**SANS Institute**
Information Security Reading Room

# Programmatic Management of Active Directory Groups

Don Quigley

Don Quigley
SANS GIAC Version 1.4b, Option 2

# Programmatic Management of Active Directory Groups

## Abstract

Management of security group memberships in midsize and larger organizations has always been a problematic issue. If individuals are not in the correct groups, they usually need to call the company's security department, explain the issue, and get approval to gain access to the security group before they can perform job related tasks. For large companies with high turnover this can result in hundreds of security requests per week. The impact to the bottom line of a company due to lost productivity and salaries for the additional help desk personnel required to handle these requests can be significant. How much money a large company loses due to these inefficiencies can been seen in a recent article from CIO Magazine:

> "Jonathan Penn, a research director for Cambridge, Mass.-based Giga Information Group, says provisioning can save as much as 50 percent of all IT time spent on user account management, such as creating new accounts, changing accounts and disabling accounts." [1]

Even if we ignore the additional cost required to manually process security requests, manually maintaining security rights can lead to an auditing nightmare. Few organizations actively monitor the membership of security groups. Even though an employee's job responsibility may change over time, access to applications and data that is no longer required is seldom removed.

I currently work at a company with a base of 160,000+ active computer users. Using some homegrown Perl code that I have written along with our metadirectory solution, we have automated our group provisioning/de-provisioning process where possible. We are currently averaging around 300 automated Windows 2000 group adds/deletes per day. This paper goes into some detail to explain the solution that was developed and includes the Perl code in the appendices (although more up to date code and documentation can be found at my website after September 10, 2003: www.donquigley.net). Although the code is designed to work with Critical Path's MetaConnect product as a constructed attribute, I have also included a program that can be used to "manually" call the subroutine so the only real requirements to use the code is an LDAP [4] accessible data store and Perl.

## Before

My company has a user base of well over 160,000 users. The users include employees, consultants (10,000+), and agents (users with system accounts and access to some of our systems but who are not direct employees). In an environment like this, some form of identity management is an absolute

necessity. The need for some manner of programmatically determining authorization based on business data is fairly well laid out in a recent white paper from PriceWaterhouseCoopers [4]:

> "It's not about just knowing who to let in and who to keep out. That decision is usually pretty clear. It's also about control. Technology controls – like authorization or authentication, for example. Just as importantly, it's about process controls – the rationalized business rules and logic that constrain users, attributes and roles are just as critical as the technical architecture."

To help address this need, we have implemented a fairly complete metadirectory system that automatically provisions and de-provisions ids to our 43 production NT 4.0 domains, miscellaneous NT 4.0 testing domains, UNIX, Lotus Notes, and our internal white pages. The information used to provision for these accounts is mostly derived from our HR database, our external associate database, and our subcontractors database.

As part of our provisioning process into NT 4.0, we have a customized Perl application that automatically provisions/deprovisions users into groups. This program is relatively straightforward and simple to use. It reads in a set of criteria files in a format similar to:

```
{attributeName1#attributeName2# ... #attributeNameN}
value1#value2 .... #valueN          NT4ProgrammaticGroupName
```

When a user's entry is processed by our metadirectory, the user's information stored in our metadirectory is compared to this list of criteria. If the user meets the requirements, the user is added to the group and the group name is added to the user's multi-valued grouplist attribute in the metadirectory. If the user has a group listed in the grouplist attribute and they no longer meet the requirements for this group, the user is automatically removed from the group.

Initially, this form of automated group population would only appear to have a limited impact. In practice, however, this simple piece of code is saving our company a lot of money every year. On an average day, the automated group process automatically provisions or deprovisions users into 100+ groups.

Programmatic groups have helped us to partially address an issue we call "group proliferation". Analysts on projects creating new web-based applications at my company often use NT groups to limit who has access to an application. If these analysts are not aware of an NT group or set of groups that already contain all of the users that need access to the group, they will have a new group created. It is extremely hard (especially with 3000+ system's employees actively working on creating new applications) to keep track of what groups have already been created and why.

Many times an analyst will find a group that contains all of the users that need access to the application in addition to a couple of extra users that do not need access to it. Rather than trying to figure out if the two extra users should no longer be in that group (which requires determining what the group was originally intended for), many analysts will take the easy route and have a new group created.

This group proliferation can quickly lead to an administrative and security nightmare. At one point in time we had 23,121 active accounts and 7273 groups in one of our production NT 4.0 domains. This type of group proliferation can lead to quite a few security risks. Determining what type of access is granted by each group and whether or not all of the users in that group should be there is a nightmare. Left to itself, group proliferation will result in a lot of users that have access to applications and data that they no longer need or should never have had access to in the first place.

Programmatic groups go a long way towards fixing the problem. With a programmatic group, the security analyst will try to determine if there is any information in any of our employee information stores (the HR database, the corporate white pages, etc) that all of the users needing access to an application or set of data have in common. This list of identifying information is then given to the directory team. More often than not, there will already be an existing programmatic group with the same set of criteria. This prevents an additional, unnecessary group from being created. Additionally, this means that anyone no longer needing access to data or an application because their job has changed will automatically be removed from the group. One of the major problems with manually maintained groups is that although organizations are really good at identifying groups a user needs to be added to (users will call the helpdesk to complain about lack of access), organizations usually do a very bad job of removing users from groups they no longer need to be members of (users almost never call the help desk to say they have too much access) [3].

**During**

Like many other organizations, my company has recently started migrating from Windows NT 4.0 to Windows 2000. As part of this migration, I have taken the opportunity to try to address some of the limitations of our existing NT 4.0 group code. Since Active Directory also gives us more options when dealing with groups, a lot more changes needed to be added to the programmatic group code. After developing a list of everything we wanted to change and all of the new features we wanted to add to the group code it was decided that I should just re-write the whole thing.

One of the first decisions we had to make was whether or not to always remove users automatically from groups if they no longer met the criteria or if we should

only remove them programmatically if they had been added programmatically. In our NT 4.0 environment, we often had users manually placed into our programmatic NT 4.0 groups. According to our HR records, these users that were manually added to these groups had nothing in common with the users that were programmatically added. This was mostly due to one fact that has always plagued our metadirectory team and resulted in a lot of Perl code to handle special exceptions. Namely, the primary purpose of the information contained in the HR database is to determine how much a person gets paid and where in the organization's hierarchy they fit. A person's actual job responsibility is determined by their manager. This means we might have a programmer in one office that is also in charge of hiring consultants for the office even though he's not in management. HR does not know and does not care that one of his job responsibilities is hiring consultants -- this additional duty assigned to the programmer by his manger has no bearing on his pay or employee benefits. This means, however, that we cannot programmatically give him access to the application that we use to hire consultants since there is a disparity between his HR defined job responsibilities and his manager assigned job responsibilities.

Because there is (and never will be) a data store that accurately reflects all of an employee's job responsibilities, it was decided that we should only programmatically remove users from groups that they were programmatically added to. This allows us to manually add users to a security group without having to worry about them getting taken out of the group every morning. To keep track of which groups a user was programmatically added to a new attribute in Active Directory called jegrouplist was created. This attribute would contain a list of all of the groups that a user had been added to programmatically. The jegrouplist attribute would also contain the group name and date of any groups that a user had programmatically been removed from.
Unfortunately, this also meant that users would not be removed from groups when their job positions changed if they were added manually. To help alleviate this, every group in our organization has at least two users assigned to monitor it's membership for accuracy. This did not work very well when these group owners were supposed to monitor the membership of 10-20 groups with 2000+ users per group. Using programmatic groups, these group owners only need to verify that user's added manually to these groups still need access. Needless to say, group owners are much more willing to thoroughly examine the access needs of a handful of users in a small number of groups than they were when they had to examine the access rights of hundreds or thousands of users. The end result is that, as an unexpected benefit of programmatic group management, manually maintained group memberships are monitored much more closely than they have been in the past.

Another decision we had to make was how flexible we wanted the code to be when finding groups to programmatically populate. By default, my code will always build programmatic groups in the ou=programmatic, ou=groups branch. If a group with the same name already exists in a different branch, the group

creation will fail. At this point in time, it would have been fairly easy to have the code add users to the pre-existing group even though the group was in the wrong location. It was decided that with the Win2k's ability to delegate, this would be a bad idea. If a not-so-trustworthy user knew we were going to create the UberUser group that has access to all date everywhere, they could create a group called UberUser in a branch that they had admin rights to (only a small number of users have any administrative rights to the programmatic group OU). This would allow that user to grant herself access to this group anytime she felt like. Although for the most part we believe that all of our 160,000+ users have the company's best interests at heart, there's that one person on the third floor with the shifty eyes that we don't quite trust so we decided that all groups not located in the programmatic group OU would have to have their DN fully specified before we would touch them. This way, if that person does create a group in an improper location, we'll notice in the error log that the code was not able to add anyone else to that group. And, as always, even if we do not notice all of the failed security group adds, our users will be more than happy to call us up and ask us why they don't have access to do their job.

Our new group code also needed to take into account several other options available to us with Active Directory. We had the option of creating 3 different types of groups (6 if we decided to include the ability to programmatically create mail groups), we could nest groups, and groups could be nested across domains. Since these are all good features, we felt that the new group code should be able to support them.

Finally, we also had to address the maximum group size limitation in Active Directory. An Active Directory group should not contain more than 5000 users [6]. With 160,000 users we have the potential to exceed this. To compensate for this limitation I added a check for a numeric group indicator. If this indicator is present, the code will add a number to the base name of the group specified. If more than 900 users are in that group, a new group will be created with the same name but a different number (i.e. AllEmployee1, AllEmployee2, etc). These numeric groups are then nested into another group (i.e. AllEmployees) that can be used to grant access to resources.

Before I step through how the group code functions, I need to mention a few things the metadirectory product that we are using: Critical Path's MetaConnect.

At a high level, the MetaConnect product has 3 parts that we will be concerned with in the paper. The first part are the connector views or CVs. The connector views are external systems that MetaConnect connects to. MetaConnect can read information from connector views, write information to connector views, or both. The metaview is MetaConnect data store. Information read in from the connector views is written to the metaview. Finally, we have the most important part which is the join engine. The join engine "joins" all of the data coming in from the CVs and writes it to the metaview. The join engine is also responsible

for updating information in the client CVs when information in the metaview changes.

MetaConnect uses a change based process rather than a batch based process. This means that user entries are only looked at if something changes. If a user's HR information does not change for a month, MetaConnect will not attempt to do anything with the user's entry for that month. The advantage of this is that our metadirectory only has to process 3000-4000 entries a day rather than all 160,000 entries every day. The disadvantage of this is that when we implement new criteria we need to force MetaConnect to look at any entries that might be impacted by the new criteria. To force MetaConnect to examine an entry, we usually just change the value of the entry's accountList attribute.

