Writing Access Control Policies for LDAP

30th January 2009

Andrew Findlay

Skills 1st Ltd

www.skills-1st.co.uk

Synopsis

Access Control systems vary from one LDAP server to the next. All of them can implement simple policies, but it may be necessary to design the DIT around the access control requirements. In more complex cases it is essential to choose a server with a very flexible access control language. There are a number of pitfalls in ACL design, and some requirements cannot be implemented by many of the commonly-used server products.

This paper suggests an approach to designing and testing access control rules. It includes worked examples to illustrate some common use-cases.

Dr Andrew Findlay
Skills 1st Ltd
2 Cedar Chase
Taplow
Maidenhead
SL6 0EU
+44 1628 782565
andrew.findlay@skills-1st.co.uk

1 Why Use Access Control?

Directories often hold sensitive information: names, addresses, user IDs, passwords, lists of permissions... the list is endless. This information always needs protection against unauthorised modification, and often has to be protected against unauthorised access. At the same time, the directory must be kept up-to-date and accessed by those authorised to do so.

Organisations usually have policies governing who may access and modify information. Access Control is the mechanism used to enforce those policies.

2 LDAP Access Control Schemes

Most directory servers provide some kind of access control language, but the exact form varies from one product to another for historical reasons.

The original X.500(1988) standard explicitly avoided defining an access control scheme ([X50088] section 7.1.2). As a result, early implementations such as Quipu were left to define their own. LDAP was originally based on X.500(1988) so it inherited this lack of standardisation. The 1993 revision of X.500 [X50093] introduced a very comprehensive access control scheme, but this was widely regarded as over-complex so it did not get implemented in server products or carried over into LDAP when other 1993 concepts were incorporated in 1997 [RFC2251][RFC2252]. By this time there were standalone LDAP servers such as Michigan's *slapd* as well as LDAP-to-X.500 gateways in use, so the number of access-control schemes continued to grow.

The situation today is that X.500(2005) contains a comprehensive – if complex – access control system [X50105], but LDAP leaves this aspect for implementers to define. It should be noted that LDAP *does* define authentication methods and security mechanisms [RFC4513].

Existing LDAP access control schemes can be broadly divided into two types:

1. Access control rules held in directory entries, usually in the part of the DIT that they control.
2. Access control rules held outside the DIT.

Rules in the first type are usually unordered sets, while the second type can resemble programming languages.

2.1 Concepts

All access control systems must deal with the same set of concepts, though the terms they use to describe them may be different.

2.1.1 Subject

The Subject is the person or other entity that is requesting access. It might be defined by the DN of a single entry, the DN of a group, or perhaps by a rule or LDAP search filter that matches a number of entries. Subjects might also be called *principals* or *users*.

Most schemes have an explicit representation of anonymous users, often in the form of a special DN such as *cn=any*.

Users can identify themselves to an LDAP server in different ways with varying levels of security, so some schemes allow the required authentication strength to be included in the access control rules.

2.1.2 Object

The Object is the thing that is being accessed. Some servers call it the *target*. Objects include entire entries, attributes, and possibly particular values of those attributes. Most LDAP operations require access to at least one entry and at least one attribute within that entry.

In access control rules, objects are often defined by filters and search strings, e.g. *"All inetOrgPerson entries under dc=example,dc=org"*

2.1.3 Access

The type of access granted will affect which LDAP operations can be performed. Most LDAP servers provide a set of access levels such as:

- Add an entry
- Delete an entry
- Access an entry (does not give access to any attributes)
- Read an attribute
- Modify an attribute
- Search an attribute

Most LDAP operations require a combination of these permissions.

2.1.4 ACI

An Access Control Item is a rule or tuple that defines a level of access for some subject(s) to some object(s).

2.1.5 ACL

An Access Control List is a list of ACIs. Depending on the language used by the server the list may be ordered or the items may have the same effect whatever order they are written in.

2.2 Related Concepts

Servers usually provide other forms of administrative control, which may or may not be integrated into the access control language.

- Size limits: these restrict the number of search results that can be obtained from a single LDAP operation.
- DIT Content Rules: these control the attributes and auxiliary object classes that can be placed in entries of a given structural class (see section 4.1.6 of [RFC4512]). They are closely related to schema, but in some servers are implemented as part of the access-control system.
- DIT Structure Rules: these define the types of entry that can appear in each part of the DIT. They are not widely implemented.

3 LDAP Servers

The access control capabilities of some representative LDAP servers are summarised in this section.

3.1 Netscape DS / Sun DSEE / Red Hat DS / iPlanet DS / Fedora DS

This is a family of similar products derived from the original University of Michigan *slapd* server. Access-control information is stored in ACI attributes, and can be placed in any entry that is a direct ancestor of the entry being controlled. The syntax is powerful, with support for:

- Target entry selection filters
- Attribute lists
- Wildcards in target specification
- Subject selection by filter
- Bind-strength specification for subjects
- Group subjects (though the group object is limited to a pre-defined list of object classes)
- Macro ACIs – a value matched in the target specification can be substituted into other fields in the rule.

Where multiple rules apply to the same target, they are combined. Deny rules always override grants.

These servers do not implement DIT Content Rules, but they do offer some control over added entries as the proposed entry is subjected to a subset of the full access checks before being created.

3.2 IBM Tivoli Directory Server

TDS provides two types of access control rules: filtered and non-filtered. Both are stored in the DIT but cannot be combined in one entry. Filtered rules have a richer syntax and are more flexible than the earlier type.

Filters can only be used in target selection – not to define subjects. They are standard LDAP search filters, and there is no regular expression or macro facility.

Rules can control the *add entry* permission, but if this is granted there is no way to control *what* is added. TDS does not implement DIT Content Rules or DIT Structure Rules.

Groups may be used in access-control rules. Dynamic groups and nested groups are supported.

There is a useful set of attribute classes, which allows access-control rules to address a whole class without having to list each attribute name. New attributes can be assigned to one of these classes, but it is not possible to create new classes.

3.3 Apache DS

This is a relatively new project which plans to implement a rich superset of LDAP from the ground upwards. Interestingly, it appears to have adopted the X.500 administrative model and the associated access control system.

3.4 OpenLDAP

OpenLDAP is unique in the list of servers considered here. It has an access control system that looks like a programming language, and clauses are evaluated in a defined order. The language is very powerful:

- Object entries can be selected by DN, subtree, LDAP search, or regular expression.
- Object attributes can be selected by name, objectclass, or list
- Individual attribute values can be selected
- Subjects can be selected by DN, subtree, LDAP search, by reference to a value in the object entry, and by substitution of values matched in a regular expression while selecting an object.
- Bind-strength and encryption requirements can be specified.
- There is a syntax for connecting objects and subjects using Venn-diagram-like sets.
- Access can be explicitly granted or denied for read, write, search, compare, authenticate, and disclose-on-error.

- Rules can terminate the access evaluation or allow execution to drop through to further rules in the set.

OpenLDAP implements DIT Content Rules, and also enforces access rules when adding entries. It is the only server considered here that is able to prevent a malicious user from discovering the existence of an entry that they cannot read (though care is required if applying this to non-leaf entries).

4 Defining the Requirements

Most designs start with a simple policy along the lines of *"Keep the bad guys out"* and progress through a number of iterations where the requirements of the "Not Bad Guys" are discovered and included.

It is worth trying to keep the policy as simple as possible, as writing access-control lists can be hard and the complexity can go up much faster than the number of lines in the policy. It is also important to remember that the policy (and thus the rules that implement it) will probably have to be changed in the future: whoever does it will have to gain a deep understanding of the existing system before they can safely move forward.

When discussing the access policy it is useful to have a diagram of the proposed DIT with some sample entries in it. You can then ask questions such as "Should *this* person be able to do *this* operation on *this* entry?" The answers help to refine the policy, and also directly contribute to the test suite.

Here are some sample policies. The early ones are fairly simple, and could be adopted without too much detailed discussion.

4.1 Public read-only policy

Everyone in the world can read all entries: they can see all attributes except userPassword.

Changes can only be done by the administrator.

4.2 User read-only policy

Authenticated users can read all entries: they can see all attributes except userPassword.

The owner of any entry (who presumably knows the existing password) can change the userPassword.

Other changes can only be done by the administrator.

4.3 Delegated administration policy

Everyone in the world can read all entries: they can see all attributes except userPassword.

Each department in the organisation has a group of clerks to maintain the directory. Clerks may create and modify person entries in the department's *people* area and group entries in the department's *groups* area. (Note that this allows an existing clerk to create a new one with the same powers.)

4.4 Controlled visibility policy

Everyone in the world can read entries that are marked as *public*: they can see all attributes except userPassword.

Authenticated users can read all entries in their own department: they can see all attributes except userPassword.

Users may change their own passwords.

4.5 Subset visibility policy

Entries are explicitly marked with a visibility category in the *exampleVisibility* attribute:

- **internet** allows the whole world to see the entry.
- **users** allows all authenticated users to see the entry.
- **department** allows authenticated users in the same department to see the entry.
- **any other value (or none)** means that only the entry owner can see it.

