [27] mieux maîtrisée au placement de *bits s* sur des fichiers exécutables : plutôt que de donner l'ensemble des privilèges *root* à un exécutable invoqué par un utilisateur non privilégié, *veriexec* permet d'attribuer uniquement les capacités nécessaires au fonctionnement nominal de cet exécutable. Il suffit à cette fin d'associer à l'exécutable une entrée *veriexec* incluant les capacités souhaitées dans ses masques permis et effectif, ce qui entraîne systématiquement le transfert de ces capacités dans les masques effectif et permis du processus résultant (cf. figure 1). Cette fonctionnalité est équivalente à celle obtenue par une approche de *file capabilities* plus standard. Elle permet de limiter les possibilités d'escalade de privilèges suite à la compromission d'un exécutable *setuid root* [28].
- Un mécanisme permettant d'attribuer des privilèges CLSM à des processus, en fonction du fichier qu'ils exécutent. Ce mécanisme est, en l'état, le seul moyen d'attribuer de tels privilèges, exception faite des règles de propagation automatique lors d'un *fork()* décrite plus haut.
  Combiné aux autres fonctionnalités du LSM CLIP, *veriexec* introduit de plus un mécanisme d'attribution à *root* de capacités qui ne serait plus incluses dans son masque de capacités par défaut, réduit par l'intermédiaire du *sysctl kernel.clip.rootcap* (cf. ??). Ce mécanisme s'appuie typiquement sur des entrées *veriexec* portant le drapeau *VRX_FLAG_NEEDROOT*, qui restreint leur utilisation au seul super-utilisateur, et incluant les capacités souhaitées dans leurs masques permis et effectif. Combinées à un masque de capacités *root* réduit, elles permettent à *root* de disposer des capacités les plus " lourdes" (*CAP_SYS_ADMIN, CAP_NET_ADMIN*, etc.) uniquement lors de la mise en œuvre de certains exécutables supposés maîtrisés. On notera que le processus *root* particulièrement privilégié qui résulte de l'exécution d'un tel exécutable fait l'objet d'une protection vis-à-vis des autres processus *root*, afin d'éviter les possibilités d'escalade de privilèges entre processus *root*. Ces protections, qui reposent sur les

27. On notera à ce sujet que l'option de compilation *CONFIG_VERIEXEC_MNTSUID* (cf. 4.4.1) permet de limiter le domaine d'application de *veriexec* aux montages permettant l'utilisation d'un exécutable avec bit *s*.

28. Outre le fait de ne pas disposer de toutes les capacités, un exécutable privilégié par *veriexec* diffère fondamentalement d'un exécutable *setuid root* par le fait qu'il s'exécute toujours sous l'identité de l'appelant, et non sous l'identité effective *root*. Ainsi, à l'exception des cas où l'entrée *veriexec* confère à l'exécutable la capacité *CAP_SETUID*, les capacités possédées par l'exécutable seront perdues lors d'un appel *exec()* (réalisé par exemple dans le cadre d'une attaque de type *return to libc*). Cette propriété, couplée à des mesures de défense en profondeur (typiquement le principe *"W^X"* pour la mémoire, apporté par exemple par *PaX* , cf. [PAX]) visant à empêcher l'injection de code arbitraire dans le processus courant, limite significativement les possibilités pour un attaquant de mettre en œuvre de manière malicieuse les capacités qui sont effectivement attribuées à l'exécutable.

différents mécanismes décrits en ??, sont similaires dans le principe à la protection des processus résultant de l'exécution d'un binaire *setuid*.

Enfin, alternativement à ce mécanisme d'attribution directe de capacités par les exécutables, *veriexec* peut être mis en œuvre selon un principe d'attribution indirecte de capacités, faisant cette fois intervenir la portion "héritable" du calcul de capacités. L'utilisation "canonique" (c'est-à-dire envisagée dans le *draft* POSIX) des masques de capacités héritables consiste à attribuer un masque héritable adapté à chaque exécutable, ainsi qu'à chaque utilisateur lors de son ouverture de session. Ainsi, chaque exécutable dispose de capacités potentielles, qui sont individuellement activées ou non selon le masque de capacités de l'utilisateur qui réalise l'appel *exec*. *Veriexec* peut naturellement être employé selon ce principe, sous réserve d'être complété par un mécanisme d'attribution de capacités héritables par utilisateur lors de l'ouverture de session (typiquement, un module PAM). Mais il peut aussi être utilisé afin de conférer des masques héritables à des processus spécifiques plutôt qu'à des utilisateurs. En effet, la présence du drapeau *VRX_FLAG_INHERIT* dans une entrée *veriexec* entraîne, lors de l'exécution du fichier associé, le transfert "forcé" du masque héritable de l'entrée au processus résultant. Cette exception à la formule de calcul POSIX (selon laquelle le masque de capacités héritable n'évolue jamais) permet de gérer des niveaux de capacités héritables hétérogènes parmi les processus d'un même utilisateur [29]. Couplé à l'attribution de capacités héritables à des utilitaires de base, cette hétérogénéité permet de gérer plus finement l'accès privilégié à des utilitaires trop versatiles pour être en soi jugés de confiance. Ainsi, on pourra attribuer une capacité *CAP_SYS_ADMIN* héritable à */bin/mount*, mais pas la capacité permise correspondante, dans la mesure où */bin/mount* ne réalise pas de contrôle spécifique de ses arguments et ne doit donc pas pouvoir être invoqué librement, même par *root*. En revanche, un processus auquel on fait confiance pour n'invoquer */bin/mount* que de manière contrôlée et non nuisible à la sécurité du système pourra se voir attribuer un masque de capacités héritables spécifiques lui permettant d'invoquer *mount* avec les capacités effectives requises. On limite ainsi l'utilisation privilégiée de *mount* à certains chemins d'exécution de confiance sous l'identité *root*. Ce principe est résumé dans la , avec *xdm* comme processus de confiance (utilisant *mount* uniquement pour monter des partitions utilisateur lors d'une ouverture de session), et *sh* comme exemple d'un processus non de confiance (même sous l'identité *root*).

4.3 Partitionnement *veriexec*

La représentation interne des entrées *veriexec* est virtualisée de manière à prendre en compte l'existence de plusieurs contextes de sécurité *vserver*. A cette fin, une notion de contexte *veriexec* est introduite. A chaque contexte *veriexec* est associée une structure *"xlist"* (*struct vrhl_xlist*) contenant principalement, comme résumé dans la figure 5 :

- un identifiant de contexte, qui correspond au *xid* du contexte de sécurité *vserver* auquel ce contexte *veriexec* est associé ;
- un *bitmap* de niveau, qui conditionne le fonctionnement de *veriexec* dans ce contexte ;
- un ensemble d'entrées *veriexec*, structurées dans une table de hachage (hachage du numéro d'*inode*) afin d'optimiser les recherches ;
- un masque de capacités maximal, qui est intersecté avec les trois masques associés à une entrée *veriexec* lors de son ajout à ce contexte ;
- un masque de privilèges maximal, qui est intersecté avec le masque de privilèges associé à une entrée *veriexec* lors de son ajout à ce contexte ;
- un verrou *spin_lock*, un compteur de références et un *handle rcu* servant à la gestion des accès concurrents, tel que décrits plus bas.

Contrairement aux entrées veriexec, les contextes ne sont pas alloués dans un cache *kmem* dédié, mais directement dans les slabs génériques du noyau (par *kmalloc()/kfree()*).

29. On notera que, comme décrit en 2.2.3, les processus privilégiés au sens des capacités héritables sont protégés vis-à-vis des autres processus du même utilisateur, comme le sont les processus privilégiés au sens des capacités effectives ou permises.