So to tie all of this together, let's look at a user who has moved to a different job within the company. HR will enter the user's new information into their database. Based on the database's change log, MetaConnect will notice that the entry has changed. So the join engine will read in the user's entry from the HR CV, process it, and write the change to the metaview. MetaConnect will now notice that the information for the user has changed in the metaview. This will trigger the join engine to look at the user's metaview entry. The join engine will then determine which client CVs contain that user's information and will update those data stores based on the information contained in the metaview and any special rules (like our group code logic).

Although the information we retrieve from HR does not always contain all of the information we wish it would, using HR as a data source does have one advantage and allows us to avoid what appears to be a common problem at other companies.

> "We typically find that about 40 percent of the valid users in the enterprise are people who no longer work there," says Jeff Drake, director of security strategy at IBM Tivoli in Austin, Texas. "Companies are very good at getting you out of the payroll system when you leave, but they're very poor at removing accesses to apps that you were granted." [2]

As this quote from Jeff Drake shows, payroll is very good at removing users in a timely fashion. Since our id and group provisioning process is tied into our payroll/HR system, our list of active users is always very accurate. Even if HR cannot provide you with any user information other than employment status, it has been our experience that it is worth whatever effort is required to get HR tied into your provisioning system

So now that we know how MetaConnect works at a very high level, we can start looking at how the code works and how it could be used at your organization.

The programmatic group program first reads in it's configuration file (b2econfig.txt). The file looks like this:

```
group_criteria_directory = [main working directory]
win2kgroup_directory = [directory containing group criteria files]
max_group_size = [maximum size of numeric groups]
mv.win2kdc = [Metaview server name or IP]
mv.username = [FQDN of user to connect to MV as]
mv.password = [Encrypted MV password]

[cv].default_base = [Base name of directory]
[cv].default_user_base = [Default location for programmatic groups]
[cv].win2kdc = [Server name or IP for this cv]
[cv].username = [User to connect to cv as]
[cv].password = [Encrypted password for this cv]
[cv].description = [Human readable description of cv]
[cv].cvname = [Internal Join Engine name for cv]
[cv].fileextension = [File extension for cv's group criteria files associated]
[cv].timeout = [LDAP connection timeout for this cv]
```

The first block of lines are universal settings and will appear only once in the configuration file. The second set of lines are CV specific. Most of the settings are pretty much self-explanatory. The [cv] name is intended to be the user friendly name of the CV. Anything that doesn't contain a '.' or '=' can be used as the [cv] name. The [cv].timeout value is the total amount of time that a connection is assumed to be valid in seconds. So if this were set to 600 seconds, then any operation occurring more than 600 seconds after the last bind will automatically drop the old connection to the CV and create a new one. The group code also detects invalid LDAP handles and will re-bind on any connection related errors so, in theory, a timeout value isn't absolutely essential to the code but it makes me feel better to have one.

One additional item to note is that the [cv].fileextension is actually used in a regular expression match against files in the group criteria directory. This means that instead of

```
CorporateDomain.fileextension = 2kCDgrp
```

you could use:

```
CorporateDomain.fileextension = (2kCDgrp|2kallcvgrp|2kallprodgrp)
```

Once the configuration file is read in, the program will examine the user attributes passed to it by the join engine. One of the attributes passed to the join engine is the destination CV. Unfortunately, the maximum length of an internal CV name in MetaConnect is limited to 6 characters. To make the configuration file a little

easier to read we define a friendly CV name for each CV. If there is no [friendlycv].cvname = [unfriendlycv] setting corresponding to the unfriendly cv that the code is currently looking at, the code will return. Otherwise, it will use the friendly cv name to determine which settings should be used while processing the current user.

Now the code will look at all of the files in the group criteria directory and will read in any files with file extensions matching the [cv].fileextension setting and build a list of programmatic groups that the user should be populated into.

Group Criteria File Parsing

Group Criteria files will look like this:

```
{employeeType}
10      ==      PRIMARY     ! NAME:BLAH_G ! TYPE:G ! NUMERIC:Y \
        ==      SECONDARY ! NAME:EMPLOYEE_DLG

{officeNum#locationNum#unitNum}
12#02#LGL  ==      PRIMARY ! NAME:CORPLAW_G ! NUMERIC:N

{companyNum#unitNum}
1#5    ==      PRIMARY ! NAME:ACCOUNTING
2#5    ==      PRIMARY ! NAME:PHSYICAL_SECURITY
5#5    ==      PRIMARY ! NAME:HR

{departmentNum}
398673        ==      PRIMARY ! NAME:A_TEST_NONNUM_G ! NUMERIC:N  \
              ==      SECONDARY ! NAME: B_TEST_NONNUM_DLG

{st#mgrCode}
IL#3[157]     ==      PRIMARY ! NAME:UNIT_%UNITNAME% ! NUMERIC:N
```

The first line ({st#mgrCode}) is the header line. This contains the list of attributes that we are trying to match for separated by a '#'.

Underneath that we have one line for each set of criteria that we are using to determine group membership.

In the last example, if a user had an st value of IL and a mgrCode of 37, then our constructed header would be IL#37. So our constructed regular expression would be:

```
IL#37 =~ /IL#3[157]/i
```

In this example, the regular expression match would be successful so the user would be added to the group.

After the first set of '==' we will have a list of primary and secondary groups along with some information on how they should be built.

The Primary Group is the group that users are added to. The Secondary Group(s) are the group(s) that the Primary Group will be nested into.

We can only have one Primary group listed for each set of criteria. We can have multiple Secondary groups associated with a Primary group (in other words we could have a Primary global group that gets nested into 35 Secondary domain local groups).

The format for a group criteria line is:

regex == primary group specifications == secondary1 specs == secondary2 specs == ... == secondary specs

Each group that is specified in the group criteria line is separated by an '=='. Each specification within a group's specs is separated by a '!'.

To specify a Primary group, we need to indicate that it is a primary group by putting the word PRIMARY by itself in the specs. At a minimum, we also need to specify the name of the group by adding a spec that looks like: NAME: GroupName.

```
{empType#officeNum}
G#68  ==      PRIMARY     !         NAME:  IlinoisInternal_G
```

Optionally, we can also specify the group type (Global, Local, Universal, Mail Global, Mail Local, Mail Universal => G|L|U|MG|ML|MU), a unique base dn for the group, and whether or not the group is numeric (Y/N).

```
{empType#officeNum}
10#13 ==  PRIMARY  !  NAME:  IllinoisInternal_G  !  TYPE: G  ! \
              NUMERIC: N  !  BASE: ou=UselessGroups,c=us
```

For the Primary Group, we will default to Numeric Global and create the group in the default group dn specified in our configuration file.

To specify a Secondary group, we need to indicate that it is a secondary group by putting the word SECONDARY by itself in the specifications. We must also specify the name of the group by adding a specification like: NAME: GroupName

```
{empType#officeNum}
```

```
10#13       == PRIMARY  !  NAME:  IlinoisInternal_G  \
            == SECONDARY  !  NAME:  Illinois_DLG
```

We can also optionally specify a BASE and TYPE for any Secondary groups. Secondary Groups will default to a domain local group built in the default group dn specified in the configuration file.

As you can probably guess, these lines could start getting really long if we fully specify everything and we're nesting into one or more groups. To help with this, you can use the line continuation symbol by itself to indicate that the line is continued on the next line. So we could have a line like:

```
10    ==    PRIMARY     ! NAME:A_ATTRIB_%OFFICENUM%_G \
            ! TYPE:G ! NUMERIC:Y \
      ==    SECONDARY ! NAME:A_ATTRIB_%MANAGERID%_DLG \ !
            TYPE:L \
      ==    SECONDARY ! NAME:A_ATTRIB_%OFFICENUM%_DLG \
            ! TYPE:L
```

This should make things a bit easier to read. We can also add comments at the end of each continuation:

```
10    ==    PRIMARY ! NAME:GROUP_G ! TYPE:G ! NUMERIC:Y \ # Blech
      ==    SECONDARY ! NAME:A_ATTRIB_%MANAGERID%_DLG \
            ! TYPE:L \  # Double-Blech
      ==    SECONDARY ! NAME:A_ATTRIB_%OFFICENUM%_DLG \
            ! TYPE:L
```

You can have white space between the '\' and the '#'. The downside to being able to use line continuation characters is that we will not be able to build any groups that have a '\' or '\#' in their name or their base dn.

NUMERIC GROUPS

If a Primary group is specified as numeric or numeric is left undefined (the code defaults to Numeric:Y), the group code will add a number to the specified group name in front of the last '_' or, if there is no '_', it will append the number to the end of the group name. The group code always begins searching for groups with available space starting at the number 1. The group code will add the user to the first numeric group that has less than the maximum number of users in it.

ATTRIBUTES IN GROUP NAMES

You can also specify one or more attributes in group names (both Primary and Secondary groups). To do this, place a '%' before and after the attribute name.

The group code will substitute the value of the user's attribute when creating the group name.

For multi-valued attributes, multiple groups will be created. If a multi-valued attribute is specified for both a primary and secondary group, the values for the attributes will be 'tied' together. For instance, if we had a Primary group of:

Multi_State_%st%_G

and a Secondary group of:

State_%st%_DLG

and the user's st values were IL and MO, we would get:

| | | |
|---|---|---|
| Multi_State_IL_G | nested into | State_IL_DLG |
| Multi_State_MO_G | nested into | State_MO_DLG |
| Multi_State_IL_G | NOT nested into | State_MO_DLG |
| Multi_State_MO_G | NOT nested into | State_IL_DLG |

This will also work if a Primary and one of it's Secondaries contains multiple multi-valued attributes.

Cross Domain Nested Groups

We also have the need to nest global groups in one domain into local groups in another domain. This can be accomplished with:

```
{workcode}
.*      ==      PRIMARY ! NAME:P_%workcode%_G ! TYPE:G \
        ==      SECONDARY ! NAME:External\P_%workcode%_DLG
```

Basically, we're just putting the name of the domain in front of the group name.

NAME: [Domain_Name\]GroupName

In the example above, if we have a user with a workcode of 121234 in the Corp domain, the userid will be placed into the global P_121234_G global group in the Corp domain. The P_121234_G domain local group in Corp will then be nested into the P_121234_DLG group in the External domain.

The domain name specified must either be the user friendly CV name (as defined in the b2econfig.txt file) or it can be the actual CV name as used by the Join Engine.

It should be noted that the groups are not immediately nested. If the global group has just been created by the group program in Corp, the External domain may not know that it exists yet and Active Directory will not allow you to add the Corp P_121234_G group to the External P_121234_DLG group. To account for this, we store the nesting relations in a .db file that is processed by our crossdom-group.pl script. When crossdom-group.pl is run, it will nest all of the groups that it can. Any groups that it cannot nest will not be deleted from the .db file so we can attempt to nest them again after replication has occurred.

After parsing through the group criteria files, we now have a proposed group list of all of the Primary programmatic groups that a user should belong to. We also have another list of Secondary groups that the Primary groups should be nested into.

Now we compare this proposed group list to the list of groups that user is currently in. If the user is currently in a group that is not listed in the current jegrouplist attribute, we will add the group to the jegrouplist attribute (no need to do an ldap modify).

