

TD 1 - LIF12 système d'exploitation

Passage d'informations sur les pipes

25 avril 2017

I ACL ldap

Un annuaire électronique est une base de données spécialisée, dont la fonction première est de retourner un ou plusieurs attributs d'un objet grâce à des fonctions de recherche multi-critères. Aujourd'hui, on utilise souvent les annuaires basés sur LDAP (*Lightweight Directory Access Protocol*). Un annuaire LDAP contient des objets, organisés de façon hiérarchique (arborescente), comme cela est illustré par la figure 1. Chaque objet peut posséder un certain nombre d'attributs, dont certains sont normalisés, comme par exemple :

- dc pour *domain component* : identifie une composante de l'annuaire ;
- dn pour *distinguished name* : nom distinct, identifiant unique de l'objet, il contient les dn des ancêtres ;
- cn pour *common name* : nom commun de l'objet ;
- o pour *organisation* : nom de l'organisation ;
- ou pour *organisation unit* : branche de l'organisation ;
- ...

Un annuaire LDAP permet à certains de s'authentifier et de manipuler les informations. Ces manipulations sont limitées par des directives d'accès sous la forme d'ACLs. Pour définir une directive d'accès vous devez utiliser :

```
access to <what> [ by <who> [ <access> ] [ <control> ] ]+
```

Les 4 parties importantes de la directive sont :

- <what> spécifie l'objet cible par exemple « le mot de passe de tout le monde » ou « la photo de d'un membre du service informatique » ;
- <who> l'objet à qui le droit est attribué, par exemple, l'administrateur ou n'importe qu'elle ordinateur ;
- <access> le droit attribué (`none|auth|compare|search|read|write|manage`) ces droits sont croissant, celui qui a le droit de lecture (**read**) peut forcément comparer (`compare`).
- <control> est permet de choisir l'action à faire ensuite. Si rien n'est précisé, c'est `stop` (arrêter d'évaluer les ACLs), mais il est possible de choisir **continue** (évalue la suite de la même directive) ou **break** (passe à la directive suivante).

Pour tester si un objet a accès à une certaine cible, la liste est évaluée par ordre d'écriture. Si une clause <what> correspond à la cible demandée, les champs <who> sont évalués. Lorsqu'un des champs correspond à l'objet authentifié, le droit obtenu est celui qui est spécifié par <access>. Le test s'arrête alors, sauf si cela est spécifié par <control>.

On considère que chaque directive est terminée implicitement par un `by * none stop` refusant l'accès aux objets qui ne sont pas nommés dans la directive. De même, la liste des ACL est supposée se terminer par

```
access to *  
by * none
```

Ce qui signifie que lorsque rien n'est précisé, personne n'a d'accès.

L'idée de l'exercice n'est pas de connaître la syntaxe des ACL propre à `openldap` mais uniquement leur fonctionnement. C'est pourquoi nous allons utiliser les abréviations suivantes

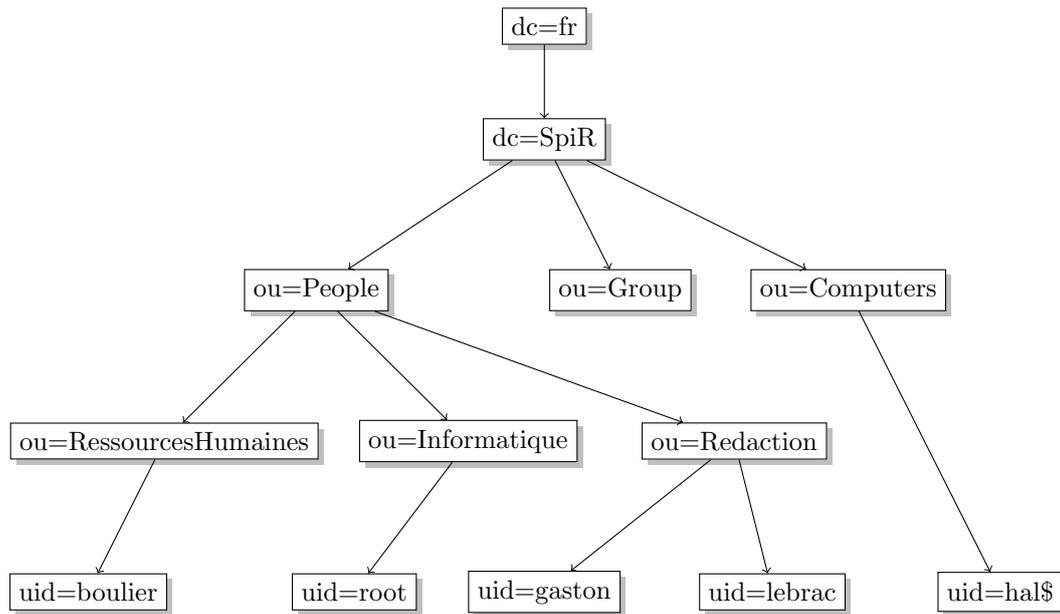


FIGURE 1 – Organisation de l’annuaire LDAP dans notre exemple.

