MEVS – SELinux

Catalin Dima

Contenu du cours – partie SELinux

- Pourquoi SELinux ?
- Architecture SELinux.
- Développement de politiques de sécurité en SELinux.
- Introduction à NuSMV.
- Analyse en NuSMV de politiques de sécurité développées en SELinux.

Évaluation (partie SELinux) :

- Sujet SELinux (créer/modifier un module, commandes, logs/messages d'erreur, etc.) : 40%.
- Sujet modélisation NuSMV (dont modélisation politique SELinux) : 60%.

Rappels de MAC et de RBAC

- Matrice de contrôle d'accès : $M : S \times O \to A$,
  - S = sujets, O = objets, A = attributs/accès.
  - Serveur de sécurité utilisant M.
  - Commandes pour modifier M (à la Harrison-Ruzzo-Ullman).
- Système typé :
  - $t : O \to Types$.
  - Matrice sur les types : $M : Types \times Types \to A$.
  - Commandes typées.
- RBAC : politique orientée utilisateur (et non pas processus).
  - Utilisateur $\to$ roles.
  - Matrice de contrôle d'accès $M : Roles \times O \to A$.

Linux, DAC et MAC

- Limitation du DAC en Linux :
  - Cible des Chevaux de Troie (scripts), par les moyens des commandes `setuid/setgid`.
  - Décisions d'autoriser l'accès prises sur l'*identité* du sujet et du *propriétaire* de l'objet.
  - Seulement deux grandes catégories d'utilisateurs : *administrateurs* et *autres*.
  - Un service qui nécessite un privilège quelconque se verra attribué *tous les privilèges*.
- Deux principes de sécurité essentiels :
  1. **Plus petit privilège** : un sujet devrait se voir attribuer seulement les privilèges *qui lui soient nécessaires* pour compléter son travail.
  2. **Séparation des tâches** : le travail d'un sujet sert de vérification complémentaire pour le travail d'un autre.
- MAC permettrait de *renforcer* ces deux principes :
  - Définition de permissions gouvernant les intéractions *sujets/objets*.
  - Décisions de sécurité prises selon toute l'information disponible sur les sujets/objets.
  - Exemple : `chmod`.

MAC en SELinux

- **Renforcement de type** :
  - Chaque sujet et objet est associé à un *type*.
  - Sujets (processus) : *type domain*.
  - Objets (programmes/fichiers/dev/sockets/ports) : *type*.
- **Contrôle basé sur les rôles** (RBAC) :
  - Utilisateurs : essentiellement les identités dans Linux (utilisateurs, démons).
  - Rôles : jouent le rôle des relations entre utilisateurs et types (domaines).
- **Sécurité multi-niveau** :
  - Prévention de la fuite d'information : *no read-up, no write-down*.
  - Hiérarchie des rôles, transitions entre rôles.

Qu'est-ce que SELinux ?

- Implémentation de mécanismes de *contrôle d'accès obligatoire* = MAC (Mandatory Access Control).
- Intervient après le contrôle d'accès *discrétionnaire* (DAC) de Linux (attributs).
- Mécanisme présent dans le noyau Linux.
  - Utilisant le cadre des LSM (Linux Security Modules).
- Basé sur l'architecture dite *Flask* :
  - Architecture générale de MAC.
  - Support flexible pour les politiques de sécurité.
  - Projet de longue date d'intégration d'une architecture de sécurite dans un système d'exploitation (Fluke).

A quoi sert SELinux ?

- On a un programme (logiciel) P contenant une faille (ex. *stack overflow*).
- Un attaquant est capable d'interagir avec P (web, cgi scripts, etc.).
- L'attaquant est capable d'attaquer le programme P, en contournant toute protection contre les stack overflows.
- L'attaque permet de lancer un shell.

SELinux peut prévenir cela !

- On définit un *domaine* D_P pour P, et un autre (distinct/disjoint !) D_S pour le shell.
- On décrit des règles SELinux qui interdisent à tout D_P de `fork`er ou `exec`-uter !) des D_S.
- Tout essai de lancer un shell à partir du domaine P sera interdit, et *audité* !

Cela peut représenter un mécanisme de détection d'intrusion.