If the user is not currently in a group listed in jegrouplist, the group code will attempt to add the user to the group. If the add fails because the group does not exist, the group code will attempt to create this group and then add the user account to it. If the creation attempt succeeds, the group code will attempt to nest the newly created group into any Secondary groups associated with it. If the nesting fails because the Secondary group does not exist, the group code will create the Secondary group and then nest. If the nested group is in a separate CV and the nesting operation fails (due to replication delays between the two domains), the nesting information will be written out to a DB file that can later be processed by our crossdom-group.pl script.

It should be noted here that normally the group code will only attempt to nest Primary groups into Secondary groups if the Primary group has just been created by the group program. This also that Secondary group(s) will not be created if the Primary group already exists. If the Primary group already exists, the group code will not do an LDAP search to verify that the Secondary group exists and has been nested into. This is by design to allow the Join Engine to run faster. If you intend to use this code in a smaller environment, you can modify the code so it always performs this check. In our environment, we are occasionally required to perform a check on all 160,000+ users. Even adding a quarter of a second to the time required to process a user can add more than two hours to our run time (assuming we have four threads running at the same time). If we are doing this full refresh during the day because of enterprise wide security issues, this additional 2 hours can have a quite an impact the company's bottom line.

The group code can be forced to check for proper nesting and Secondary group existence, however. If the user's MV accountList attribute has a value of 2112 or

2113 when the user entry is processed by the Join Engine, the group program will attempt to create the Secondary group and nest the Primary group into it whether or not the Primary group already exists.

Next, the Group code will remove the user from any groups listed in the current jegrouplist attribute that have criteria the user no longer meets. When the group is removed, the value is replaced with the group name and a timestamp for auditing purposes. If the user cannot be removed from a group (i.e. directory error or the group was deleted manually), the group will not be removed from the jegrouplist attribute. We do this so that the next time the user is processed the group code will attempt to remove the user again. For this reason, groups should not be manually deleted until all of their users are removed programmatically. If you make it a habit to delete groups manually before all users are removed by the join engine's group code, you can end up with a lot of orphaned jegrouplist entries.

Manual Group Runs

Group population can occur programmatically outside the MetaConnect process. This can be useful if you do not want the Join Engine's performance to be affected during a mass group update. This can be even more useful if you don't have a Join Engine To manually update a group of users you will need to use the program man-jegrouplist2.pl. The program will prompt you to specify the target's user-friendly CV name (as listed in the b2econfig.txt file) and an LDAP search filter. The LDAP search filter will be used to search the Metaview and retrieve a list of users. If you don't have a Metaview, you can have the program use any LDAP directory by setting the mv.* parameters in the b2e_config.txt file to point to your LDAP directory. The users found as a result of this search will then be run through the same group code. This will perform all of the standard group maintenance tasks on these users that the join engine's group code normally would and will update their jegrouplist attribute.

**AFTER**

We have currently had our programmatic group code in production for almost 6 months now. Although we are only halfway through our Windows 2000 roll out, we have already programmatically created and populated over 32,700 programmatic groups with a little over 90,000 migrated Windows 2000 users.

Many of the security groups are based on an office's physical location so confidential customer information cannot be shared between different offices. The company I work at actually has more physical locations to provide customer support than there are McDonalds in the U.S. These locations usually have 3-5 users in them and experience a turnover rate of more than 110%. Because of the high turnover, we usually have over 100 group adds and 100 group deletes for the groups associated with these physical locations every day. These are

users that, in the past, would have had to request access and have it manually granted by an administrator.

Not only has the automatic population of physical location groups resulted in a cost savings of at least $5000 per work day (it costs the company approximately $25 for any internal technical support calls), but this has made our environment much more secure. It is not uncommon for a single user to move from one physical location to another multiple times in the year. In the past, these users were only removed from their old physical location groups if the manager at the location remembered to fill out the forms to get their access removed or during an audit. This resulted in some users retaining access to confidential customer information even though there was no longer any business need for them to have that access. With our new automated group provisioning process, these users are automatically being removed in a timely manner if they move offices or quit. This has also eliminated the problem of users being manually added to the wrong group. Additionally, a simple ldap search:

((&(!(jegrouplist=groupName))(memberof=FQDN_of_group)))

can help us identify any users in a group that were not added programmatically (the rogue admin problem).

We have also had a similar success with groups that are being populated based on job information rather than physical location. Although these groups make up less than 2% of the total groups managed to date, these have historically been the most difficult groups to manage. With the new group code, we have already doubled the number of groups that we are able to programmatically manage even though we are not quite to the half-way point in our migration. As a result of being able to use regexes in our group criteria and having more options available to us (such as cross-domain nesting), we are currently managing programmatically a little over 80% of our groups that are not based on physical locations.

Now that manually maintained access groups have become the exception rather than the rule, we are better able to audit the membership of these groups on a periodic basis. For example, in one of our old NT domains we had over 6000 groups that were manually maintained. In the equivalent Windows 2000 domain, we now have only 117 groups that are manually maintained despite the fact that the Windows 2000 domain has more users in it.

Another benefit of the new code is how the user base views our security department. In the past, if a user did not have access to an application, the help desk would need to determine what security group restricted access to that application. Then the help desk would need to find the individuals responsible for the membership of that group and get their approval to add the user to the group. For a new employee or an employee that had changed jobs, this process could

be repeated several times before the user had all of the access they needed. With the new group code, the user has access to almost everything they need for their new job just as soon as HR enters the user's new information into their database. From the user's perspective, the security department is now a helpful department that gets all of their access set up before they need it instead of a department that is preventing them from being able to do their job. Since these users are now happy (or, at the very least, less angry) with security, they are also more willing to work with security when issues arise rather than trying to work around security.

In summary, the programmatic group code has done quite a few things for my company. It is saving a lot of money every year in administrative costs. Users are rarely in security groups that they should not be in. We rarely encounter users that cannot get work done because they are not in the security groups that they need to be in. And, finally, users are much happier with the security department.

## References


[1] Levinson, Meridith. "Who Goes There?". 1 December 2002. URL: http://www.cio.com/archive/120102/et_article.html (8/25/2003)

[2] Margulius, David L. "Tackling Security Threats from Within". 28 April 2003. URL:http://mail.toadworld.org/PublicFolders/Misc%20Newsletters/TEST%20CENTER%20REPORT%20from%20InfoWorld.com,%20April%2028,%202003.EML (8/25/2003)

[3] Carter, Gerald. LDAP System Administration. O'Reilly & Associates. March 2003.

[4] PriceWaterhouseCoopers. "The Value of Identity Management for the Communications Industry". 2003. URL: http://www.pwcglobal.com/Extweb/service.nsf/8b9d788097dff3c9852565e00073c0ba/9a7f8cade2d39dbc85256cde006a1d1c/$FILE/IdMForCommInd.pdf

[5] Blank-Edelman, David N. Perl for System Administration. O'Reilly & Associates. July 2000.

[6] Microsoft. "Exchange 2000 Resource Kit: Chapter 4 - Active Directory Design". 2003. URL: http://www.microsoft.com/technet/treeview/default.asp?url=/technet/prodtechnol/exchange/exchange2000/reskit/part2/c04names.asp (8/25/2003)

Appendix A: Win2kgroups.pl

This is the primary program used to programmatically add and removed users from Active Directory groups. This program is designed to be used as a constructed from within Critical Path's MetaConnect program. It can also be used in conjunction with the man-jegrouplist2.pl program found in Appendix B.

```
# win2kgroups.pl
# V 1.1
# Created by Don Quigley
# quigley@techie.com
# 4/1/2003

# If this is used with Critical Path's MetaConnect product, this can be set up as
# a constructed attribute.  Otherwise, this can be called from man-sfjegrouplist2.pl

package ProgGroup2;
require "ldap-conn.pl";
use Net::LDAP;
use Net::LDAP::Util qw( ldap_error_name
                        ldap_error_text) ;

# Need to add ability to nest groups across domains
# Need to create negative name generation routines (I tried using an anti-name
generation routine,
# but every time an anti-name touched a name it blew up the join engine).
# Ought to add a check for EOF at a continuation line in the criteria file


sub sf2kGroup {
    my @returned_list;

    eval {

    my %Details = ();   # User attributes passed from join engine
    my %Setting = ();   # Configuration settings from config file
    my %CVNames = ();   # Map of Join Engine CV name to user friendly CV name
    my %proposed_groups = ();               # Groups MetaConnect thinks user should be
in
    my %negative_groups = ();               # Groups MetaConnect should not place user
into
    my %current_jegrouplist = ();# Groups MetaConnect has placed user into
    my %current_memberof = ();              # All groups that the user is currently a
member of
    my %primary_group = ();
    my %secondary_group = ();
    my %constructed_secondary_info = ();
    my %constructed_primary_info = ();


    #################
    # GET USER INFO #
    #################
    # Read in all of the attributes passed to us by the join engine
    # Multi-valued attributes are stored in the key as a tab delimited string
    foreach $element (@_) {
     $element =~ tr/a-z/A-Z/;
     my ($attrib, $val) = split(/: /, $element);
     $val =~ s/\s+$//;
     if(exists $Details{uc($attrib)}) {
              $Details{$attrib} .= "\t" . $val;
     } else {
              $Details{$attrib} = $val;
     }
     }

    # These are added in so we can use CV/MV names as a criteria
    my $userdn = $Details{'CV.DN'};
    $Details{'CV.CV'} = $Details{'CV'};
    $Details{'MV.MV'} = $Details{'MV'};

