MEDI/MEPO – SELinux
Cours 2

Catalin Dima

Booléens et règles conditionnelles

- Règles conditionnelles pour la configuration de la politique en exécution.
  - Permettent de autoriser/invalider des fragments entiers de la politique.
  - L'auteur de la politique crée les configurations permises.
- Plus sûr que plusieurs politiques
  - Toutes les configurations possibles sont connues par avance, ce qui permet l'analyse.
  - Permet d'accorder aux domaines des permissions limitées de modification de politique.
- Permet d'implémenter des politiques dynamiques :
  - Mobilité, IDS.

Définitions et syntaxe

- Booléens = variables déclarées dans la politique.
  - Valeur booléene : `true/false`.
  - Valeur modifiable en ligne !
- Syntaxe de la déclaration : `bool nom val_par_defaut`.
  - Exemple : `bool allow_ypbind false`.
  - Fichiers de configuration : `/etc/selinux/[nompol]/booleans` et `/etc/selinux/[nompol]/booleans.local`.
  - Valeurs par défaut – `policy.conf` – écrasées par les valeurs dans le fichier de configuration.
- Deux valeurs associées :
  - Valeur actuelle.
  - Valeur *en attente* – les changements “en ligne” ne sont pas immédiats.

Gestion “en ligne”

- Valeurs (actuelle et en attente) dans `/selinux/booleans/nom_de_booleen`.
- Commandes :
  - Obtention de la valeur : `getsebool`.
  - Modification de la valeur : `setsebool` – modifie la valeur actuelle *et* la valeur en attente !
- Modifier “en ligne” la valeur d’un booléen *ne modifie pas* sa valeur par défaut au redémarrage.
  - `setsebool -P` pour “permanentiser” la modification.

Expressions booléennes

- Booléens combinées en expressions booléennes employant les opérateurs logiques :
  - `&&, ||, !, ^, ==, !=`.
- Instructions conditionnelles :
  - Blocs d'instructions de la politique, contrôlés par des expressions booléennes.
  - Structures de type `if-then-else`.
- Les blocs d'instructions d'un if-then-else peuvent contenir :
  - Des règles AV `allow, dontaudit`.
  - Des règles de transition de type.
- Les blocs d'instructions d'un if-then-else *ne peuvent pas* contenir :
  - Déclarations de type/alias/utilisateur/rôle.
  - Règles de type `role_transition`.

```
bool user_ping false;
if (user_ping) {
   domain_auto_trans(unpriv_user_domain, ping_exec_t, ping_t)
   allow ping_t { ttyfile ptyfile } : chr_file rw_file_perms;
}
```

Multilevel Security (MLS)

- Le modèle TE de SELinux est censé protéger l'*intégrité* du système, pas la sécurité des données.
  - On veut confiner l'interaction entre processus.
  - `apol` peut nous aider à trouver les flux d'information – *mais* tâche ardue.
- Protection de la confidentialité = modèle *Bell-LaPadula* :
  - Lattice de classes de sécurité.
  - No read-up, no write-down.
  - Protège les classes contre (un certain type de) fuites d'information.
- Implémentable dans des systèmes de contrôle d'accès, mais à un prix parfois trop grand.
  - Prévoir de coder les niveaux de sécurité dans les `permissions`.
  - Si le système permet aux objets de voir changer leur classe de sécurité – gestion complexe des permissions...
- Solution SELinux : politiques MLS/MCS.

MLS et MCS en SELinux

- Le contexte de chaque sujet/objet possède un 4e et un 5e champs, composés de 2 éléments :
  - *Niveau de sensibilité* : reflecte la sensitivité de l'information contenue dans l'objet.
  - *Catégorie de sécurité* : reflecte l'appartenance à une classe de sécurité par rapport au flux d'information.
- Sensibilités : strictement hiérarchisées = treillis linéaire.

```
sensitivity s0;
...
sensitivity s3;
dominance { s0 s1 s2 s3 }; # s0 < s1 < s2 < s3
```

- Catégories : sans relation d'ordre.

```
category c0 alias comptabilite;
category c1 alias ressources_humaines;
...
```