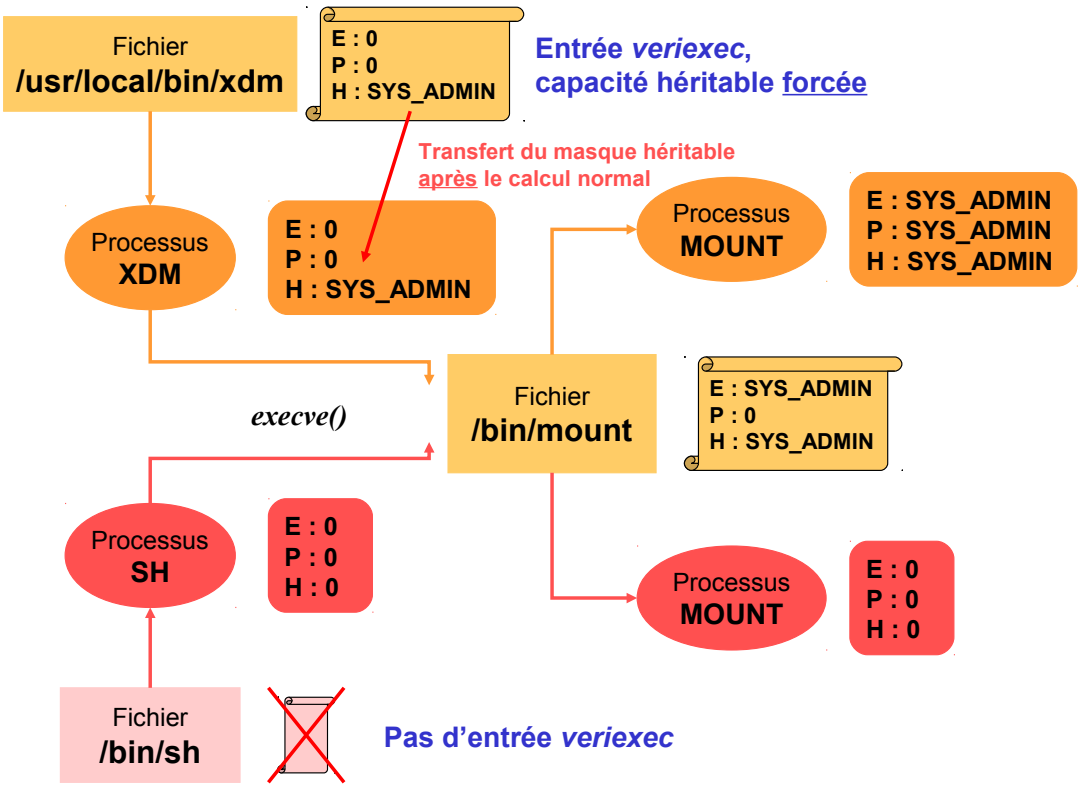


FIGURE 4 – Principe de propagation des capacités par *veriexec*.

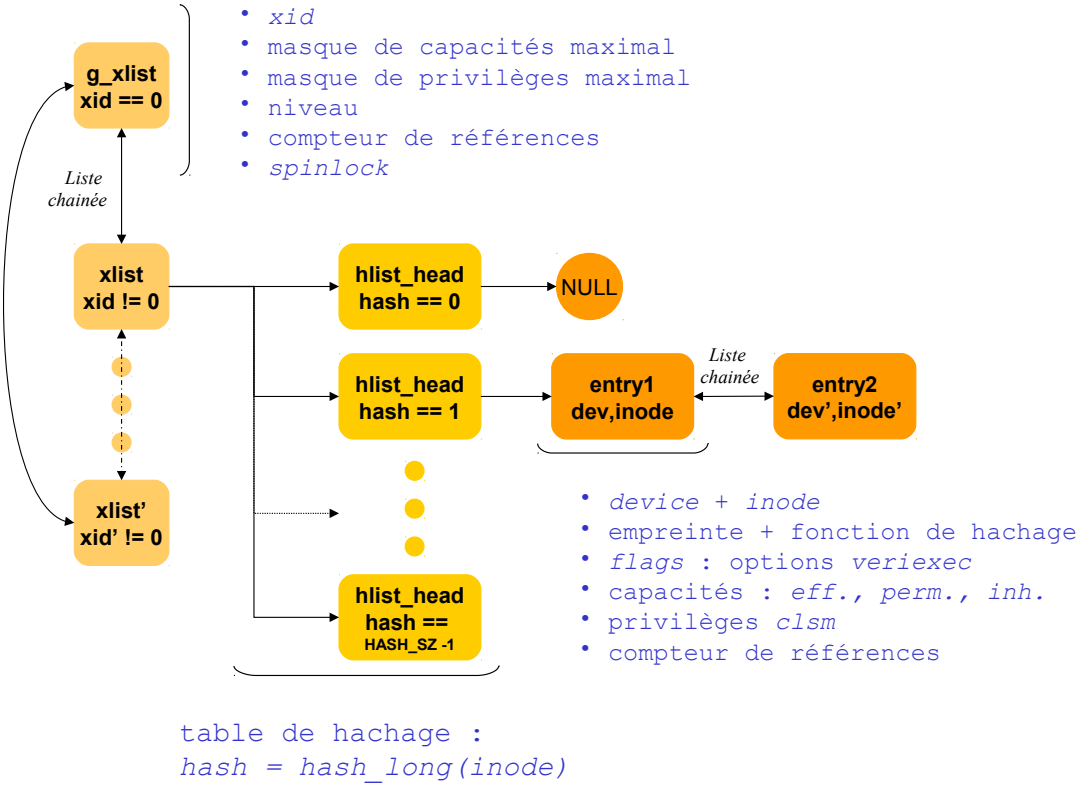


FIGURE 5 – Structure de la base d'entrées *veriexec*, partitionnée par contextes *vserver*

Chaque contexte *veriexec* constitue une représentation complète, et indépendante des autres contextes (en particulier, une même entrée peut être présente dans différents contextes). Toutes les opérations *veriexec* sont redirigées " silencieusement" vers le contexte *veriexec* -- si un tel contexte existe -- dont l'identifiant correspond au *xid* du contexte de sécurité *vserver* associé à la tâche courante. On délègue de cette manière à une représentation dédiée par instance *vserver* non seulement la recherche des entrées *veriexec* associées aux exécutables et bibliothèques de cette instance, mais aussi les fonctions d'ajout et de suppression d'entrées (sauf si le niveau du contexte l'interdit explicitement par le drapeau *VRXLVL_SELF_IMMUTABLE*, cf. tableau 7, et dans tous les cas uniquement dans le même contexte). Le contexte ADMIN (*xid 0*), qui correspond au contexte *vserver* du même nom et est le seul créé automatiquement au démarrage, est en revanche le seul à avoir accès aux opérations les plus privilégiées, comme la création ou la suppression d'autres contextes, ou l'ajustement du niveau ou des masques maximaux d'un contexte. Ce contexte ADMIN possède de plus un privilège spécifique, lui permettant d'ajouter ou de supprimer des entrées dans d'autres contextes, sauf si le niveau de ces contextes l'interdit explicitement (drapeau *VRXLVL_ADMIN_IMMUTABLE*). Par ailleurs, un autre contexte spécifique, différent de ADMIN, peut être défini [30] : le contexte UPDATE, dont le *xid* est configuré par l'*ioctl VERIEXEC_IO_SETUPDATE*. Ce contexte peut lui-aussi ajouter ou supprimer des entrées dans les autres contextes, y compris ADMIN, là encore sauf si le niveau d'un contexte l'interdit explicitement (drapeau *VRXLVL_UPDATE_IMMUTABLE*). Comme son nom l'indique, ce contexte permet de déléguer les mises à jour du système, ou du moins d'une partie du système, à un contexte *vserver* dédié, comme cela est fait dans le système CLIP.

Le niveau associé à chaque contexte correspond à un *bitmap* d'options déterminant le comportement de *veriexec* dans ce contexte. Il permet en premier lieu d'activer ou de désactiver *veriexec* indépendamment dans chaque contexte. Dans l'état désactivé, les appels à *veriexec* effectués par la couche LSM retournent immédiatement et sans erreur, aucune recherche d'entrée, vérification d'empreinte ou attribution de privilèges n'étant réalisée. Par ailleurs, le niveau du contexte définit les possibilités d'ajout ou de suppression d'entrées dans ce contexte, qui peuvent être interdites en fonction du contexte réalisant l'opération (ADMIN, UPDATE, ou le contexte cible lui-même, indépendamment), ou, de manière décorrélée du contexte appelant, pour tout entrée correspondant à un fichier auquel les options de montage interdisent l'accès en écriture. Cette dernière option est prévue dans l'optique de mises à jour dynamiques de la base *veriexec* accompagnant la mise à jour des exécutables : seules peuvent être modifiées les entrées correspondant à des fichiers effectivement susceptibles, du fait des options de montage, d'être mis à jour. Enfin, le niveau du contexte détermine celles des opérations d'administration les plus privilégiées qui sont autorisées pour ce contexte : modification du niveau lui-même, modification des masques maximaux de capacités et de privilèges CLSM, ajout et suppression d'autres contextes. Les différents drapeaux sont résumés dans le . Il est rappelé que la modification du niveau d'un contexte est réservée au contexte ADMIN.

