# Apache and LDAP

Before we get started, I would like to explain to you a bit about who I am and what I do. I like to do this to help prove that I really am a working programmer and not a marketing type. I'm an Instructional Technology Engineer for WebCT, Inc. We're the leading provider of tools and services for Web based course delivery. Previously I was the Web Administrator for the University of North Texas, the fourth largest university in Texas. I've also written a book on LDAP, *Implementing LDAP,* and have contributed to several other books and magazines as well as have presented several times on LDAP/directory services.

At UNT we used LDAP for four main reasons. One is user authentication. The second is for directory services (e.g. looking up someone's email addresses). The third reason is for user authorization (e.g. does this user have the rights to access a particular system or application). Finally we are using LDAP and email to improve communication with our students and to reduce the amount of paper we send out annually to students. This "bulk email system", as we call it, allows the university administration to send email to our students based on a query in the LDAP server. It's rather trivial really to send a message to all students. It's more interesting when you can do things like "send a message to all freshmen who are majoring in history and let them know that noted historian Dr. Smith from the Smithsonian will be speaking next week." We also give faculty the ability to send messages to students in their courses. The faculty interface is very much like a listserv, but the bulk mail system is broadcast only. We use LDAP to control who can send the messages and to store the attributes we can query on.

My presentation today is about how you can use LDAP and Apache to improve your authentication, authorization and configuration management.

First let's look at what a directory service is. A directory service is essentially a specialized database that has been optimized for reads. The data in a directory service is a collection of attributes and values that are usually arranged in a hierarchical fashion. A directory service typically is designed to be available over the network.

Why do we need directory services, in particular when you already have a relational database management system? The answer is that a directory service usually provides quicker responses to applications that need directory data (like what is Mark Wilcox's email address or a list of everyone in the engineering deparment). Because a directory service uses a hierarchical fashion, they are usually easier to use when managing company personnel information. In the end, you usually use a directory service because of its quick read responses. You use a RDBMS for applications that need transaction management (e.g. the application succeeds only if all of the steps required in the transaction succeeded).

LDAP, the Lightweight Directory Access Protocol, is an open standard for connecting to directories, in particular over the Internet. It was originally inspired by the X.500 directory service developed by the ISO and International Telephone Union (ITU). For a variety of reasons like the fact the X.500 required OSI networking and it's server requirements were so mammoth, some guys at University of Michigan led by Tim Howes, decided to implement X.500 but without all of the weight of X.500. Thus LDAP was created.

LDAP has become the de-facto standard for any type of directory service. It's probably harder to find a mainstream software company that's not doing something with LDAP than it is to find one that is. All of your major operating systems (Microsoft, Sun, Novell and even some Linux distributions) are shipping with LDAP capability built in. Lucent and Cisco are using LDAP to make it easier to manage their network hardware like routers. Big name web sites like CNN are using LDAP to provide improved services.

However, when we talk about LDAP, it can get a bit confusing. That's because in some ways LDAP is like the old Saturday Night Live skit, "it's not only a dessert topping, it's also a floor wax".

When we talk about LDAP we mean one of four things. We are either talking about its access model, its security model, its data model or its naming model.

The LDAP access model means the actual communication protocol. This is where we define how a LDAP client will talk to an LDAP server. This includes the commands to query and update an LDAP server. By the way if I start talking about a directory server instead of an LDAP server, they mean the same thing. At least they do in this topic.

The LDAP security model defines how the data in the directory server is secured. LDAP has a very flexible model. One thing you should note is that by default the data in the LDAP server is only accessible by the Directory Manager (aka the directory superuser) account. Typically LDAP servers use access control lists to define directory permissions. These ACLs are inheritable, so that if you set a permission at a branch, it's permissions flow through all of its "leaves". The LDAP protocol also enables you to use SSL/TLS and the Simple Authentication and Security Layer or SASL to secure communications. SSL/TLS is easier to implement, but SASL gives you more flexibility. SASL providers can range from CRAM-MD5 to Kerberos to biometrics.

The LDAP data model defines how the entries (which are similar to database records) in a directory server should look. An LDAP entry is made up of two or more attributes. These attributes can contain one or more values. The values can be text or binary. Because the LDAP communications protocol is actually a binary protocol, there is no penalty in the protocol for using binary data. However, LDAP is optimized for searching and its hard to search on binary data. So it's usually better to limit the amount of binary data you store in your LDAP server. Besides, if you have an LDAP server, you most likely have access to a Web or anonymous FTP server. These services are optimized for delivery of files. Instead of storing the actual binary data in your LDAP server, point to their real location via a URL.

By the way, if you've ever looked at an LDAP API and wondered why you have to understand BER to understand LDAP, it's because LDAP translates everything into binary, which must then be encoded into text for cross-platform transport.

An LDAP attribute is similar to a database field. Each entry must have a special attribute named objectclass. The objectclass attribute tells the LDAP server what attributes are required and which attributes are allowed as defined in the LDAP server's schema. Objectclasses are lot like a database table, except that they can be extended. Thus you can have an objectclass called person and then create a new objectclass called companyperson. The companyperson objectclass will have all of the attributes of the person objectclass plus new ones you have defined for companyperson.

