

**NAME**

slapd.access – access configuration for slapd, the stand-alone LDAP daemon

**SYNOPSIS**

/etc/ldap/slapd.conf

**DESCRIPTION**

The **slapd.conf**(5) file contains configuration information for the **slapd**(8) daemon. This configuration file is also used by the **slurpd**(8) replication daemon and by the SLAPD tools **slapacl**(8), **slapadd**(8), **slapauth**(8), **slapcat**(8), **slapdn**(8), **slapindex**(8), and **slaptest**(8).

The **slapd.conf** file consists of a series of global configuration options that apply to **slapd** as a whole (including all backends), followed by zero or more database backend definitions that contain information specific to a backend instance.

The general format of **slapd.conf** is as follows:

```
# comment - these options apply to every database
<global configuration options>
# first database definition & configuration options
database <backend 1 type>
<configuration options specific to backend 1>
# subsequent database definitions & configuration options
...
```

Both the global configuration and each backend-specific section can contain access information. Backend-specific access control directives are used for those entries that belong to the backend, according to their naming context. In case no access control directives are defined for a backend or those which are defined are not applicable, the directives from the global configuration section are then used.

If no access controls are present, the default policy allows anyone and everyone to read anything but restricts updates to rootdn. (e.g., "access to \* by \* read"). The rootdn can always read and write EVERYTHING!

For entries not held in any backend (such as a root DSE), the directives of the first backend (and any global directives) are used.

Arguments that should be replaced by actual text are shown in brackets <>.

**THE ACCESS DIRECTIVE**

The structure of the access control directives is

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

Grant access (specified by <access>) to a set of entries and/or attributes (specified by <what>) by one or more requestors (specified by <who>).

**THE <WHAT> FIELD**

The field <what> specifies the entity the access control directive applies to. It can have the forms

```
dn[.<dnstyle>]=<dnpattern>
filter=<ldapfilter>
attrs=<attrlist>[ val[/matchingRule][.<attrstyle>]=<attrval>]
```

with

```
<dnstyle>={ {exact|base(object)}|regex
|one(level)|sub(tree)|children}
<attrlist>={ <attr>[ [!|@ ]<objectClass> } [,<attrlist> ]
<attrstyle>={ {exact|base(object)}|regex
|one(level)|sub(tree)|children}
```

The statement **dn=<dnpattern>** selects the entries based on their naming context. The <dnpattern> is a string representation of the entry's DN. The wildcard \* stands for all the entries, and it is implied if no **dn** form is given.

The **<dnstyle>** is optional; however, it is recommended to specify it to avoid ambiguities. **Base** (synonym of **baseObject**), the default, or **exact** (an alias of **base**) indicates the entry whose DN is equal to the **<dnpattern>**; **one** (synonym of **onelevel**) indicates all the entries immediately below the **<dnpattern>**; **sub** (synonym of **subtree**) indicates all entries in the subtree at the **<dnpattern>**; **children** indicates all the entries below (subordinate to) the **<dnpattern>**.

If the **<dnstyle>** qualifier is **regex**, then **<dnpattern>** is a POSIX ("extended") regular expression pattern, as detailed in **regex(7)** and/or **re\_format(7)**, matching a normalized string representation of the entry's DN. The regex form of the pattern does not (yet) support UTF-8.

The statement **filter=<ldapfilter>** selects the entries based on a valid LDAP filter as described in RFC 2254. A filter of (**objectClass=\***) is implied if no **filter** form is given.

The statement **attrs=<attrlist>** selects the attributes the access control rule applies to. It is a comma-separated list of attribute types, plus the special names **entry**, indicating access to the entry itself, and **children**, indicating access to the entry's children. ObjectClass names may also be specified in this list, which will affect all the attributes that are required and/or allowed by that objectClass. Actually, names in **<attrlist>** that are prefixed by **@** are directly treated as objectClass names. A name prefixed by **!** is also treated as an objectClass, but in this case the access rule affects the attributes that are not required nor allowed by that objectClass. If no **attrs** form is given, **attrs=@extensibleObject** is implied, i.e. all attributes are addressed.

Using the form **attrs=<attr> val[/matchingRule][.<attrstyle>]=<attrval>** specifies access to a particular value of a single attribute. In this case, only a single attribute type may be given. The **<attrstyle>** **exact** (the default) uses the attribute's equality matching rule to compare the value, unless a different (and compatible) matching rule is specified. If the **<attrstyle>** is **regex**, the provided value is used as a POSIX ("extended") regular expression pattern. If the attribute has DN syntax, the **<attrstyle>** can be any of **base**, **onelevel**, **subtree** or **children**, resulting in base, onelevel, subtree or children match, respectively.

