Authentication using LDAP

New in Django Development version.

Django includes an LDAP authentication backend to authenticate against any LDAP server. To enable, add `django.contrib.auth.contrib.ldap.backend.LDAPBackend` to `AUTHENTICATION_BACKENDS`. LDAP configuration can be as simple as a single distinguished name template, but there are many rich options for working with `User` objects, groups, and permissions. This backend depends on the Python ldap module.

Note

`LDAPBackend` does not inherit from `ModelBackend`. It is possible to use `LDAPBackend` exclusively by configuring it to draw group membership from the LDAP server. However, if you would like to assign permissions to individual users or add users to groups within Django, you'll need to have both backends installed:

```
AUTHENTICATION_BACKENDS = (
    'django.contrib.auth.contrib.ldap.backend.LDAPBackend',
    'django.contrib.auth.backends.ModelBackend',
)
```

Configuring basic authentication

If your LDAP server isn't running locally on the default port, you'll want to start by setting `AUTH_LDAP_SERVER_URI` to point to your server.

```
AUTH_LDAP_SERVER_URI = "ldap://ldap.example.com"
```

That done, the first step is to authenticate a username and password against the LDAP service. There are two ways to do this, called search/bind and simply bind. The first one involves connecting to the LDAP server either anonymously or with a fixed account and searching for the distinguished name of the authenticating user. Then we can attempt to bind again with the user's password. The second method is to derive the user's DN from his username and attempt to bind as the user directly.

Because LDAP searches appear elsewhere in the configuration, the `LDAPSearch` class is provided to encapsulate search information. In this case, the filter parameter should contain the placeholder `%(user)s`. A simple configuration for the search/bind approach looks like this (some defaults included for completeness):

```
import ldap
from django.contrib.auth.contrib.ldap.config import LDAPSearch

AUTH_LDAP_BIND_DN = ""
AUTH_LDAP_BIND_PASSWORD = ""
AUTH_LDAP_USER_SEARCH = LDAPSearch("ou=users,dc=example,dc=com",
    ldap.SCOPE_SUBTREE, "(uid=%(user)s)")
```

This will perform an anonymous bind, search under `"ou=users,dc=example,dc=com"` for an object with a uid matching the user's name, and try to bind using that DN and the user's password. The search must return exactly one result or authentication will fail. If you can't search anonymously, you can set `AUTH_LDAP_BIND_DN` to the distinguished name of an authorized user and `AUTH_LDAP_BIND_PASSWORD` to the password.

To skip the search phase, set `AUTH_LDAP_USER_DN_TEMPLATE` to a template that will produce the authenticating user's DN directly. This template should have one placeholder, `%(user)s`. If the previous example had used `ldap.SCOPE_ONELEVEL`, the following would be a more straightforward (and efficient) equivalent:

```
AUTH_LDAP_USER_DN_TEMPLATE = "uid=%(user)s,ou=users,dc=example,dc=com"
```

Working with groups

Working with groups in LDAP can be a tricky business, as there isn't a single standard grouping mechanism. This module includes an extensible API for working with any kind of group and includes implementations for the most common ones. `LDAPGroupType` is a base class whose concrete

Last update:
Oct 23, 2009

subclasses can determine group membership for particular grouping mechanisms. Three built-in subclasses cover most grouping mechanisms:

- PosixGroupType
- MemberDNGroupType
- NestedMemberDNGroupType

posixGroup objects are somewhat specialized, so they get their own class. The other two cover mechanisms whereby a group object stores a list of its members as distinguished names. This includes groupOfNames, groupOfUniqueNames, and Active Directory groups, among others. The nested variant allows groups to contain other groups, to as many levels as you like. For convenience and readability, several trivial subclasses of the above are provided:

- GroupOfNamesType
- NestedGroupOfNamesType
- GroupOfUniqueNamesType
- NestedGroupOfUniqueNamesType
- ActiveDirectoryGroupType
- NestedActiveDirectoryGroupType

To get started, you'll need to provide some basic information about your LDAP groups. AUTH_LDAP_GROUP_SEARCH is an LDAPSearch object that identifies the set of relevant group objects. That is, all groups that users might belong to as well as any others that we might need to know about (in the case of nested groups, for example). AUTH_LDAP_GROUP_TYPE is an instance of the class corresponding to the type of group that will be returned by AUTH_LDAP_GROUP_SEARCH. All groups referenced elsewhere in the configuration must be of this type and part of the search results.

```
import ldap
from django.contrib.auth.contrib.ldap.config import LDAPSearch, GroupOfNamesType

AUTH_LDAP_GROUP_SEARCH = LDAPSearch("ou=groups,dc=example,dc=com",
    ldap.SCOPE_SUBTREE, "(objectClass=groupOfNames)"
)
AUTH_LDAP_GROUP_TYPE = GroupOfNamesType()
```

The simplest use of groups is to limit the users who are allowed to log in. If AUTH_LDAP_REQUIRE_GROUP is set, then only users who are members of that group will successfully authenticate:

```
AUTH_LDAP_REQUIRE_GROUP = "cn=enabled,ou=groups,dc=example,dc=com"
```

More advanced uses of groups are covered in the next two sections.