There are several popular schemas that have been defined for use in Internet directory services. These include schemas for defining people, binary code (e.g. serialized Java objects) and network objects (e.g. Directory Enabled Network). You can also define your own schema for your directory server.

Each LDAP entry has a special attribute that is an unique identifier for that entry. This attribute is called the Distinguished Name or DN. The DN is comprised of attributes that are found in the directory and the entry itself. The DN reads from left (most unique component) to right (least unique component). The left most component of a DN is called a Relative Distinguished Name or RDN.

An example DN is like this:

uid=mewilcox,ou=people,dc=unt,dc=edu

The RDN is uid=mewilcox.

## Apache and LDAP

Now we start to get into why you're here. Why do you care about using LDAP with Apache. In my experience there are three basic reasons why you might want to use LDAP with Apache.

The first two deal with access control. That is you use LDAP to authenticate and authorize users.

The third reason is to make it easier to configure Apache, in particular if you are running a server farm. If you need to setup a redirect and you have 300 copies of Apache running, you won't want to have to edit 300 httpd.conf files. Instead you want to manage those server's configurations from a centralized data store, which in this case would be LDAP.

One of the greatest things about the Apache Web server is that it is so expandable. It is expandable because you have full access to its server API. You add a dash of Perl via mod_perl and you get the best of all possibilities (in my opinion). That is the ability to expand Apache's capabilities via Perl.

In our authentication and authorization examples we are going to use the Net::LDAP module written by Graham Barr. You could also use Netscape's PerLDAP. I've chosen Net::LDAP for three reasons. One reason is its portability. The module is written entirely in Perl, thus it is extremely portable. The second reason is that it's just about the only LDAP API that doesn't have any ties to an LDAP vendor. Thus it works really well in testing between LDAP vendors. The final reason it's the only module that is still going under active development.

Our first example is going to be authenticating with LDAP. While LDAP does allow us to authenticate through a standard userid and password or an X.509 certificate or even Kerberos, we're only going to focus on userid and password authentication.

Here's a sample configuration for Apache::AuthNetLDAP in a Directory section in httpd.conf (see slide).

The Apache configuration is pretty simple. It simply says that only a valid-user (which means someone who's entered a correct id and password) can see files in this directory. The set of PerlSetVar parameters set variables for the PerlAuthHandler parameter. This tells Apache to use the Apache::AuthNetLDAP module for authentication.

To the novice programmer, it's a bit tricky to authenticate a user with LDAP, when you are just given an userid and password. This is because to test for successful authentication with LDAP, it requires the user's DN to test. Thus the first thing you must do is retrieve the user's DN. You do this by searching the LDAP server, using the user's userid to build the search filter.

We use standard mod_perl code to get the user's uid and password (to see the code, please see the slides). Then we connect to the LDAP server.  Please note that I've left out the error checking code here for brevity.

Unless the search returned one or more entries, we return an authorization failure. We really should check to see if there is more than one entry returned and return a failure.

Finally we make a call to the bind function which is the function we use to authenticate to the LDAP server.  Note that this reuses the same LDAP connection we did the search with. This saves time and network resources because you don't have to open another TCP/IP connection. One option I've been toying with is to add the ability to do an LDAP compare instead of a bind. The LDAP compare takes a DN, an attribute and a value for that attribute. What you are saying to the server is "Does the entry at this DN contain an attribute with this value?". The compare returns either true or false. If we are going to compare encrypted passwords, we must first encrypt the given password in the proper format, whether that is SHA-1,  MD5 or UNIX crypt.

A compare operation is actually preferable because it does not require you to either reuse a TCP/IP resource or open a new connection. It's also easier on the server because the server only has to return a result code, not a full set of search results.

However, you can lose on some server features when you authenticate this way. For example Netscape does not enforce any of its password protections (e.g. limit the number of retries, password expiration, etc) in a compare. But some LDAP server's like Novell's NDS 8 do. Then some LDAP servers, the openLDAP server actually encourages to authenticate this way. This is one of those gray areas, you as a programmer have to do as you see fit.

I'd also like to point out that my module can accept any attribute as the userid. While 99% of the time you are going to want to stick with the uid attribute, there are cases where you might want to use the mail attribute or even commonname. With this module it's easy to do that without having to change the code.

Our next example deals with Authorization via LDAP groups.

Groups in LDAP give people more problems than just about anything else. And I'm not exactly sure why. I think one reason is that there is not a standard group object. There are a couple of group objects, groupOfNames and groupOfUniqueNames that are fairly common, but neither are written down in stone. Netscape even added more confusion (along with a great deal more flexibility) with their latest version of their LDAP server when they added dynamic group functionality.

Also I think people either get confused or worried when they realize that to get a particular value from all of the group's members (e.g. all of their email addresses) you must do an individual search on each member. There isn't a simple getGroupValue API call that does the magic.