Visibility of attributes is also to be controlled. In non-person entries, only *objectclass*, *o*, *ou*, *dc*, *description* attributes shall be visible. Person entries shall be controlled in detail:

- Anonymous users can see: *objectclass, cn, sn, displayname, mail, uniqueIdentifier*
- Authenticated users can also see: *telephoneNumber*
- Same-department users can also see: *uid*

- Users can also see their own: *description, exampleVisibility*

Users may change their own *userPassword* and *exampleVisibility* attributes.

All other access is denied

Note that this is a default-deny policy, so it does not need to say anything about protecting *userPassword* visibility.

5 Design Principles

There are no hard-and-fast rules for writing access control lists, and the varying capabilities of different LDAP servers sometimes require very different approaches. However, I have found these principles to be a useful guide:

1. ACLs are programs - they should be handled by programmers, not by data administrators.
2. Place ACLs on the smallest possible number of entries. If there are lots of similar ACLs controlling different parts of the DIT then it becomes hard to change them if the policy changes.
3. Try to keep ACLs out of entries that need to be routinely created and deleted. This can be difficult on some servers, particularly where regular expressions and macros are not available.
   This follows from principles 1 and 2.
4. DIT and schema design may be affected by ACL capabilities. If different entries should have different access rules then either the content of the entry, its membership of some groups, or its position in the DIT must be used to select which rules to apply.
5. Add new attributes to the schema to drive access control. Administrators can control the values of these attributes without having to understand the ACL implementation.
6. Write the tests *first*, as this helps to clarify exactly what the requirements are. Keep the test suite up to date and run it frequently when working on ACLs.
7. Try to write ACLs on the 'default deny' principle. It is easier to understand and verify the ACLs if you do not mix 'grant' and 'deny' rules. This is more important in the 'ACLs as attributes' system than it is where ACLs behave like programs.
8. Don't write individual account IDs into ACLs: give permissions to groups and allow administrators to control membership of the groups.

9. Dynamic groups can often be used to identify subjects if the access control language is not flexible enough to do it directly.
10. Don't forget that some LDAP servers do not enforce the same rules when adding an entry that they do when modifying the same entry later. This can make it impossible to control the type of entries that administrators can create.
11. DIT Content Rules can be useful to control which auxiliary classes can be used with each structural class. They can also add to the MUST and MAY lists and subtract from the MAY list of attributes. This is often better than using ACLs to control the content of entries. Not all servers implement these rules.
12. DIT Structure Rules can be useful in controlling the shape of the DIT, but very few servers implement them.
13. Search limits can also be useful in access control.
14. Design the DIT so that DNs do not expose sensitive information. Many LDAP servers are unable to prevent an attacker from detecting an entry whose name they have guessed. Some will even return an entry (and thus its RDN) when the RDN attribute is not supposed to be visible to the requesting user.
15. Avoid spaces and punctuation in RDNs, as this can make regular expressions and dynamic group attributes harder to construct. Naming entries with opaque hexadecimal or Base64 values of *uniqueIdentifier* satisfies this principle and the previous one.
16. Remember to give ANON enough access to the root DSE. Some LDAP clients will not start properly if they cannot read certain attributes from the root DSE.
17. ANON may need some access to user entries to permit authentication. This can be particularly awkward where LDAP clients expect to use uid or mail address rather than DN to identify the entry to bind.
18. Be wary of subtree replication: the replica servers may not contain all the ACLs (or entries providing data for ACLs).

6 Schema Design

LDAP stores information as attributes in entries and also in the structure of the DIT. There are trade-offs to be made and the final design must take account of access control policy as well as the structure of the information to be stored.

6.1 DIT shape

In *LDAP Schema Design* [FINDL05] I suggest collecting all *person* entries into a single location. This is a good policy for a DIT serving a single organisation. If the access control policy is to be based on membership of particular departments or groups, there needs to be some way to define that membership. The two obvious choices are:

1. An attribute in each entry
2. A group object elsewhere in the DIT

Each has advantages and disadvantages, and the choice will depend to some extent on how the entries are to be managed. In fact the two concepts are not mutually exclusive: most LDAP servers support dynamic groups whose membership is defined by a search string (and thus by entry attributes). Using such a dynamic group to define access rights is an obvious development.

Where a single DIT must serve a number of organisations (e.g. in a service-provider environment) it is likely that access policies will be defined in terms of position in the DIT. Servers that implement macros or regular expressions in their access control language have an advantage here, as a single rule can be written to apply to all organisations. If using a less-capable LDAP server it will be necessary to define one or more dynamic groups for each organisation and to create ACLs referring to them by name.

6.2 Attributes

It is often useful to add new attributes specifically to drive access-control rules. People often ask for an 'ex-directory' attribute, but in keeping with the design principles in section 5 above, I prefer to use an attribute that *grants* specific levels of access. Thus a design for “The Example Organisation” would include a text attribute called *exampleVisibility* with a defined set of values. An entry with no value in this attribute would not be visible (except perhaps to administrators). Values 'department', 'organisation', 'world' would give progressively wider visibility to the entry.

6.3 Using groups

Another approach to controlling visibility would be to make *exampleVisibility* a DN-valued attribute and use it to list groups that in turn list users who may see the entry. A potential problem with this scheme is that some LDAP servers may not be able to represent anonymous users in groups.

7 The Process of Writing the ACLs

7.1 Requirements capture

As mentioned in section 4 above, it is likely that this will be an iterative process tightly coupled with schema design. In the first round I generally try to establish a broad picture:

- What are the *subjects* (section 2.1.1) and how can they be grouped into classes?
- What are the *objects* (section 2.1.2) that we must control access to? Don't forget the non-leaf objects that make up the structure of the DIT.
- What is the security posture of the organisation – open to the world or tightly closed?
- How will entries be created and managed? If the directory will be the master source of data, who will be administering it?
- What will the directory be used for? What access is required for each application to work?

Working from that information I produce one or more outline designs, concentrating on the overall shape of the DIT. These designs form the basis for a second round of discussion where specific use-cases can be fitted into the model. It is often useful to work through some real examples to see how the data will be represented and used. At this point it is possible to point to an *object* in the DIT and ask what access specific *subjects* should have to it. Each of these questions and answers should be recorded carefully, as each one should provide material for at least one test in the test suite.

7.2 Build a Test Suite

Once the initial schema design has been agreed (and sometimes before that if there are significant choices to be made) I always build a sample DIT. This has the main structural components and a number of example leaf-entries representing the main types of data to be stored. There are no ACLs yet, but there are enough entries to allow each question from the first stage to be represented.

The sample DIT can be given to application developers at this stage, so they can make progress while the ACLs are being written.

The next step is to start writing the ACL test suite. I normally use Perl for this, with the Net::LDAP and Test::Simple modules. The structure follows a simple pattern:

1. Open a number of LDAP connections to the server and bind each as a different *subject*. This usually includes:

   Anonymous

   The LDAP server manager (*rootdn*) – this user is not subject to access control.

   One or more accounts representing different classes of user: ordinary people, admin staff, automatic processes etc.
2. For each class of *object*, test the read access granted to each class of *subject*.
3. Test write access.
4. Test creation and deletion of entries.

Even a simple access policy can give rise to 50 or more tests.

7.3 Choose a Style

Access control languages may not be as expressive as Perl, but in most cases there is *still* "more than one way to do it".

7.3.1 Grants only or mixed?

ACLs can be written entirely using "grant" rules, leaving all non-defined access to be denied by the "default deny" rule that is built into most LDAP servers. This is probably the safest option, and is usually the easiest to understand when working with non-ordered ACLs (most servers that keep ACLs in attributes within the main DIT).

At first sight this approach would seem to generate very large ACIs as each attribute has to be explicitly controlled, but mechanisms like TDI's attribute classes and OpenLDAP's use of objectclasses in ACLs help to keep the size in check.

For very simple cases, a mixture of "grant" and "deny" rules can result in a smaller ACL to achieve a given result. This is commonly used where the access policy says something like *"Everyone can read everything except userPassword"*. There are risks though: *userPassword* may not be the only attribute that stores password data and this rule would leave the other forms unprotected.

7.3.2 Match and define or accumulate permissions?

Most servers allow more than one ACI to contribute to a given access decision. This allows for two distinct styles:

- Each ACI matches a particular case and defines the entire access for that case.

- The access granted by several ACIs is combined to make the final decision.

If using the second style it is best to avoid “deny” ACIs as the rules for combining these with overlapping “grants” can be difficult to deal with (even if they are documented, which is not always the case!)

7.3.3 Separate rules for read and for write?

If accumulating permissions it is possible to use separate rules for read/search operations and for write/add/delete operations. In complex cases this may help to make the rules easier to understand.

7.4 Choose control points

In many cases, some rules apply only within a particular subtree of the DIT. The root of each such subtree is an obvious place to put the rules (or at least to have them take effect from).

The design principles in section 5 above suggest that there should be as few control points as possible, so rules will normally take effect from high up in the DIT.

Where rule targets are defined by filters or regular expressions it may be possible to place every rule at the root suffix of the DIT. This has the advantage that any later extension of the DIT will be covered automatically.

7.5 Write and Test ACLs

This tends to be an iterative process, starting with a simple rule giving basic access to anonymous users. I would expect to write one or two rules and then run the test suite to check the effect.