User objects

Authenticating against an external source is swell, but Django's auth module is tightly bound to the django.contrib.auth.models.User model. Thus, when a user logs in, we have to create a User object to represent him in the database.

The only required field for a user is the username, which we obviously have. The User model is picky about the characters allowed in usernames, so LDAPBackend includes a pair of hooks, ldap_to_django_username() and django_to_ldap_username(), to translate between LDAP usernames and Django usernames. You'll need this, for example, if your LDAP names have periods in them. You can subclass LDAPBackend to implement these hooks; by default the username is not modified. User objects that are authenticated by LDAPBackend will have an ldap_username attribute with the original (LDAP) username. username will, of course, be the Django username.

LDAP directories tend to contain much more information about users that you may wish to propagate. A pair of settings, AUTH_LDAP_USER_ATTR_MAP and AUTH_LDAP_PROFILE_ATTR_MAP, serve to copy directory information into User and profile objects. These are dictionaries that map user and profile model keys, respectively, to LDAP attribute names:

```
AUTH_LDAP_USER_ATTR_MAP = {"first_name": "givenName", "last_name": "sn"}
AUTH_LDAP_PROFILE_ATTR_MAP = {"home_directory": "homeDirectory"}
```

Only string fields can be mapped to attributes. Boolean fields can be defined by group membership:

```
AUTH_LDAP_USER_FLAGS_BY_GROUP = {
    "is_active": "cn=active,ou=groups,dc=example,dc=com",
    "is_staff": "cn=staff,ou=groups,dc=example,dc=com",
    "is_superuser": "cn=superuser,ou=groups,dc=example,dc=com"
}
```

By default, all mapped user fields will be updated each time the user logs in. To disable this, set `AUTH_LDAP_ALWAYS_UPDATE_USER` to `False`.

If you need to access multi-value attributes or there is some other reason that the above is inadequate, you can also access the user's raw LDAP attributes. `user.ldap_user` is an object with two public properties:

- `dn`: The user's distinguished name.
- `attrs`: The user's LDAP attributes as a dictionary of lists of string values.

Python-ldap returns all attribute values as utf8-encoded strings. For convenience, this module will try to decode all values into Unicode strings. Any string that can not be successfully decoded will be left as-is; this may apply to binary values such as Active Directory's objectSid.

Note

Users created by `LDAPBackend` will have an unusable password set. This will only happen when the user is created, so if you set a valid password in Django, the user will be able to log in through `ModelBackend` (if configured) even if he is rejected by LDAP. This is not generally recommended, but could be useful as a fail-safe for selected users in case the LDAP server is unavailable.

Permissions

Groups are useful for more than just populating the user's `is_*` fields. `LDAPBackend` would not be complete without some way to turn a user's LDAP group memberships into Django model permissions. In fact, there are two ways to do this.

Ultimately, both mechanisms need some way to map LDAP groups to Django groups. Implementations of `LDAPGroupType` will have an algorithm for deriving the Django group name from the LDAP group. Clients that need to modify this behavior can subclass the `LDAPGroupType` class. All of the built-in implementations take a `name_attr` argument to `__init__`, which specifies the LDAP attribute from which to take the Django group name. By default, the `cn` attribute is used.

The least invasive way to map group permissions is to set `AUTH_LDAP_FIND_GROUP_PERMS` to `True` `LDAPBackend` will then find all of the LDAP groups that a user belongs to, map them to Django groups, and load the permissions for those groups. You will need to create the Django groups yourself, generally through the admin interface.

Note

After the user logs in, subsequent requests will have to determine group membership based solely on the `User` object of the logged-in user. We will not have the user's password at this point. This means that if `AUTH_LDAP_FIND_GROUP_PERMS` is `True`, we must have access to the LDAP directory through `AUTH_LDAP_BIND_DN` and `AUTH_LDAP_BIND_PASSWORD`, even if you're using `AUTH_LDAP_USER_DN_TEMPLATE` to authenticate the user.