```

```
61
62
63
64          ############################
65          # READ IN CONFIGURATION FILE #
66          ############################
67          # Open up our file that contains all of the configuration information we need.
68          open(IN,"b2e_config.txt") || ProgError(4,"Can't open b2e_config.txt");
69          # Read in the b2e_config.txt configuration settings
70          while (<IN>) {
71           if (/^\[/ or /^\s*$/) {next;}          # Let's ignore blank lines and header
lines (lines with [ text ]
72           chomp;
73           s/\s*#.*$//;                           # Get rid of whitespaces in front of #
(beginning of comments)
74           my @line = split(/\s*=\s*/,$_,2);
75           $line[0] =~ tr/a-z/A-Z/;
76           $line[1] =~ tr/a-z/A-Z/;
77           $Setting{$line[0]} = $line[1];
78           if ($line[0] =~ /(^\S+)\.cvname/i) {
79               my $temp = uc($1);
80               $CVNames{uc($line[1])} = $temp;
81           }
82           }
83          close(IN);
84
85          ############################
86          # DEFINE CV SPECIFIC SETTINGS #
87          ############################
88          if (! exists $CVNames{$Details{'CV.CV'}}) {
89           return "";          # We don't know nothing about this CV.  It ain't not in our
config file
90          }
91
92          my $cvname = $CVNames{$Details{'CV.CV'}};
93          my $server = $Setting{"$cvname.WIN2KDC"};
94          my $username = $Setting{"$cvname.USERNAME"};
95          my $basedn = $Setting{"$cvname.DEFAULT_BASE"};
96          my $userbasedn = $Setting{"$cvname.DEFAULT_USER_BASE"};
97          my $fileextension = $Setting{"$cvname.FILEEXTENSION"};
98          my $maxsize = $Setting{'MAX_GROUP_SIZE'};
99
100         my $ldap = B2EGenLDAP::GetLDAP($CVNames{$Details{'CV.CV'}});        # Retrieve
connection to our LDAP server
101
102
103         ######################################
104         # GET FILENAMES OF GROUP CRITERIA FILES #
105         ######################################
106         my @files;
107
108         # Change directories to the working directory defined in the config file
109         unless (chdir($Setting{'GROUP_CRITERIA_DIRECTORY'})) {
110          ProgError(1,"Can't change to directory $setting{'WIN2KGROUP_DIRECTORY'}");
111          die;
112         }
113
114         # Open up the directory that all of the group files are in and populate the
@files array
115         # with the names of all of the group files
116         unless (opendir(GROUP_DIR,$Setting{'WIN2KGROUP_DIRECTORY'})) {
117          ProgError(1,"Can't open directory $setting{'WIN2KGROUP_DIRECTORY'}");
118          die;
119         }
120
121         # Get a list of all of the 2kgrp files in the group directory
122         foreach my $file (sort readdir(GROUP_DIR)) {
123             if ($file !~ /$fileextension$/i) {next;}
124             $file = $Setting{'WIN2KGROUP_DIRECTORY'}."\\$file";
125             push(@files,$file);
126         }
127
128         ########################
129         # INITIALIZE GROUP LISTS #
```

```
130       ##########################
131
132       foreach my $temp (split (/\t/,$Details{'CV.JEGROUPLIST'})) {
133        $current_jegrouplist{uc($temp)} = 1;
134       }
135
136       foreach my $temp (split (/\t/,$Details{'CV.MEMBEROF'})) {
137        $temp =~ s/,.*//;
138        $temp =~ s/cn=//i;
139        $current_memberof{uc($temp)} = 1;
140       }
141
142
143       ##############################################
144       # READ IN CONFIG FILES AND FIND MATCHING LINES #
145       ##############################################
146
147       foreach $file (@files) {
148
149        my $builtline = "";          # Used to build up the current line
150        my @headers = ();            # Used to store match criteria
151        my $user_header_values = "";
152
153        open(IN,$file) || ProgError(1,"Can't open file $file");
154        while(<IN>) {
155
156            if (/^#/ || /^\s*$/) {next;}      # If the line contains just whitespace or
starts with
157                                              # a # (indicating a comment
line) we ignore it
158
159            s/\s*$//;                # Get rid of trailing whitespace --> super-chomp
160
161            if (/^\{.*/) { # If line starts with '{', then it's telling us attribute
names
162                                     # So we take these attribute names in the order
that they are
163                                     # given to us and create a string containing the
corresponding
164                                     # attribute values for the user we're looking at.
For instance,
165                                     # for [userid#recordtype] we would construct a
string
166                                     # that looks like bob15#Employee
167                tr/a-z/A-Z/;         # Make everything uppercase
168                s/\}.*$//;           # Get rid of everything after the }.  Now we can
add comments
169                s/[\{\}]//g;         # Get rid of any { and }.  We'll assume we won't be
given any attributes
170                                     # with a { or } in the name.
171                @headers = split(/#/,$_);    # List of attributes names are delimited
by a #
172
173                # We replace all of the attribute names with the attribute values for
the user
174                # so we can then use the string to do a regex match.  It should be
noted that if
175                # we have a multi-valued attribute, there will be tabs in this string
since we're
176                # storing multivalued attributes in %Details as a tab delimited string
177                $user_header_values = "";
178                foreach $header (@headers) {
179                    # If CV or MV isn't specified in the filter, we'll assume we're
talking about the MV
180                    if ($header !~ /^CV./i && $header !~ /^MV./i) {
181                         $header = 'MV.'.$header;
182                    }
183                    if ($user_header_values eq "") {
184                         $user_header_values = $Details{$header};
185                    } else {
186                         $user_header_values =
$user_header_values.'#'.$Details{$header};
187                    }
188                }
189                next;
```

```
190             }
191
192             # If we get here, we're looking at a criteria line.
193             if (/\\\s*$/ || /\\\s*#/) {       # This criteria line continues onto the
next line
194                   $line = $_;
195                   $line =~ s/\\$//;
196                   $line =~ s/\\#.*$//;
197                   $builtline = $builtline.$line;
198                   next;
199             }
200
201             # If we get here, we're looking at a fully built criteria line.
202             $line = $builtline.$_;
203             if ($line =~ /^\s*$/) {next;}
204             $builtline = "";
205             $line = uc $line;
206
207             # Now we see if the criteria match.
208             # We wrap the regex evaluation in an eval so that an invalid regex in an
209             # input file doesn't crash the program.
210             my ($regex,$actions) = split(/\s*==\s*/,$line,2);
211             eval {
212                   if ($user_header_values =~ /$regex/i) {
213                       $primary_group{$actions}{'ACTION'} = 1;
214                   }
215             };
216             if ($@) {
217                   ProgError(3,"Error in regex group check: $user_header_values compare
to $regex");
218             }
219       }
220       close(IN);
221       }
222
223       ############################
224       # DONE WITH REGEX COMPARES #
225       ############################
226
227       # Now we have a list of group lines that matched the regex criteria.  Now we
need to act on the information
228       # contained in the group lines.
229
230       ########################
231       # FULLY POPULATE HASHES #
232       ########################
233
234       # Cycle through each line that was read in that meets the regex criteria
235       foreach my $line (keys %primary_group) {
236        my %temp_primary_group = ();
237        my %temp_secondary_group = ();
238        # Split out the specifications for each group listed
239        foreach my $specs (split (/\s*==\s*/,$line)) {
240             if ($specs =~ /^\s*PRIMARY[\s!]+/i) {       # primary group stuff
241                   foreach (split(/\s*!\s*/,$specs)) {    # Look at each of the specs
242                       my($setting,$value) = split(/\s*:\s*/,$_);
243                       $setting = uc $setting;
244                       if ($setting =~ /^PRIMARY$/i) {
245                            next;
246                       }
247                       $primary_group{$line}{$setting} = $value;
248                   }
249             } else {                                              # assume secondary
group
250                   my %temp_temp_secondary_group = ();
251                   foreach (split(/\s*!\s*/,$specs)) {    # Look at each of the specs
252                       my ($setting,$value) = split(/\s*:\s*/,$_);
253                       $setting = uc $setting;
254                       $temp_temp_secondary_group{$setting} = $value;
255                   }
256                   my $basename = $temp_temp_secondary_group{'NAME'};
257                   my $temp_userbasedn = $userbasedn;
258                   if ($basename =~ /(.+)\\/) {
```

```
259                     my $temp_cv = $1;
260                     if (exists $Setting{"$temp_cv.DEFAULT_USER_BASE"}) {
261                           $temp_userbasedn = $Setting{"$temp_cv.DEFAULT_USER_BASE"};
262                     } elsif (exists $CVNames{$temp_cv}) {
263                           $temp_cv = $CVNames{$temp_cv};
264                           $temp_userbasedn = $Setting{"$temp_cv.DEFAULT_USER_BASE"};
265                     } else {
266                           ProgError(3,"$basename does not have a valid cv");
267                           next;
268                     }
269                 }
270                 $temp_secondary_group{$basename}{'TYPE'} =
$temp_temp_secondary_group{'TYPE'} || "L";      # Default to DLG
271                 $temp_secondary_group{$basename}{'BASE'} =~ s/\s*,\s*/,/g;
272                 $temp_secondary_group{$basename}{'BASE'} =~ s/\s*=\s*/=/g;
273                 $temp_secondary_group{$basename}{'BASE'} =
$temp_temp_secondary_group{'BASE'} || $temp_userbasedn; # Default basedn for groups
274             }
275         }
276         $primary_group{$line}{'TYPE'} = $primary_group{$line}{'TYPE'} || "G";        #
Default to global
277         $primary_group{$line}{'NUMERIC'} = $primary_group{$line}{'NUMERIC'} || "Y";  #
Default to Yes
278         $primary_group{$line}{'BASE'} =~ s/\s*,\s*/,/g;
279         $primary_group{$line}{'BASE'} =~ s/\s*=\s*/=/g;
280         $primary_group{$line}{'BASE'} = $primary_group{$line}{'BASE'} || $userbasedn;          # Default to
281
282         foreach $group (keys %temp_secondary_group) {
283             $secondary_group{$line}{$group}{'TYPE'} =
$temp_secondary_group{$group}{'TYPE'};
284             $secondary_group{$line}{$group}{'BASE'} =
$temp_secondary_group{$group}{'BASE'};
285             #$secondary_group{$line}{$group}{'GROUPS'} = [];
286         }
287
288         $primary_group{$line}{'2NDARY'} = $secondary_group{$line};
289         undef %temp_primary_group;
290         }
291
292         # OK, now our %primary_group and %secondary_group hashes are almost fully
populated.  The only thing
293         # left to do is construct the actual group name(s) and place them into the hash.
We do this at the same
294         # time we add users to the groups since we need to do some lookups for numeric
groups.
295
296
297         ########################
298         # GENERATE GROUP NAMES #
299         ########################
300
301         # Keep track of what groups each primary group should be nested into.
302         my %nesting_relationships = ();
303
304         foreach $line (keys %primary_group) {
305          # Call out to get a list of groupnames.
306
          &BuildGroupName(\%Details,\%Setting,\%{$primary_group{$line}},$secondary_group{$
line},
307                               \%nesting_relationships,\%negative_groups,\%proposed_groups,
308
          \%constructed_secondary_info,\%constructed_primary_info,$line);
309
310         # Now we have all of the group names created.  We also have the nesting
relationships defined.
311         # $nesting_relationships{primary-group} = [ groups to be nested into ]
312
313         }
314
315         ########################
316         # PUT USERS INTO GROUPS #
317         ########################
318
319         foreach my $group (keys %proposed_groups) {
```

```
320         if ($group =~ /^\s*$/) {next;}  # Ok, this is the cheap way to do it
321
322         if (exists $current_memberof{$group} && $Details{'MV.ACCOUNTLIST'} !~ /211[23]/)
{   # User is already in this group
323             next;
324         }
325
326         my $numeric_flag = 0;
327         if ($constructed_primary_info{$group}{'NUMERIC'} =~ /^Y/i ) {
328             foreach my $current (keys %current_memberof) {
329                   $current =~ /(.+)\d([_\w])$/;
330                   $current = $1.$2;
331                   if (exists $current_memberof{$current}) {
332                       $numeric_flag=1;
333                       last;
334                   }
335             }
336         }
337
338         if ($numeric_flag == 1 && $Details{'MV.ACCOUNTLIST'} !~ /211[23]/) {  #THIS
MAYBE NOT A GOOD IDEA
339             next;
340         }                           # User is already in this numeric group
341
342         my $res =
Luser2Group($userdn,$group,\%constructed_primary_info,\%constructed_secondary_info,
343
\%nesting_relationships,\%Setting,$cvname,\%Details,\%CVNames,$maxsize,\%proposed_groups)
;
344
345         if ($res ne "YEP") {
346             ProgError(3,"Can't add $userdn to $group: $res");
347             # Take the user out of the proposed_groups list so we'll try to add them
again
348             # the next time we process the account
349             delete $proposed_groups{$group};
350             next;
351         } else {
352             my $tempid = $userdn;
353             $tempid =~ s/,.*//;
354             my $tempgroup = $group;
355             $tempgroup =~ s/,.*//;
356             ProgError(5,"ADD: $tempid ==> $tempgroup : $userdn ==> $group");
357         }
358
359         }
360
361         # Now we should have a list of all of the groups that we have programmatically
determined that
362         # the user should be in.  Now we want to compare this list to the previous list
of of
363         # programmatically determined groups.  We want to remove the user from any
groups that they
364         # should no longer be a member of
365
366         ############################
367         # DELETE USERS FROM GROUPS #
368         ############################
369
370         foreach $key (keys %current_jegrouplist) {
371         if ($key =~ /#\d+$/) { # If group name has a date in it, that means we were
removed
372             $proposed_groups{$key} = 1;
373             next;
374         }
375         if ($key =~ /^cn=/i) { # The old jegrouplist had the FQDN of the group
376             $key =~ s/^cn=//i;
377             $key =~ s/,.*$//;
378         }
379         if (! exists $proposed_groups{$key}) {
380             # If a value is returned from the delete user subroutine, that means the
user wasn't
381             # really removed so we won't remove the group from the list of programmatic
groups
```

```
382             # We should probably check periodically to see if any programmatic groups
were manually
383             # deleted without first de-provisioning all of the users.  Otherwise the
manually deleted
384             # group will always be in this list
385             my $temp = DeleteUserFromGroup($key,\%Setting,$userdn,$cvname);
386             if ($temp != 1) {
387                   $proposed_groups{$key} = 1; # If delete fails keep group in list
388             } else {
389                   my $tempdate = &GetDate;
390                   $tempdate =~ s/ .*$//;
391                   $key = $key."#$tempdate";
392                   $proposed_groups{$key} = 1;
393                   my $tempid = $userdn;
394                   $tempid =~ s/,.*//;
395                   my $tempgroup = $key;
396                   $tempgroup =~ s/,.*//;
397                   ProgError(5,"DELETE: $tempid ==> $tempgroup : $userdn ==> $group");
398             }
399       }
400       }
401
402       # Now we want to get rid of any duplicate group#date entries (we only want to
keep the most
403       # recent removal).
404
405       my $prevkey = "";
406       foreach $key (sort keys %proposed_groups) {
407        if ($key =~ /#\d+-\d+-\d+$/) {next;}
408        my $prevtemp = $prevkey;
409        my $newtemp = $key;
410        $prevtemp =~ s/#\d+-\d+-\d+$//;
411        $newtemp =~ s/#\d+-\d+-\d+$//;
412        if ($newtemp eq $prevtemp) {
413            delete($proposed_groups{$prevkey});
414        }
415        $prevkey = $key;
416       }
417
418        foreach $key (keys %proposed_groups) {
419       push(@returned_list,$key);
420       }
421
422       # Return a list of all MetaConnect managed groups that the user should be in
423       my $count = @returned_list;
424
425       # Return nothing if we don't have any programmatic groups.  This will make
MetaConnect
426       # remove the attribute from the user's entry
427       if ($count == 0) { # No programmatic groups
428        return("");
429       }
430
431   };
432   if ($@) {
433       ProgError(2,"sub died with $@");
434   }
435   return(@returned_list);
436   }
437
438   sub BuildGroupName
439   {
440       my ($Details_ref,$Setting_ref,$primary_group_ref,$secondary_group_ref,
441        $nesting_relationships_ref,$negative_groups_ref,$proposed_groups_ref,
442        $constructed_secondary_info_ref,$constructed_primary_info_ref)
443       = @_;
444
445       my (@secondary_groups);
446        my $primary_group_base = $$primary_group_ref{'BASE'};
447       my $primary_group_basename = $$primary_group_ref{'NAME'};
448       my $primary_group_type = $$primary_group_ref{'TYPE'};
449       my $primary_group_numeric = $$primary_group_ref{'NUMERIC'};
450
```

```
451       my %group_attributes = ();              # Used to keep track of attributes used to
construct global group
452                                               # names.  These values will be used when
naming secondary groups
453
454       my $test = $primary_group_basename;
455       $test = s/,.*//;
456       $test = s/^\w+=//;
457       if (exists $$negative_groups_ref{$test}) {return "";}
458
459       my @primary_groups =
PrimaryGroupConstruct($primary_group_basename,$Details_ref,\%group_attributes);
460
461       foreach $group (@primary_groups) {
462        my $test = $group;
463        $test = s/,.*//;
464        $test = s/^\w+=//;
465        if (exists $$negative_groups_ref{$test}) {
466            next;
467        }
468        $$proposed_groups_ref{$group} = 1;
469        $$constructed_primary_info_ref{$group}{'TYPE'} = $primary_group_type;
470        $$constructed_primary_info_ref{$group}{'BASE'} = $primary_group_base;
471        $$constructed_primary_info_ref{$group}{'NUMERIC'} = $primary_group_numeric;
472
473        foreach my $secondary_group (keys %$secondary_group_ref)  {
474            my @secondary_groups =
SecondaryGroupConstruct($group,$secondary_group,$Details_ref,\%group_attributes);
475            foreach my $sec_group (@secondary_groups) {
476                 if ($sec_group =~ /^\s*$/) { next; }  # Shouldn't need this, but
doesn't hurt to have it
477                 $$nesting_relationships_ref{$group}{$sec_group} = 1;