- *root* l’utilisateur administrateur : la définition serait exactement `dn.exact="uid=root,ou=Informatique,ou=People,dc=SpiR,dc=fr"` ;
- *machines* descendants de l’objet `Computer` : `dn.children="ou=Computer, dc=SpiR, dc=fr"` ;
- *employés* tous les descendants de `People` : `dn.children="ou=People, dc=SpiR, dc=fr"` ;
- *infos employés* tous les attributs des employés ;
- *infos perso* les attributs représentant les informations personnelles : `homePostalAddress,homePhone,street` ;
- *RH* tous les employés descendant de `RessourcesHumaines`.

nous proposons le tableau de directives suivant (les directives en *italique* sont implicites) :

directive	what	who	access	control
1	userPassword	"root" "machines" self anonymous *	manage read write auth none	<i>stop</i> <i>stop</i> <i>stop</i> <i>stop</i> <i>stop</i>
2	*	"root" self * *	manage write <i>none</i> <i>none</i>	<i>stop</i> <i>stop</i> break <i>stop</i>
3	"infos employés"	"RH" * *	write <i>none</i> <i>none</i>	<i>stop</i> break <i>stop</i>
4	telephonenumber, displayName	* *	read <i>none</i>	<i>stop</i> <i>stop</i>
5	"infos perso"	*	<i>none</i>	<i>stop</i>
6	*	*	read	<i>stop</i>

Q.I.1) - Pour les communications internes, les telephones intérogent l’annuaire grâce à une connexion anonyme. Peuvent-ils connaitre l’identité de celui qui appel ? Quelles directives sont concernées ?

Les téléphones doivent rechercher le nom (`displayName`) d'un utilisateur à partir de son numéro de téléphone (`telephoneNumber`).

- la directive 1 ne correspond pas car elle concerne le mot de passe ;
- la directive 2 correspond car elle concerne tout
 - l'auteur de la demande est anonyme et donc la première ACE qui s'applique est la 3e : * none **break**
 - elle n'accorde aucun droit (`none`)
 - le control associé est `break` ce qui signifie qu'on continue à lire les règles suivantes.
- la directive 3 s'applique car elle concerne tous les attributs des employés. Comme la précédente elle se termine par * none **break**
- la directive 4 s'applique et donne * **read stop** ce qui accorde le droit de lecture sur les 2 attributs et stop la lecture des ACLs.

Au final, un utilisateur anonyme comme un téléphone a le droit de lire le numéro de téléphone et le nom des employés. Il peut donc aussi faire une recherche en fonction du numéro afin d'obtenir ce nom (le droit de recherche est inférieur à celui de lecture). Donc l'affichage du nom est possible.

Q.I.2) - Que se passe-t'il si on retire la ligne « `self, write` » de la seconde directive (celle concernant *) ? Que peuvent faire sur leurs propres données :

- l'utilisateur `uid=gaston,ou=Redaction,ou=People,dc=SpiR,dc=fr`
- l'utilisateur `uid=boulier,ou=RessourcesHumaines,ou=People,dc=SpiR,dc=fr`

Donnez le droit maximum obtenu pour chacun des attributs `telephonenumber` (le téléphone de travail), `homePostalAddress` (l'adresse personnelle), `userPassword` et `jpegPhoto`.

Si on retire la directive, un utilisateur n'a plus le droit de modifier ses propres données par défaut. Il peut toujours modifier son propre mot de passe (directive 1) mais pour les autres champs, il n'y a pas de droit spécial accordé sur ses données. Donc :

- Gaston, peut :
 - modifier son mot de passe `userPassword` (directive 1)
 - n'obtient rien par la directive 2 (il n'est pas `root`)
 - n'obtient rien par la directive 3 il n'est pas dans Ressource Humaine)
 - peut lire son nom et son numéro de téléphone `telephonenumber` (directive 4)
 - n'a aucun droit sur ses informations personnelles : donc ne peut ni lire ni même comparer la valeur du champ `homePostalAddress` (directive 5)
 - peut lire le reste (directive 6) donc peut lire `jpegPhoto`.
- M. Boulier peut :
 - modifier son mot de passe `userPassword` (directive 1)
 - n'obtient rien par la directive 2 (il n'est pas `root`)
 - obtient le droit d'écrire ses informations personnelles car il est dans *RH* (directive 3) il peut donc écrire `telephonenumber`, `homePostalAddress`, et `jpegPhoto`.