To minimize traffic to the LDAP server, `LDAPBackend` can make use of Django's cache framework to keep a copy of a user's LDAP group memberships. To enable this feature, set `AUTH_LDAP_CACHE_GROUPS` to `True`. You can also set `AUTH_LDAP_GROUP_CACHE_TIMEOUT` to override the timeout of cache entries (in seconds).

```
AUTH_LDAP_CACHE_GROUPS = True
AUTH_LDAP_GROUP_CACHE_TIMEOUT = 300
```

The second way to turn LDAP group memberships into permissions is to mirror the groups themselves. If `AUTH_LDAP_MIRROR_GROUPS` is `True`, then every time a user logs in, `LDAPBackend` will update the database with the user's LDAP groups. Any group that doesn't exist will be created and the user's Django group membership will be updated to exactly match his LDAP group membership. Note that if the LDAP server has nested groups, the Django database will end up with a flattened representation.

This approach has two main differences from `AUTH_LDAP_FIND_GROUP_PERMS`. First, `AUTH_LDAP_FIND_GROUP_PERMS` will query for LDAP group membership either for every request or according to the cache timeout. With group mirroring, membership will be updated when the user authenticates. This may not be appropriate for sites with long session timeouts. The second difference is that with `AUTH_LDAP_FIND_GROUP_PERMS`, there is no way for clients to determine a user's group memberships, only their permissions. If you want to make decisions based directly on group membership, you'll have to mirror the groups.

Logging

`LDAPBackend` uses the standard logging module to log debug and warning messages to the logger named `'django.contrib.auth.contrib.ldap'`. If you need debug messages to help with configuration issues, you should add a handler to this logger.

More options

Miscellaneous settings for `LDAPBackend`:

- `AUTH_LDAP_GLOBAL_OPTIONS`: A dictionary of options to pass to python-ldap via `ldap.set_option()`.
- `AUTH_LDAP_CONNECTION_OPTIONS`: A dictionary of options to pass to each LDAPObject instance via `LDAPObject.set_option()`.

Performance

`LDAPBackend` is carefully designed not to require a connection to the LDAP service for every request. Of course, this depends heavily on how it is configured. If LDAP traffic or latency is a concern for your deployment, this section has a few tips on minimizing it, in decreasing order of impact.

1. **Cache groups**. If `AUTH_LDAP_FIND_GROUP_PERMS` is `True`, the default behavior is to reload a user's group memberships on every request. This is the safest behavior, as any membership change takes effect immediately, but it is expensive. If possible, set `AUTH_LDAP_CACHE_GROUPS` to `True` to remove most of this traffic. Alternatively, you might consider using `AUTH_LDAP_MIRROR_GROUPS` and relying on `ModelBackend` to supply group permissions.
2. **Don't access user.ldap_user.***. These properties are only cached on a per-request basis. If you can propagate LDAP attributes to a `User` or profile object, they will only be updated at login. `user.ldap_user.attrs` triggers an LDAP connection for every request in which it's accessed. If you're not using `AUTH_LDAP_USER_DN_TEMPLATE`, then accessing `user.ldap_user.dn` will also trigger an LDAP connection.
3. **Use simpler group types**. Some grouping mechanisms are more expensive than others. This will often be outside your control, but it's important to note that the extra functionality of more complex group types like `NestedGroupOfNamesType` is not free and will generally require a greater number and complexity of LDAP queries.
4. **Use direct binding**. Binding with `AUTH_LDAP_USER_DN_TEMPLATE` is a little bit more efficient than relying on `AUTH_LDAP_USER_SEARCH`. Specifically, it saves two LDAP operations (one bind and one search) per login.