Concepts SELinux

- *Type* : comme dans les systèmes de contrôle d'accès typés.
- *Domaine* : type pour les programmes (= "sujets").
- *Contexte de sécurité* : étiquette de sécurité associé à un sujet/objet (utilisateur, role, type, éventuellement deux autres étiquettes MLS/MCS).

  `root:sysadm_r:sysadm_t:s0:c0.c128`
- *Moniteur de sécurité* (RM, Reference Monitor) : décide des permissions accordées aux opérations, sur la base des informations fournies par le PMS.
- *Access Vector Cache* : mémoire locale fournissant des décisions rapides sur les permissions, se basant sur des décisions antérieures du RM.

Interaction SELinux / Système d'Exploitation

1. Sujet essaie d'effectuer un appel système impliquant un objet.
   - Tout d'abord, le contrôle d'accès Linux intervient et prend une décision.
   - Si DAC Linux autorise l'accès, alors le SELinux intervient à son tour.
2. Les contextes de sécurité du sujet et de l'objet entrent en jeu !
3. On cherche d'abord une règle dans l'AVC dans laquelle les deux contextes de sécurité interviennent.
4. Si solution trouvée dans l'AVC, *permission accordée/refusée*.
5. Sinon on appelle le RM pour chercher la solution dans la politique binaire, chargée dans le noyau du système, qui renvoie *permission accordée/interdite*
6. AVC retient la solution pour utilisation ultérieure.
7. Les permissions interdites sont journalisées dans `/var/log/...`

Appels système

- `open/read/write/...` impliquant un processus et un fichier/répertoire ou le système de fichiers.
- `fork/exec`.
- `exit/kill/waitpid/shmat/mmap...` et autres communications/interactions entre processus.
- `setuid` et autres opérations de contrôle d'accès.
- Etc. etc...
- Un appel système peut impliquer plusieurs actions, chaque action élémentaire doit se voir accorder la permission d'exécution par le RM.

Contexte de sécurité

- **LA** notion permettant de définir les règles d'accès.
- Ensemble de propriétés définissant un certain domaine de sécurité.
  - Utilisateur, rôle, domaine/type, étiquette MLS/MCS.
- E.g. `system_u:system_r:proc_t` pour les processus.
- E.g. `user_u:object_r:tmp_t` pour les fichiers.
- Implémenté par les attributs étendus (xattrs) de Linux.
  - Associés avec chaque sujet/objet.
  - Mais aussi peut être associé avec un système de fichiers entier (CD).
- Supplémentaires aux attributs Linux :
  - Les attributs de processus (ID réel et effectif d'utilisateur et de groupe) continuent d'exister.
  - Les attributs de fichiers (modes d'accès, autres attributs `d`, `l`, `p`...) continuent d'exister.

Éléments constitutifs pour les règles de controle d'accès

- Classes d'objets = les cibles des permissions.
- Permissions = comme dans les modèles de contrôle d'accès.
  - Macros de permissions.
- Types = permettent de définir des ensembles des sujets ou des objets représentant un domaine de sécurité.
  - Regroupables en attributs, peuvent avoir des alias.

Classes d'objets

- Servent à identifier la cible des permissions.
- Spécifiées en terme de permissions d'accès des sujets aux objets.
  - Sujets : processus,
  - Objets : 40 classes différentes d'objets dans le noyau.
  - Permissions : chaque classe vient avec sa spécification de permissions.
- Rarement besoin de les modifier.
  - Seulement quand les classes d'objets changent dans l'administrateur d'objets dans le noyau.

Classes d'objets

- Classes relatives aux fichiers :
  - `file, dir, fd, lnk_file, chr_file, blk_file, sock_file, fifo_file, filesystem`.
- Classes relatives au réseau :
  - `node, netif, tcp_socket, udp_socket, rawip_socket`...
- Classes spécifiques aux protocôles :
  - `netlink_route_socket, netlink_ip6fw_socket,netlink_tcpdiag_socket, netlink_selinux_socket`...
- Classes spécifiques aux IPC :
  - `sem, msgq, msg, shm`.
- Classes spéciales :
  - `process, security, system, capability`.