Lors du démarrage du système, seul le contexte ADMIN est créé, avec un niveau nul (*veriexec* désactivé, aucune restriction d'accès), un masque maximal de capacité égal au *cap_bset* (c'est-à-dire normalement à cet instant *CAP_INIT_EFF_SET*, toutes les capacités sauf *CAP_SETPCAP*), et un masque maximal de privilèges comportant tous les bits à un. D'autres contextes peuvent ensuite être créés depuis le contexte ADMIN, si son niveau l'y autorise, par l'*ioctl VERIEXEC_IO_ADDCTX*, en spécifiant dans les arguments, le *xid* ainsi que le niveau et les masques initiaux du nouveau contexte. Ces masques doivent être chacun inférieur au masque du même type du contexte ADMIN (évalués à l'instant de la création du nouveau contexte). Par ailleurs, les masques maximaux d'un contexte peuvent être ultérieurement réduits, mais pas augmentés, depuis le contexte ADMIN uniquement. On notera que dans le cas d'une telle réduction, le nouveau masque ne s'applique pas aux entrées déjà présentes dans la base, mais uniquement à celles ajoutées par la suite.

La gestion des accès concurrents aux structures *veriexec* en mémoire noyau se fait selon le principe *Read-Copy-Update* (RCU -- cf. par exemple [LDD] pour une description de la logique et des primitives associées),

30. Cette définition n'est possible qu'une unique fois dans la vie du système, il est impossible de modifier le numéro du contexte UPDATE une fois que celui-ci a été défini.

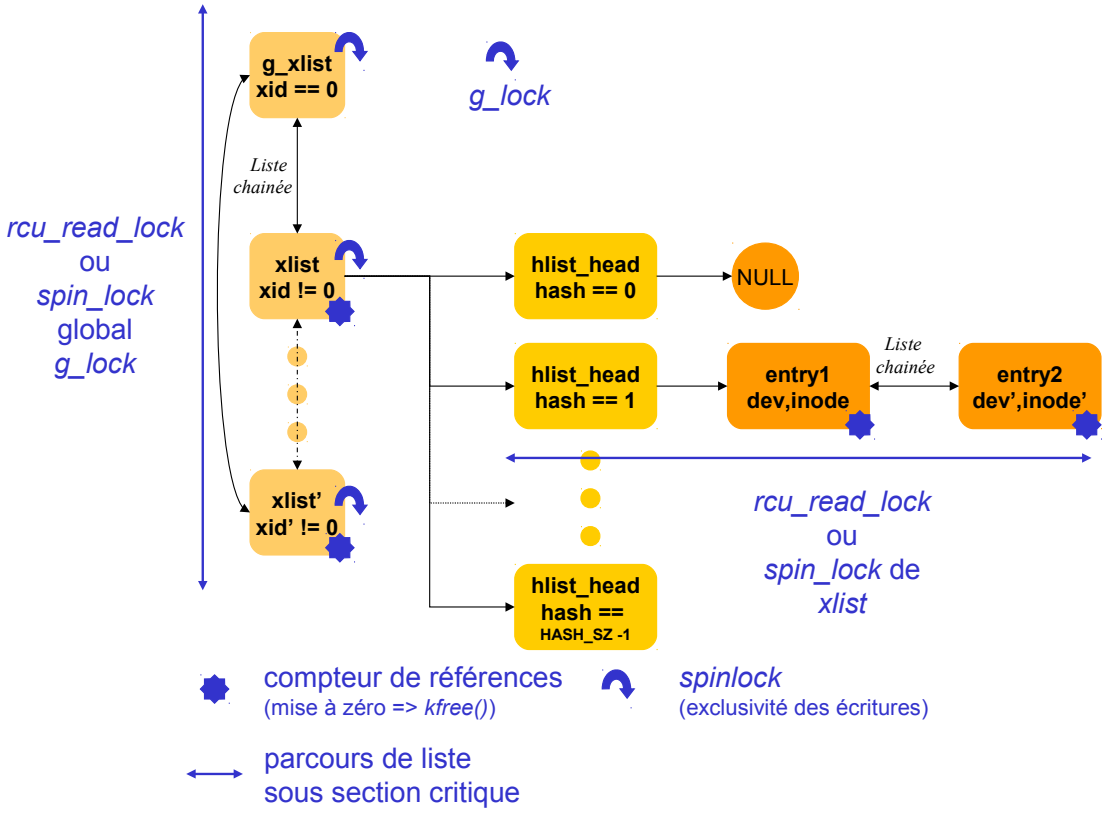


FIGURE 6 – Gestion des accès concurrents à la base d'entrées *veriexec*

Drapeau	Signification
VRXLVL_ACTIVE	*Veriexec* est actif dans ce contexte.
VRXLVL_LVL_IMMUTABLE	Interdiction de retirer des drapeaux de niveaux à ce contexte.
VRXLVL_SELF_IMMUTABLE	Interdiction d'ajouter ou retirer des entrées à ce contexte depuis lui-même.
VRXLVL_ADMIN_IMMUTABLE	Interdiction d'ajouter ou retirer des entrées à ce contexte depuis le contexte ADMIN (0).
VRXLVL_UPDATE_IMMUTABLE	Interdiction d'ajouter ou retirer des entrées à ce contexte depuis le contexte UPDATE (s'il est défini).
VRXLVL_CTX_IMMUTABLE	Interdiction de créer ou supprimer des contextes *veriexec* (n'a de sens que pour le contexte ADMIN).
VRXLVL_CTXSET_IMMUTABLE	Interdiction de réduire le *cap-bset* (au sens *veriexec*) de ce contexte.
VRXLVL_ENFORCE_MNTRO	Interdiction d'ajouter ou retirer dans ce contexte des entrées correspondant à des fichiers auxquels les droits de montages interdisent l'accès en écriture.

TABLE 7 – Drapeaux de niveau *veriexec*

afin d'optimiser les accès en lecture, qui sont seuls nécessaires en dehors des fonctions d'administration. Les accès en lecture sont mutuellement non exclusifs et ne nécessitent aucune synchronisation en dehors d'une désactivation locale de la préemption (pas de prise de verrou), les accès en écriture sont mutuellement exclusifs, et les accès en lecture restent possibles durant un accès en écriture. Un accès en lecture (recherche d'une entrée) nécessite de traverser successivement deux sections critiques RCU : une pour le parcours de la liste chainée des contextes *veriexec*, puis une pour le parcours du *hash bucket* approprié dans la table de hachage référençant les entrées du contexte retenu. La liste de contextes *veriexec* est protégée vis-à-vis des accès en écriture par un verrou *spinlock* statique dédié. Chaque contexte possède son propre *spinlock*, contrôlant les accès en écriture à l'ensemble des listes d'entrées (*hash buckets*) de sa table de hachage. De ce fait, l'administration locale d'un contexte n'a pas d'impact bloquant en termes de synchronisation sur les autres contextes, ce qui limite les possibilités de déni de service depuis un contexte non privilégié. Les désallocations de contextes et d'entrées *veriexec* font appel à un comptage de référence, selon le modèle *get()/put()* très largement utilisé au sein du noyau, associés à un *call-back* RCU pour assurer la synchronisation avec les lecteurs en cours d'itération sur la liste. Ces différents éléments sont repris dans la figure 6.

4.4 Interfaces utilisateur et contrôle d'accès

4.4.1 Options de compilation

Le sous-système *veriexec* peut-être paramétré à la compilation par les options suivantes, accessibles par le sous-menu "*Verified Execution*" du menu de configuration *CLIP-LSM*.