If using the “grant rules only” (default deny) scheme it should be possible to write rules that do not interfere with each other, but in practice this is quite hard so you should expect to go back and modify earlier rules as the ACLs develop.

The process of writing ACLs usually suggests more tests, and these should be written straight away.

8 Server specifics

Each family of LDAP servers has its own access-control language. The principles are similar at first sight, but implementation details vary considerably. This means that the approach to implementing a particular policy could be very different for each server: in fact there are policies that some servers cannot implement at all.

It is essential to have the server's technical documentation to hand and to read it very carefully. ACLs can have effects that are not intuitively obvious,

and it is sometimes necessary to run servers in debug mode to understand why they behave in particular ways.

This section points out one or two features of each server's ACL implementation. It is a long way from being comprehensive.

8.1 Netscape DS / Sun DSEE / Red Hat DS / iPlanet DS / Fedora DS

ACIs can be placed in any ancestor of the entry they are to control. The control point is defined by the *target* clause, which is an LDAP URL.

Groups used in access-control must be one of the standard group classes. Inventing a new group class and adding member attributes does not work.

Macros are a powerful feature that can be used to make a single ACI apply to many similar targets. For example, if an ACI's target is defined as `"ldap:///($dn),dc=example,dc=org"` then the associated subject (userdn) can be defined as `"ldap:///dc=people,[$dn],dc=example,dc=org??sub?"` - the effect is to make the rule apply where a user is accessing entries in their own department (see examples 10.3 and 10.4).

The server applies ACLs to the content of an entry that is about to be added, but it allows the add to proceed if it finds *any* writeable attributes. This provides some control over the creation of entries, but is not enough to be completely safe.

8.2 IBM Tivoli Directory Server

The default ACL looks like this:

```
aclentry: group:CN=ANYBODY:system:rsc:normal:rsc:restricted:rsc
```

It gives world read-only access to the common attributes of all entries.

However, if you add *any* ACL to the DIT it will totally override the default in its subtree. If you want to combine extra permissions with the default then you must add the above ACI explicitly.

TDS supports two types of ACL for historical reasons. For any non-trivial task, it is probably best to use filtered ACLs alone.

There is no way to control the type and content of new entries: once you grant an account 'a' permission on a node they can create anything they like under it even if they cannot then modify what they have created. If the attacker remembers to make themselves the owner of the entry when they create it then they have total freedom to do what they want with it in the future.

TDS does not support macros or regular expressions in ACLs. This makes it very hard to satisfy the first three design principles in section 5.

8.3 OpenLDAP

OpenLDAP supports one ACL per backend plus a default ACL. The default is *appended* to each backend-specific ACL. It is therefore wise to end every ACL with a catch-all statement such as:

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

Always turn on the checking of added entries:

```
add_content_acl yes
```

This feature was introduced in version 2.4.13 and it causes ACLs to be applied fully to the content of entries that are about to be added. Use DIT content rules to gain further control of added entries.

OpenLDAP allows an object class name to be used to define the list of attributes that an ACI applies to. This is a very useful feature, particularly if you define new object classes specifically for the purpose. See example 10.5 where this is used extensively.

If basing an access rule on a value match you must make sure that the attribute has a suitable matching rule defined in the schema. If it does not, then you can nominate a rule:

```
access to attrs=namingContext
        val/distinguishedNameMatch="o=example"
        by * none
```

9 Gotchas

However hard you try, there are some things that ACLs cannot protect. The exact list varies from one server to the next, but a few seem to be very common.

1. Protecting a non-leaf object does not protect objects below it in the DIT. Thus, even if a user has no access at all to `ou=deptA,dc=example,dc=org` they could still issue a search that would return data from `cn=private,ou=deptA,dc=example,dc=org`
2. It is very hard to prevent a malicious user from proving the existence of an entry whose DN they have guessed. Even if the user has no access to `cn=secret,dc=example,dc=org` they can use it as the base of a search: if it exists they will get no answers but no error, whereas if it does not exist they will get a “no such object” error.
   OpenLDAP provides the “disclose” permission to control this, but is still unable to prevent the disclosure of non-leaf objects that have accessible objects beneath them. Most other servers do not attempt to control this disclosure at all.

3. Many LDAP servers do not apply access control to the contents of an entry that is about to be added. Thus a user who has permission to create addressbook entries may be able to create new accounts, groups etc. Even if access control *is* applied to new entry contents, it is not always done as thoroughly as when modifying existing entries.

10 Examples

In these scenarios, the Example Organisation has a DIT rooted at `dc=example,dc=org`. The diagrams show the structure of the DIT, with heavy outlines indicating those entries that are created before the test-suite runs.

A set of tests is outlined for each example, and sample ACLs for several different LDAP servers are given for comparison. Lines have been folded for readability so the text here may not be directly loadable: refer to the example distribution pack for usable files.

Where servers have default ACLs it is assumed that they have not been changed. In OpenLDAP the ACLs presented here are intended to be used in the backend configuration, along with a simple world-read ACL in the global configuration:

```
access to * by * read
```

A distribution pack containing all the files for these examples is available from: http://www.skills-1st.co.uk/pub/code/acl-examples.tgz

10.1 A Simple Account Registry

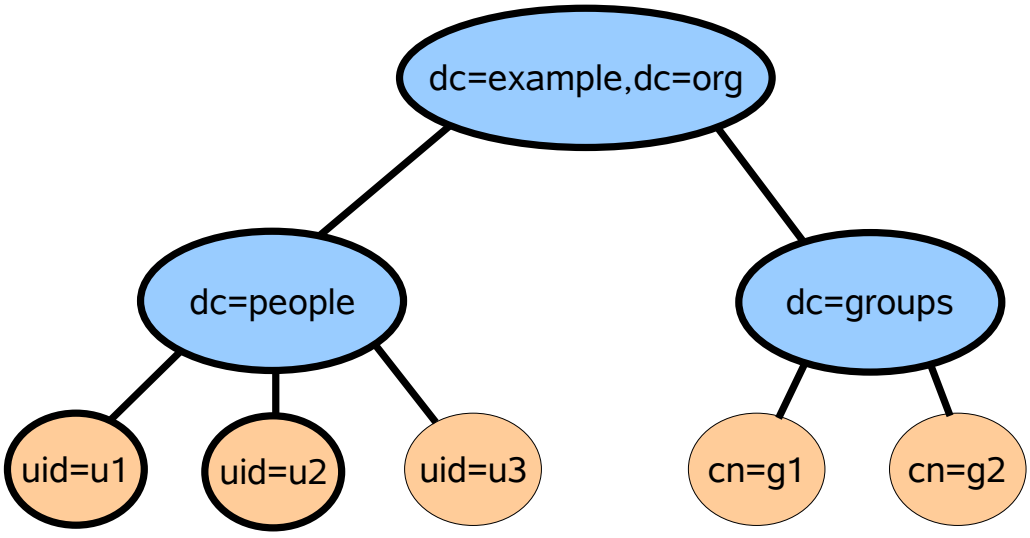

Figure 1: Simple user registry

A very simple DIT (Figure 1) and a very simple access policy. The DIT has one arc for users and one for groups. Everyone (including anon users) can

read all attributes except for *userPassword*. Users can change their own passwords. This setup might be used to support a simple website. It is called "simple" in the distribution pack.

The obvious control point is `dc=example,dc=org` and the expression of the policy suggests that we might use a "deny" rule to control *userPassword*.

10.1.1 Tests

This is a very simple scenario but it still needs quite a lot of tests:

- Open LDAP connections and bind them as *Anon*, *rootdn*, and *uid=u1,dc=people,dc=example,dc=org*
- Read `dc=example,dc=org` using each connection and check that basic data is visible.
- Read `dc=people,dc=example,dc=org` using each connection and check that basic data is visible.
- Read `dc=groups,dc=example,dc=org` using each connection and check that basic data is visible.
- Read `uid=u1,dc=people,dc=example,dc=org` using each connection and check that basic data is visible. Check that the *anon* and *u1* connections cannot read *userPassword* from u1's entry.
- Check that the *u1* connection can read everything except *userPassword* from u2's entry.
- Use the *u1* connection to change u1's password. Check that the new password works.
  I like my tests to return everything to the basic starting point, so I would use the *u1* or the *rootdn* connection to reset the password at this point.
- Check that the *u1* connection cannot create or modify entries. Check that the correct error codes are returned (LDAP_INSUFFICIENT_ACCESS in this case)

10.1.2 ACLs for Sun / Netscape servers

The documentation for these servers says that they allow world read access by default, but that is no longer the case in the version that I tested (Sun DSEE 6.0). The first ACI therefore provides public read access to all attributes except *userPassword*. The ACI is stored at `dc=example,dc=org` and has effect on the entire subtree below that point.

Note the use of the negative match on the target attributes. This can be risky (as with other negative forms) because two or more such ACIs may not combine in the way that the designer expects.