Permissions

- Toute classe d'objets possède des permissions, dénommées **vecteur d'accès** (AV).
- Définitions dans `/etc/selinux/[nompol]/src/policy/flask/access_vectors`.
- Identificateurs de permissions définis de deux façons :
  - Déclaration *commune* – utile pour ne pas redéclarer les permissions dans chaque classe les incluant.
  - Déclaration de classe *spécifique*.
- Les concepteurs de politiques changent rarement les AV.
  - Comme les classes d'objets, sont codés en dur dans le noyau.
  - Certaines *actions* demandent plusieurs permissions pour plusieurs classes d'objets à la fois.

Permissions communes

- Déclarées avec le mot-clé `common`.
- Définissent un groupe d'identificateurs de permissions,
- ...associées avec une classe d'objets en tant que groupe.
- Affectées aux classes d'objets via des déclarations de 2e type.

```
common file
{
    ioctl
    read
    write
    create
....
}
```

Permissions pour les classes d'objets

- Affectent des permissions pour chaque objet.
- Peut assigner des permissions communes ou spécifiques à la classe :

```
class file
inherits file
{
    execute_no_trans
    entrypoint
    execmod
}
```

  - Héritage : classe `file` hérite de l'ensemble de permissions `file`.

Quelques permissions

- Permissions d'un processus sur un fichier :
  - `read, write, append, create, getattr, setattr, ioctl, unlink, link, lock`,
  - `execute` : executer le fichier.
  - `execute_no_trans` :executer le fichier sans transition de domaine (on verra).
  - `entrypoint` : entrer dans un nouveau domaine par ce programme.
  - `execmod` : contrôler la mémoire pendant l'exécution par des fichiers projetés en mémoire.
- Permissions d'un processus sur un autre processus :
  - `fork, sigchld, sigkill,..., ptrace, getsched, setsched, getsession, getpid, ... , setpgid`.
  - `transition` : changer le contexte de sécurité via `exec` – implique le contexte précédent et le nouveau contexte.
  - `dyntransition` : changer le contexte de sécurité dans une autre situation que `exec`.
- `dyntransition` – **pas de propriété de tranquilité** !

Définitions de permissions

- Macros de permissions : `.../policy/macros`.
  - Définies avec le langage de processeur de macros `m4`.
- Qqs exemples :
  - `rx_file_perms` : permissions pour lire et exécuter un fichier.
    - `{ read getattr lock execute ioctl }`
  - `r_dir_perms` : permissions pour traverser un répertoire.
    - `{ read getattr lock search ioctl }`
  - `create_dir_perms` : crée et utilise un nouveau répertoire.
    - `{ create read getattr lock setattr link unlink rename search add_name remove_name reparent write rmdir }`

Règles de renforcement de types (TE)

- La partie la plus importante dans SELinux :
  - Permet de spécifier *tous* les accès permis.
  - Aucun accès ne peut réussir sans qu'une règle TE le permette.
- Forme la majeure partie de la politique SELinux :
  - Pas extraordinaire d'avoir des dizaines de milliers de règles.
- Trois composants majeurs :
  1. Déclarations : types, attributs et alias.
  2. Règles AV : `allow`, `dontallow`, `auditallow`, `neverallow`.
  3. Règles de transition de type : `type_transition`, `type_change`, `type_member`.
- TE est ciblé sur les *programmes*, pas sur les utilisateurs (comme le DAC).

Types

- Type = Identificateur non-ambigu de sécurité associé à chaque sujet/objet.
  - Permettant de grouper les sujets ou objets dans des classes d'équivalence.
  - Créé par le concepteur de la politique.
  - Tous les sujet ou objets ayant le même type auront les mêmes accès.
- Le sens d'un type est créé par son utilisation
  - Les règles de la politique donnent un sens aux types.
  - E.g. `shadow_t, user_home_t,...`
- Interviennent dans les décisions de contrôle d'accès :
  - Types associé au sujet et à l'objet impliqué dans une décision d'accès.
- Règles de *contrôle d'accès pour le renforcement de type*.
  - Similaires aux **matrices de contrôle d'accès**.
  - À la différence des classes/permissions, les types ne sont pas codés dans le noyau !