But groups aren't that bad. In fact I think the way LDAP handles groups gives you much more flexibility than you think is possible.

There are three basic types of groups in use by LDAP today. They are groupOfNames, which stores its members in the members attribute. The next group is the groupOfUniqueNames, which stores its members in the uniquemembers attribute. Both of these groups members values are the DNs of their member's entries. Their members can also be groups.

The final group is Netscape's dynamic group object. This object stores its members as LDAP URLs in the memberurls attribute. The objectclass is groupOfURLs. The URL value is actually an LDAP query which you can use to perform an LDAP search or to simply compare attribute(s)/value(s) with a given entry.

The configuration allows us to use a couple of standard authorization parameters, "require valid-user" and "require user". Note that we can either take a userid or another value like an user's email address if you are using a non-standard userid.

The last two parameters are what we're interested in. The first parameter "require group", takes the DN of the group we require the user to be a member of. If you are using this module, the user could be a direct member of this group, a member of a group that is a direct or child member of this group, or who's entry matches an LDAP query contained inside the group, if the group is a dynamic group.

The second parameter, "require ldap-url" takes an LDAP URL. Access is given if the user's entry matches successfully with the given query.

Now let's take a look at the actual code.

Until we get to the subroutine calls, the code is exactly like the authenticate module.

The checks for valid user and matching on user value are pretty simple.

The subroutines are more relevant to us.

The first subroutine we call is _getIsMember().

This routine takes a handle to an LDAP connection, to the Apache request object, to the user's DN and to the group DN.

We first check membership using an LDAP compare. As I said earlier, LDAP compare is much quicker and efficient than doing an LDAP search. We check to see if the user's DN is the value for either the group's uniquemember or member value. If the search returns true (result code 6), then we return with a positive match.

Else we then check to see if any of the member entries of the group entry are groups themselves. We then retrieve those groups and then call _getIsMember() recursively.

The _getIsMember and _checkURL subroutines both are capable of handling LDAP queries. The _getIsMember uses this capability to support Netscape dynamic groups.

The _checkURL subroutine is included so that you can provide dynamic group capabilities without having a Netscape directory server.

A reason why Netscape created the concept of dynamic groups and the reason why I use them is because they make group membership so much easier to manage. As we've learned, group memberships are stored in a multivalued attribute such as uniquemember. Multi-valued attributes can be difficult to manage because you can't easily replace a single value. To replace even a single value, you must get all of the values, replace the offending value and then readd ALL of the values. If you are only dealing with 5 or 6 members, no problem. But at UNT I use groups to describe course memberships. Being a large public university I have several courses with 2 or 3 hundred students. I even have 1 or two that have 1000 plus students.

Thus to make it more manageable, I store course memberships using dynamic groups. I create a group an entry for each course. An attribute in each student's entry keeps track (via the course's DN) of what courses the student is enrolled in. The course object contains an LDAP URL that contains the search query that will retrieve all of the students (useful for determining membership or sending email). Thus it's like a many to many relationship.

To determine membership, we first must get the search filter. We can ignore the rest of the URL. Then we perform a search setting the search base to the user's DN. This will only search the user's entry, reducing the amount of time needed for the search.

If the search returns with a returned entry, it's a match. If it doesn't, then it's not a match. Pretty simple. Perhaps we could break the filter into components and do a compare, but I think that wouldn't be any quicker than an LDAP search, in particular if the search is using indexed attributes (a server controlled feature).

There are several things I'm considering for the future of these modules. These are partly driven by personal experience and partially driven by talking to other people.

One thing I would like to add is the ability for the user to choose whether or not to use compare instead of bind for the password authentication. Second, I would like to cache results to improve performance. Or I could possibly also add the ability to only check during the initial request. This would tell the modules to return true in any sub-request. It would also be nice to have the user be able to tell the module to DECLINE an authentication request if it can't find the user, so that it can let a later module have a crack at it. Finally I need to do things like clean up the code, test it and document it. You know all of the fun stuff.

The final thing I would like to talk about is using LDAP to configure an Apache server. This is a topic that I'm just beginning to research. What you would need to do is to define an LDAP schema for Apache. This is the toughest part. After that it's pretty simple. You write some code inside of a <Perl> directive in your httpd.conf file. This populates the Apache configuration variables.

Some of the things that you need to consider are security. For example who can see the configuration information. This is particularly import if you're doing SSL HTTP or have confidential Web sites. There's also a performance penalty to be paid due to the overhead of TCP. MessagingDirect has an LDAP server that runs over UDP & I suppose someone someday may  come up with an implementation of openLDAP that works via UDP, but until then you'll either have to pay MessagingDirect or pay the price of TCP.

Of course you might ask, why would I want to configure an Apache server via LDAP. The answer is you won't unless you have a server farm. In that case you'll want to have a central configuration "file" instead of managing several hundred text files.

That wraps up my presentation. If you have any questions you can send them to me at mark@mjwilcox.com.