Example configuration

Here is a complete example configuration from `settings.py` that exercises nearly all of the features. In this example, we're authenticating against a global pool of users in the directory, but we have a special area set aside for Django groups (ou=django,ou=groups,dc=example,dc=com). Remember that most of this is optional if you just need simple authentication. Some default settings and arguments are included for completeness.

```
import ldap
from django.contrib.auth.contrib.ldap.config import LDAPSearch, GroupOfNamesType


# Baseline configuration.
AUTH_LDAP_SERVER_URI = "ldap://ldap.example.com"

AUTH_LDAP_BIND_DN = "cn=django-agent,dc=example,dc=com"
AUTH_LDAP_BIND_PASSWORD = "phlebotinum"
AUTH_LDAP_USER_SEARCH = LDAPSearch("ou=users,dc=example,dc=com",
    ldap.SCOPE_SUBTREE, "(uid=%(user)s)")
# or perhaps:
# AUTH_LDAP_USER_DN_TEMPLATE = "uid=%(user)s,ou=users,dc=example,dc=com"

# Set up the basic group parameters.
AUTH_LDAP_GROUP_SEARCH = LDAPSearch("ou=django,ou=groups,dc=example,dc=com",
    ldap.SCOPE_SUBTREE, "(objectClass=groupOfNames)"
)
AUTH_LDAP_GROUP_TYPE = GroupOfNamesType(name_attr="cn")

# Only users in this group can log in.
AUTH_LDAP_REQUIRE_GROUP = "cn=enabled,ou=django,ou=groups,dc=example,dc=com"

# Populate the Django user from the LDAP directory.
AUTH_LDAP_USER_ATTR_MAP = {
    "first_name": "givenName",
```

```
    "last_name": "sn",
    "email": "mail"
}

AUTH_LDAP_PROFILE_ATTR_MAP = {
    "employee_number": "employeeNumber"
}

AUTH_LDAP_USER_FLAGS_BY_GROUP = {
    "is_active": "cn=active,ou=django,ou=groups,dc=example,dc=com",
    "is_staff": "cn=staff,ou=django,ou=groups,dc=example,dc=com",
    "is_superuser": "cn=superuser,ou=django,ou=groups,dc=example,dc=com"
}

# This is the default, but I like to be explicit.
AUTH_LDAP_ALWAYS_UPDATE_USER = True

# Use LDAP group membership to calculate group permissions.
AUTH_LDAP_FIND_GROUP_PERMS = True

# Cache group memberships for an hour to minimize LDAP traffic
AUTH_LDAP_CACHE_GROUPS = True
AUTH_LDAP_GROUP_CACHE_TIMEOUT = 3600


# Keep ModelBackend around for per-user permissions and maybe a local
# superuser.
AUTHENTICATION_BACKENDS = (
    'django.contrib.auth.contrib.ldap.backend.LDAPBackend',
    'django.contrib.auth.backends.ModelBackend',
)
```

API reference

Configuration

class **LDAPSearch**

__init__(*base_dn*, *scope*, *filterstr='(objectClass=*)'*)

- `base_dn`: The distinguished name of the search base.
- `scope`: One of `ldap.SCOPE_*`.
- `filterstr`: An optional filter string (e.g. '(objectClass=person)'). In order to be valid, `filterstr` must be enclosed in parentheses.

class **LDAPGroupType**

The base class for objects that will determine group membership for various LDAP grouping mechanisms. Implementations are provided for common group types or you can write your own. See the source code for subclassing notes.

__init__(*name_attr='cn'*)

By default, LDAP groups will be mapped to Django groups by taking the first value of the cn attribute. You can specify a different attribute with `name_attr`.

class **PosixGroupType**

A concrete subclass of `LDAPGroupType` that handles the `posixGroup` object class. This checks for both primary group and group membership.

__init__(*name_attr='cn'*)

class **MemberDNGroupType**

A concrete subclass of `LDAPGroupType` that handles grouping mechanisms wherein the group object contains a list of its member DNs.

__init__(*member_attr*, *name_attr='cn'*)

- `member_attr`: The attribute on the group object that contains a list of member DNs. 'member' and 'uniqueMember' are common examples.

class **NestedMemberDNGroupType**

Similar to `MemberDNGroupType`, except this allows groups to contain other groups as members. Group hierarchies will be traversed to determine membership.

__init__(*member_attr*, *name_attr='cn'*)
As above.

class **GroupOfNamesType**

A concrete subclass of `MemberDNGroupType` that handles the `groupOfNames` object class. Equivalent to `MemberDNGroupType('member')`.

__init__(*name_attr='cn'*)

class **NestedGroupOfNamesType**

A concrete subclass of `NestedMemberDNGroupType` that handles the `groupOfNames` object class. Equivalent to `NestedMemberDNGroupType('member')`.

__init__(*name_attr='cn'*)

class **GroupOfUniqueNamesType**

A concrete subclass of `MemberDNGroupType` that handles the `groupOfUniqueNames` object class. Equivalent to `MemberDNGroupType('uniqueMember')`.

__init__(*name_attr='cn'*)

class **NestedGroupOfUniqueNamesType**

A concrete subclass of `NestedMemberDNGroupType` that handles the `groupOfUniqueNames` object class. Equivalent to `NestedMemberDNGroupType('uniqueMember')`.

__init__(*name_attr='cn'*)

class **ActiveDirectoryGroupType**

A concrete subclass of `MemberDNGroupType` that handles Active Directory groups. Equivalent to `MemberDNGroupType('member')`.

__init__(*name_attr='cn'*)

class **NestedActiveDirectoryGroupType**

A concrete subclass of `NestedMemberDNGroupType` that handles Active Directory groups. Equivalent to `NestedMemberDNGroupType('member')`.

__init__(*name_attr='cn'*)

Backend

class **LDAPBackend**

ldap_to_django_username(*username*)
Returns a valid Django username based on the given LDAP username (which is what the user enters). By default, `username` is returned unchanged. This can be overriden by subclasses.

django_to_ldap_username(*username*)
The inverse of `ldap_to_django_username()`. If this is not symmetrical to `ldap_to_django_username()`, the behavior is undefined.