Vue d'ensemble des types, attributs et alias

- *Types* : identificateurs uniques associés aux sujets et objets.
  - Souvent, on parle de *domaine* = type d'un *processus* (sujet), à la différence de types banaux, associés aux objets.
- *Attributs* et *alias* : employés pour simplifier l'utilisation des types.
  - *Attribut* : regouper des types.
  - *Alias* : autre nom pour un type.
- Types et alias utilisés dans les
  - Règles de renforcement de types (Type Enforcement, TE).
  - Contextes, étiquettage de déclarations.
- Attributs utilisés *seulement* dans les règles TE.
- SELinux *ne possède pas* de types, attributs, alias *prédéfinis* (à la différence des classes d'objets et de permissions !).

Déclarations de types

- `type load_policy_exec_t, file_type, exec_type, sysadmfile;`
  - Déclaration de type `load_policy_exec_t`, ayant les attributs `file_type, exec_type, sysadmfile`.
  - Même espace de noms pour les types, attributs, alias !
  - Alias optionnel, devant les attributs :
    - `type load_policy_exec_t alias lpex, file_type, exec_type, sysadmfile;`
  - Attributs optionnels eux-aussi.
- `typealias load_policy_exec_t alias { lpex mapolitique };`
  - Pour définir un alias pour un type déjà déclaré.
- Déclarations de types – dans les fichiers `*.te`.

Attributs de types

- Utilisés pour mieux gérer les politiques complexes.
  - Raison similaire à l'introduction des classes d'objets + permissions.
- Peuvent être utilisés dans les règles de renforcement de politiques.
- Déclarations :
  - `attribute file_type;`.
  - Seulement une réservation de nom.
  - Une fois déclaré, peut servir pour associer des types.
- On ne peut pas utiliser des attributs dans les contextes !
- Associer des attributs aux types : `typeattribute tmp_t file_type;`
- Autre possibilité : `type tmp_t, file_type;`
  - Associer un ou plusieurs attributs à la déclaration de type.

Conventions communes type/attribut

- Identificateurs de types se terminent en `_t`.
- Noms d'attributs = mots décrivant le groupe.
- Types en relation partagent la racine du nom : `user_t, user_tmp_t`.
- Les types des fichiers exécutables : suffixe `_exec_t`.
  - Exécutables qui peuvent provoquer une transition de domaine.
- Exemples d'attributs :
  - `domain` = affecté à tous les domaines par convention.
  - `file_type` = affectés à tous les types associés aux objets de type fichiers.
  - `auth_chkpwd` = pour tous les domaines dont l'intention est d'authentifier des utilisateurs.
  - `privlog` = communication avec `syslog`.
- Exemple :
  - `type passwd_t, domain, privlog, auth, privowner;.`
  - `type passwd_exec_t, file_type, sysadmfile, exec_type;.`

Règles de TE

- Spécifient les relations entre types.
- Deux types de règles :
  - Règles d'accès (access vector rules, AVR) $\sim$ 50000.
  - Règles de transition de type $\sim$ 2000.
- But primaire : contrôle d'accès pour les programmes.
  - Règles de type $\sim$ matrice de contrôle d'accès !

Règles de TE

- Implémentation des politiques pour renforcer :
  - La protection des programmes.
  - La protection du noyau et de ses ressources.
  - La protection de la politique elle-même.
  - Le principe du plus petit privilège.
  - La limitation de la propagation d'erreurs (confinés dans le domaine de la source).
  - Autres propriétés de sécurité : flux d'information.
- Règles ciblées sur les accès domaine, et non pas sur les utilisateurs.
- Différence avec les politiques RBAC :
  - En RBAC ce sont les *rôles* qui associent des permissions aux processus.
  - En TE, ce sont les *domaines*, indépendants des utilisateurs, qui associent des permissions aux processus.

Règles d'accès

- Par défaut, **accès interdit** !
- Quatre règles :
  1. `allow` : accès permis.
  2. `auditallow` : accès permis, mais enregistrement de l'accès.
  3. `dontaudit` : pas d'enregistrement en cas d'accès interdit.
  4. `neverallow` : c'est une assertion, utilisée au moment de la compilation de la politique avec `checkpolicy` en tant qu'invariant.
- Syntaxe :
  `nomderegle typesrc typedest : classes permissions;`
  - Accorde à un processus de type `typesrc` le droit d'accéder à un objet de type `typedest`, si l'objet est dans une des `classes` et l'action requiert une des `permissions` déclarées dans la règle.
  - Granularité des spécifications d'accès :
    - Classes & permissions.
  - Utilisation d'alias et d'attributs pour identifier les domaines (types) source.