4.4.2 Fichier spécial */dev/veriexec*

Toutes les opérations d'administration sont réalisées à travers un *device* spécifique, *veriexec*, fichier spécial en mode caractère de numéro de majeur 1 et de mineur 14. Ce *device* ne supporte que trois types d'opérations : ouverture (*open*, pour laquelle des droits d'accès en lecture-écriture sont systématiquement exigés), fermeture (*release*) et *ioctl*. C'est par cette méthode que sont passées toutes les commandes d'administration, telles que décrites dans le . La plupart de ces appels *ioctl* sont potentiellement bloquants (du fait principalement d'alloca-

Option	Signification
CONFIG_VERIEXEC_MNTSUID	Interdit l'utilisation des fonctionnalités *veriexec* lorsqu'un fichier est exécuté depuis un montage possédant l'option *nosuid*, empêchant ainsi le contournement de ces options de montage par l'intermédiaire de *veriexec* (utilisé comme substitut à un bit `s').
CONFIG_VERIEXEC_CACHE	Active la mise en cache des vérifications *veriexec* (cf. 4.1.4).
CONFIG_VERIEXEC_ALLOWSCRIPT	Permet la prise en compte des drapeaux *VRX_FLAG_SCRIPT* dans les entrées *veriexec* (cf. et 4.1.2). Ne doit pas être activé dans un système en production.
CONFIG_VERIEXEC_HASH_BITS	Nombre de bits de battement dans la table de hachage stockant les entrées d'un contexte. Permet d'ajuster le compromis consommation mémoire / temps de recherche selon le nombre d'entrées envisagé.
CONFIG_VERIEXEC_DEBUG	Génère des messages *printk* de *debug* pour les opérations *veriexec*
CONFIG_VERIEXEC_DEBUG_EXTRA	Génère des messages *printk* de *debug* plus verbeux
CONFIG_VERIEXEC_DEBUG_MEMLEAK	Tient à jour un compteur des entrées allouées à un instant donné (tous contextes confondus), afin de permettre une mesure de la consommation de mémoire par *veriexec* (à des fins de *debug* ou éventuellement d'audit).

Table 8 – Options de compilation *veriexec*

Option	Signification
CONFIG_VERIEXEC_DIG_MD5	Support de la fonction de hachage MD5.
CONFIG_VERIEXEC_DIG_SHA1	Support de la fonction de hachage SHA1.
CONFIG_VERIEXEC_DIG_SHA256	Support de la fonction de hachage SHA256.
CONFIG_VERIEXEC_DIG_CCSD	Support de la fonction de hachage CCSD.
CONFIG_VERIEXEC_PROC	Active l'interface de *debug* / audit : */proc/veriexec*.
CONFIG_VERIEXEC_STORE_NAMES	Sauvegarde les noms de fichiers dans les entrées *veriexec*, afin de faciliter la lecture de */proc/veriexec*.

Table 9 – Options de compilation *veriexec* (suite)

tions de mémoire noyau). Le drapeau *O_NONBLOCK* n'est pas pris en compte à l'ouverture. Enfin, l'ouverture du *device* n'est possible que dans un contexte *vserver* pour lequel un contexte *veriexec* a été créé, une erreur *--EPERM* étant retournée en l'absence de contexte *veriexec*.

On notera que les commandes d'ajout / suppression d'entrées et de lecture / écriture du niveau d'un contexte permettent de spécifier un contexte cible dans leurs paramètres. Ce paramètre peut être laissé à -

Numéro d'*ioctl*	Opération
VERIEXEC_IO_LOAD	Ajout d'une entrée dans le contexte spécifié dans les arguments.
VERIEXEC_IO_UNLOAD	Suppression d'une entrée dans le contexte spécifié dans les arguments.
VERIEXEC_IO_GETLVL	Lecture du niveau du contexte spécifié dans les arguments.
VERIEXEC_IO_SETLVL	Ecriture du niveau du contexte spécifié dans les arguments.
VERIEXEC_IO_ADDCTX	Ajout d'un contexte *veriexec*.
VERIEXEC_IO_DELCTX	Suppression d'un contexte *veriexec*.
VERIEXEC_IO_SETCTX	Modification d'un contexte *veriexec préexistant (masque de capacités maximal du contexte uniquement).*
VERIEXEC_IO_SETUPDATE	Définition du contexte UPDATE.
VERIEXEC_IO_MEMCHK	Lecture du nombre d'entrées allouées, tous contextes confondus (CONFIG_VERIEXEC_DEBUG_MEMLEAK).

TABLE 10 – Numéros d'*ioctl* de configuration *veriexec*, supportés par */dev/veriexec*.

1, auquel cas le contexte de l'appelant est automatiquement utilisé comme contexte cible. Les opérations de création, suppression et modification de contextes nécessitent en revanche la spécification d'un numéro de contexte cible explicite. Enfin, la notion de contexte cible n'existe pas pour les appels *SETUPDATE* et *MEMCHK*, qui sont transverses à tous les contextes : le contexte désigné comme UPDATE devient le contexte de mise à jour de tous les autres contextes, et la lecture du nombre d'entrées allouées se fait tous contextes confondus.

Par ailleurs, outre le contrôle d'accès au *device* (qui doit normalement être réservé à *root*), chacune de ces opérations est soumise à des contraintes spécifiques portant aussi bien sur les privilèges (POSIX et CLSM) de l'appelant que sur son contexte et le contexte cible de l'opération. Ces différentes contraintes sont répertoriées dans le tableau 11.

4.4.3 Interface *proc*

Une interface en lecture seule peut par ailleurs fournie, à des fins de mise au point ou de supervision, à travers un fichier du */proc*, */proc/veriexec*. Ce fichier n'est créé que si l'option *CONFIG_VERIEXEC_PROC* a été sélectionnée à la compilation. Il peut être ouvert dans chaque contexte *vserver* pour lequel un contexte *veriexec* existe. Sa lecture donne dans ce cas une description du contexte (numéro *xid* et masques maximaux de capacités et de privilèges CLSM), suivie d'une description de chaque entrée (identifiants, privilèges, capacités et options *veriexec* associés, type de fonction de hachage et empreinte cryptographique). On notera que la représentation par défaut ne garde pas trace des noms des fichiers, qui sont donc identifiés uniquement par un couple (numéro de *device kdev_t* / numéro d'*inode*). L'option de compilation *CONFIG_VERIEXEC_STORE_NAMES* permet un affichage plus explicite incluant le nom de fichier tel que passé en argument lors de la création de l'entrée, au prix d'un coût accru en consommation et fragmentation de la mémoire noyau.

L'ouverture du fichier */proc/veriexec* est aussi possible dans le contexte *vserver* WATCH (*xid 1*). Dans ce cas, la lecture du fichier donne une concaténation de toutes les représentations, tous contextes confondus. L'ouverture du fichier dans un contexte autre que WATCH auquel aucun contexte *veriexec* n'est associé renvoie une erreur *--ENOENT*.

Opération	Contexte de l'appelant	Privilèges requis
LOAD	**IDENT | ADMIN | UPDATE**	CAP_CONTEXT si ADMIN et contexte cible différent ; CLSM_PRIV_VERICTL si contexte appelant actif.
UNLOAD	**IDENT | ADMIN | UPDATE**	CAP_CONTEXT si ADMIN et contexte cible différent ; CLSM_PRIV_VERICTL si contexte appelant actif.
GETLEVEL	**IDENT | ADMIN**	CAP_CONTEXT si ADMIN et contexte cible différent.
SETLEVEL	**ADMIN**	CAP_CONTEXT si contexte cible différent de ADMIN ; CLSM_PRIV_VERICTL si contexte appelant actif.
ADDCTX	**ADMIN**	CAP_CONTEXT et CAP_SYS_ADMIN ; CLSM_PRIV_VERICTL si contexte appelant actif.
DELCTX	**ADMIN**	CAP_CONTEXT et CAP_SYS_ADMIN ; CLSM_PRIV_VERICTL si contexte appelant actif.
SETCTX	**ADMIN**	CAP_CONTEXT et CAP_SYS_ADMIN ; CLSM_PRIV_VERICTL si contexte appelant actif.
SETUPDATE	**ADMIN**	CAP_CONTEXT et CAP_SYS_ADMIN ; CLSM_PRIV_VERICTL si contexte appelant actif ; Le contexte UPDATE ne doit pas encore avoir été défini.
MEMCHK	**ADMIN | WATCH**	Aucun.