- Les combinaisons peuvent être limitées :

```
level s1:c1.c3;
level s2:c0.c2,c4;
```

- En absence de déclarations de catégories pour une sensibilité : *toutes les catégories* déclarées acceptables.

```
user_u:user_r:user_t:s1  # implicitement c1.c3 !
```

Intervalle de sécurité

- Les deux champs (4e et 5e) représentent l'*intervalle* de sécurité :
  - min de l'intervalle = niveau courant.
  - max de l'intervalle = *clearance* (appx homologation/autorisation).
- La clearance représente :
  - Pour un processus : le niveau maximal de sécurité auquel il peut accéder.
  - Pour un objet : c'est le cas d'objets *multiniveau* – la plage maximale de données qui peut être stockée.
- L'intervalle doit être cohérent : min plus petit que max.
  - Y compris l'inclusion des catégories !
  - Contre-exemple : `user_u:user_r:user_t:s0`.

Accès et MLS/MCS

- Accès processus $\rightarrow$ objet :
  - Couple (niveaux, catégories) du processus doit *dominer* couple (niveaux, catégories) de l'objet.
  - Dominance :
    - intervalle de niveaux du sujet doit contenir niveau de l'objet et
    - ensemble de classes du sujet doit contenir ensemble de classes de l'objet.
  - Pas de distinction "write-down/read-up" pour les catégories !
- Cas particulier de l'exécution :
  - intervalle de niveaux du processus lanceur doit contenir *intervalle* de niveaux du processus lancé.

Exemples

- Affectation d'étiquette MLS/MCS :

```
user_u:system_r:unconfined_t:s0-s0:c0,c3,c15
root:system_r:unconfined_t:s0-s15:c0.c1023
root:system_r:kernel_t:s15:c0.c1023
```

- Translations
  - `s0=SystemLow`
  - `s15:c0.c1023=SystemHigh`
  - `s0-s15:c0.c1023=SystemLow-SystemHigh`
  - `s0:c1,c3=CompanyConfidential`
  - Présents dans divers emplacements : `/etc/selinux/semanage.conf`, ou `/etc/mcs.conf`, ou...

Transition d'intervalle

- Définition d'intervalle lors d'une transition de type.
- Syntaxe :

```
range_transition typesource typedest : classe étiquetteMLSMCS

range_transition init_t init_exec_t : process s0
range_transition kernel_t udev_exec_t : process s0 - s0:c0.c1023
```

- Peuvent être limitées à leur tour par des *contraintes* MLS.

Roles et utilisateurs

- RBAC traditionnel :
  - Attribuer des privilèges et des autorisations aux roles.
  - Attribuer des utilisateurs à un ou plusieurs rôles.
- RBAC en SELinux :
  - Attribuer des privilèges et des autorisations aux domaines.
  - Attribuer des domaines aux rôles.
  - Assigner un ou plusieurs rôles à un utilisateur.
  - Permet de définir une hiérarchie de rôles.
- Identificateur de rôle :
  - Pour les sujets, représente le rôle courant.
  - Pour les objets, typiquement `object_r`.

Déclarations de rôles

- `role ident types liste;`
  - Convention pour les identificateurs : terminaison en `_r`.
  - Espace de noms séparé par rapport aux types.
  - Pour les types, on peut employer les attributs/alias, si plusieurs, entourés d'accolades.
  - Seulement les déclarations de rôles se voyant attribuer des domaines ont un sens.
- On peut avoir plusieurs déclarations d'un même rôle.
  - Attribution de l'union des domaines dans toutes les déclarations.