478                 $$constructed_secondary_info_ref{$sec_group}{'TYPE'} =
$$secondary_group_ref{$secondary_group}{'TYPE'};
479                 $$constructed_secondary_info_ref{$sec_group}{'BASE'} =
$$secondary_group_ref{$secondary_group}{'BASE'};
480            }
481       }
482       }
483   }
484
485   sub PrimaryGroupConstruct
486   {
487       my ($group,$Details_ref,$grp_attrs) = @_;
488
489       if ($group !~ /%[^%]+%/) {    # Group name doesn't contain any attributes
490        return ($group);
491       }
492
493       $group =~ /%([^%]+)%/;
494       my $attr = $1;       # We get the attribute name from the previous match
495       if ($attr !~ /^CV\./i && $attr !~ /^MV\./i) {
496        $attr = 'MV.'.$attr;
497       }
498
499       my @groups;
500
501       foreach $val (split(/\t/,$$Details_ref{$attr})) {
502        my $tempgroup = $group;
503        my $tempattr = $attr;
504        $tempattr =~ s/^[MC]V\.//;
505        $tempgroup =~ s/%[MC]?V?\.?$tempattr%/$val/i;
506        push(@groups,$tempgroup);
507        foreach $tempkey (keys %$grp_attrs) {
508            $$grp_attrs{$tempgroup}{$tempkey}=$$grp_attrs{$group}{$tempkey};
509        }
510        $$grp_attrs{$tempgroup}{$attr}=$val;
511       }
512
513       my @final_groups;
514
515       foreach $group (@groups) {
516        my @tempgroups = PrimaryGroupConstruct($group,$Details_ref,$grp_attrs);      # me
so clever
```

```
517         push(@final_groups,@tempgroups);
518         }
519
520         return @final_groups;
521     }
522
523     sub SecondaryGroupConstruct
524     {
525         my ($primary_group,$secondary_group,$Details_ref,$grp_attrs) = @_;
526
527         if ($secondary_group !~ /%([^%]+)%/) { # Group name doesn't contain any
attributes
528         my @temp_group;
529         $temp_group[0] = $secondary_group;
530         return (@temp_group);
531         }
532
533         my $attr = $1;      # We get the attribute name from the previous match in the if
statement
534         if ($attr !~ /^CV\./i && $attr !~ /^MV\./i) {
535         $attr = 'MV.'.$attr;
536         }
537
538         my @groups;
539
540         if (exists $$grp_attrs{$primary_group}{$attr}) {
541         my $tempattr = $attr;
542         $tempattr =~ s/^[MC]V\.//;
543         $secondary_group =~
s/%[MC]?V?\.?$tempattr%/$$grp_attrs{$primary_group}{$attr}/i;
544         push(@groups,$secondary_group);
545         } else {
546         foreach $val (split(/\t/,$$Details_ref{$attr})) {
547             my $tempgroup = $secondary_group;
548
549             my $tempattr = $attr;
550             $tempattr =~ s/^[MC]V\.//;
551             $tempgroup =~ s/%[MC]?V?\.?$tempattr%/$val/i;
552             push(@groups,$tempgroup);
553         }
554         }
555
556         my @final_groups;
557
558         foreach $group (@groups) {
559         my @tempgroups =
SecondaryGroupConstruct($primary_group,$group,$Details_ref,$grp_attrs);      # me so
clever
560         push(@final_groups,@tempgroups);
561         }
562
563         return (@final_groups);
564     }
565
566
567
568     sub DeleteUserFromGroup
569     {
570         my ($group,$Setting_ref,$userdn,$cv) = @_;
571         my $ldap = B2EGenLDAP::GetLDAP($cv);
572
573
574
575         my $filter = "(cn=$group)";
576         my $scope = "sub";
577         my $basedn = $$Setting_ref{"$cv.DEFAULT_BASE"};
578
579         my $res = $ldap->search(base => $basedn, scope => $scope,filter => $filter);
580         if (&ldap_connect_error($res->code)) { # Will return false if there's a problem
with the ldap handle
581         $ldap = B2EGenLDAP::RefreshLDAP($cv);
582         $res = $ldap->search(base => $basedn, scope => $scope,filter => $filter);
583         }
584
585         if ($res->is_error) {
586         $mesg = ldap_error_name($res->code).": ".ldap_error_name($res->code)."-
".ldap_error_text($res->code)."\t\n";
587
```

```
588         ProgError(3,"Search $filter to delete $userdn from -- not removing user from
$group: $mesg\n");
589         return();
590         }
591
592         my $count = $res->count;
593         if ($count != 1) {
594         ProgError(3,"Got back $count results for $filter.  Not removing user from
$group\n");
595         return;
596         }
597
598         my $entry = $res->pop_entry;
599         my $group_dn = $entry->dn;
600
601         $res = $ldap->modify($group_dn, delete => {'member' => $userdn});
602         if (&ldap_connect_error($res->code)) { # Will return false if there's a problem
with the ldap handle
603         $ldap = B2EGenLDAP::RefreshLDAP($cv);
604         $res = $ldap->modify($group_dn, delete => {'member' => $userdn});
605         }
606
607         if ($res->is_error) {
608         $mesg = ldap_error_name($res->code).": ".ldap_error_name($res->code)."-
".ldap_error_text($res->code)."\t\n";
609
610         ProgError(3,"Delete call $filter to remove $userdn from -- not removing user
from $group: $mesg\n");
611         return();
612         }
613
614         return(1);
615
616   }
617
618   sub Luser2Group
619   {
620         my
($userdn,$group,$constructed_primary_info_ref,$constructed_secondary_info_ref,
621         $nesting_relationships_ref,$setting_ref,$cv,$Details_ref,$CVNames_ref,$maxsize,$
proposed_groups_ref) = @_;
622
623         my $ldap = B2EGenLDAP::GetLDAP($$CVNames_ref{$$Details_ref{'CV.CV'}});
624
625
626         # Now that we're adding, we'll check to see if the group is numeric.  If it is,
we'll
627         # figure out which number should be added to the group name
628
629         my $groupnum = "";
630         if ($$constructed_primary_info_ref{$group}{'NUMERIC'} =~ /^Y/i) {
631         $groupnum =
DetermineGroupNum($group,$maxsize,$cv,0,$setting_ref,$$constructed_primary_info_ref{$grou
p}{'BASE'});
632         if ($groupnum eq "FAIL") {
633             return "fail";
634         }
635         }
636         my $newgroup = $group;
637         $newgroup =~ s/(_[^_]*)$/$groupnum$1/;
638
639         if ($newgroup ne $group) {
640         $$proposed_groups_ref{$newgroup} = 1;
641         delete $$proposed_groups_ref{$group};
642         }
643
644         $newgroup =~ s/.*\\//;        # Get rid of cv if groupname is something like
opr\P_StupidGroup_G
645         my $group_dn =
"cn=".$newgroup.",".$$constructed_primary_info_ref{$group}{'BASE'};
646
647         my $res = $ldap->modify($group_dn, add => {'member' => $userdn});
648         if (&ldap_connect_error($res->code)) { # Will return false if there's a problem
with the ldap handle
649         $ldap = B2EGenLDAP::RefreshLDAP($cv);
650         $res = $ldap->modify($group_dn, add => {'member' => $userdn});
```

```
651         }
652
653         if ( ($res->code && $res->code != 68) || $$Details_ref{'MV.ACCOUNTLIST'} =~
/211[23]/) {  # 68 is LDAP_ALREADY_EXISTS == someone else has added the user already so
we're good
654         $mesg = ldap_error_name($res->code).": ".ldap_error_name($res->code)."-
".ldap_error_text($res->code)."\t\n";
655         my $type = $$constructed_primary_info_ref{$group}{'TYPE'};
656
657         # If we build the global, we should go ahead and nest/create the locals
658         if (BuildADGroup($cv,$group_dn,$type,$setting_ref) =~ /CREATED/i ||
$$Details_ref{'MV.ACCOUNTLIST'} =~ /211[23]/) {
659             # Cycle through each nested group associated with the primary group
660             foreach $nested (keys %{$$nesting_relationships_ref{$group}})
661             {
662                 my $nested_cv = $cv;
663                 if ($nested =~ /(.+)\\/) {   # group names has a back slash ==>  cv-
name\groupname
664                     $nested_cv = $1;
665                 }
666                 my $nested_ldap = B2EGenLDAP::GetLDAP($nested_cv);
667                 my $type = $$constructed_secondary_info_ref{$nested}{'TYPE'};
668                 my $local_group_dn = "cn=".$nested;
669                 $local_group_dn = $local_group_dn.",";
670                 $local_group_dn =
$local_group_dn.$$constructed_secondary_info_ref{$nested}{'BASE'};
671                 my $local_group_add_dn = $local_group_dn;
672                 $local_group_add_dn =~ s/.*\\//;
673                 if ($local_group_add_dn !~ /^cn=/i) {
674                     $local_group_add_dn = "cn=".$local_group_add_dn;
675                 }
676                 $res = $nested_ldap->modify($local_group_add_dn, add => {'member' =>
$group_dn });       # Add primary to nested group
677                 if ($res->code && $res->code != 68) {  # 68 is LDAP_ALREADY_EXISTS ==
someone else has added the user already so we're good
678                     # If we can't add, maybe the group doesn't exist so let's try to
create it
679                     my $results =
BuildADGroup($nested_cv,$local_group_add_dn,$type,$setting_ref);
680                     if ($results =~ /CREATED/i) {
681
682                         # If we created the secondary (nested) group, let's go ahead
and try to add
683                         # the primary group to it again.
684                         $res = $nested_ldap->modify($local_group_add_dn, add =>
{'member' => $group_dn});
685                         if (&ldap_connect_error($res->code)) { # Will return false if
there's a problem with the ldap handle
686                             $nested_ldap = B2EGenLDAP::RefreshLDAP($nested_cv);
687                             $res = $nested_ldap->modify($local_group_add_dn, add =>
{'member' => $group_dn});
688                         }
689                         if ($res->code && $res->code != 68) {
690                             ProgError(3,"Can't nest $group_dn into
$local_group_add_dn");
691                         }
692                     } elsif ($results =~ /EXISTS/i && $nested_cv ne $cv) {
693                         ProgError(3,"Probably domain replication issue -- creating
batch\n");
694                         my $dbfile = $cv."2".$nested_cv.".nested.db";
695                         my %db;
696                         dbmopen (%db,"e:\\data\\cp\\scripts\\$dbfile",0666);
697                         $db{$group_dn} = $local_group_add_dn;
698                         dbmclose %db;
699                     } else {
700                         ProgError(3,"Can't create nested group $local_group_add_dn");
701                     }
702                 }
703             }
704         }
705
706         my $res = $ldap->modify($group_dn, add => {'member' => $userdn});
```

```
707
708         if (&ldap_connect_error($res->code)) { # Will return false if there's a problem
with the ldap handle
709             $ldap = B2EGenLDAP::RefreshLDAP($cv);
710             $res = $ldap->modify($group_dn, add => {'member' => $userdn});
711         }
712
713         if ($res->code && $res->code != 68) {  # 68 is LDAP_ALREADY_EXISTS == someone
else has added the user already so we're good
714             $mesg = ldap_error_name($res->code).": ".ldap_error_name($res->code)."-
".ldap_error_text($res->code)."\t\n";
715             ProgError(3,"Can't put user $userdn into $group_dn -- continuing processing:
$mesg\n");
716             return("fail");
717         }
718         }
719         return('YEP');
720     }
721
722     sub DetermineGroupNum
723     {
724         my ($group,$maxsize,$cv,$num,$Setting_ref,$basedn) = @_;
725
726         my $testgroup = $group;
727         $num++;
728         $testgroup =~ s/(.*)(_[^_]*)$/$1$num$2/;
729
730         if ($num > 150) {  # Depending on how many users you have and your max group
size setting, you may need
731                            # need to change this number.  This is just here to keep us
out of an infinite loop
732                            # It should be safe to remove this check, but it doesn't hurt
to have it.
733         ProgError('3',"Numeric Group Max Limit Reached: 150 for $group\n");
734         return("FAIL");
735         }
736
737         my $ldap = B2EGenLDAP::GetLDAP($cv);
738
739         my $filter = "(cn=$testgroup)";
740         my $scope = "sub";
741
742         my $res = $ldap->search(base => $basedn, scope => $scope,filter => $filter);
743         if (&ldap_connect_error($res->code)) { # Will return false if there's a problem
with the ldap handle
744         $ldap = B2EGenLDAP::RefreshLDAP($cv);
745         $res = $ldap->search(base => $basedn, scope => $scope,filter => $filter);
746         }
747         if ($res->is_error) {
748         $mesg = ldap_error_name($res->code).": ".ldap_error_name($res->code)."-
".ldap_error_text($res->code)."\t\n";
749
750         ProgError(3,"Search $filter to add user to numeric $testgroup -- $mesg\n");
751         return('FAIL');
752         }
753
754         my $count = $res->count;
755
756         if ($count == 0) {
757         return($num);
758         }
759
760         my $entry = $res->pop_entry;
761         my @temp = $entry->get_value('member');
762         my $size = @temp;
763         if ($size > $maxsize) {
764         $num = DetermineGroupNum($group,$maxsize,$cv,$num,$Setting_ref,$basedn);
765         }
766         return($num);
767
768     }
769     sub ProgError
770     {
771
772         my ($level,$msg) = @_;
773         chomp($msg);
```

```
774       my $time = &GetDate;
775       $time =~ /([^\s]*)\s+(.*)/;
776       $date = $1;
777
778
779       if ($level == 5) {
780        my $file = ">>MDS-JEGRP-INFO-$date.log";
781        open(PROGERR,$file);
782        my $fh = select(PROGERR);
783        $| = 1;
784        select($fh);
785        print PROGERR "$time:   Group: $msg\n";
786        close(PROGERR);
787       } else {
788        my $file = ">>MDS-JEGRP-ERR-$date.log";
789        open(PROGERR,$file);
790        my $fh = select(PROGERR);
791        $| = 1;
792        select($fh);
793        print PROGERR "$time: Level: $level  Error: $msg\n";
794        close(PROGERR);
795       }
796       if ($level == 2) {
797        #print OUT "Level 2 -- die now\n";
798       }
799       return;
800   }
801
802
803   sub GetDate
804   {
805       #If you're not putting this in MetaConnect log folder or if you are not using
MetaConnect
806       #you may want to change the gmtime call to localtime
807       my @localtime=gmtime(time()); # Get time and don't convert it from GMT but still
call it localtime to be confusing
808       $localtime[5] += 1900;       # Add 1900 to year.
809       $localtime[4]++;
810       if ($localtime[4] < 10) {
811           $localtime[4] = "0".$localtime[4];
812       }
813       if ($localtime[3] < 10) {
814           $localtime[3] = "0".$localtime[3];
815       }
816       if ($localtime[2] < 10) {
817           $localtime[2] = "0".$localtime[2];
818       }
819       if ($localtime[1] < 10) {
820           $localtime[1] = "0".$localtime[1];
821       }
822       if ($localtime[0] < 10) {
823           $localtime[0] = "0".$localtime[0];
824       }
825       my $date = $localtime[5].$localtime[4].$localtime[3]."
".$localtime[2].":".$localtime[1].":".$localtime[0];
826       return($date);
827   }
828
829   sub ldap_connect_error
830   {
831       my $code = $_[0];   # ldap return code
832       my @connect_errors = (0x30,0x31,0x32,0x33,0x34,0x35,0x36,0x50,0x51,
833                             0x52,0x55,0x58,0x59,0x5a,0x5b,0x5c,0x5d);
834       my $errflag = 0;
835       foreach $err (@connect_errors) {
836        if ($code == $err) {
837            $errflag = 1;
838        }
839       }
840       return $errflag;
841       # Probably don't need most of these, but hey, can't hurt
842       # LDAP_TIMELIMIT_EXCEEDED (0x03)
843       # LDAP_ADMIN_LIMIT_EXCEEDED { 0x0b } # V3
```

```
844         # LDAP_UNAVAILABLE_CRITICAL_EXT 0x0c
845         #sub LDAP_INAPPROPRIATE_AUTH        () { 0x30 }
846         #sub LDAP_INVALID_CREDENTIALS       () { 0x31 }
847         #sub LDAP_INSUFFICIENT_ACCESS       () { 0x32 }
848         #sub LDAP_BUSY                      () { 0x33 }
849         #sub LDAP_UNAVAILABLE               () { 0x34 }
850         #sub LDAP_UNWILLING_TO_PERFORM      () { 0x35 }
851         #sub LDAP_LOOP_DETECT               () { 0x36 }
852         #sub LDAP_OTHER                     () { 0x50 }
853         #sub LDAP_SERVER_DOWN               () { 0x51 }
854         #sub LDAP_LOCAL_ERROR               () { 0x52 }
855         #sub LDAP_TIMEOUT                   () { 0x55 }
856         #sub LDAP_USER_CANCELED             () { 0x58 }
857         #sub LDAP_PARAM_ERROR               () { 0x59 }
858         #sub LDAP_NO_MEMORY                 () { 0x5a }
859         #sub LDAP_CONNECT_ERROR                () { 0x5b }
860         #sub LDAP_NOT_SUPPORTED                () { 0x5c }
861         #sub LDAP_CONTROL_NOT_FOUND     () {   0x5d }
862     }
863
864
865
866
867     sub BuildADGroup
868     {
869         my ($cv,$group,$grouptype,$setting_ref) = @_;
870
871         if ($grouptype =~ /^G/i) {              # AD Global Security group
872          $grouptype = '-2147483646';
873         } elsif ($grouptype =~ /^L/i) {         # AD Local Security group
874          $grouptype = '-2147483644';
875         } elsif ($grouptype =~ /^U/i) {         # AD Universal Security group
876          $grouptype = '-2147483640';
877         } elsif ($grouptype =~ /^MG/i) {        # AD Global Distribution group
878          $grouptype = '-4294967294';
879         } elsif ($grouptype =~ /^ML/i) {        # AD Local Distribution group
880          $grouptype = '-4294967292';
881         } elsif ($grouptype =~ /^MU/i) {        # AD Universal Distribution group
882          $grouptype = '-4294967288';
883         } else {
884          ProgError(3,"Invalid grouptype $grouptype specified for $group");
885          return();
886         }
887
888         $object = B2EGenLDAP::GetObj($cv);
889
890         my $base = $$setting_ref{"$cv.DEFAULT_BASE"};
891         $group =~ s/,$base//i;
892         if ($group !~ /^cn=/i) {
893          $group = "cn=".$group;
894         }
895
896         my $groupobject = $object->Create("Group",$group);
897         if (Win32::OLE->LastError()) {
898          $error = Win32::OLE->LastError();
899          ProgError(3,"Can't create Win32::OLE group object to create group $group --
continuing run: $error");
900          return();
901         }
902
903         $groupobject->Put("Description","Metaconnect Managed Group");
904         if (Win32::OLE->LastError()) {
905          $error = Win32::OLE->LastError();
906          ProgError(3,"Can't put description into Win32::OLE group object to create group
$group -- continuing run: $error");
907          return();
908         }
909
910         my $cn = $group;
911         $cn =~ s/^CN=//i;
912         $cn =~ s/,.*$//;
913         $groupobject->Put("cn",$cn);
914         if (Win32::OLE->LastError()) {
915          $error = Win32::OLE->LastError();
```

```
916         ProgError(3,"Can't put CN into Win32::OLE group object to create group $group --
continuing run: $error");
917         return();
918         }
919
920         $groupobject->Put("sAMAccountName",$cn);
921         if (Win32::OLE->LastError()) {
922         $error = Win32::OLE->LastError();
923         ProgError(3,"Can't put sAMAccountName into Win32::OLE group object to create
group $group -- continuing run: $error");
924         return();
925         }
926
927         $groupobject->Put("groupType",$grouptype);
928         if (Win32::OLE->LastError()) {
929         $error = Win32::OLE->LastError();
930         ProgError(3,"Can't put groupType into Win32::OLE group object to create group
$group -- continuing run: $error");
931         return();
932         }
933
934         $groupobject->SetInfo();
935         if(Win32::OLE->LastError()) {
936         $error = Win32::OLE->LastError();
937         if ($error =~ /The object already exists/i) {
938             return "EXISTS";
939         }
940         ProgError(3,"Can't SetInfo on Win32::OLE group object to create group $group --
continuing run: $error");
941         return();
942         }
943
944         undef $groupobject;
945         return "CREATED";
946     }
947     1;
```