The DN "ldap:///anyone" is used by this family of servers to include all users including anonymous ones.

```
dn: dc=example,dc=org
changetype: modify
add: aci
aci: (targetattr != "userPassword")
  (version 3.0; acl "Make public objects visible to all";
     allow (read, compare, search)
     (userdn = "ldap:///anyone") ;)
```

One more ACI is required, to give users the ability to change their own passwords. In fact, this *is* part of the default set but including it explicitly does no harm. Note that the *write* permission granted here does *not* also grant read, search or compare.

```
# Allow users to change their own passwords
#
dn: dc=example,dc=org
changetype: modify
add: aci
aci: (targetattr = "userPassword")
  ( version 3.0; acl
     "allow userpassword self modification";
     allow (write) userdn = "ldap:///self";)
```

The two ACIs are both stored in the same entry, where they will each become a separate value of the *aci* attribute. Note that there is no order of preference: they could be evaluated in any order.

This ACL passes all the tests in the test suite.

10.1.3 ACLs for IBM TDS

TDS switches to default-deny mode as soon as it finds *any* ACI that applies to the object being considered. We therefore start by adding an ACI to explicitly permit read access to desired attributes. The control point is dc=example,dc=org and I have chosen to use filter ACLs throughout.

Note the use of attribute class “normal”. This class contains ordinary informational attributes such as *cn, sn, telephoneNumber* etc. The *userPassword* attribute belongs to the “critical” class so it will not be exposed.

TDS uses the special group DN cn=anybody to denote all users including anonymous ones.

```
dn: dc=example,dc=org
changetype: modify
add: ibm-filterAclEntry
ibm-filterAclEntry:
         group:CN=ANYBODY:(objectclass=*):normal:rsc
```

A second ACI is needed to allow users to change their own passwords. This uses another special DN (`cn=this`) to denote the user represented by the entry itself. This ACI grants write access to the single attribute *userPassword*.

```
dn: dc=example,dc=org
changetype: modify
add: ibm-filterAclEntry
ibm-filterAclEntry: access-id:
 cn=this:(objectclass=*):at.userPassword:grant:w
```

Two ACIs are sufficient to do the job. They are both stored in the same entry and the order in which they are defined is irrelevant.

This ACL passes all the tests in the test suite.

10.1.4 ACLs for OpenLDAP

OpenLDAP does not place ACIs in the DIT that they are controlling. Current (2.4.x) versions allow the use of configuration files or a configuration database accessed through LDAP. All the examples here use the file format as that is slightly simpler than the LDIF form.

An OpenLDAP ACL has a distinct flow of control. Directives are evaluated in order until a terminating statement is reached.

The first directive in this example deals with the *userPassword* attribute, wherever it may occur in the DIT. If the attribute is being accessed by the user represented by the entry under consideration (`by self`) then the access "=w" is granted. This means that write operations are permitted but others are not. The case of a user accessing their own password is completely defined by this statement. Any other user trying to access the *userPassword* attribute would drop through to the next "by" field, which is designed to match all users including anonymous ones. The access granted is "auth" which is the minimum necessary to support authentication. All possible subjects wanting access to the *userPassword* attribute have now been covered, so evaluation would stop here.

```
access to attrs="userPassword"
        by self =w
        by * auth
```

The second directive handles access to all other attributes and pseudo-attributes. It allows everyone to read everything. This does not expose *userPassword* as the first directive covered all access to that attribute.

```
access to *
        by * read
```

This ACL passes all the tests in the test suite.

10.2 Adding a Data Administrator

This is an extension of the account registry example. Power is to be delegated to a data administrator, who will be allowed to create and modify person and group entries. To preserve sanity, the administrator is limited to creating each type of entry in the correct part of the DIT and is not permitted to add any auxiliary object classes.

Note that the administrator is *not* the “directory manager” or “rootdn”: these terms refer to an all-powerful account that is not subject to access control.

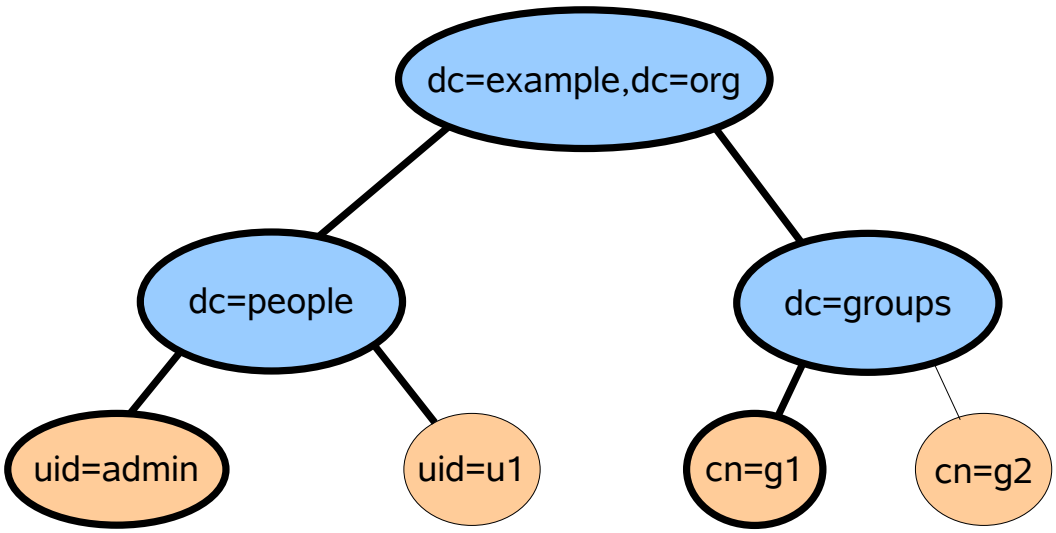

Figure 2: Delegated administration

This is called “structure-control” in the distribution pack.

The requirement to limit what type of entry can be created under each arc means that there will have to be three control points:

- `dc=example,dc=org`
- `dc=people,dc=example,dc=org`
- `dc=groups,dc=example,dc=org`

10.2.1 Tests

Most of the tests listed for the simple scenario in section 10.1.1 also apply here. In addition, tests are needed for:

- The delegated administrator should be able to create and modify person entries under `dc=people,dc=example,dc=org` and groups under `dc=groups,dc=example,dc=org`
- The delegated administrator should *not* be able to create entries of the wrong objectclass in either location.
- The delegated administrator should *not* be able to add unwanted attributes or objectclasses to any entry.

10.2.2 ACLs for Sun / Netscape servers

The first ACI is the same as for the simple account registry in section 10.1: it gives public read-only access to all attributes apart from *userPassword*.

```
dn: dc=example,dc=org
changetype: modify
add: aci
aci: (targetattr != "userPassword")
  (version 3.0; acl "Make public objects visible to all";
  allow (read, compare, search)
  (userdn = "ldap:///anyone") ;)
```

The next ACI controls what the delegated administrator can do in the `dc=people` subtree. Note that the ACI itself is placed at `dc=example,dc=org` but has its target set to `dc=people,dc=example,dc=org`.

The ACI has a target filter that restricts it to objects of class *person* or *account*. As this family of servers applies some access-control rules to the content of new entries, this should prevent the administrator from creating or editing entries of other types. Note that it cannot prevent them from adding auxiliary object classes.

As a further protection, the ACI has a list of target attributes. This restricts the attributes that the administrator can modify in existing entries. Unfortunately this protection does not apply when creating new entries, so the administrator can add any attributes they like provided the new entry has a structural object class of *person* or *account*.

```
dn: dc=example,dc=org
changetype: modify
add: aci
aci: (target = "ldap:///dc=people,dc=example,dc=org")
     (targetfilter =
       "(|(objectclass=person)(objectclass=account))")
     (targetattr = "cn || sn || uid || description ||
       userPassword || objectclass")
     (version 3.0; acl
       "Admins may add and modify users in the people arc";
       allow (write, add, delete)
       (userdn =
         "ldap:///uid=admin,dc=people,dc=example,dc=org")
     ;)
```

The remaining ACI controls access to the `dc=groups` subtree. It has filters and attribute lists analogous to those used for `dc=people`.

```
dn: dc=example,dc=org
changetype: modify
add: aci
aci: (target = "ldap:///dc=groups,dc=example,dc=org")
  (targetfilter = "(objectclass=groupOfNames)")
  (targetattr =
    "objectclass || cn || member || description")
  (version 3.0; acl
    "Admins may add and modify groups in the groups arc";
    allow (write, add, delete)
  (userdn =
    "ldap:///uid=admin,dc=people,dc=example,dc=org") ;)
```

The ACLs implement most of the requirements of the policy, but they fail to control the content of new entries (apart from requiring them to contain the desired objectclass). There does not appear to be a way around this problem in Sun DSEE 6.0.