Exemples

```
allow auth shadow_t : file { getattr read };
allow { file_type device_type ttyfile } fs_t : filesystem associate;
allow user_t bin_t : file { read getattr execute_no_trans };
allow user_t self : process *;
allow userdomain shell_exec_t : file { read getattr lock execute ioctl }
dontaudit named_t root_t : file { getattr read };
auditallow rshd_t security_t : security { load_policy setenforce setbool };
neverallow domain ~domain : process { transition dyntransition };
```

- On peut utiliser `*` et `~`.
- Si plusieurs règles sur la même paire `typesrc/typedest`, alors *union des permissions*.
- Règles `neverallow` – dans `assert.te`.

Petit resumé

- Chaque processus possède un domaine (type).
  - Y compris `bash/firefox/httpd` ou votre commande/processus ls/a.out lancés en ligne de commande !
- Chaque autre objet dans le système d'exploitation possède un type.
  - Y compris les fichiers se trouvant sur les supports amovibles !
- L'accès d'un processus à un fichier se fait seulement lorsque le RM trouve une règle de type `allow/dontaudit/auditallow` qui donne une permission au domaine source d'avoir accès au type selon l'opération désirée.
- Le même processus est enclenché lorsqu'un processus veut exécuter une opération impliquant un autre processus, ou le processus même.
  - Y compris lors des `fork/exec` et les autres interactions/communications interprocessus, ou autre `setuid` !

La suite : qu'est-ce qui se passe lorsqu'on crée de nouveaux processus/fichiers/objets ?...

Transition de types en SELinux

- Spécifient le type par défaut pour un nouvel objet !
- Deux formes :
  1. Type par défaut associé au moment d'un `execv()` pour un nouveau processus.
  2. Type par défaut associé à un nouvel objet de type fichier.
- Ne représentent pas une permission !
- Plusieurs règles de TE doivent contribuer pour permettre la transition :
  - Le domaine original devrait avoir accès au fichier exécutable.
  - Le domain original devrait avoir la permission de faire une transition au nouveau domaine.
  - Le nouveau domaine devrait se voir accorder la permission d'entrer via un programme.
  - Autres...

Syntaxe

`type_transition type_src type_dest : classes type_par_defaut;`

- `type_src` et `type_dest` : un ou plusieurs types/attributs/alias.
- `type_par_defaut` : un seul type.
- `classes = process` : transition de type processus.
  - Si un processus dans le domaine `type_src` lance en `exec`ution un programme de type `type_dest`, le processus fils sera mis dans le domaine `type_par_defaut`.
  - Si tous les accès sont permis par les règles de TE.
- `classes = file` (ou autre classe de fichier) : transition de création de fichier.
  - Si un processus dans le domaine `type_src` crée un objet de classe fichier *dans un répertoire* de type `type_dest`, le nouveau fichier se verra attribuer le type `type_par_defaut`.
  - Si tous les accès sont permis par les règles de TE.

Exemples :

- `type_transition userdomain passwd_exec_t : process passwd_t`
  - Processus dans un domaine ayant l'attribut `userdomain` veut lancer en exécution programme dans le domaine `passwd_exec_t`
  - Si les règles de TE le permettent, le fils sera lancé et mis par défaut dans le domaine `passwd_t`.

`type_transition` et règles `allow` correspondantes

```
bash$
execlp("monprog.out","/home/dima/monprog.out",NULL)
```

- Le programme courant tourne dans le domaine `shell_t`.
  - Par exemple, du fait qu'on lance `monprog.out` dans un shell.
  - Donc il y a un `fork` qui précède le `exec`!!
- Mais dans la conception de la politique de sécurité du système, on veut que `"monprog.out"` tourne dans son propre domaine (`mondom_t`).
  - La permission `transition`!
- L'appel système `exec` doit impliquer un accès du domaine `shell_t` au domaine `mondom_t`.
- D'autre part, pour charger le code de `monprog.out`, il faut lire un fichier.
  - Donc, interdiction d'exec sans avoir des permissions `read/getattr/execute` du domaine `shell_t` sur le type du fichier (disons, `monprog_exec_t`.
- Enfin, un dernier droit particulier : permettre de lancer un programme qui tourne dans un domaine à partir de fichiers ayant un type particulier.
  - Permission `entrypoint`.
- Pour éviter que, par exemple, un `firefox` attaqué lance un `shell`!