The **dn**, **filter**, and **attrs** statements are additive; they can be used in sequence to select entities the access rule applies to based on naming context, value and attribute type simultaneously.

## THE <WHO> FIELD

The field **<who>** indicates whom the access rules apply to. Multiple **<who>** statements can appear in an access control statement, indicating the different access privileges to the same resource that apply to different accessesee. It can have the forms

```
*
anonymous
users
self[.<selfstyle>]

dn[.<dnstyle>[,<modifier>]]=<DN>
dnattr=<attrname>

realanonymous
realusers
realself[.<selfstyle>]

realdn[.<dnstyle>[,<modifier>]]=<DN>
realdnattr=<attrname>

group[/<objectclass>/<attrname>]]
[.<groupstyle>]=<group>
peername[.<peernamestyle>]=<peername>
sockname[.<style>]=<sockname>
domain[.<domainstyle>[,<modifier>]]=<domain>
sockurl[.<style>]=<sockurl>
set[.<setstyle>]=<pattern>
```

```

ssf=<n>
transport_ssf=<n>
tls_ssf=<n>
sasl_ssf=<n>

aci[=<attrname>]
dynacl/name[/<options>][.<dynstyle>][=<pattern>]

```

with

```

<style>={exact|regex|expand}
<selfstyle>={level{<n>}}
<dnstyle>={ {exact|base(object)}|regex
             |one(level)|sub(tree)|children|level{<n>}}
<groupstyle>={exact|expand}
<peernamestyle>={<style>|ip|path}
<domainstyle>={exact|regex|sub(tree)}
<setstyle>={exact|regex}
<modifier>={expand}

```

They may be specified in combination.

The wildcard \* refers to everybody.

The keywords prefixed by **real** act as their counterparts without prefix; the checking respectively occurs with the *authentication* DN and the *authorization* DN.

The keyword **anonymous** means access is granted to unauthenticated clients; it is mostly used to limit access to authentication resources (e.g. the **userPassword** attribute) to unauthenticated clients for authentication purposes.

The keyword **users** means access is granted to authenticated clients.

The keyword **self** means access to an entry is allowed to the entry itself (e.g. the entry being accessed and the requesting entry must be the same). It allows the **level{<n>}** style, where <n> indicates what ancestor of the DN is to be used in matches. A positive value indicates that the <n>-th ancestor of the user's DN is to be considered; a negative value indicates that the <n>-th ancestor of the target is to be considered. For example, a "*by self.level{1} ...*" clause would match when the object "*dc=example,dc=com*" is accessed by "*cn=User,dc=example,dc=com*". A "*by self.level{-1} ...*" clause would match when the same user accesses the object "*ou=Address Book,cn=User,dc=example,dc=com*".

The statement **dn=<DN>** means that access is granted to the matching DN. The optional style qualifier **dnstyle** allows the same choices of the dn form of the <what> field. In addition, the **regex** style can exploit substring substitution of submatches in the <what> dn.regex clause by using the form **\$<digit>**, with **digit** ranging from 0 to 9 (where 0 matches the entire string), or the form **\${<digit>+}**, for submatches higher than 9. Since the dollar character is used to indicate a substring replacement, the dollar character that is used to indicate match up to the end of the string must be escaped by a second dollar character, e.g.

```

access to dn.regex="^(.+)?uid=(^[,]+),dc=[^,]+,dc=com$"
by dn.regex="uid=$2,dc=[^,]+,dc=com$$" write

```

The style qualifier allows an optional **modifier**. At present, the only type allowed is **expand**, which causes substring substitution of submatches to take place even if **dnstyle** is not **regex**. Note that the **regex** dnstyle in the above example may be of use only if the <by> clause needs to be a regex; otherwise, if the value of the second (from the right) **dc**= portion of the DN in the above example were fixed, the form

```

access to dn.regex="^(.+)?uid=(^[,]+),dc=example,dc=com$"
by dn.exact,expand="uid=$2,dc=example,dc=com" write

```

could be used; if it had to match the value in the <what> clause, the form

```

access to dn.regex="^(.+)?uid=(^[,]+),dc=(^[,]+),dc=com$"

```

by dn.exact,expand="uid=\$2,dc=\$3,dc=com" write  
could be used.