Changement de rôle

- `allow role_courant nv_role`
  - Permet au rôle courant de faire une transition vers un nouveau rôle.
  - Peut comporter plusieurs nouveaux rôles.
  - D'un usage limité dans les politiques courantes :
  - La plupart des rôles peuvet faire une transition vers un autre.
  - Des contraintes peuvent être imposées pour limiter les changements de rôles (à voir).
- Multi-Level Security (MLS) : `dominance { role qconque { role sysadm_t; role user_t`
  - On peut définir les roles seulement de cette manière !
  - Les rôles héritent des types qu'ils dominent.
  - Pas utilisés dans la politique par défaut.
- `role_transition firstboot_r initrc_exec_t system_r`
  - Permet aux `sysadm_r` de démarrer les démons dans le rôle `system_r` avec des scripts d'initialisation.
  - Changement de rôle en se basant sur le rôle courant du processus et le type de fichier.

Utilisateurs en UNIX

- Identités des utilisateurs SELinux :
  - Orthogonale aux utilisateurs Linux.
  - L'abilité de changement l'utilisateur est contrôlée par la politique.
  - L'identité des utilisateurs SELinux et Linux peut ne pas être la même : on peut avoir des identités génériques.
- Définissent les rôles que les utilisateurs sont autorisés à prendre.
- Les programmes de login (`login, sshd, etc.`) déterminent l'identité d'utilisateur.
- Contextes de login par défaut :
  - `policy/contexts/users/username` et `policy/contexts/default_contexts`.
- Nouvelle syntaxe `ssh` : `ssh user/role@hostname`
  - Le changement de rôle devrait être permis par la politique.
  - Existence du domaine par défaut nécessaire.

Contraintes

- Restrictions globales sur certaines permissions, indépendamment des règles `allow`.
- Basées sur des expressions sur let entre utilisateurs, rôles et/ou types.
- Exemples de contraintes de la politique :
  - `/etc/selinux/src/policy/constraints`
- Place de l'évaluation des contraintes dans le mécanisme de décisions de sécurité de SELinux :
  - Le SS (Serveur de Sécurité) crée d'abord un masque de bits des permissions autorisées par les règles d'accès.
  - Ensuite, SS enlève les bits correspondant aux permissions qui ne satisfont les contraintes.
  - C'est ce masque de bits qui est fourni au Serveur de Renforcement de Politiques.

Syntaxe des contraintes

```
constrain class_set perm_set expression;
```

- `class_set` : une ou plusieurs classes d'objets.
- `perm_set` : une ou plusieurs permissions.
  - Chaque permission doit être valide pour chaque classe d'objets.
- Les caractères joker *, - et ne sont pas supportés.
- `expression` = expression booléenne
  - Définit une relation entre l'utilisateur/le rôle/le type source et cible.
  - Emploie des opérateurs de comparaison.
  - Mots-clés :
    - `u1` : utilisateur source, `u2` : utilisateur cible.
    - `r1` : rôle source, `r2` : rôle cible.
    - `t1` : type source, `t2` : type cible.

Opérateurs employés dans les contraintes

- `t1 == t2` : test d'égalité de types ; similaire pour les rôles et les utilisateurs ; opérateur identique : `eq`.
- `t1 == attribut` : test si `t1` possède l'attribut ; similaire si on place un `type_name` ; similaire pour `t2`, ainsi que pour les rôles et les utilisateurs.
- `t1 != t2` : test d'inégalité de types ; similaire pour les rôles et les utilisateurs.
- `t1 != attribut` : test si `t1` ne possède pas l'attribut ; similaire si on place un `type_name` ; similaire pour `t2`, ainsi que pour les rôles et les utilisateurs.
- `r1 dom r2` : test de dominance, `r1` doit être déclaré comme dominant `r2`.
- `r1 domby r2` : test inverse de dominance, `r2` doit être déclaré comme dominant `r1`.
- `r1 incomp r2` : test d'absence de dominance, `r1` et `r2` ne doivent pas se trouver en relation de dominance ou de dominance inverse.

Exemples

`constrain process transition ( u1 == u2 )`

- Restrictionne de la permission de faire une transition pour les processus,
- Garantit l'absence de modification d'identité.

`constrain process transition ( u1 == u2 or t1 == privuser )`

- Rajoute une exception à la contrainte précédente.
- Donne la permission de faire des transitions aux domaines possédant l'attribut `privuser`.
- E.g. : processus login.