## Appendix B: Man-jegrouplist2.pl

```
# ma-jegrouplist2.pl
# V 1.1
# Created by Don Quigley
# quigley@techie.com
# 4/1/2003

package SF2KGROUP;

use Net::LDAP;
use Net::LDAP::Util qw( ldap_error_name
                        ldap_error_text) ;
use Carp;
use Win32::OLE;
require "win2kgroups.pl";

open(IN,"b2e_config.txt");
while (<IN>) {
      if (/^\[/ or /^\s*$/) {next;}          # Let's ignore blank lines and header
lines (lines with [ text ]
      chomp;                                 # The politically correct implentation of
chop chop
      s/\s*#.*$//;                           # Get rid of whitespaces in front of #
(beginning of comments)
      my @line = split(/\s*=\s*/,$_,2);      # heh.
      $line[0] =~ tr/a-z/A-Z/;
      $setting{$line[0]} = $line[1];
}
close(IN);

print "Enter search filter to find users to process:\n";
$filter = <STDIN>;
chomp $filter;

print "Valid CV's are: ";
foreach $key (sort keys %setting) {
    if ($key =~ /DESCRIPTION/i) {
     $view = $key;
     $view =~ s/.DESCRIPTION//i;
     print "$view: $setting{$key}\n";
     }
}
print "Enter the name of the CV you wish to populate: ";
my $cv = <STDIN>;
chomp $cv;
$cv = uc $cv;
print "\n\n";
print "Server: $setting{$cv.'.WIN2KDC'}\n";
print "BaseDN: $setting{$cv.'.DEFAULT_BASE'}\n";
print "Is this correct? [yN]: \n";
$temp = <STDIN>;
if ($temp !~ /^\s*y/i) {die;}


my ($server,$username,$password,$ldap,$basedn,$userbasedn);

# GET INFO FOR DESTINATION SERVER
$server = $setting{$cv.'.WIN2KDC'};
$username = $setting{$cv.'.USERNAME'};
$password = $setting{$cv.'.PASSWORD'};
$ldap = Net::LDAP->new($server,port => '389');
$basedn = uc($setting{$cv.'.DEFAULT_BASE'});
$userbasedn = uc($setting{$cv.'.DEFAULT_USER_BASE'});

# Decrypt password
# @pad is our 'one-time' pad that we use over and over.  Note: Passwords longer than
1000 characters will be problematic
my @pad = ();
my $i = 0;
my @c = split(/\s+/,$password);
my @d;
foreach $padval (@c) {
```

```
68        $padval = $padval - $pad[$i];
69        $d[$i] = $padval;
70        $i++;
71    }
72    $password = pack("C*",@d);
73
74    @pad = ();
75    $i = 0;
76    @c = split(/\s+/,$setting{'MV.PASSWORD'});
77    @d =();
78    foreach $padval (@c) {
79        $padval = $padval - $pad[$i];
80        $d[$i] = $padval;
81        $i++;
82    }
83    $setting{'MV.PASSWORD'} = pack("C*",@d);
84    # THIS IS THE HASH THAT WILL STORE GROUP NAMES AND MEMBERS
85    # The key will be the group name and the value will be an array reference.
86    # The array will contain a list of users.
87    my %prop_programmatic_groups = ();
88    my %nested_ldg = ();
89
90    my $ldpuser = $username;
91    $ldpuser =~ s/[^\\]*\\//;
92    $ldap = Net::LDAP->new($setting{'MV.WIN2KDC'},port => '389') || die "cannot mv";
93    $ldap->bind(dn => $setting{'MV.USERNAME'},password => $setting{'MV.PASSWORD'},version
=> '3') || die "Bind failed\nerk\n";
94
95    # NOW WE WANT TO SEARCH THROUGH AND FIND USER ENTRIES.
96
97    print "searching on $filter\n";
98    $searchobj = $ldap->search(scope => 'sub',filter => $filter,base => 'o=state
farm,c=us');
99
100   foreach $entry ($searchobj->entries) {
101       # Put all of the attributes into a hash.  We add MV. so it fits better with our
102       # existing criteria logic
103       my @attrs;
104       push(@attrs,"cv: $cv");
105       my $dn = $entry->dn;
106       print "$dn\n";
107       foreach $attr ($entry->attributes) {
108        my @temp = $entry->get_value($attr);
109        foreach $val (@temp) {
110            push(@attrs,"MV.$attr: $val");
111            my @jegrouplist = SF2KGROUP::sf2kGroup(@attrs);
112            $res = $ldap->modify($dn,replace => {'sfjegrouplist' => \@jegrouplist});
113
114            if ( $res->code && $res->code != 68) {  # 68 is LDAP_ALREADY_EXISTS ==
someone else has added the user already so we're good
115                $mesg = ldap_error_name($res->code).": ".ldap_error_name($res-
>code)."-".ldap_error_text($res->code)."\t\n";
116                print $mesg,"\n";
117            }
118       }
119       }
120   }
121
```

