Yes, Virginia, There Is Still LDAP

CHRIS "MAC" MCENIRY

Chris "Mac" McEniry is a practicing sysadmin responsible for running a large e-commerce and gaming service. He's been working and developing in an operational capacity for 15 years. In his free time, he builds tools and thinks about efficiency. cmceniry@mit.edu

With the current trend of adapting web-based single-sign-on solutions, it is easy to forget about one of the most prominent authentication and user information systems still in use: LDAP. At its base, LDAP is a collection of objects that:

1. have a Distinguished Name to identify them,
2. have attributes that follow predetermined schema, and
3. are structured and related to each other as nodes on a branching tree.

The above properties affect how you identify and manipulate them.

Two of the most common implementations of LDAP are Microsoft's Active Directory and OpenLDAP. Microsoft's Active Directory (AD) underpins many corporate infrastructures. While you may not want to use it for all AD operations, AD provides LDAP as a first-class way of searching and modifying objects inside of it. OpenLDAP is commonly found in many open source shops and large cluster installations.

In this article, we will look at two common interactions with LDAP:

- How do you find a user in LDAP?
- How do you add a user to a group in LDAP?

Along with properly assigned group ownership, these two can be used to help users manage their own groups.

To help us out, we're going to focus on the go-ldap library (https://github.com/go-ldap/ldap). In addition to that, we will use the Go Subrepository library for password handling (https://golang.org/x/crypto/ssh/terminal).

Setup

The code for this is found in the `useldap` directory of the GitHub repository (https://github.com/cmceniry/login). It includes Gopkg configurations to pull in dependencies. Both the `search` and the `group` commands are expected to be run with a simple `go run ...` command.

In addition to the code, you will need access to an LDAP server. If you are familiar with LDAP, you can probably modify the examples as necessary for your situations.

If you are new to LDAP, one of the fastest ways to get up and running is to run OpenLDAP as a docker container:

```
docker run --hostname ldap.example.com \
   --name ldap -d -p 389:389 -p 636:636 \
   osixia/openldap
```

Once up and running, you will want to load the included `data.ldif` file:

```
ldapadd -H ldap://localhost \
   -D "cn=admin,dc=example,dc=org" -w admin \
   -f ./data.ldif
```

While there are common conventions that appear between LDAP installs, the specific locations and paths used for objects can vary. In the examples here, we limit our users to the `ou=people,dc=example,dc=org` subtree, and our groups to the `ou=groups,dc=example,dc=org` subtree. If you are attempting the same thing against Active Directory, its structure will depend entirely on your Forest, Domains, and Organizational Unit structures. You may have to change the search filters widely to find the appropriate objects there.

For the sake of brevity, we will ignore TLS in this example. However, if you are using LDAP, you should be using it securely. Luckily, the LDAP Go library referenced here has simple support for TLS. Add the TLS configuration after the `ldap.Dial` calls:

```
err = l.StartTLS(&tls.Config{
   …
})
```

Safely Reading Passwords

LDAP does not maintain a constant session across multiple connections, but does require authentication, known as "binding" inside of LDAP. Our examples are simple command line tools which will create new connections every time that they are invoked. This means that we're going to have to authenticate every time as well. To do that, we'll want a safe and cross-platform way to obtain the user's password. In this case, it is the simple "admin" password, but we should still handle it safely.

passwd.go: GetPassword.

```
func GetPassword() (string, error) {
   fmt.Printf("Password: ")
   pw, err := terminal.ReadPassword(int(os.Stdin.Fd()))
   fmt.Println()
   if err != nil {
      return "", err
   }
   return string(pw), nil
}
```

We begin the above function by asking for a password via our "Password:" prompt. We don't end this `Printf` with a new line in order to preserve it as a prompt. This doesn't change the behavior of it, but it is the common convention for the user interface. The magic comes in the form of `terminal.ReadPassword`, which is the cross-platform method of obtaining input without echoing it back to the screen.

We finish the main prompting with the `Println` for two reasons. First, since `terminal.ReadPassword` disables echo, any new line entered by the user will not be echoed and so the next printed characters will end up on this line. In addition, the `Println` statement resets the echo state of the terminal. Any `Print*` would do, but we are taking out two birds with one stone.

Finding a User

When doing group changes, the first step is to identify the users to be added or removed from the group. Our first utility will help us identify users. In the simple case, we're going to accept a command line option, the name of a user to find, and return the distinguished name (DN) for that user.