10.2.3 ACLs for IBM TDS

As in the first example, an ACL at `dc=example,dc=org` will grant overall read permission and will also allow users to change their own passwords.

```
dn: dc=example,dc=org
changetype: modify
replace: ibm-filterAclEntry
ibm-filterAclEntry:
 group:CN=ANYBODY:(objectclass=*):normal:rsc
ibm-filterAclEntry:
 access-id:cn=this:(objectclass=*):at.userPassword:grant:w
```

The next task is to allow the admin account to work on person entries in the `dc=people` subtree. This ACL has two ACIs: one to permit the creation of new entries and one to permit existing entries to be modified or deleted. Each ACI has a filter that selects where it takes effect. Attributes in the "normal" class are given read, write, search and compare permissions, but *userPassword* is only given write: this allows passwords to be set and changed, but not read.

```
dn: dc=people,dc=example,dc=org
changetype: modify
replace: ibm-filterAclEntry
ibm-filterAclEntry:
 access-id:uid=admin,dc=people,dc=example,dc=org:
 (objectclass=organizationalUnit):object:grant:a
ibm-filterAclEntry:
 access-id:uid=admin,dc=people,dc=example,dc=org:
 (|(objectclass=account)(objectclass=person))
 :object:grant:d:normal:rwsc:at.userPassword:grant:w
```

The ACL for the groups subtree is similar, but does not have to account for passwords.

```
dn: dc=groups,dc=example,dc=org
changetype: modify
replace: ibm-filterAclEntry
ibm-filterAclEntry:
 access-id:uid=admin,dc=people,dc=example,dc=org:
 (objectclass=organizationalUnit):object:grant:a
ibm-filterAclEntry: access-
id:uid=admin,dc=people,dc=example,dc=org:
 (objectclass=groupOfNames):object:grant:d:normal:rwsc
```

TDS does particularly badly on this test. It is completely unable to control the content of new entries so the policy limiting the actions of the administrator cannot be implemented at all.

10.2.4 ACLs for OpenLDAP

The first step in implementing this policy is not an access rule at all, but a pair of DIT Content Rules. These constrain the schema so that no auxiliary classes can be added to person or group entries. Once a content rule is in force, even the *rootdn* cannot break it.

Each rule starts with the OID of the class that it applies to. Note that it has no effect on superclasses or subclasses: if you want to control those then they each need an explicit rule. In this case the two rules are enough because later rules will constrain the administrator to working on *inetOrgPerson* and *groupOfNames* entries, and neither has any subclasses in the schema in use.

```
# Content rule for inetOrgPerson
#
ditcontentrule ( 2.16.840.1.113730.3.2.2
        NAME 'dcrPerson'
        DESC 'Limit aux classes on inetOrgPerson entries'
        )

# Content rule for groupOfNames
#
ditcontentrule ( 2.5.6.9
        NAME 'dcrGroup'
        DESC 'Limit aux classes on group entries'
        )
```

The next step is to turn on strict enforcement of ACLs when new entries are being added. This feature was introduced in version 2.4.13 and is highly recommended, though it is not strictly needed in this case because the DIT Content Rules provide enough control.

```
add_content_acl yes
```

The first access rule controls the *userPassword* attribute. It is very similar to the rule in the simple example in section 10.1 above, but now has a field giving write-only access to the delegated administrator.

```
access to attrs="userPassword"
     by dn.exact="uid=admin,dc=people,dc=example,dc=org" =w
     by self =w
     by * auth
```

The second rule permits the administrator to create new entries directly under `dc=people`. This is done by granting write access to the “children”

pseudo-attribute. The rule has no effect on other users due to the "`by * break`" field at the end.

```
access to dn.exact="dc=people,dc=example,dc=org"
     attrs="children"
     by dn.exact="uid=admin,dc=people,dc=example,dc=org"
        write
     by * break
```

Creating entries also requires access to the entry to be added. The next rule grants this to the administrator, but only where the objectclass is *inetOrgPerson*. This prevents other entry types from being created of modified here. Again, the rule has no effect on other users.

```
access to dn.onelevel="dc=people,dc=example,dc=org"
     filter="(objectClass=inetOrgPerson)"
     by dn.exact="uid=admin,dc=people,dc=example,dc=org"
        write
     by * break
```

A similar pair of rules controls the creation and modification of entries under `dc=groups`:

```
access to dn.exact="dc=groups,dc=example,dc=org"
     attrs="children"
     by dn.exact="uid=admin,dc=people,dc=example,dc=org"
        write
     by * break

access to dn.onelevel="dc=groups,dc=example,dc=org"
     filter="(objectClass=groupOfNames)"
     by dn.exact="uid=admin,dc=people,dc=example,dc=org"
        write
     by * break
```

Finally, the default access covering all other cases is set:

```
access to * by * read
```

The ACLs pass all the tests, showing how useful DIT Content Rules can be. It would be very hard to implement this policy completely using access-control alone.

10.3 Limiting Visibility of Entries

In this example we consider an organisation with several departments. Some entries are marked for public visibility using the *exampleVisibility* attribute. Any that are not marked can only be seen by users in the same department.

This is the “local-visibility” example in the distribution pack.

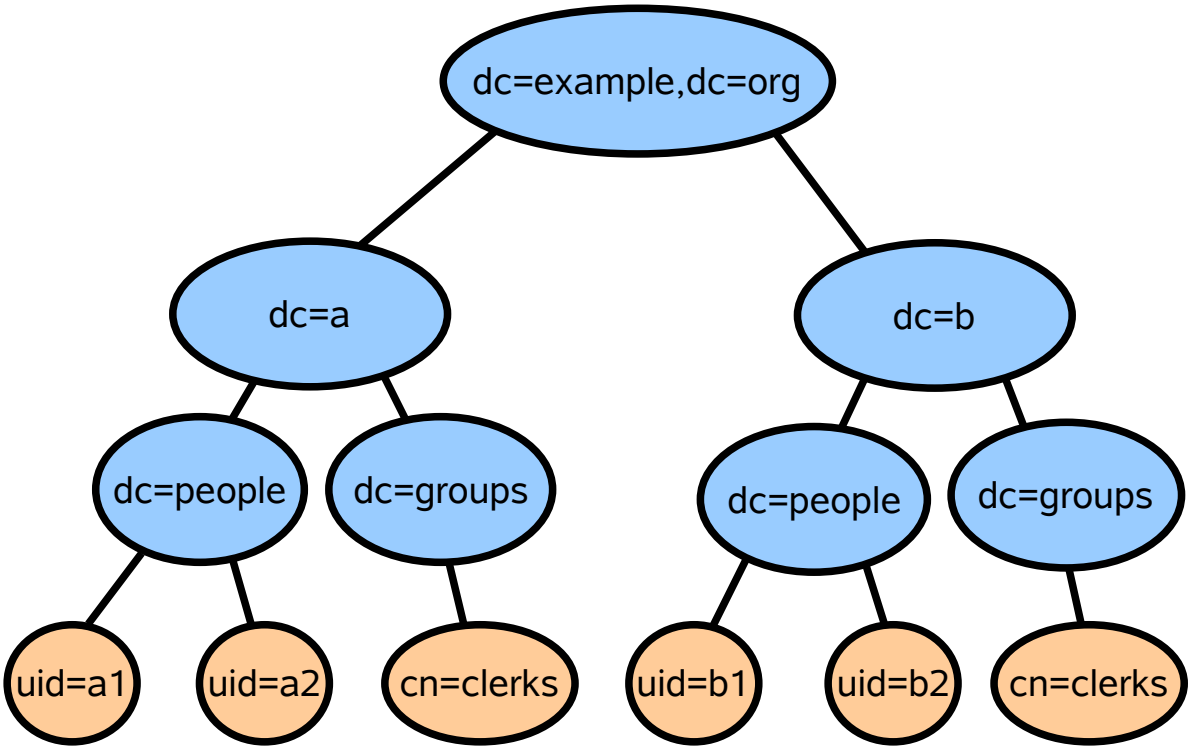

Figure 3: The Local-Visibility scenario

The departments `dc=a` and `dc=b` are marked for public visibility, as is the suffix node `dc=example,dc=org`. All other entries should be protected.

10.3.1 Tests

- Authenticate as users a1, a2, b1, b2
- Check that all users (and ANON) can read the public entries
- Check that ANON cannot read or search other entries
- Check that a1 can see a2 and the clerks group in department A
- Check that b1 cannot see anything inside department A

The approach to implementing this policy is very different for each server family.

10.3.2 ACLs for Sun / Netscape servers

These servers have a very useful macro facility, which allows one ACL at the root of the tree to control any number of departments.

Public visibility is implemented using a target filter that triggers when *exampleVisibility* is "public":

```
dn: dc=example,dc=org
changetype: modify
add: aci
aci: (targetfilter = "(exampleVisibility=public)")
  (targetattr != "userPassword")
  (version 3.0; acl "Make public objects visible to all";
   allow (read, compare, search)
   (userdn = "ldap:///anyone") ;)
```

As before, the "change own password" ACI is copied in from the default set:

```
dn: dc=example,dc=org
changetype: modify
add: aci
aci: (targetattr = "userPassword")
  ( version 3.0; acl "allow password self modification";
  allow (write) userdn = "ldap:///self";)
```

The value of the macro facility becomes evident when controlling access to non-public entries in departments. The target URL contains the string "`($dn)`" which matches all entries in the tree under `dc=example,dc=org` and saves the matched string. The matched string is then substituted into the user DN using "[$dn]" to identify users in the same department as the entry being accessed. The use of square brackets in the substitution means that the matched string is first tried whole, and components are repeatedly removed until a match is found or the string is empty. This allows the rule to apply to an entire subtree.

```
dn: dc=example,dc=org
changetype: modify
add: aci
aci: (target = "ldap:///($dn),dc=example,dc=org")
  (targetscope = "subtree")
  (targetattr != "aci || userPassword")
  (version 3.0; acl
    "Users may see entries in their own department";
    allow (read, compare, search)
    (userdn =
       "ldap:///dc=people,[$dn],dc=example,dc=org??sub?")
  ;)
```

This ACL passes all the tests in the test suite.