TABLE 11 – Privilèges et contextes requis pour les opérations d'administration *veriexec*.

Le fichier */proc/veriexec* est par défaut lisible par tous les utilisateurs.

4.4.4 Journalisation

Le sous-système *veriexec* journalise son activité par des *printk*, précédés du mot-clé "*VERIEXEC :*". Aucune trace n'est générée pendant le fonctionnement normal, sauf lorsque les options de compilation *CONFIG_VERIEXEC_DEBUG / DEBUG_EXTRA* ont été activées. L'échec de toute opération d'administration pour cause de privilèges insuffisants est journalisé avec la priorité *KERN_WARNING*. L'échec des opérations de vérification (erreur dans le calcul d'empreinte ou empreinte invalide) est journalisé avec la priorité *KERN_ERROR*.

5 Insertion du LSM CLIP dans le noyau

Le LSM CLIP peut être compilé aussi bien comme un module, à charger dynamiquement après l'initialisation du noyau, que comme un sous-système statique, compilé dans le même exécutable que le noyau et initialisé en même temps que celui-ci. Le code du LSM est ainsi bien isolé du code standard du noyau, et confiné à des fichiers distincts des fichiers standards. Cependant, l'intégration du module au noyau nécessite la modification de certains fichiers préexistant de ce dernier, afin en particulier :

– d'introduire des points de contrôle (*hooks*) supplémentaires dans l'interface LSM standard

- de supporter des options de montage supplémentaires
- de supporter des mécanismes spécifiques au LSM, comme la protection en écriture systématique des fichiers projetés en exécution, ou le blocage des appels *fork()* pendant l'initialisation du module.

Ces modifications sont insérées dans le noyau par un *patch clsm.patch*, qui doit être systématiquement appliqué au noyau, même lorsque le LSM CLIP est compilé en un module séparé. Elles ne sont activés que lorsque le noyau est compilé avec l'option *CONFIG_CLIP_LSM_SUPPORT*.

Par ailleurs, l'initialisation du LSM CLIP doit être adaptée selon qu'elle est réalisée en même temps que celle du reste du noyau (cas statique), ou ultérieurement (cas modulaire). On notera que, même lorsqu'il est compilé en module, le LSM CLIP ne peut pas être déchargé du noyau une fois chargé.

La présente section détaille ces différents éléments d'insertion du LSM CLIP au sein du noyau Linux.

5.1 *Hooks* ajoutés à l'interface LSM

5.1.1 *Hooks* sur les *inodes*

inode_blkdev_open(*inode, mask*)

Autorise ou non l'ouverture du *device* en mode bloc correspondant à *inode* avec les options *mask*. Cette fonction n'est appelée que pour les *block devices*, et uniquement après la vérification des droits d'ouverture standards par le *hook* LSM *inode_open()*. Elle est utilisée pour réaliser les contrôles d'accès du sous-système *devctl*.

inode_memdev_open(*inode, file*)

Fonction d'ouverture de *devices* complémentaires en mode caractère de majeur 1, appelée lorsqu'une tâche tente d'ouvrir un périphérique du numéro majeur 1 et de numéro mineur non reconnu par le noyau standard. Elle permet de supporter des *devices* supplémentaires, en mettant à jour le pointeur *file_operations* de *file* pour pointer vers le *file_operations* approprié pour le *device*, et en retournant 0, lorsque le numéro mineur de *inode* correspond à l'un des *devices* complémentaires supportés. Cette fonction renvoie l'erreur *-ENXIO* lorsque *inode* ne correspond à aucun de *devices* supportés.

Elle est à ce stade utilisée pour réaliser l'ouverture des *devices /dev/veriexec* et */dev/devctl* spécifiques au LSM CLIP, ainsi que des *devices* propres aux autres modules spécifiques à CLIP, en particulier le module *ccsd*, qui définit éventuellement un *device /dev/crandom* (cf. [CLIP 1205]). Le choix des *devices* supportés est fonction uniquement des options de compilation, la fonction ne supportant pas de méthode d'ajout dynamique de nouveaux *devices*.

inode_write_access(*inode*)

Fonction appelée (en détenant le verrou *inode->i_lock*) lorsqu'un accès en écriture est accordé sur *inode*. Cette fonction ne prend aucune décision, mais permet la mise à jour des informations de sécurité de *inode*. Elle est utilisée pour gérer la mise en cache des vérifications *veriexec* (et en particulier le retrait d'un *inode* du cache avant un accès en écriture).

5.1.2 *Hooks* sur les fichiers

file_mmap_exec(*file*)

Fonction appelée avant la projection en mémoire par *mmap()*, avec des droits autorisant l'exécution (*PROT_EXEC*), de tout ou partie du fichier *file*. Cette fonction peut autoriser ou interdire la projection, et au besoin mettre à jour les informations de sécurité et les capacités de la tâche réalisant la projection. Elle n'est appelée que pour des projections en exécution associées à un fichier (*file* est non-NULL), et uniquement lorsque la projection a été au préalable autorisée par le *hook* standard *file_mmap()*, lequel interdit notamment

dans le LSM CLIP la projection en exécution d'un fichier sur lequel l'appelant ne dispose pas des droits discrétionnaires en exécution. A la différence de *file_mmap()*, *file_mmap_exec()* n'est appelée qu'après l'interdiction de tout accès en écriture à *file*. Cette fonction est utilisée dans le LSM CLIP pour la vérification de bibliothèques par *veriexec*.

file_mprotect_exec(*vma*)

Fonction appelée avant l'attribution par *mprotect()* de droits autorisant l'exécution (*PROT_EXEC*) sur une projection mémoire *vma*. Cette fonction est l'équivalent pour *mprotect()* de ce que *file_mmap_exec()* est à *mmap()* : elle n'est appelée que sur des projections exécutables et associées et après l'interdiction de tout accès en écriture à *file*. Elle peut autoriser ou non l'appel, mais aussi mettre à jour les informations de sécurité et les capacités de la tâche appelante. Elle est utilisée par le LSM CLIP pour la vérification de bibliothèques par *veriexec*.

file_interpreter(*file*)

Fonction appelée lors du traitement d'un appel *exec()*, avant le chargement de l'interpréteur binaire *file* dans l'espace mémoire du processus appelant. Elle peut autoriser ou non le chargement, et mettre à jour les informations de sécurité et capacités de la tâche appelante. Elle est utilisée par le LSM CLIP pour la vérification d'interpréteur par *veriexec*.

file_fsignum(*file, sig*)

Fonction appelée lors du traitement d'un appel *fcntl(F_SETSIG, sig)* sur le fichier *file*. Elle peut autoriser ou non l'appel, en fonction des informations de sécurité de la tâche appelante, et au besoin allouer ou modifier l'étiquette de sécurité de *file* pour y reporter ces informations. Elle complète le contrôle réalisé par le LSM CLIP de l'envoi de signaux entre processus.

file_swapon(*file, path*)

Fonction appelée avant la création d'un *swap* sur le fichier *file* (ouvert avec le chemin *path*). Elle peut autoriser ou non l'appel, en fonction des informations de sécurité de la tâche appelante. Elle est utilisée par le LSM CLIP pour interdire la création de nouveaux *swap* une fois le *sysctl kernel.clip.mount* activé (positionné à zéro).

file_swapoff_open(*path*)

Fonction appelée pour ouvrir (en se substituant à la fonction *filp_open()* normalement utilisée) un fichier de chemin *path* avant de réaliser un appel *swapoff* sur ce fichier, et uniquement dans ce cas. Elle peut autoriser ou non l'accès. Dans le premier cas, elle doit ensuite réaliser l'ouverture proprement dite, et renvoyer la structure *struct file* correspondante.

Cette fonction permet de réaliser une ouverture de fichier avec un contrôle d'accès moins contraignant, et en particulier, dans le cas du LSM CLIP, sans vérification *devctl*, dans le cadre très spécifique d'un appel *swapoff*. Elle permet ainsi de gérer une exception, nécessaire lors de l'arrêt du système, à une configuration *devctl* qui n'autoriserait aucun accès au *device* associé au *swap*.