We start by getting the admin password using our `terminal.ReadPassword` wrapper. In this example, we're going to `panic` if anything goes wrong.

search/main.go: getpw.

```
pw, err := useldap.GetPassword()
if err != nil {
   panic(err)
}
```

With password in hand, we open our connection to the LDAP server. `ldap.Dial` follows the same form that any of the `Dial` functions do: protocol and hostname:port. After checking for error, we defer closing the connection so that it will properly shut that down when we are finished (probably not needed in this case, but good practice nonetheless).

search/main.go: connect.

```
l, err := ldap.Dial("tcp", "localhost:389")
if err != nil {
   panic(err)
}
defer l.Close()
```

After connecting, we need to identify ourselves. In LDAP terms, this is called binding. Binding takes a distinguished name and a password. Our DN is the LDAP admin account.

search/main.go: bind.

```
err = l.Bind("cn=admin,dc=example,dc=org", pw)
if err != nil {
   panic(err)
}
```

Once fully into the server, we can perform our search with the `Search` method of our LDAP connection. `Search` takes one argument, `*SearchRequest` which is constructed with the general `NewSearchRequest` func. `NewSearchRequest` takes nine arguments:

1. The base DN or section of the tree to search under
2. The scope or how deeply into the tree to search
3. The "Deref" flag to show if there are any objects pointed to
4. The limit on the number of resulting entries to get (this can be further restricted by the server, so the response may not always be the same)
5. The time limit to wait for a response
6. The "TypesOnly" flag to indicate whether to show attributes' names only or names and values

7. The filter to use to search what matches which attributes to select one
8. The limit of the attributes to return
9. The controls that affect how a search is processed (e.g., to support paging of results)

Of these, the most common one to change is the search filter, or what to search for (no. 7), and the second most commonly changed is the base DN, or where to search for it (no. 1). For our example, we want to look only under the `ou=people,dc=example,dc=org` part of the tree and only for those entries where the common name, or `cn`, attribute matches our command line options.

search/main.go: search.

```
results, err := l.Search(ldap.NewSearchRequest(
   "ou=people,dc=example,dc=org",
   ldap.ScopeWholeSubtree, ldap.NeverDerefAliases,
   0, 0, false,
   fmt.Sprintf("(cn=%s)", os.Args[1]),
         nil, nil,
))
```

Now we show the output with three loops. The `results` struct has a primary field, `Entries`, which is an array of all of the returned LDAP objects. We can iterate over the array of objects. Each object has a DN and an array of attributes. By iterating over this array, we can see that each attribute can have multiple values (e.g., multiple `member` attributes for group membership), so we finally iterate over those and display them.

search/main.go: show.

```
for _, r := range results.Entries {
   fmt.Printf("------- %s -------\n", r.DN)
   for _, attr := range r.Attributes {
      for _, v := range attr.Values {
         fmt.Printf("%s: %s\n", attr.Name, v)
      }
   }
}
```

Updating a Group

Once we have the reference to the user object, we can make sure that that is a member of the group. In our second tool, `group`, we're going to accept a DN (**note: not user cn or name**) and ensure that that exists on our mygroup group (i.e., add it if it doesn't exist, or just leave it there if it does).

We start by getting the password, connecting, and binding as we did before:

group/main.go: getpw,connect,bind.

```
pw, err := useldap.GetPassword()
...
l, err := ldap.Dial("tcp", "localhost:389")
...
err = l.Bind("cn=admin,dc=example,dc=org", pw)
```

Modifying an LDAP object with something it already has results in an error. So we first want to check that our user addition doesn't already exist on the group. We perform an LDAP search, but this time on the group.

group/main.go: search.

```
results, err := l.Search(ldap.NewSearchRequest(
   "ou=groups,dc=example,dc=org",
   ldap.ScopeWholeSubtree, ldap.NeverDerefAliases,
   0, 0, false,
   "(cn=mygroup)",
   nil, nil,
))
```

With this result, we iterate through the `member` values and exit out successfully if the DN is already there.

group/main.go: exist.

```
members := results.Entries[0].GetAttributeValues("member")
for _, v := range members {
   if v == os.Args[1] {
      os.Exit(0)
   }
}
```

Once we confirm the addition isn't already there, we proceed to update the group object. Similar to the `NewSearchRequest`, we construct a `NewModifyRequest` that we can feed to `Modify`. The main difference between using the two is that we create the request struct and then add our modifications to it. In this case, we `Add` our DN as a value for the `member` attribute. Again, attributes can have multiple values, so we add the array (even if it's only one value).

group/main.go: modify.

```
m := ldap.NewModifyRequest(
   "cn=mygroup,ou=groups,dc=example,dc=org",
)
m.Add("member", []string{os.Args[1]})
err = l.Modify(m)
if err != nil {
   panic(err)
}
```

And with that, we've ensured that our user is on the group.

Conclusion

This example shows that Go has the chops to exercise even what many forgot is a common protocol underlying a lot of infrastructures. The above could be done with the appropriate invocations of `ldapsearch` and `ldapmodify`, but we can encode some of our conventions (tree structure, attribute names) and simplify what we must know to achieve our goals. Add to that that we can distribute these tool binaries as single files, and we can provide simple interfaces for our users and ourselves to manage our resources. This is a very useful method to keep operations running smoothly in any organization.