10.3.3 ACLs for IBM TDS

TDS does not have ACL macros so we have to find another way to give users access to entries in their own department. The first job is to create a dynamic group for each department:

```
dn: cn=users,dc=groups,dc=a,dc=example,dc=org
changetype: add
objectclass: groupOfURLs
objectclass: ibm-dynamicGroup
cn: users
memberURL: ldap:///dc=people,dc=a,dc=example,dc=org??
  sub?(objectclass=*)

dn: cn=users,dc=groups,dc=b,dc=example,dc=org
changetype: add
objectclass: groupOfURLs
objectclass: ibm-dynamicGroup
cn: users
memberURL: ldap:///dc=people,dc=b,dc=example,dc=org??
  sub?(objectclass=*)
```

Membership of each group is defined by an LDAP search URL. It selects all entries in the department's `cn=people` subtree.

The first ACI is the usual one for public read access, with a filter to select just those entries marked with *exampleVisibility*=public.

```
dn: dc=example,dc=org
changetype: modify
replace: ibm-filterAclEntry
ibm-filterAclEntry:
  access-id:cn=this:(objectclass=*):at.userPassword:grant:w
ibm-filterAclEntry: group:CN=ANYBODY:
  (exampleVisibility=public):normal:rsc
```

Providing the “local department” access requires one ACI per department. They are all very similar, with the department name substituted in relevant positions. Each ACI grants read, search and compare access to a departmental subtree, using the dynamic group to select the users who should have access:

```
dn: dc=a,dc=example,dc=org
changetype: modify
replace: ibm-filterAclEntry
ibm-filterAclEntry:
  group:cn=users,dc=groups,dc=a,dc=example,dc=org:
  (objectclass=*):normal:rsc

dn: dc=b,dc=example,dc=org
changetype: modify
replace: ibm-filterAclEntry
ibm-filterAclEntry:
  group:cn=users,dc=groups,dc=b,dc=example,dc=org:
  (objectclass=*):normal:rsc
```

The ACL passes all the tests in the test suite, but it breaks a number of design principles. Every department needs a special group and its own ACI, and both of these need to be created whenever a new department is added. A templating system could do the job, but it does mean that administrative users cannot use LDAP directly to manage departments.

An organisation with hundreds of departments (or a service provider with thousands of customers) would end up with a very large set of ACLs. This would be a problem if a global policy change were required, and may have a performance impact.

10.3.4 ACLs for OpenLDAP

The use of regular expressions allows for a very elegant solution here.

The first directive is the usual one to allow authentication and to permit users to change their own passwords.

```
access to dn.subtree="dc=example,dc=org"
        attrs="userPassword"
        by self =w
        by * auth
```

Public-access entries are simply handled by a filter rule:

```
access to filter="(exampleVisibility=public)"
        by * read
```

A regular expression selects entries inside departments and gives access to the appropriate local users:

```
access to dn.regex="(dc=[^,]+,dc=example,dc=org)$"
        by dn.subtree,expand="dc=people,$1" read
        by * break
```

Finally, a default-deny rule takes care of everything else:

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

The regular expression is worth looking at in more detail. It acts on the DN of the entry being considered. There is a “$” to anchor the end of the pattern, but no “^” to anchor the start. Inside the parentheses is a pattern that matches exactly one level of entries named “dc=”. This means that the pattern will match any of these names:

- `dc=a,dc=example,dc=org`
- `dc=people,dc=a,dc=example,dc=org`
- `uid=a1,dc=people,dc=a,dc=example,dc=org`
- `dc=groups,dc=a,dc=example,dc=org`
- `cn=clerks,dc=groups,dc=a,dc=example,dc=org`

It will cause the DN of the department entry to be saved as “$1” (that is what the parentheses are for).

The “by” field is now able to construct a definition for the set of users who may read the entries matched by the pattern. “dn.subtree,expand” means that the value saved in “$1” will be substituted into the string which will then be used as a subtree specification. Thus for the example entries above,

any user in the subtree `dc=people,dc=a,dc=example,dc=org` will be granted read access. Other users including ANON are left to later rules by the "`by * break`" field.

Strictly speaking, there is a risk in the regular expression: it will also match entries where the RDN attribute name *ends with* "dc" so there might be others that are not domain-component names. This could be fixed with a little extra complexity.

The ACL passes all of the tests in the test suite.

10.4 Delegated Administration of Users and Groups

An organisation with the same DIT structure used in 10.3 above wants to give local administrative power to a group of clerks in each department. For simplicity, all entries have public read access.

This is the "local-admins" example in the distribution pack.

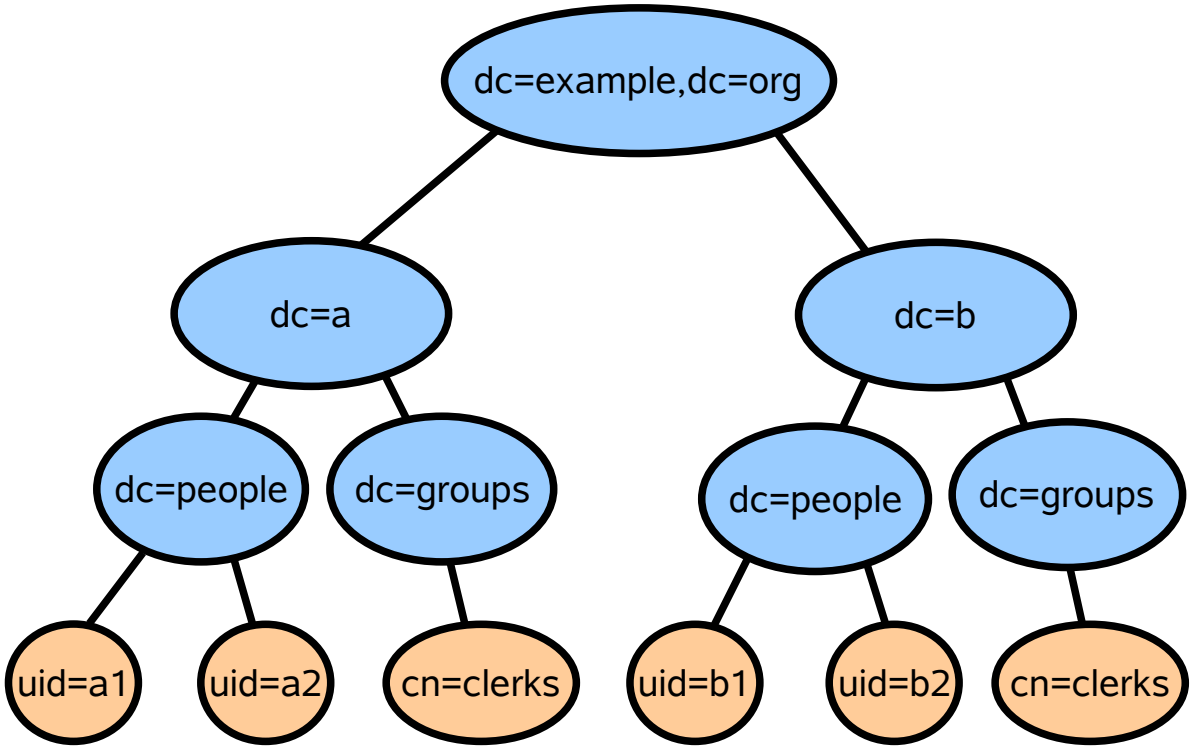

Figure 4: Delegated Administration

The principles are very similar, but this time we are granting write access, and the subjects are defined in groups. In the tests, user a1 is a member of the department A clerks group and b1 is a member of the department B clerks group.

10.4.1 Tests

- Authenticate as users a1, a2, ANON
- Check that all users can read each type of entry
- Check that a1 can create and modify entries in department A
- Check that a1 cannot create or modify entries in department B
- Check that a2 cannot create or modify entries in either department

10.4.2 ACLs for Sun / Netscape servers

The usual userPassword and public-item ACIs are used. The interesting ACI is the one controlling the clerks' access:

```
dn: dc=example,dc=org
changetype: modify
add: aci
aci: (target = "ldap:///dc=people,($dn),dc=example,dc=org")
  (targetfilter =
     "(|(objectclass=person)(objectclass=account))")
  (targetattr = "cn || sn || uid || description ||
       userPassword || objectclass")
  (version 3.0; acl "Clerks may add and modify users in
     their own department";
     allow (write, add, delete)
     (groupdn =
     "ldap:///cn=clerks,dc=groups,[$dn],dc=example,dc=org")
  ;)
```

As in the previous example, the target DN is selected by an LDAP URL containing a macro “($dn)”. Objectclasses and attributes are limited, and the clerks group is given access using a macro substitution.

The ACLs pass most of the tests, but are unable to prevent the clerks from creating entries with unwanted auxiliary objectclasses and attributes.