`constrain process transition (u1 == u2 or t1 == privuser ) and ( r1 == r2 or t1 == privrole )`

- Rajoute une contrainte similaire pour les rôles.

Versions pour MLS : `mlsconstrain, mlsvalidatetrans` (nouveau, support seulement pour les objets permanents).

Contraintes de transitions

- Contrôle de la capacité de changer les contextes de sécurité.
- Permet de spécifier le contexte de l'objet avant et après le changement, *ainsi que* le contexte du processus qui le fait.
- Syntaxe :

  ```
  validatetrans { classes } expression;
  ```

  - `classes` et `expression` = même interprétation que pour les `constrain`.
  - Mais troisième ensemble de mots-clés : `u3, r3, t3` = utilisateur, rôle ou type du processus qui effectue l'opération.
- Actuellement, seulement admise pour les objets de type fichier/système de fichiers.

Exemples

- Application dans l'assurance de l'intégrité :
  - Supposons que des fichiers de type `user_tmp_t` sont créés par des utilisateurs dans le `/tmp`.
  - On veut empêcher des domaines privilégiés de réétiqueter `user_tmp_t` en autre type hautement critique (ex. : `shadow_t`) :

    ```
    validatetrans { file lnk_file }
    ( t2 != shadow_t or t1 != user_tmp_t );
    ```
- 3e type de paramètres : rajout de domaines qui peuvent réétiqueter des fichiers non-intègres.

```
validatetrans { file lnk_file } ( ( t2 != shadow_t or t1 != user_tmp_t )
          or ( t3 == relabelany ) );
```

Contraintes MLS

- Privilèges MLS/MCS implémentés avec des attributs
  - `mlsfileread, mlsfilewrite, ...`
  - Domaines associés aux attributs : `typeattribute consoletype_t mlsfileread`.
- Processus peut acquerir des privilèges en exécutant un domaine qui possède l'attribut associé.
- La politique MLS définit les effets de ces privilèges.
- Règles `mlsconstrain` :
  - Basés sur les combinaisons classe-permission.
  - Extension du langage des contraintes SELinux.
  - Syntaxe : `mlsconstrain classes permissions (expression)`
  - Deux paramètres : sujet (initiateur) et objet.

Contraintes – mlsconstrain

- Rappel : expressions pour `constrain`
  - Identificateurs : `u1, u2, r1, r2, t1, t2, privrole, privuser`.
  - Opérations : `==, !=, dom, domby, incomp`.
- Nouvelles expressions pour `mlsconstrain` :
  - Identificateurs `l1, l2, h1, h2`.
- Exemple :

```
mlsconstrain file read ( (l1 dom l2)
   or ((t1 == mlsfilereadtoclr) and (h1 dom h2))
   or (t1 == mlsfileread) );
```

Contraintes – mlsvalidatetrans

- Contraintes à satisfaire lorsqu'on upgrade ou downgrade un objet.
- Pas liées à une permission particulière
  - Définies pour toute la classe de sécurité.
  - Le test de permissions est censé être assuré par les règles AV.
- Syntaxe : `mlsvalidatetrans classes expression;`
  - *Trois* paramètres : (objet source, objet destination, processus).
  - Extension d'identificateurs : `u3, r3, t3, l3`.
  - Exemple :

```
mlsvalidatetrans file ( (l1 eq l2)
  or ((t3 == mlsfileupgrade) and (l1 domby l2))
  or (t3 == mlsfiledowngrade) );
```

Autres composants MLS/MCS

- Base de données utilisateurs : `/etc/selinux/[politique]/src/policy/policy/users`
  - Étiquette par défaut.
  - Intervalle MLS/MCS par défaut.
  - Utilisés lors du login.
  - Exemple :

    ```
    user jdoe {user_r} level s2 range s0-s9:c0.c127;
    ```
- Arguments MLS/MCS pour commandes Linux :
  - `runcon -l` et `chcon -l` : niveau de sécurité (translations utilisés).
  - Les transitions doivent satisfaire les contraintes MLS !