Appendix C: Ldap-conn.pl

This subroutine is used to make LDAP connections to all of the directories specified in the b2e_config.txt file. These connections will be persistent across calls to the subroutine so that if win2kgroups.pl is used as a constructed attribute within MetaConnect, there won't any performance degradation caused my multiple binds/unbinds.

```
# ldap-conn.pl
# V 1.1
# Created by Don Quigley
# quigley@techie.com
# 4/1/2003

package B2EGenLDAP;


BEGIN {     # Hopefully the begin will force this to run when metaconnect starts up


    use Net::LDAP;
    use Net::LDAP::Util qw( ldap_error_name
                            ldap_error_text) ;
    use Carp;
    use Win32::OLE;
    my %ldap = ();
    my %Setting = ();
    my %oOpenDSObject = ();
    my %CVNames = ();
    my %object = ();


    ##############################
    # READ IN CONFIGURATION FILE #
    ##############################
    # Open up our file that contains all of the configuration information we need.
    open(IN,"b2e_config.txt") || ProgError(4,"Can't open b2e_config.txt");
    # Read in the b2e_config.txt configuration settings
    while (<IN>) {
     if (/^\[/ or /^\s*$/) {next;}          # Let's ignore blank lines and header
lines (lines with [ text ]
     chomp;
     s/\s*#.*$//;                           # Get rid of whitespaces in front of #
(beginning of comments)
     my @line = split(/\s*=\s*/,$_,2);
     $line[0] = uc $line[0];
     if ($line[0] !~ /.+\.password/i) {
         $line[1] = uc $line[1];
     }
     $Setting{$line[0]} = $line[1];
     if ($line[0] =~ /(^\S+)\.cvname/i) {
         $CVNames{$line[1]} = $1;
     }
     }
    close(IN);


    my @pad = ();

    foreach $cv (values %CVNames) {

     Win32::OLE->Initialize(Win32::OLE::COINIT_MULTITHREADED);

     my $server = $Setting{"$cv.WIN2KDC"};
     $ldap{$cv}{'port'} = $Setting{"$cv.PORT"} || 389;
     $ldap{$cv}{'password'} = $Setting{"$cv.PASSWORD"};
     $ldap{$cv}{'username'} = $Setting{"$cv.USERNAME"};
     $ldap{$cv}{'timeout'} = $Setting{"$cv.TIMEOUT"} || 120;
     $ldap{$cv}{'time'} = 0; # used to be time
     $ldap{$cv}{'ldpuser'} = $ldap{$cv}{'username'};