Exemples

- Exemple de combinaison de règles de TE et de transition de types pour le lancement de programmes à partir d'un shell :

```
type_transition shell_t monprog_exec_t : process monprog_t;
allow shell_t monprog_exec_t : file { read getattr execute };
allow monprog_t monprog_exec_t : file entrypoint;
allow shell_t monprog_t : process transition;
```

  Ces règles s'appliquent pour l'`exec`, mais pas pour le `fork` !

  - Tout programme pour lequel on ne donne pas de règle particulière sera lancé dans le même domaine que celui de son père.

- Exemple de combinaison de règles de TE et de transition de types pour la création de fichiers :

```
attribute privhome;
type unconfined_t, .... privhome;
type_transition privhome user_home_dir_t:
        { file lnk_file ....} user_home_t;
allow privhome user_home_dir_t:dir { ... ioctl write };
allow privhome user_home_t:file { create ioctl append write ... };
```

Contextes de sécurité et étiquettage

- On sait (presque...) comment on étiquette les “sujets”
  - Chaque lancement en exécution d’un processus est accompagné d’un étiquettage du nouveau-né (`allow, type_transition`...)
  - Et `init` reçoit un contexte par défaut.
- Les règles étudiées jusqu’à maintenant spécifient comment associer des contextes aux *nouveaux* objets.

```
type_transition user_t user_home_dir_t : file user_home_t;
```

  - Le rôle est hérité du rôle du processus créateur.
  - Utilisateur `user_u`.
- Mais comment on étiquette de manière plus automatique des répertoires/systèmes de fichiers entiers ?
  - Sûrement utile lors de l’installation du SELinux !
  - Mais aussi lors du montage, ou de toute modification d’une politique existante.

Étiquettage des objets

- Pour des raisons historiques, SELinux définit des **SIDs** (identificateurs initiaux) pour chaque objet.
  - À chercher dans `.../src/policy/flask/initial_sids`.
  - Chaque SID est associé à un contexte de sécurité par défaut.
  - Fichiers : `.../src/policy/initial_sid_contexts`

Autres types d'étiquettage :

- Étiquettage par défaut.
- Étiquettage par les règles de la politique.
- Étiquettage *à la demande* :
  - Pour des systèmes de fichiers qu'on veut étiqueter la première fois en SELinux.
  - Pour réparer.

Étiquetage à la demande

- Au montage : `mount -o context=user_u:object_r:user_home_t`.
  - Autres options : `fscontext, defcontext`.
- À n'importe quel instant : `chcon -u -r -t -l`
- Réétiquetage :
  - `setfiles, fixfiles` (script !), `restorecon, genhomedircon, matchpathcon`.
  - Réétiquetage selon les règles d'étiquetage décrites dans la politique (on va voir !).
- Au lancement en exécution :
  - `newrole -r -t -l` – lancement d'un shell avec un nouveaur rôle/type.
  - `runcon -u -r -t -l` – lancement en exécution d'un nouveau processus avec un contexte spécifié.

Étiquetage par les règles de la politique – systèmes de fichiers

- Régies aussi par des règles `allow` mentionnant une des deux permissions `relabelto, relabelfrom`!

```
allow user_t user_home_t : file relabelfrom ;
allow user_t httpd_user_content_t : file relabelto ;
```

- Règles définies dans la politique
  - Fichiers `*.fc`.
- Quatre mécanismes d'étiquetage :
  - Attributs étendus : règle de type `fs_use_xattr`, concernant less classes d'objets `ext2, ext3, xfs, jfs, reiserfs`.
  - Basé sur les tâches : règle de type `fs_use_task` concernant les classes d'objets `pipefs, sockfs`.
  - Basé sur les transitions : règle de type `fs_use_trans` concernant les classes d'objets `devpts, tmpfs, shm, mqueue`.
  - Généralisé : règle de type `genfscon` pour tout le reste, utilisé pour étiqueter des pseudo-systèmes de fichiers ou des systèmes de fichiers non-linux.