Fichiers *.fc

- Fichiers décrivant de manière plus fine l'étiquettage initial.
- Lignes = expressions régulières décrivant l'identification des fichiers :

```
/bin(/.*)?                          system_u:object_r:bin_t
/bin/bash                  --       system_u:object_r:shell_exec_t
/bin/d?ash                 --       system_u:object_r:shell_exec_t
/bin/zsh.*                 --       system_u:object_r:shell_exec_t
/bin/ls                    --       system_u:object_r:ls_exec_t
/u?dev/dm-[0-9]+           -b       system_u:object_r:fixed_disk_device_t
/u?dev/sg[0-9]+            -c       system_u:object_r:scsi_generic_device_t
/opt(/.*)?/lib(64)?(/.*)?           system_u:object_r:lib_t
/opt(/.*)?/.*\.so(\.[^/]*)*         --      system_u:object_r:shlib_t
/dev/pts(/.*)?                      <<none>>
```

Politique référence

- Politique modulaire de sécurité, permettant le rajout ou la suppression des modules de politiques sans recompilation de l'ensemble.
- Basée sur la politique *targeted*, donc avec un domaine *unconfined* et des domaines spécifiques aux services.
- Fichier `.pp` contenant les modules binaires.
- Fichier `.te` contenant les déclarations de types et règles de controle d'accès.
- Fichier `.fc` contenant les règles d'étiquettage par défaut des objets spécifiques au module.
- Fichier `.if` contenant les déclarations de macros utilisables dans d'autres modules.

Développement d'un nouveau module

- Nouveau module censé *confiner* les interactions d'un nouveau service/nouvelle application avec le reste du système d'exploitation.
  - Pour éviter que les possibles failles dans les autres services représentent des dangers pour le nouveau service.
  - Pour éviter que, en absence de garantie sur un certain type d'activité du nouveau service, la présence de failles dans celui-ci dé-fiabilisé l'architecture de sécurité du système.
- Problématique :
  - Quels sont les interactions permises ? avec qui ?
  - Quels sont les objets que le service/l'application manipule ? Sont-ils les mêmes que ceux utilisés par d'autres services/applications ?
  - Quels sont les interactions avec les utilisateurs ?
  - Quels sont les fichiers de configuration ? Qui a le droit de les manipuler ?
  - Qui a le droit de lancer en exécution le service/l'application ? Quels domaines ont le droit de communiquer avec les processus de l'application/du service, et à travers quel mécanisme ?
  - A-t-on besoin de créer un seul domaine pour notre service ou plusieurs ? Même question pour les types assignés aux objets manipulés.

Exemple de fichiers `.te` et `.if` (1)

- Déclaration des types et autres contraintes :

```
policy_module(monappli,1.0.0);
require {
  type unconfined_t;      # on en aura besoin...
}
type monappli_t;          # notre domaine !
type monappli_exec_t;     # le type de nos exécutables

application_domain(monappli_t,monappli_exec_t)
role user_r types monappli_t;

allow unconfined_t monappli_exec_t : file { read execute };
```

- Déclaration d'une macro d'interface :

```
interface(‘monappli_domtrans’,‘
   gen_require(‘
        type monappli_t;
        type monappli_exec_t;
   ’)
   domtrans_pattern($1, monappli_exec_t, monappli_t)
’)
```

Fichiers `.te`

- Sections indispensables à la modularité de la politique : `require`, `gen_require`, etc.
- Sections spécifiques : déclaration de types/domaines utilisés, définition de booléens, règles AV manipulant les types nouvellement déclarés ou les types préexistants, etc.
- Une grande variété de *macros* prédéfinis, englobant des règles AV.
- Longue liste de macros trouvable ici : `http://oss.tresys.com/docs/refpolicy/api/`.