5.1.3 *Hooks* sur les tâches

task_ctx_migrate(*tsk*)

Fonction appelée immédiatement après la migration de la tâche *tsk* dans un nouveau contexte *vserver*. Cette fonction ne prend aucune décision, mais permet la mise à jour des informations de sécurité de la tâche courante.

task_ctx_migrated(*tsk*)

Fonction permettant de tester si la tâche *tsk* a changé de contexte *vserver* depuis son dernier appel *execve()* (ou, autrement dit, si la tâche n'a pas réalisé d'appel *execve()* depuis sa migration). Elle est utilisée dans le LSM CLIP pour autoriser spécifiquement un processus venant de migrer dans un contexte *vserver*, typiquement *vsctl* (cf. [CLIP 1202]), à tuer le processus *child_reaper* de ce contexte.

task_kill_vserver(*p, c, sig*)

Fonction permettant d'autoriser la tâche *p*, enfermée dans un contexte *vserver* non privilégié, à envoyer un signal *sig* à la tâche *c* n'appartenant pas au même contexte. Cette fonction permet dans le LSM CLIP la mise en oeuvre du privilège *CLSM_PRIV_SIGUSR*.

task_chroot(*tsk*)

Fonction appelée immédiatement avant un changement de racine *VFS* de la tâche *tsk* par un appel *chroot()*, afin de d'autoriser ou non l'appel. Elle est utilisée dans le LSM CLIP pour interdire les appels *chroot* lorsque l'appelant dispose de descripteurs de fichiers ouverts sur des répertoires.

task_chrooted(*tsk*)

Fonction permettant de tester si la tâche courante est enfermée dans une cage *chroot*. On notera qu'une tâche enfermée dans un contexte *vserver* n'est pas automatiquement considérée comme *chrootée*, même si la racine du contexte diffère de la racine du système. Par ailleurs, une tâche est automatiquement considérée comme non *chrootée* après un changement de contexte *vserver*, même si elle était *chrootée* avant la migration.

task_badness(*tsk*)

Fonction retournant un entier non nul par lequel diviser la *badness* de la tâche *tsk*, lors de la recherche de tâches à tuer par l'*out of memory killer* du noyau. Cette fonction est utilisée dans le LSM CLIP afin d'assurer une protection supplémentaire, en cas de saturation mémoire, aux tâches possédant le privilège *CLSM_PRIV_IMMORTAL*.

task_proc_pid(*tsk, buf, len*)

Fonction permettant d'afficher des informations complémentaires dans le fichier */proc/<pid>/status* correspondant à la tâche *tsk*. Les informations supplémentaires sont à écrire dans le *buffer* **buf*, avec une longueur maximal de *len*. Le pointeur *buf* doit être avancé au premier caractère suivant l'écriture avant le retour de la fonction. Cette fonction est utilisée par le LSM CLIP pour afficher les privilèges et drapeaux CLSM et *veriexec* de chaque tâche.

task_procfd(*p, c*)

Fonction permettant d'autoriser ou de refuser à la tâche *p* l'accès aux fichiers de */proc/<pid>/fd/** et */proc/<pid>/maps*, avec *<pid>* correspondant à la tâche *c*, lorsque *p* ne dispose pas de privilèges suffisant à un attachement *ptrace* à *c*. Cette fonction est utilisée dans le LSM CLIP pour gérer le privilège *CLSM_PRIV_PROCFD*.

5.1.4 *Hooks* IPsec

xfrm_policy_add(*dir, xp*)

Fonction permettant d'autoriser ou de d'interdire l'ajout d'une politique de sécurité *xp* dans la direction *dir* à la base de politiques de sécurité IPsec (SPD). Les retraits de cette même base sont contrôlés par le *hook* standard *xfrm_policy_delete_security()*. Cette fonction est utilisée dans le LSM CLIP pour la gestion du privilège *CLSM_PRIV_XFRMSP*.

xfrm_state_add(*x*)

Fonction permettant d'autoriser ou de d'interdire l'ajout ou la mise à jour d'une association de sécurité *x* à la base d'associations de sécurité IPsec (SAD). Les retraits de cette même base sont contrôlés par le *hook* standard *xfrm_state_delete_security()*. Cette fonction est utilisée dans le LSM CLIP pour la gestion du privilège *CLSM_PRIV_XFRMSA*.

5.1.5 Autres *hooks*

syslog_vserver(*type*)

Fonction permettant d'autoriser une opération *syslog* de type *type* à une tâche appartenant à un contexte *vserver* non privilégié. Lorsque la fonction autorise le traitement, l'opération est traitée comme elle le serait dans le contexte ADMIN. Dans le cas contraire, le traitement spécifique à *vserver* (c'est-à-dire l'appel de la fonction *vx_do_syslog()*, qui rend en général un code d'erreur positif sans accès réel aux journaux) est déclenché. Cette fonction est utilisée dans le LSM CLIP pour gérer le privilège *CLSM_PRIV_KSYSLOG*.

inotify_addwatch(*nd*)

Fonction appelée avant la création d'une veille *inotify* sur un fichier associé à *nd*, et permettant d'autoriser ou de refuser cette veille. Cette fonction est utilisée dans CLIP pour interdire la création de veilles *inotify* sur les fichiers appartenant à des montages *NOLOCK*.

5.2 Autres modifications des sources du noyau

5.2.1 Options de montage

Le *patch clsm.patch* ajoute deux options de montage à celles normalement supportées par le noyau :

- *MNT_NOSYMFOLLOW* : lorsque cette option est présente sur un montage, la résolution des liens symboliques est interdite au sein de ce dernier. Le traitement de cette option est réalisé directement dans la fonction de recherche de chemin (*fs/namei.c*), modifiée par *clsm.patch*, plutôt que dans un *hook* spécifique du LSM CLIP, pour des raisons de performances.
- *MNT_NOLOCK* : lorsque cette option est présente sur un montage, toute pose de verrou ou création de veille *inotify* est interdite sur les fichiers du montage. Une telle interdiction permet de supprimer des canaux de communication possibles entre deux cages qui partageraient un même montage en lecture seule (cf. [CLIP 1202]). Les contrôles correspondants sont réalisés par les *hooks file_lock()* et *inotify_addwatch()* du LSM.

5.2.2 Interdiction des accès en écriture aux projections exécutables

Le blocage des accès en écriture aux fichiers projetés en exécution par un processus est indispensable à la gestion sécurisée des niveaux de sécurité hétérogènes permise par le LSM CLIP. Elle s'accompagne d'une interdiction de projeter en exécution un fichier sur lequel l'appelant ne dispose pas des droits discrétionnaires en exécution, afin de contrer les possibilités de déni de service liées à ce traitement particulier. Ces mécanismes sont détaillés en section ??. La suppression des accès en écriture est implantée directement dans les fonctions correspondantes du noyau, *mm/mmap.c :do_mmap_pgoff()* et *mm/mprotect.c :mprotect_fixup()*, pour des raisons de performances. Le test de droits discrétionnaires en exécution est quant à lui en revanche réalisé, avant la suppression des accès en écriture, par les *hooks file_mmap()* et *file_mprotect()*.

5.2.3 Blocage temporaire des *fork()*

L'initialisation du LSM CLIP, dans le cas modulaire (cf. 5.3.2), nécessite de bloquer temporairement les appels *fork()*, afin de pouvoir labelliser l'ensemble des tâches du système. A cette fin, un sémaphore en lecture / écriture, *fork_sem*, est défini (et exporté vers les modules) dans *kernel/fork.c*, et le traitement de *fork()* est modifié de manière à prendre ce sémaphore en lecture avant de dupliquer une tâche. Ainsi, il suffit au LSM de prendre le sémaphore en écriture pour bloquer temporairement ces appels *fork()*.