Forms of the **<what>** clause other than regex may provide submatches as well. The **base(object)**, the **sub(tree)**, the **one(level)**, and the **children** forms provide **\$0** as the match of the entire string. The **sub(tree)**, the **one(level)**, and the **children** forms also provide **\$1** as the match of the rightmost part of the DN as defined in the **<what>** clause. This may be useful, for instance, to provide access to all the ancestors of a user by defining

```
access to dn.subtree="dc=com"
  by dn.subtree,expand="$1" read
```

which means that only access to entries that appear in the DN of the **<by>** clause is allowed.

The **level{<n>}** form is an extension and a generalization of the **onelevel** form, which matches all DNs whose **<n>**-th ancestor is the pattern. So, *level{1}* is equivalent to *onelevel*, and *level{0}* is equivalent to *base*.

It is perfectly useless to give any access privileges to a DN that exactly matches the **rootdn** of the database the ACLs apply to, because it implicitly possesses write privileges for the entire tree of that database. Actually, access control is bypassed for the **rootdn**, to solve the intrinsic chicken-and-egg problem.

The statement **dnattr=<attrname>** means that access is granted to requests whose DN is listed in the entry being accessed under the **<attrname>** attribute.

The statement **group=<group>** means that access is granted to requests whose DN is listed in the group entry whose DN is given by **<group>**. The optional parameters **<objectclass>** and **<attrname>** define the objectClass and the member attributeType of the group entry. The defaults are **groupOfNames** and **member**, respectively. The optional style qualifier **<style>** can be **expand**, which means that **<group>** will be expanded as a replacement string (but not as a regular expression) according to **regex(7)** and/or **re\_format(7)**, and **exact**, which means that exact match will be used. If the style of the DN portion of the **<what>** clause is regex, the submatches are made available according to **regex(7)** and/or **re\_format(7)**; other styles provide limited submatches as discussed above about the DN form of the **<by>** clause.

For static groups, the specified attributeType must have **DistinguishedName** or **NameAndOptionalUID** syntax. For dynamic groups the attributeType must be a subtype of the **labeledURI** attributeType. Only LDAP URIs of the form **ldap:///<base>??<scope>?<filter>** will be evaluated in a dynamic group, by searching the local server only.

The statements **peername=<peername>**, **sockname=<sockname>**, **domain=<domain>**, and **sockurl=<sockurl>** mean that the contacting host IP (in the form **IP=<ip>:<port>**) or the contacting host named pipe file name (in the form **PATH=<path>** if connecting through a named pipe) for **peername**, the named pipe file name for **sockname**, the contacting host name for **domain**, and the contacting URL for **sockurl** are compared against **pattern** to determine access. The same **style** rules for pattern match described for the **group** case apply, plus the **regex** style, which implies submatch **expand** and regex match of the corresponding connection parameters. The **exact** style of the **<peername>** clause (the default) implies a case-exact match on the client's **IP**, including the **IP=** prefix and the trailing **:<port>**, or the client's **path**, including the **PATH=** prefix if connecting through a named pipe. The special **ip** style interprets the pattern as **<peername>=<ip>[%<mask>][{<n>}]**, where **<ip>** and **<mask>** are dotted digit representations of the IP and the mask, while **<n>**, delimited by curly brackets, is an optional port. When checking access privileges, the IP portion of the **peername** is extracted, eliminating the **IP=** prefix and the **:<port>** part, and it is compared against the **<ip>** portion of the pattern after masking with **<mask>**. As an example, **peername.ip=127.0.0.1** allows connections only from localhost, **peername.ip=192.168.1.0%255.255.255.0** allows connections from any IP in the 192.168.1 class C domain, and **peername.ip=192.168.1.16%255.255.255.240{9009}** allows connections from any IP in the 192.168.1.[16-31] range of the same domain, only if port 9009 is used. The special **path** style eliminates the **PATH=** prefix from the **peername** when connecting through a named pipe, and performs an exact match on the given pattern. The **<domain>** clause also allows the **subtree** style, which succeeds when a fully qualified name exactly matches the **domain** pattern, or its trailing part, after a **dot**, exactly matches the

**domain** pattern. The **expand** style is allowed, implying an **exact** match with submatch expansion; the use of **expand** as a style modifier is considered more appropriate. As an example, **domain.subtree=example.com** will match `www.example.com`, but will not match `www.anotherexample.com`. The **domain** of the contacting host is determined by performing a DNS reverse lookup. As this lookup can easily be spoofed, use of the **domain** statement is strongly discouraged. By default, reverse lookups are disabled. The optional **domainstyle** qualifier of the `<domain>` clause allows a **modifier** option; the only value currently supported is **expand**, which causes substring substitution of submatches to take place even if the **domainstyle** is not **regex**, much like the analogous usage in `<dn>` clause.

The statement **set=<pattern>** is undocumented yet.