```

```
64        # Decrypt password
65        # @pad is our 'one-time' pad that we use over and over.
66
67        # Note: The @pad array needs to be defined as a set of numbers between 1-999.
68        # The array needs to contain at least as many elements as there are characters
69        # in the longest password that will be used.
70
71        # Obviously, the more passwords you use this pad to "encrypt", the less secure
it
72        # will be.  Plus the pad that you use is stored in plain text in all of the
programs.
73        # At least it's better than being in cleartext in the config file.  The best
74        # solution is to tie the passwords to a hardware crypto device.
75
76
77        my $i = 0;
78        my @c = split(/\s+/,$ldap{$cv}{'password'});
79        my @d;
80        foreach $padval (@c) {
81            $padval = $padval - $pad[$i];
82            $d[$i] = $padval;
83            $i++;
84        }
85        $ldap{$cv}{'password'} = pack("C*",@d);
86        }
87
88        sub GetLDAP
89        {
90        my $cv = $_[0];
91
92        foreach $key (keys %CVNames) {
93            if ($key =~ /^$cv$/i) {
94                  $cv = uc $CVNames{$key};
95            }
96        }
97
98        $cv = uc $cv;
99
100       $a = time;
101       $b = $ldap{$cv}{'time'};
102       $c = $ldap{$cv}{'timeout'};
103       if ( (time - $ldap{$cv}{'time'}) > $ldap{$cv}{'timeout'} ) {
104         RefreshLDAP($cv);
105       }
106       return ($ldap{$cv}{'ldap'});
107       }
108
109       sub RefreshLDAP
110       {
111       my $cv = uc $_[0];
112
113       # If passed friendly name, need to convert to unfriendly name
114       foreach $key (keys %CVNames) {
115           if ($key =~ /^$cv$/i) {
116                 $cv = uc $CVNames{$key};
117           }
118       }
119
120       my $server = $Setting{"$cv.WIN2KDC"};
121
122       $error = 0;
123       if ($ldap{$cv}{'time'} != 0) {  # If 0, we haven't bound to this directory yet
124           if ($ldap{$cv}{'ldap'} -> unbind) {
125                 $ldap{$cv}{'time'} = 0;
126           }
127       }
128
129       $error = 0;
130       $ldap{$cv}{'ldap'} = new Net::LDAP($server,port => $ldap{$cv}{'port'}) or $error
= 1;
131       if ($error == 1) {
132           B2EGenLDAP::ProgError(3,"Can't connect to $server on cv $cv-- continuing");
133       }
134       $ldpuser = $ldap{$cv}{'username'};
135       $ldpuser =~ s/[^\\]*\\//;
```

```
136         $ldap{$cv}{'ldap'}->bind(dn => $ldpuser,password => $ldap{$cv}{'password'},
version => 3) or
137                                             B2EGenLDAP::ProgError(3,"Can't bind to
$server");
138
139         $ldap{$cv}{'time'} = time;
140
141         my $adspath = "LDAP://".$server;
142         $oOpenDSObject{$cv} = Win32::OLE->GetObject("LDAP:");
143         if (Win32::OLE->LastError()) {
144             $error = Win32::OLE->LastError();
145             B2EGenLDAP::ProgError(3,"Can't open OLE object for $cv on Refresh in ldap-
conn.pl -- continuing run: $error");
146         }
147
148         $object{$cv} = $oOpenDSObject{$cv}-
>OpenDSObject($adspath,$ldap{$cv}{'username'},$ldap{$cv}{'password'},'1');
149         if (Win32::OLE->LastError()) {
150             $error = Win32::OLE->LastError();
151             B2EGenLDAP::ProgError(3,"Can't open OLE object for $cv on Refresh in ldap-
conn.pl -- continuing run: $error");
152         }
153        }
154
155
156
157        sub GetObj
158        {
159        my $cv = $_[0];
160        foreach $key (keys %CVNames) {
161            if ($key =~ /^$cv$/i) {
162                  $cv = uc $CVNames{$key};
163            }
164        }
165
166        $a = time;
167        $b = $ldap{$cv}{'time'};
168        $c = $ldap{$cv}{'timeout'};
169        if ( (time - $ldap{$cv}{'time'}) > $ldap{$cv}{'timeout'} ) {
170          RefreshLDAP($cv);
171        }
172
173        return $object{$cv};
174        ##########################################
175        ##########################################
176        my $server = $Setting{"$cv.WIN2KDC"};
177        $oOpenDSObject{$cv} = Win32::OLE->GetObject("LDAP:");
178
179        my $adspath = "LDAP://".$server;
180        $object{$cv} = $oOpenDSObject{$cv}-
>OpenDSObject($adspath,$ldap{$cv}{'username'},$ldap{$cv}{'password'},'1');
181        if (Win32::OLE->LastError()) {
182            $error = Win32::OLE->LastError();
183            B2EGenLDAP::ProgError(3,"Can't open OLE object to $cv -- continuing run:
$error");
184        }
185        return $object{$cv};
186        }
187
188
189        sub ProgError
190        {
191        my ($level,$msg) = @_;
192        chomp($msg);
193        my $time = &GetDate;
194        $time =~ /([^\s]*)\s+(.*)/;
195        $date = $1;
196
197
198        if ($level == 5) {
199            my $file = ">>MDS-JEGRP-LDP-$date.log";
200            open(PROGERR,$file);
201            my $fh = select(PROGERR);
202            $| = 1;
203            select($fh);
```

```
204             print PROGERR "$time:   Group: $msg\n";
205             close(PROGERR);
206         } else {
207             my $file = ">>MDS-JEGRP-LDP-$date.log";
208             open(PROGERR,$file);
209             my $fh = select(PROGERR);
210             $| = 1;
211             select($fh);
212             print PROGERR "$time: Level: $level  Error: $msg\n";
213             close(PROGERR);
214         }
215         if ($level == 2) {
216             die;
217         }
218         return;
219         }
220
221         sub GetDate
222         {
223         my @localtime=gmtime(time()); # Get time and don't convert it from GMT but still
call it localtime to be confusing
224         $localtime[5] += 1900;       # Add 1900 to year.
225         $localtime[4]++;
226         if ($localtime[4] < 10) {
227             $localtime[4] = "0".$localtime[4];
228         }
229         if ($localtime[3] < 10) {
230             $localtime[3] = "0".$localtime[3];
231         }
232         if ($localtime[2] < 10) {
233             $localtime[2] = "0".$localtime[2];
234         }
235         if ($localtime[1] < 10) {
236             $localtime[1] = "0".$localtime[1];
237         }
238         if ($localtime[0] < 10) {
239             $localtime[0] = "0".$localtime[0];
240         }
241         my $date = $localtime[5].$localtime[4].$localtime[3]."
".$localtime[2].":".$localtime[1].":".$localtime[0];
242         return($date);
243         }
244     }
245     1;
246
247
```

Appendix D: Genpwd.pl

This program is used to create the "encrypted" passwords stored in the b2e_config.txt file. It is important that the @pad array is identical to the ones stored in ldap-conn.pl and man-jegrouplist2.pl.

```
# genpwd.pl
# V 1.1
# Created by Don Quigley
# quigley@techie.com
# 4/1/2003

# Used to generate the "encrypted" passwords stored in the b2e_config.txt file
# used by the Win2kGroups.pl program.

# Note: The @pad array needs to be defined as a set of numbers between 1-999.
# The array needs to contain at least as many elements as there are characters
# in the longest password that will be used.

# Obviously, the more passwords you use this pad to "encrypt", the less secure it
# will be.  Plus the pad that you use is stored in plain text in all of the
programs.
# At least it's better than being in cleartext in the config file.  The best
# solution is to tie the passwords to a hardware crypto device.

# Usage:  perl genpwd.pl "password_to_encrypt"

@pad = ();

my $a = $ARGV[0];
@b = unpack("C*",$a);
$i = 0;
my @c;
foreach $letterval (@b) {
    $letterval = $letterval + $pad[$i];
    $c[$i] = $letterval;
    $i++;
}
print join " ",@c;
```

Appendix E: Sample b2e_confix.txt File

```
# Location of required perl scripts and log files
group_criteria_directory = e:\groups\scripts
# Location of group criteria files
win2kgroup_directory = e:\groups\scripts
# Set max. size of numeric groups to 900
max_group_size = 900
# Server name of LDAP instance storing our join of user information
mv.win2kdc = bigdude.mycomp.com
# LDAP user to bind to directory as
mv.username = cn=manager
# "Encrypted" password for user to bind as
mv.password = 157 910 403 205 930 454 1047 386

# Information for our connector view
# This connector view is the employee domain in our forest
# Base dn for domain
employee.default_base = dc=employee,dc=mycomp,dc=com
# Default dn under which programmatic groups are created
employee.default_user_base = ou=prog,ou=groups,dc=employee,dc=mycomp,dc=com
# DC in employee domain to which all updates will be made
employee.win2kdc = superdude.employee.mycomp.com
employee.username = employee\MetaDirAcct
employee.password = 122 914 404 187 910 454 1043 396 415 199 354 562
employee.description = Production AD Connector to the Employee Domain
# Define the internal CV name used by MetaConnect
employee.cvname = emp
employee.fileextension = empgrp
# How many seconds we should use an LDAP handle to this CV before
# unbinding and binding again.  Any setting over 5 minutes or so should
# have a negligible impact to performance.
employee.timeout = 300

agent.default_base = dc=extagents,dc=mycomp,dc=com
agent.default_user_base = ou=prog,ou=groups,dc=extagents,dc=mycomp,dc=com
agent.win2kdc = coffeedude.extagents.mycomp.com
agent.username = extagents\MetaDirAcct
agent.password = 122 914 404 187 910 454 1043 396 415 199 354 563
agent.description = Production AD Connector to the External Agent Rep Domain
agent.cvname = ext
agent.fileextension = extgrp
agent.timeout = 300
```