5.2.4 Export de symboles supplémentaires

Les dépendances du module LSM CLIP nécessitent d'exporter vers les modules externes quelques symboles noyau normalement non exportés. Plus précisément, les symboles suivants sont exportés uniquement lorsque *CONFIG_CLIP_LSM_SUPPORT* est définie :

- *kernel/pid.c :init_pid_ns*
- *fs/namei.c :open_namei()*
- *fs/exec.c :get_task_comm()*
- *fs/open.c :nameidata_to_filp()*
- *fs/super.c :sb_lock()*
- *fs/super.c :__put_super()*
- *fs/super.c :user_get_super()*
- *fs/filesystems.c :put_filesystem()*
- *drivers/char/mem.c :mem_class*

Par ailleurs, les symboles normalement réservés aux modules sous licence GPL (exportés par *EXPORT_SYMBOL_GPL()*) sont dans ce cas exportés aussi aux modules non-GPL (dont le LSM CLIP fait partie).

5.2.5 Adaptation de *grsecurity*

Les fichiers *grsecurity* sont adaptés de telle sorte que les fonctions *grsec* de durcissement des prisons *chroot* (cf. [CLIP 1203]) déterminent la racine *VFS* hors prison *chroot* en considérant la racine du contexte *vserver* courant, plutôt que celle de *init,* ce qui rend ces tests valides dans une cage *vserver*. Cette adaptation est réalisée à l'aide d'une fonction *inline* définie directement par *clsm.patch*, plutôt que par un *hook* CLIP LSM, pour des raisons de performance. Par ailleurs, le test d'enfermement d'un processus dans une prison *chroot* est adapté de manière à appeler le *hook task_chrooted()* spécifique au LSM CLIP, plutôt que par comparaison de la racine du processus à celle de *init*.

5.3 Initialisation du LSM CLIP

L'initialisation du LSM CLIP soulève deux difficultés principales, qui se déclinent différemment selon que le LSM est compilé statiquement dans le noyau, ou comme un module séparé. D'une part, le LSM doit être initialisé relativement tôt dans la séquence d'initialisation du noyau (comme un *security_initcall()*), mais doit aussi créer des *devices* et des *sysctl* pour la configuration du module et des sous-systèmes *veriexec* et *devctl,* ce qui n'est possible que plus tard dans le démarrage du noyau. D'autre part, les *hooks* définis par le LSM sur les tâches sont écrits en faisant l'hypothèse que toute tâche du système (autre que la "tâche" *idle*) possède une étiquette de sécurité valide (cf. 2.1.1). L'approche inverse, consistant à tester dans chaque *hook* la définition de toutes les étiquettes de sécurité, a été écartée car elle aurait conduit à une complexité excessive du code. De ce fait, il est important de s'assurer, avant la mise en place des *hooks*, que toutes les tâches du système sont labellisées.

5.3.1 Cas de la compilation statique

Dans le cas d'une compilation statique, le LSM est entièrement initialisé avant la création de la première tâche du système, ce qui assure trivialement la labellisation des tâches. En revanche, la création des *devices* CLIP-LSM est dans ce cas problématique, dans la mesure où l'initialisation en *security_initcall()* du LSM est réalisée avant celle de la classe des *character devices* et de la table globale de *sysctl*. Ce problème est traité dans ce cas par l'éclatement en deux appels de l'initialisation du LSM. Un premier appel, *clsm_init()*, appelé comme un *security_initcall()*, assure l'initialisation du module à l'exception de ses interfaces de configuration, c'est-à-dire, dans cet ordre :

- L'initialisation des variables *sysctl* du module (sans les exposer dans la table de *sysctl*).
- La création des caches mémoire (*kmem_cache*) associés aux différents types d'étiquettes de sécurité.
- L'initialisation des sous-systèmes *veriexec* et *devctl*, consistant principalement en la création de leurs propres caches mémoire.
- L'enregistrement du module comme LSM, ce qui positionne les différents *hooks*, par un appel *register_security()* (en tant que *"primary module"* uniquement, l'appel échoue si un autre module LSM est déjà présent).

Une deuxième fonction d'initialisation, *clsm_device_init()*, est ensuite appelée, comme un *late_initcall()*, c'est-à-dire après l'initialisation des classes de *devices* et de la table de *sysctl*, pour :

- Insérer les variables *sysctl* CLIP-LSM dans la table *sysctl* globale.
- Créer les *devices veriexec* et *devctl*.

5.3.2 Cas de la compilation modulaire

Dans le cas où CLIP LSM est compilé comme un module externe, son chargement n'est possible qu'après l'initialisation complète du noyau. Ainsi, le problème d'initialisation des *devices* et *sysctl* du module ne se pose pas dans ce cas. En revanche, le système comporte un nombre arbitraire de tâches actives non labellisées au moment du chargement du module. Il est donc nécessaire de procéder à leur labellisation avant l'insertion des *hooks* CLIP dans l'interface LSM.

L'initialisation du module est dans ce cas réalisée par une fonction unique, *clsm_module_init()*, qui réalise les traitements suivants dans cet ordre :

- Traitement de *clsm_init()*, à l'exception de l'enregistrement des *hooks*, c'est-à-dire initialisation des variables *sysctl* et des caches mémoire.
- Labellisation de toutes les tâches existantes : les tâches du système sont parcourues par une boucle *for_each_process()*, et une étiquette vide (tous champs à zéro) est allouée pour chaque tâche. Les *fork()* sont interdits pendant cette boucle, par la prise du sémaphore *fork_sem* (cf. 5.2.3) en écriture, afin d'éviter la création de nouvelles tâches non labellisées pendant l'exécution de la boucle.
- Insertion des *hooks* du module (en tant que *primary module"* uniquement, l'appel échoue si un autre module LSM est déjà présent) par un appel *register_security()*.
- Création des *devices* et fichiers *sysctl* par un appel à *clsm_device_init()*.

Par ailleurs, et comme évoqué en préambule de cette section, le module CLIP-LSM ne peut à ce stade pas être retiré avant l'arrêt du système. Ainsi, la fonction *module_exit()* du module appelle directement *BUG()* pour interrompre le traitement et remonter une erreur.

6 Utilitaires *verictl* et *devctl*

L'utilitaire *verictl* est une application en couche utilisateur destinée à faciliter l'administration du sous-système *veriexec* en réalisant les différents *ioctl* sur le *device /dev/veriexec*. De manière similaire, *devctl* permet l'administration de la base de permissions sur les *block devices*, en réalisant des *ioctl* sur */dev/devctl*. Ces deux utilitaires sont installés par le paquetage *app-clip/verictl*.

6.1 Utilitaire *verictl*

Le prototype d'une ligne de commande *verictl* est le suivant :

```
verictl [-dDehlmpuxXy] [--f <fichier>] [-c <argument>]
        [--L <niveau>] [--U <contexte>
```

Les différentes options de la ligne de commande permettent de réaliser les opérations suivantes.

6.1.1 Ajout / Suppression d'entrées

- **--l** : ajout d'une ou plusieurs entrées.
- **--u** : suppression d'une ou plusieurs entrées.

La ou les entrées affectées par ces commandes sont définies soit directement sur la ligne de commande avec l'option **--c <entrée>**, avec <entrée> une définition d'entrée au format décrit ci-dessous, soit dans un fichier avec l'option **--f <fichier>**, avec <fichier> le chemin vers un fichier contenant une ou plusieurs définitions d'entrées, une par ligne. Dans les deux cas, le format d'une ligne de définition d'entrée est le suivant (tous les champs sur une même ligne, séparés par un ou plusieurs espaces ou tabulations) :

```
<fichier> <ctx> <options> <cap_eff> <cap_perm> <cap_inh> \
                    <privs> <fonction> <empreinte>
```

avec :

- **<fichier>** le chemin complet du fichier pour lequel l'entrée doit être créée (ou plus exactement, d'un lien dur associé à l'*inode* auquel l'entrée doit être associée).
- **<ctx>** le contexte dans lequel doit être créée l'entrée (valeur numérique), ou --1 pour utiliser le contexte dans lequel *verictl* est invoqué.
- **<options>** les options *veriexec* associées à l'entrée, concaténation de lettres-clés telles que décrites dans le tableau 12.
- **<cap_eff>**, **<cap_perm>**, **<cap_inh>** les masques de capacités effectif, permis et héritable associés à l'entrée, sous forme numérique (décimale, octale ou hexadécimale, selon les conventions du langage C)
- **<privs>** les privilèges CLSM associés à l'entrée, concaténation de lettres clés telles que décrites dans le tableau 13.
- **<fonction>** la fonction de hachage utilisée pour générer l'empreinte cryptographique. Les valeurs possibles sont **md5**, **sha1**, **sha256** et **ccsd**.
- **<empreinte>** l'empreinte cryptographique du fichier, sous forme de chaîne hexadécimale (deux caractères de *0* à *9* ou *a* à *f* pour chaque octet).