The statement **aci[=<attrname>]** means that the access control is determined by the values in the **attrname** of the entry itself. The optional `<attrname>` indicates what attributeType holds the ACI information in the entry. By default, the **OpenLDAPaci** operational attribute is used. ACIs are experimental; they must be enabled at compile time.

The statement **dynacl/<name>[/<options>][.<dynstyle>][=<pattern>]** means that access checking is delegated to the admin-defined method indicated by `<name>`, which can be registered at run-time by means of the **moduleload** statement. The fields `<options>`, `<dynstyle>` and `<pattern>` are optional, and are directly passed to the registered parsing routine. Dynacl is experimental; it must be enabled at compile time. If dynacl and ACIs are both enabled, ACIs are cast into the dynacl scheme, where `<name>=aci` and, optionally, `<pattern>=<attrname>`. However, the original ACI syntax is preserved for backward compatibility.

The statements **ssf=<n>**, **transport\_ssf=<n>**, **tls\_ssf=<n>**, and **sasl\_ssf=<n>** set the minimum required Security Strength Factor (ssf) needed to grant access. The value should be positive integer.

## THE <ACCESS> FIELD

The field `<access> ::= [[real]self]{<level>|<priv>}` determines the access level or the specific access privileges the **who** field will have. Its component are defined as

```
<level> ::= none|disclose|auth|compare|search|read|write
<priv> ::= {=|+|-}{w|r|s|c|x|d|0}+
```

The modifier **self** allows special operations like having a certain access level or privilege only in case the operation involves the name of the user that's requesting the access. It implies the user that requests access is authorized. The modifier **realself** refers to the authenticated DN as opposed to the authorized DN of the **self** modifier. An example is the **selfwrite** access to the member attribute of a group, which allows one to add/delete its own DN from the member list of a group, without affecting other members.

The **level** access model relies on an incremental interpretation of the access privileges. The possible levels are **none**, **disclose**, **auth**, **compare**, **search**, **read**, and **write**. Each access level implies all the preceding ones, thus **write** access will imply all accesses.

The **none** access level disallows all access including disclosure on error.

The **disclose** access level allows disclosure of information on error.

The **auth** access level means that one is allowed access to an attribute to perform authentication/authorization operations (e.g. **bind**) with no other access. This is useful to grant unauthenticated clients the least possible access level to critical resources, like passwords.

The **priv** access model relies on the explicit setting of access privileges for each clause. The = sign resets previously defined accesses; as a consequence, the final access privileges will be only those defined by the clause. The + and - signs add/remove access privileges to the existing ones. The privileges are **w** for write, **r** for read, **s** for search, **c** for compare, **x** for authentication, and **d** for disclose. More than one of the above privileges can be added in one statement. **0** indicates no privileges and is used only by itself (e.g., +0). If no access is given, it defaults to +0.

## THE <CONTROL> FIELD

The optional field `<control>` controls the flow of access rule application. It can have the forms

```
stop
continue
```

break

where **stop**, the default, means access checking stops in case of match. The other two forms are used to keep on processing access clauses. In detail, the **continue** form allows for other **<who>** clauses in the same **<access>** clause to be considered, so that they may result in incrementally altering the privileges, while the **break** form allows for other **<access>** clauses that match the same target to be processed. Consider the (silly) example

```
access to dn.subtree="dc=example,dc=com" attrs=cn
by * =cs break
```

```
access to dn.subtree="ou=People,dc=example,dc=com"
by * +r
```

which allows search and compare privileges to everybody under the "dc=example,dc=com" tree, with the second rule allowing also read in the "ou=People" subtree, or the (even more silly) example

```
access to dn.subtree="dc=example,dc=com" attrs=cn
by * =cs continue
by users +r
```

which grants everybody search and compare privileges, and adds read privileges to authenticated clients.

One useful application is to easily grant write privileges to an **updatedn** that is different from the **rootdn**. In this case, since the **updatedn** needs write access to (almost) all data, one can use

```
access to *
by dn.exact="cn=The Update DN,dc=example,dc=com" write
by * break
```

as the first access rule. As a consequence, unless the operation is performed with the **updatedn** identity, control is passed straight to the subsequent rules.

## OPERATION REQUIREMENTS

Operations require different privileges on different portions of entries. The following summary applies to primary database backends such as the BDB and HDB backends. Requirements for other backends may (and often do) differ.

The **add** operation requires **write (=w)** privileges on the pseudo-attribute **entry** of the entry being added, and **write (=w)** privileges on the pseudo-attribute **children** of the entry's parent. When adding the suffix entry of a database, write access to **children** of the empty DN ("") is required.