10.4.3 ACLs for IBM TDS

As in the previous example, the lack of macros and regular expressions means that each department needs its own ACLs. This is the one that gives the clerks group in department A access to create and modify person entries:

```
dn: dc=people,dc=a,dc=example,dc=org
changetype: modify
replace: ibm-filterAclEntry
ibm-filterAclEntry:
  group:cn=clerks,dc=groups,dc=a,dc=example,dc=org:
  (objectclass=organizationalUnit):object:grant:a
ibm-filterAclEntry:
  group:cn=clerks,dc=groups,dc=a,dc=example,dc=org:
  (|(objectclass=account)(objectclass=person)):
  object:grant:d:normal:rwsc:at.userPassword:grant:w
```

As in example 10.2 above, the ACLs give the clerks enough access to do their jobs, but TDS does not control the content of the entries that they add.

10.4.4 ACLs for OpenLDAP

I have included the complete set of access directives here to illustrate another style. The earlier examples used “match and define” ACLs, where this one uses the “accumulate permissions” style. The basic difference is that each access directive adds something to the permissions and allows control to drop through so all directives are involved in all decisions.

This example does not use DIT Content Rules; it controls entry content directly in the ACLs, so the `add_content_acl` directive is essential.

```
add_content_acl yes
```

The first access directive deals with clerks creating new person entries. It applies only to the “children” attribute of the `dc=people` entry. Note the use of “+w” to add write permission while leaving others alone, and the “break” control to allow execution of the ACL to continue at the next directive.

The DN is selected by a regular expression so that part of it can be used to define the group that is being given access.

```
access to
    dn.regex="^dc=people,(dc=[^,]+,dc=example,dc=org)$"
    attrs="children"
    by group/groupOfNames/member.expand=
      "cn=clerks,dc=groups,$1" +w break
    by * break
```

Clerk access to the actual entries under dc=people is controlled by the next directive. The DN is again selected by regular expression, but we also limit this rule to the desired objectclasses and attributes.

The attribute definition is interesting: it contains the pseudo-attribute “entry” and also three objectclasses. The effect is that the rule only permits clerks to write attributes that appear in one of those classes. This control is not as strong as a DIT Content Rule as it cannot prevent the addition of auxiliary classes (though it does prevent the addition of any extra attributes that they might allow).

Again, access is given by “+w” and execution continues with the next directive.

```
access to
  dn.regex="^[^,]+,dc=people,(dc=[^,]+,dc=example,dc=org)$"
  filter="(|(objectClass=inetOrgPerson)
            (objectClass=account))"
  attrs=
      "entry,@inetOrgPerson,@account,@simpleSecurityObject"
  by group/groupOfNames/member.expand=
     "cn=clerks,dc=groups,$1" +w break
  by * break
```

The rules for the `dc=groups` subtrees are very similar:

```
access to
  dn.regex="^dc=groups,(dc=[^,]+,dc=example,dc=org)$"
  attrs="children"
  by group/groupOfNames/member.expand=
     "cn=clerks,dc=groups,$1" +w break
  by * break

access to
  dn.regex="^[^,]+,dc=groups,(dc=[^,]+,dc=example,dc=org)$"
  filter="(objectClass=groupOfNames)"
  attrs="entry,@groupOfNames"
  by group/groupOfNames/member.expand=
     "cn=clerks,dc=groups,$1" +w break
  by * break
```

Treatment of the userPassword attribute is slightly different in this style: users are given write and authenticate access, and everyone else is just given authenticate. It is important to allow ANON to authenticate as all sessions start off anonymous!

```
access to attrs="userPassword"
  by self +wx break
  by * +x break
```

As we have not explicitly blocked read access to the userPassword attribute we must be very careful about what we do allow to be read. In the spirit of the "default deny" style, the next directive gives global read access to a very short list of attributes. In fact the access given is "+rscxd" which permits read, search, compare, authenticate, and disclose-on-error.

```
access to *
  attrs=
   "entry,objectclass,dc,uid,cn,sn,o,ou,description,member"
  by * +rscxd break
```

At the end of the list is a directive that just stops the execution and uses the accumulated permissions:

```
access to * by * stop
```

10.5 Detailed Control of Attribute and Entry Visibility

With the same DIT structure as 10.3 and 10.4 above, this example demonstrates the use of visibility attributes for entries in conjunction with rules limiting the set of attributes visible to different requesters.

Entry visibility is defined by the *exampleVisibility* attribute, using one of these values:

- **internet** – the entry is visible to everyone including anonymous users.
- **users** – the entry is visible to authenticated users.
- **department** – the entry is visible to users in the same department.
- Any other value (or none at all) – the entry is only visible to the user that it represents.

Attribute visibility for person entries is defined by the policy:

- Anonymous users can see: *cn, sn, displayname, mail, uniqueIdentifier*
- Authenticated users can also see: *telephoneNumber*
- Same-department users can also see: *uid*
- Users can also see their own: *description* and *exampleVisibility*

In *organization* and *organizationalUnit* entries, the following attributes are visible to anyone who can see the entry itself: *objectclass, o, ou, dc, description*.

This is the “subset-visibility” example in the distribution pack.

10.5.1 Tests

This policy does not define any write permissions, but it still needs a lot of tests for the various combinations of read permission.

Four person entries are created across two departments, and each is given a different *exampleVisibility* value. Access to each entry is then tested using:

- Anon
- A user in the other department
- A user in the same department
- The user represented by the entry

While testing combinations where the subject should not see the object, a further test for *detectability* is included. This tests whether the ACLs prevent the subject from using error-message analysis to detect whether the entry exists.

10.5.2 ACLs for Sun / Netscape

This family of servers does not explicitly control access to entries: an entry is visible if any of its attributes are visible.

The design of this set of ACLs assumes that the suffix entry `dc=example,dc=org` is marked with `exampleVisibility=internet`, and it combines organisational and personal entries in each rule on the basis that the attribute sets do not overlap.

The first ACI sets the minimum level of access for public entries:

```
dn: dc=example,dc=org
changetype: modify
add: aci
aci: (targetfilter = "(exampleVisibility=internet)")
    (targetattr = "cn || sn || displayname || mail ||
                   uniqueIdentifier || objectclass ||
                   o || ou || dc")
    (version 3.0;
     acl "Make public attributes visible to all";
    allow (read, compare, search)
    (userdn = "ldap:///anyone") ;)
```

Authenticated access allows a slightly larger set of attributes. It also permits access to entries marked for “users” visibility.

```
dn: dc=example,dc=org
changetype: modify
add: aci
aci: (targetfilter = "(|(exampleVisibility=internet)
                        (exampleVisibility=users))")
     (targetattr = "telephoneNumber || cn || sn ||
                    displayname || mail ||
                    uniqueIdentifier || objectclass ||
                    o || ou || dc")
     (version 3.0;
     acl "Make public objects visible to all";
     allow (read, compare, search)
     (userdn = "ldap:///all") ;)
```

Same-department users have a larger set of attributes and require a macro to identify them:

```
dn: dc=example,dc=org
changetype: modify
add: aci
aci: (target = "ldap:///($dn),dc=example,dc=org")
     (targetfilter = "(|(exampleVisibility=internet)
                        (exampleVisibility=users)
                        (exampleVisibility=department))")
     (targetattr = "uid || telephoneNumber || cn || sn ||
                    displayname || mail ||
                    uniqueIdentifier || objectclass ||
                    o || ou || dc")
     (version 3.0;
      acl "Users may see exampleVisibility=dept entries in
           their own department";
     allow (read, compare, search)
     (userdn =
         "ldap:///dc=people,[$dn],dc=example,dc=org??sub?")
   ;)
```

Users can always see their own entry, but the set of attributes is still limited:

```
dn: dc=example,dc=org
changetype: modify
add: aci
aci: (targetattr = "description || exampleVisibility ||
                    uid || telephoneNumber || cn || sn ||
                    displayname || mail ||
                    uniqueIdentifier || objectclass ||
                    o || ou || dc")
     (version 3.0;
      acl "Users may see their own entries entries";
      allow (read, compare, search)
      (userdn = "ldap:///self")
   ;)
```

This set of ACLs passes most of the tests in the test suite, but it fails to prevent non-visible entries from being detected.

The lack of explicit control on access to entries means that each ACI must deal with complete sets of attributes rather than each adding some permissions. This is clumsy, but at least we do not require one ACI for each principal/access combination.

10.5.3 ACLs for IBM TDS

As usual, the main problem is dealing with the “same department” parts of the policy. This requires one dynamic group and one ACL per department.

Here are the dynamic groups for departments A and B:

```
dn: cn=users,dc=groups,dc=a,dc=example,dc=org
changetype: add
objectclass: groupOfURLs
objectclass: ibm-dynamicGroup
cn: users
memberURL: ldap:///dc=people,dc=a,dc=example,dc=org??
           sub?(objectclass=*)

dn: cn=users,dc=groups,dc=b,dc=example,dc=org
changetype: add
objectclass: groupOfURLs
objectclass: ibm-dynamicGroup
cn: users
memberURL: ldap:///dc=people,dc=b,dc=example,dc=org??
           sub?(objectclass=*)
```

Most of the policy for attribute and entry visibility can be handled in a single ACL at the top of the DIT. TDS does not explicitly control read access to

entries: an entry is visible if the requesting user is allowed to see the RDN attribute.