Lorsqu'un fichier de déclaration d'entrées est passé en ligne de commande par **--f <fichier>**, toutes les lignes de déclaration sont lues et parsées, et les informations correspondantes stockées dans une liste chaînée allouée sur le tas du processus, avant d'ouvrir */dev/veriexec* et d'enchaîner les commandes *ioctl* correspondantes. Le traitement est interrompu à la première erreur d'analyse du fichier de configuration, ou au premier code de retour non nul d'un *ioctl*. On notera que le passage de `**--f** -' dans les options entraîne la lecture de ces lignes sur l'entrée standard de *verictl*, ce qui permet par exemple de concaténer plusieurs fichiers de déclaration avant de les passer à *verictl*, afin d'optimiser les accès.

6.1.2 Lecture / Modification de niveau

- **--p** : affichage du niveau (valeur numérique) du contexte courant.
- **--e** : activation (positionnement de *VRXLVL_ACTIVE*) de *veriexec* dans les contexte courant.
- **--d** : désactivation (remise du niveau à zéro) de *veriexec* dans le contexte courant.
- **--L <niveau>** : modification générique du niveau.

Dans ce dernier cas, le paramètre *<niveau>* doit être de la forme suivante :

```
[<ctx>-]<mot-clé-1> :<mot-clé-2> :... :<mot-clé-n>
```

Option *veriexec*	Lettre-clé *verictl*
0	-
VRX_FLAG_EXE	e
VRX_FLAG_LIB	l
VRX_FLAG_NEEDROOT	r
VRX_FLAG_NEEDLIB	N
VRX_FLAG_CHECKLIB	L
VRX_FLAG_INHERIT	I
VRX_FLAG_SCRIPT	s

TABLE 12 – Correspondance entre options *veriexec* et lettres-clés *verictl*

Privilège CLSM	Lettre-clé *verictl*
0	-
CLSM_PRIV_CHROOT	C
CLSM_PRIV_VERICTL	V
CLSM_PRIV_NETCLIENT	c
CLSM_PRIV_NETSERVER	s
CLSM_PRIV_NETOTHER	n
CLSM_PRIV_PROCFD	P
CLSM_PRIV_SIGUSR	S
CLSM_PRIV_RECVSIG	r
CLSM_PRIV_NETLINK	N
CLSM_PRIV_KSYSLOG	k
CLSM_PRIV_IMMORTAL	I
CLSM_PRIV_KEEPPRIV	K

TABLE 13 – Correspondance entre privilèges CLSM et lettres-clés *verictl*

avec :

- **<ctx>** un numéro de contexte optionnel. Lorsqu'un tel numéro est spécifié, c'est au contexte correspondant que s'applique le changement de niveau. Sinon, ce changement porte sur le contexte courant.
- **<mot-clé-1>**… **<mot-clé-n>** une suite de mots-clés de définition de niveau, tels que définis dans le tableau 14, séparés par des ` :'.

6.1.3 Ajout / Suppression / Modification d'un contexte

- **--x** : ajout d'un contexte.
- **--X** : suppression d'un contexte
- **--y** : modification d'un contexte

Dans les trois cas, il est nécessaire de passer un argument par **--c <argument>**, avec *<argument>* de la forme suivante :

```
<ctx> <niveau> <masque capacités> <masque privileges>
```

avec :

- **<ctx>** le numéro du contexte.
- **<niveau>** le niveau initial du contexte, sous la forme d'une suite de mots-clés parmi ceux définis dans le tableau 14, séparés par des ` :'. Ce champ n'est pris en compte que par **--x**, sa valeur est ignorée par les deux autres commandes.

Niveau *veriexec*	Mot-clé *verictl*
0	*inactive*
VRXLVL_ACTIVE	*active*
VRXLVL_LVL_IMMUTABLE	*lvl_immutable*
VRXLVL_SELF_IMMUTABLE	*self_immutable*
VRXLVL_ADMIN_IMMUTABLE	*admin_immutable*
VRXLVL_UPDATE_IMMUTABLE	*update_immutable*
VRXLVL_CTX_IMMUTABLE	*ctx_immutable*
VRXLVL_CTXSET_IMMUTABLE	*ctxset_immutable*
VRXLVL_ENFORCE_MNTRO	*enforce_mntro*

TABLE 14 – Correspondance entre niveaux veriexec et mots-clés *veriexec*

- **<masque capacités>** le masque maximal de capacités, valeur numérique (selon les conventions du langage C). Ce masque est ignoré par **--X**. Pour **--x** il représente le masque initial du contexte (qui doit être inférieur au masque du contexte ADMIN) ; pour **--y** il s'agit du nouveau masque (qui est intersecté avec l'ancien).
- **<masque privilèges>** le masque maximal de privilèges CLSM, concaténation de lettres-clés tels que décrites dans le tableau 13. Ce masque est ignoré par **--X**. Pour **--x** il représente le masque initial du contexte (qui doit être inférieur au masque du contexte ADMIN) ; pour **--y** il s'agit du nouveau masque (qui est intersecté avec l'ancien).

6.1.4 Autres opérations

- **--U <contexte>** : définition du contexte <*contexte*> (numéro de contexte) comme contexte UPDATE.
- **--D** : debug. Combiné à n'importe quelles autres options, remplace l'ouverture de */dev/veriexec* et les différents *ioctl* par l'affichage sur la sortie de standard de descriptifs de ces opérations.
- **--m** : affiche le nombre d'entrées *veriexec* présentement allouées tous contextes confondus sur la sortie standard (si le noyau est compilé avec l'option *CONFIG_VERIEXEC_DEBUG_MEMLEAK*).
- **--h** : affiche une aide synthétique sur la sortie standard et retourne immédiatement.
- **--v** : affiche la version du paquetage *verictl* sur la sortie standard et retourne immédiatement.

6.2 Utilitaire *devctl*

Le prototype d'une ligne de commande *devctl* est le suivant :

```
devctl [-uvh] [-c <device>]
```

Les options supportées sont :

- **--c <device>** : traiter l'entrée définie par *<device>*. Le comportement par défaut est d'ajouter cette entrée, sauf si *--u* est passé sur la ligne de commande. Le format de *<device>* est :

  ```
  <maj> <min> <larg> <prio> <perms>
  ```

  avec :
  - **<maj>**, **<min>**, **<larg>** et **<prio>** les majeur, mineur, largeur et priorité, respectivement, de l'entrée, sous la forme d'entiers notés selon les conventions du langage C.
  - **<perms>** le masque de permissions associé, sous la forme d'une chaîne de lettres-clés, telles que décrites dans le tableau 15.
- **--u** : supprime l'entrée définie à l'aide de *--c*, au lieu de l'ajouter
- **--h** : affiche une aide synthétique sur la sortie standard et retourne immédiatement.
- **--v** : affiche la version du paquetage *verictl* sur la sortie standard et retourne immédiatement.

Option *devctl*	Lettre-clé *devctl*
DEVCTL_PERM_NONE	-
DEVCTL_PERM _RO	r
DEVCTL_PERM _RW	w
DEVCTL_PERM _EXEC	x
DEVCTL_PERM _SUID	s
DEVCTL_PERM _DEV	d

TABLE 15 – Lettres-clés reconnues par l'utilitaire *devctl*

Références

[CLIP 1202] Documentation CLIP, 1202, *Patch Vserver*, ANSSI.

[CLIP 1203] Documentation CLIP, 1203, *Patch Grsecurity*, ANSSI.

[CLIP 1204] Documentation CLIP, 1204, *Privilèges Linux*, ANSSI.

[CLIP 1205] Documentation CLIP, 1205, *Intégration de CCSD en couche noyau*, ANSSI.

[LDD] *Linux Device Drivers, 3rd Edition.* http://lwn.net/images/pdf/LDD3/.

[PAX] *PaX.* http://pax.grsecurity.net.