The **bind** operation, when credentials are stored in the directory, requires **auth (=x)** privileges on the attribute the credentials are stored in (usually **userPassword**).

The **compare** operation requires **compare (=c)** privileges on the attribute that is being compared.

The **delete** operation requires **write (=w)** privileges on the pseudo-attribute **entry** of the entry being deleted, and **write (=w)** privileges on the **children** pseudo-attribute of the entry's parent.

The **modify** operation requires **write (=w)** privileges on the attributes being modified.

The **modrdn** operation requires **write (=w)** privileges on the pseudo-attribute **entry** of the entry whose relative DN is being modified, **write (=w)** privileges on the pseudo-attribute **children** of the old and new entry's parents, and **write (=w)** privileges on the attributes that are present in the new relative DN. **Write (=w)** privileges are also required on the attributes that are present in the old relative DN if **deleteoldrdn** is

set to 1.

The **search** operation, requires **search (=s)** privileges on the **entry** pseudo-attribute of the **searchBase** (NOTE: this was introduced with 2.3). Then, for each entry, it requires **search (=s)** privileges on the attributes that are defined in the filter. The resulting entries are finally tested for **read (=r)** privileges on the pseudo-attribute **entry** (for read access to the entry itself) and for **read (=r)** access on each value of each attribute that is requested. Also, for each **referral** object used in generating continuation references, the operation requires **read (=r)** access on the pseudo-attribute **entry** (for read access to the referral object itself), as well as **read (=r)** access to the attribute holding the referral information (generally the **ref** attribute).

Some internal operations and some **controls** require specific access privileges. The **authzID** mapping and the **proxyAuthz** control require **auth (=x)** privileges on all the attributes that are present in the search filter of the URI regexp maps (the right-hand side of the **authz-regexp** directives). **Auth (=x)** privileges are also required on the **authzTo** attribute of the authorizing identity and/or on the **authzFrom** attribute of the authorized identity.

Access control to search entries is checked by the frontend, so it is fully honored by all backends; for all other operations and for the discovery phase of the search operation, full ACL semantics is only supported by the primary backends, i.e. **back-bdb(5)**, and **back-hdb(5)**.

Some other backend, like **back-sql(5)**, may fully support them; others may only support a portion of the described semantics, or even differ in some aspects. The relevant details are described in the backend-specific man pages.

## CAVEATS

It is strongly recommended to explicitly use the most appropriate **<dnstyle>** in **<what>** and **<who>** clauses, to avoid possible incorrect specifications of the access rules as well as for performance (avoid unnecessary regexp matching when an exact match suffices) reasons.

An administrator might create a rule of the form:

```
access to dn.regex="dc=example,dc=com"
by ...
```

expecting it to match all entries in the subtree "dc=example,dc=com". However, this rule actually matches any DN which contains anywhere the substring "dc=example,dc=com". That is, the rule matches both "uid=joe,dc=example,dc=com" and "dc=example,dc=com,uid=joe".

To match the desired subtree, the rule would be more precisely written:

```
access to dn.regex="^(.+)?dc=example,dc=com$"
by ...
```

For performance reasons, it would be better to use the subtree style.

```
access to dn.subtree="dc=example,dc=com"
by ...
```

When writing submatch rules, it may be convenient to avoid unnecessary **regex <dnstyle>** use; for instance, to allow access to the subtree of the user that matches the **<what>** clause, one could use

```
access to dn.regex="^(.+)?uid=(^[^,]+),dc=example,dc=com$"
by dn.regex="^uid=$2,dc=example,dc=com$$" write
by ...
```

However, since all that is required in the **<by>** clause is substring expansion, a more efficient solution is

```
access to dn.regex="^(.+)?uid=(^[^,]+),dc=example,dc=com$"
```

by dn.exact,expand="uid=\$2,dc=example,dc=com" write  
by ...

In fact, while a `<dnstyle>` of **regex** implies substring expansion, **exact**, as well as all the other DN specific `<dnstyle>` values, does not, so it must be explicitly requested.

## FILES

/etc/ldap/slapd.conf  
default slapd configuration file

## SEE ALSO

**slapd**(8), **slapd**-(5), **slapacl**(8), **regex**(7), **re\_format**(7)

"OpenLDAP Administrator's Guide" (<http://www.OpenLDAP.org/doc/admin/>)

## ACKNOWLEDGEMENTS

**OpenLDAP** is developed and maintained by The OpenLDAP Project (<http://www.openldap.org/>). **OpenLDAP** is derived from University of Michigan LDAP 3.3 Release.