Q.I.3) - Sous Active Directory, un utilisateur non authentifié (`anonymous`) ne peut pas consulter le contenu de l'arbre. La seule possibilité offerte est de s'authentifier ce qui impose de fournir son login et son mot de passe et de demander au serveur ldap de les vérifier (droit `auth` sur le mot de passe).

Que faut-il changer aux ACL proposées pour obtenir le même comportement ?

Entre les directives 1 et 2 il faut ajouter une directive :

directive	what	who	access	control
1.5	*	anonymous *	none none	stop break

Ce qui ajoute une règle concernant tous les objets. Cette dernière n'accorde aucun droit à *anonymous* et arrête la lecture des ACL. Pour tous les autres utilisateurs, on n'accorde rien non plus, mais on continue la lecture des ACL. Les droits de ces derniers ne sont donc pas modifiés par cette règle. Le droit d'*anonymous* est bien réduit à la seule authentification (acquis par la directive 1) sans aucun autre.

II Lecture de données

De nombreux protocoles de communication ont besoin d'échanger des données de taille variables. Cela pose problème car lors de l'échange, celui qui lit les données doit avoir un moyen de reconnaître la fin du message.

L'une des méthodes pour résoudre ce problème est de choisir un fanion. C'est à dire un octet spécial (ou un groupe d'octets) qui marque la fin du message.

Q.II.1) - Donnez 2 autres méthodes permettant résoudre le même problème.

On peut :

- Transmettre un entête de taille connue contenant la taille totale du message. Celui qui lit peut alors lire l'entête, extraire la taille du reste depuis l'entête puis lire le reste. Par exemple, cette technique est utilisée dans les websockets, les paquets IPs, ...
- Fermer la socket à la fin du message. Celui qui lit est toujours prévenu de la fermeture de cette socket et est donc informé qu'il n'y a plus rien à lire. Par exemple, cette technique est utilisée par le protocole HTTP dans la première version, le protocole FTP lors de la transmission de données, ...
- Découper le message en petits morceaux de taille fixes. Il y a alors un morceau spéciale qui marque la fin du message.

Pour pouvoir utiliser les fanions, il faut être capable de lire les données jusqu'à ce dernier, sans en oublier et sans lire de données en trop.

Q.II.2) - Donnez le code d'une fonction :

```
vector<char> read_until(int sock, char until);
```

Qui lit exactement de qui est envoyé sur la socket `sock` jusqu'au prochain caractère `until`

Cette fonction est difficile à écrire car les primitives qui permettent de lire sur le file descriptor `sock` consomment les données. C'est à dire que si on lit par erreur les données qui se trouvent après le fanion de fin. Une fois lu, on ne peut plus revenir en arrière.

Dans le cas de certains file descriptor, il est possible de trouver une solution. Par exemple, dans le cas des fichiers, les données n'ont pas disparues, seul le marqueur de la lecture dans le fichier à été déplacé. Il suffit de revenir en arrière (voir la fonction `lseek`). Dans le cas des sockets il est possible de tester la lecture en laissant les données dans le flux (voir l'option `MSG_PEEK` de la fonction `recv`). Mais dans le cas général, on ne peut que lire caractère par caractère.

```
vector<char> read_until(int sock, char fanion) {
    // le résultat est un tableau vide au départ
    vector<char> res;

    while(1) {
        char b;

        int r = read(sock, &b, 1);
        if (r == -1) { // c'est une erreur de lecture
        }
        if (r == 0) {
            // la socket est fermée avant le fanion
            cerr << "Attention, fermeture de la socket" << endl;
            break;
        }
        res.push_back(b);
        if (b == fanion) {
            // c'est fini
            break;
        }
    }

    return res;
}
```

Q.II.3 - La solution proposée n'est pas très efficace. Comment peut-on faire mieux.

La lecture caractère par caractère demande beaucoup d'appels système pour une simple lecture d'un message complet. Ce n'est donc pas très efficace.

On peut utiliser la solution de java, c'est à dire créer une classe qui englobe les sockets et contient un buffer (voir les `java.io.BufferedReader`. Les octets lus en trop sont stockés dans ce buffer jusqu'à la prochaine lecture.

Cette méthode a un gros inconvénient, la prochaine lecture **doit** être fait en tenant compte du buffer. Donc elle rend inutilisable la socket elle même.