Exemples

- Règles déclarées dans `../src/policy/policy/modules/kernel/*.te`
- Pour les systèmes de fichiers qui possèdent les `xattrs` Linux :
  - `fs_use_xattr ext3 system_u:objetc_r:fs_t`
  - `fs_use_task pipefs system_u:objetc_r:fs_t`
  - `fs_use_trans devpts system_u:objetc_r:devpts_t`
- Systèmes de fichiers qui ne possèdent pas les `xattrs` :
  - `genfscon fs_type pathprefix context` :
  - Associe le contexte par défaut à tout fichier qui se trouve dans le système de fichiers identifié par le `pathname`.
  - Exemple : `genfscon proc /sys system_u:object_r:proc_t`.
  - On peut avoir un argument optionnel de type de fichier (`b, c, d, p, l, s, -` (se rappeler des types de fichiers Linux).

Contextes pour les objets réseau

- `portcon tcp 7 system_u:object_r:inetd_child_port_t` :
  - Pour les messages via le port TCP 7.
  - Par défaut, le SID nommé “`port`”.
- `netifcon eth0 system_u:object_r:netif_lo_t`
  - Premier contexte : le contexte de `eth0`
  - Le 2e : contexte de tout paquet qui arrive par `eth0`.
  - Par défaut, le SID nommé “`netif`”.
- `nodecon 127.0.0.1 255.255.255.255 system_u:object_r:node_lo_t`
  - Par défaut, le SID nommé “`node`”.

Quelques commandes

- Afficher les contextes de sécurité :
  - `id -Z` : contexte de sécurité pour le bash courant (plus uid/gid). Le seul contexte : `getcon`.
  - `ls -Z, -lcontext, -scontext` : contexte de sécurité pour les fichiers. Pour un fichier particulier : `getfilecon`.
  - `ps -Z` : contexte de sécurité pour les processus. Pour un PID donné : `getpidcon`.
- `cp -Z` : recopie les contextes (si la politique le permet !).
- Autres commandes modifiées : `lsof/netstat -Z`, `rsync -xattr` etc.
- Archivage avec contexte de sécurité : `tar`, `zip`, `amanda`.
- `getfattr/setfattr` – modifiés pour prendre en compte les attributs SELinux.

Nouvelles commandes SELinux

- `secon,getfilecon` : visualiser le contexte de sécurité.
- Changer le contexte de sécurité des fichiers :
  - `chcon system_u:objetc_r:shadow_t /tmp/foo` (ou comme pour `runcon`).
  - `setfiles/fixfiles` : remet à jour les contextes de sécurité des fichiers (selon la définition dans la politique – à voir !).
  - `restorecon` : revient au contexte par défaut (pas au précédent !)
- Montage d'un système de fichiers : `mount -o context=<...>`.
- Information sur le système :
  - `sestatus` : info sur l'état de SELinux.
  - `seinfo` : infos sur les éléments de base de la politique.
- Audit des messages et analyse :
  - `audit2why/audit2allow` : analyse de messages en ligne de commande.
  - `seaudit` : interface graphique.
- `semodule` : compilation et chargement d'un nouveau module dans la politique (essentiel pour le développement de nouveaux domaines de sécurité !).
- `getsebool/setsebool` – manipulations des booléens SELinux (à voir plus tard).

Nouvelles commandes SELinux

- Changer le contexte de sécurité de l'utilisateur :
  - `newrole -r role -t type`.
  - Modifie les privilèges de l'utilisateur.
  - Demande la re-authentification de l'utilisateur.
  - Peut **ne pas aboutir** : le changement de privilèges doit être permis par la politique !
- Lancer un processus dans un nouveau contexte :
  - `runcon -t type -r role -u user`.
  - Le nouveau contexte doit être correcte dans la politique.
  - Mais aussi la politique doit autoriser la *transition de contexte* (implicite car présence d'`exec` !) !
  - Ce qui veut dire qu'il faut avoir des règles permettant une `type_transition` entre `unconfined_t` et le type donné !