The ACI for *organization* and *organizationalUnit* entries is fairly simple, though I have been lazy here and used the “normal” attribute class rather than listing the exact attributes that the policy allows.

```
dn: dc=example,dc=org
changetype: modify
replace: ibm-filterAclEntry
ibm-filterAclEntry: group:CN=ANYBODY:
    (&(exampleVisibility=internet)
      (|(objectclass=organization)
        (objectclass=organizationalUnit))):normal:rsc
```

The ACL continues with items dealing with the simpler cases of access to person entries. Each rule lists the complete set of attributes available to the class of user making the request. We cannot use the accumulate-permissions style here, as the rules have conditions on both the user and the visibility.

```
ibm-filterAclEntry: group:CN=ANYBODY:
    (&(exampleVisibility=internet)(objectclass=person)):
    at.objectclass:grant:rsc:at.cn:grant:rsc:
    at.sn:grant:rsc:at.displayname:grant:rsc:
    at.mail:grant:rsc:at.uniqueIdentifier:grant:rsc

ibm-filterAclEntry: group:CN=AUTHENTICATED:
    (&(|(exampleVisibility=internet)
        (exampleVisibility=users))
      (objectclass=person)):
    at.objectclass:grant:rsc:at.cn:grant:rsc:
    at.sn:grant:rsc:at.displayname:grant:rsc:
    at.mail:grant:rsc:at.uniqueIdentifier:grant:rsc:
    at.telephoneNumber:grant:rsc

ibm-filterAclEntry: access-id:CN=THIS:
    (objectclass=person):
    at.objectclass:grant:rsc:at.cn:grant:rsc:
    at.sn:grant:rsc:at.displayname:grant:rsc:
    at.mail:grant:rsc:at.uniqueIdentifier:grant:rsc:
    at.telephoneNumber:grant:rsc:at.uid:grant:rsc:
    at.description:grant:rsc:
    at.exampleVisibility:grant:rsc:
    at.userPassword:grant:w
```

To complete the job we need to add per-department ACIs to implement the local-department access policy.

```
dn: dc=people,dc=a,dc=example,dc=org
changetype: modify
replace: ibm-filterAclEntry
ibm-filterAclEntry:
    group:cn=users,dc=groups,dc=a,dc=example,dc=org:
    (&(|(exampleVisibility=internet)
        (exampleVisibility=users)
        (exampleVisibility=department))
      (objectclass=person)):
    at.objectclass:grant:rsc:at.cn:grant:rsc:
    at.sn:grant:rsc:at.displayname:grant:rsc:
    at.mail:grant:rsc:at.uniqueIdentifier:grant:rsc:
    at.telephoneNumber:grant:rsc:at.uid:grant:rsc

dn: dc=people,dc=b,dc=example,dc=org
changetype: modify
replace: ibm-filterAclEntry
ibm-filterAclEntry:
    group:cn=users,dc=groups,dc=b,dc=example,dc=org:
    (&(|(exampleVisibility=internet)
        (exampleVisibility=users)
        (exampleVisibility=department))
      (objectclass=person)):
    at.objectclass:grant:rsc:at.cn:grant:rsc:
    at.sn:grant:rsc:at.displayname:grant:rsc:
    at.mail:grant:rsc:at.uniqueIdentifier:grant:rsc:
    at.telephoneNumber:grant:rsc:at.uid:grant:rsc
```

This set of ACLs passes most of the tests in the test suite, but it fails to prevent non-visible entries from being detected. It also breaks several design principles by placing ACLs in entries that might be created routinely. Each ACI contains a long list of attributes, most of which are the same from one ACI to the next: this would be hard to maintain properly if the policy were to change in future.

10.5.4 ACLs for OpenLDAP

For this example I have chosen the “default deny” style with accumulating permissions. The policy defines varying access levels for particular sets of attributes, so the first job is to define objectclasses to represent each set. This is not essential but it does make the ACLs easier to read and maintain. Note that these objectclasses are *not* intended to be used in directory entries – they are just being used as a convenient way of referring to a set of attributes in ACLs.

```
objectclass ( 1.2.826.0.1.3458854.666.3.1
        NAME 'attrsetAnonVisible'
        DESC 'Attributes visible to anonymous users'
        AUXILIARY
        MAY ( objectclass $ cn $ sn $ displayname $
              mail $ uniqueIdentifier )
        )

objectclass ( 1.2.826.0.1.3458854.666.3.2
        NAME 'attrsetUserVisible'
        DESC 'Attributes visible to authenticated users'
        AUXILIARY
        MAY ( telephoneNumber )
        )

objectclass ( 1.2.826.0.1.3458854.666.3.3
        NAME 'attrsetDeptVisible'
        DESC 'Attributes visible to same-department users'
        AUXILIARY
        MAY ( uid )
        )

objectclass ( 1.2.826.0.1.3458854.666.3.4
        NAME 'attrsetSelfVisible'
        DESC 'Attributes visible to self'
        AUXILIARY
        MAY ( description $ exampleVisibility )
        )

objectclass ( 1.2.826.0.1.3458854.666.3.5
        NAME 'attrsetSelfWrite'
        DESC 'Attributes that users may change
              in their own entry'
        AUXILIARY
        MAY ( userPassword $ exampleVisibility )
        )

objectclass ( 1.2.826.0.1.3458854.666.3.6
        NAME 'attrsetOrgVisible'
        DESC 'Attributes that are visible in
              Organization and OrganizationalUnit entries'
        AUXILIARY
        MAY ( objectclass $ o $ ou $ dc $ description )
        )
```

The first group of access directives deals with access to entries, though it does not grant any access to the attributes contained in them.

The suffix entry is globally readable, and access to all other entries is controlled by the *exampleVisibility* attribute. Each possible value of *exampleVisibility* is treated in turn; the only complex rule is the one dealing with same-department users. Users are given write access to their own entry so that they can modify *userPassword* and *exampleVisibility*.

```
access to dn.exact="dc=example,dc=org"
        by * read

access to filter="(exampleVisibility=internet)"
        attrs="entry"
        by * read

access to filter="(exampleVisibility=users)"
        attrs="entry"
        by users read
        by * none

access to filter="(exampleVisibility=department)"
        dn.regex="(dc=[^,]+,dc=example,dc=org)$"
        attrs="entry"
        by dn.subtree,expand="dc=people,$1" read
        by * none

access to attrs="entry"
        by self write
        by * none
```

The next directive deals with access to attributes in *organization* and *organizationalunit* entries. It uses the *attrsetOrgVisible* class rather than listing the individual attributes.

```
access to filter=
        "(|(objectclass=organization)
           (objectclass=organizationalunit))"
        attrs="@attrsetOrgVisible"
        by * read
```

Attributes in person entries have different visibility depending on who is asking for them. This requires several directives, and makes good use of the accumulate-permissions style.

```
access to filter="(objectclass=person)"
        attrs="@attrsetAnonVisible"
        by * +rsc break
```

```
access to filter="(objectclass=person)"
        attrs="@attrsetUserVisible"
        by users +rsc break
        by * break

access to filter="(objectclass=person)"
        dn.regex="(dc=[^,]+,dc=example,dc=org)$"
        attrs="@attrsetDeptVisible"
        by dn.subtree,expand="dc=people,$1" +rsc break
        by * break

access to filter="(objectclass=person)"
        attrs="@attrsetSelfVisible"
        by self +rsc break
        by * break
```

Writeable attributes are handled by a similar rule. Note that *userPassword* is in *attrsetSelfWrite* but not in *attrsetSelfVisible,* so the user can set it but not read it back.

```
access to filter="(objectclass=person)"
        attrs="@attrsetSelfWrite"
        by self +w break
        by * break
```

Although not mentioned in the policy, we must grant authenticate access to *userPassword* for the anonymous user. This is required to support authentication.

```
access to filter="(objectclass=person)"
        attrs="userPassword"
        by anonymous +x break
        by * break
```

The last directive stops the accumulation process and uses the result.

```
access to * by * stop
```

This ACL passes all the tests in the test suite including tests for detectability. Unfortunately there is still a snag. If a department entry is marked for local visibility only then the entry itself is indeed protected from anonymous access, but entries below it marked for internet visibility still appear in search results. This exposes the existence of the supposedly-invisible department. There is some debate over what the correct behaviour really is in such a case: this can only be resolved by going back to the policy-makers with a specific example.

11 References

References

X50088: X.500/ISO9594-1 The Directory - Overview of Concepts, Models and Services, ISO/CCITT, Geneva, 1988

X50093: X.500/ISO9594-1 The Directory - Overview of Concepts, Models and Services, ISO/CCITT, Geneva, 1993

RFC2251: M Wahl, T Howes, S Kille, Lightweight Directory Access Protocol (v3), IETF, 1997

RFC2252: M Wahl, A Coulbeck, T Howes, S Kille, Lightweight Directory Access Protocol (v3): Attribute Syntax Definitions, IETF, 1997

X50105: X.500/ISO9594, The Directory: Models, ITU-T, Melbourne, 2005

RFC4513: R Harrison (ed), Lightweight Directory Access Protocol (LDAP): Authentication Methods and Security Mechanisms, IETF, 2006

RFC4512: K Zeilenga, LDAP Directory Information Models, IETF, 2006

FINDL05: A Findlay, LDAP Schema Design,Proceedings of the UKUUG Winter Conference, 2005