Règles AV dans fichiers `.te`

- Macro `domtrans_pattern($1,$2,$3)` : domaine $1 lance en exécution un exécutable $2, et le résultat sera dans le domaine $3.
- Macro `application_domain($1,$2)` : le 1er argument est un domaine, et sa porte d'entrée est le type en 2e argument.
- Macro `unconfined_domtrans_to($1,$2)` : unconfined peut lancer des applications du domaine $1, par exécution de fichiers de type $1.
- Notre nouvelle macro `monappli_domtrans($1)` : on peut lancer des applications dans notre domaine à partir d'applications dont le domaine est le 1er argument.
  - C'est là qu'on a besoin de `require` : sorte de déclaration `external` dans notre module de `unconfined_t`, nécessaire à la compilation.

Macros

- Notre application peut utiliser les bibliothèques : `libs_use_ld_so($1)` et autres `libs_use_...`
- Notre application peut avoir besoin d'interagir avec l'utilisateur (lire/écrire) à travers un terminal (X ou autre) : `term_use_controlling_term($1)` et autres `term_use_...`.
- Macros `m4` utiles pour la création de nouveaux macros : `http://selinuxproject.org/page/NB_RefPolicy`, section `Reference_Policy_Support_Macros`.

Macros trouvables dans `system` ou `kernel`.

Fichiers propres à l'application/au service

- On peut avoir besoin de déclarer des fichiers propres.

```
type monappli_rw_t;
files_type(monappli_rw_t);    # pour déclarer qu'il s'agit de type fichier
```

- Macro `files_config_file($1)` : type fichier de configuration ; autres `files_read_...` dans la partie `kernel`.
- Mais aussi règles AV directes :

```
allow monappli_t monappli_rw_t : file { read write ioctl } ;
type_transition monappli_t user_home_t : dir monappli_rw_t ;
```

Développement du domaine

- Commencer par y mettre le domaine et le type exécutable.
- Insérer les permissions basiques, telles que nécessaires pour l'entrée dans le domaine par la *porte d'entrée* `unconfined_t`.
- Mettre SELinux en mode permissif.
- Lancer l'application et aller chercher les `avc : denied`.
- Trouver la raison pour le déni.
- Si déni incorrect, trouver la bonne macro à utiliser et l'insérer dans le fichier `.te`, avec ses bons paramètres.
- Toujours utiliser la macro qui donne le moins de droits à l'application ! (principe de séparation des privilèges).
- Rajouter aussi l'interface `.if`.
- *Outil pour débutants* : dans le gestionnaire de politique, onglet `Policy module`, création de nouveau module.
- Il faut presque toujours rajouter des nouvelles règles dans les fichiers créés par cet outil !

Fichiers `.fc`

- Rappel : nécessaires pour l'étiquettage par défaut des fichiers/objets du système.
- Si on veut éviter un étiquettage manuel, après création de module, d'un grand nombre de répertoires/fichiers qui seront réservés à notre application.
- Syntaxte : déjà vue :

  `/home/info/try/myapp --gen_context(unconfined_u:object_r:myapp_exec_t,s0)`
- Si on n'a que très peut de fichiers, on peut toujours utiliser `chcon`.
- Par contre, attention aux permissions accordées, si `unconfined_t` ne peut pas réétiqueter !
- Régies aussi par des règles `allow` mentionnant une des deux permissions `relabelto, relabelfrom`.
- En général, `unconfined_t` a bcp de droits de réétiquettage.

Inclure un nouveau module dans la politique

- Compiler : `$ make -f /usr/share/selinux/targeted/include/Makefile`
  - Recherche automatiquement les fichiers `.te, .if, .fc` modifiés depuis la dernière création du module.
- Insérer le nouveau module dans la politique : `semodule -i myapp.pp`
- Tester si le module existe : `semodule -l`
- Voir manpages pour les autres options de `semodule`!
- On peut passer directement à l'étape `semodule` si le makefile est dans le répertoire de travail.

Exemple : `exec` et ses actions

Appel système	Action		Source	Cible
	Classe	Permission	(Processus)	(Ressource)
`execve`	`dir`	`search`	domaine crt	répertoire
	`file`	`execute`	domaine crt	fichier
	`process`	`transition`	domaine crt	nv domaine
	`process`	`entrypoint`	nv domaine	fichier
	`fd`	`inherit`	nv domaine	descr fich
	`process`	`ptrace`	parent	nv domaine