Modes et bibliothèques

- Politique reference :
  - Tous les processus s'exécutent dans un "domaine" commun (unconfined), *sauf* certains processus/démons *ciblés* (confined).
- Politique strict :
  - Chaque processus s'exécute dans un domaine confiné.
- Mode permissif : utilisé pour le développement de politiques.
- Mode renforcé : utilisé pour le déploiement d'une politique.

Interfaces :

- *Selinuxfs* : système de fichiers pour la communication entre le noyau et les utilisateurs.
- Monté à `selinux`.
- *Libselinux* : bibliothèque d'appels système pour les applications compatibles SELinux.

Quelques contextes dans la politique reference

- Quelques domaines confinés : `httpd_t`, `passwd_t`, `samba_t` etc. etc.
- Domaine non-confiné : `unconfined_t`.
  - C'est le domaine dans lequel les shells fonctionnent – et tous les programmes utilisateur lancés par le shell.
  - Relancer les services/démons par les scripts – les scripts ont des contextes particuliers qui les font s'exécuter dans les domaines appropriés !
  - Aller chercher dans `/var/run`.
- Domaine non-confiné $\neq$ domaine qui peut exécuter tout !
  - Exemple : accès à `/etc/shadow`.
- Utilisateurs et rôles confinés et non-confinés (à voir).
- Contextes particuliers pour les fichiers spéciaux (`/dev`).
- **Macros** particulières pour la création des règles d'autorisation d'accès (à voir aussi).

Transitions de domaine au démarrage

- Le processus 1 est lancé avec `kernel_t`.
- ... qui lance `init`, en le mettant en `init_t` (et le type du fichier dans lequel on trouve le code d'`init` est `init_exec_t`).
- ... qui lance le script d'initialisation `initrc` dans le domaine `initrc_t`, ainsi que la console dans `getty_t`.
- La console lance le processus de saisie de login/mot de passe dans le domaine `login_t`.
- Lorsqu'on se connecte, notre shell sera placé en `unconfined_t` à la suite d'une transition de domaine enclenchée par `login_t`.
- Tous les démons lancés par `initrc` sont placés dans leur domaine par des règles de transition de type appropriées.

Pséudo-répertoire `proc` et SELinux

- Répertoire `/proc/[pid]/attr` donnant les attributs SELinux :
  - `current, exec, prev, fscreate`.
  - Image du AV Cache.

Journalisation des décision de refus d'accès

- Messages d'audit dans `/var/log/messages` ou `/var/log/audit/`.
  - `avc: denied` (si pas empêché par une règle dontaudit).
  - `avc: granted` si auditallow.
  - Mode permissif : seulement la première violation est enregistrée.
  - Mode renforçant : toutes les violations enregistées, sauf si taux limite dépassé.
- Exemple :

```
Jul 10 02:44:17 new2 kernel: audit(1089441857.053:0): avc: denied { read }
for pid=4121 exe=/bin/bash name=.bashrc dev=hda2 ino=130311
scontext=root:system_r:bin_t tcontext=root:object_r:staff_home_t tclass=file
```

- Permission refusée : `read`
- Processus : PID = 4121, nom = `/bin/bash`
- Contexte source : `root:system_r:bin_t`.
- Contexte cible : `root:object_r:staff_home_t`.
- Classe de l'objet cible : `file`.

Fichiers de politique

- `/etc/selinux/config` : configuration, incluant la politique courante.
- `/etc/selinux/[nompol]` : répertoire pour la politique courante (`targeted,strict`).
- `/etc/selinux/targeted/contexts/files/` : contextes de fichiers.
- `/etc/selinux/[nompol]/policy/policy.[version]` : fichier binaire de politique, chargé dans le noyau.
- `/etc/selinux/[nompol]/src/policy/policy.conf` : code source lisible pour la politique.
- `/etc/selinux/[nompol]/src/policy/` : fichiers de code source utilisés pour construire la politique.
- `/selinux` : système de fichiers utilisé pour la communication entre le noyau et les utilisateurs (processus), similaire au `procfs,sysfs`.
- `libselinux` : bibliothèque de fonctions utilisables par les applications/programmeurs.