PART II
Practical Applications

10
User Management

IN THIS SECTION, we take many of the abstract concepts we learned in the first part of the book and begin to apply them to real-world problems. One of the most common programming activities that most developers will perform against Active Directory and ADAM is user account management.

Active Directory user management is fairly straightforward once we know all of the basic rules. However, there are many interesting rules and behaviors concerning how attribute values in Active Directory are used to determine how user objects behave in Windows. In order to get the results we want, it is important to learn these rules and behaviors. We try to demonstrate most of them in this chapter.

To a lesser extent, we also investigate user account management in ADAM. ADAM is very similar to Active Directory in most respects, but it has a few significant differences worth pointing out.

Finding Users

When we speak of user objects for the remainder of this chapter, we are really talking about the `user` class in both Active Directory and ADAM. This is the class that acts as a security principal for Active Directory (and often, for ADAM). It is typically the class that we care the most about. The first thing we are likely to want to do with user accounts in Active Directory is find them. Given all that we know about searching from Chapters 4 and 5, this should be easy. Essentially, we just need to know where we want to search and what filter to use to find users.

Let's start with the LDAP filter. Our goal is to build an LDAP filter that will find exactly what we want and that will be as efficient as possible. A few attributes in Active Directory distinguish `user` objects from other object types that we can use to build a filter:

- `objectCategory`
- `objectClass`
- `sAMAccountName`
- `sAMAccountType`

The `objectCategory` attribute has the advantage of being single-valued and indexed by default on all versions of Active Directory. This attribute is meant to be used to group common types of objects together so that we can search across all of them. Many of the schema classes related to users share the same value of `person` as their object category. While this is useful for searches in which we want to find information across many different types of user-related objects, it is not as useful for finding the `user` objects we typically care about (which are usually security principals). For example, in Active Directory, since both `user` and `contact` classes share the same `objectCategory` value of `person`, it alone will not tell them apart.

The `objectClass` attribute seems like a no-brainer, as `(objectClass=user)` will always find `user` objects exclusively. The problem here is that in many forests, the `objectClass` attribute is not indexed and it is always multivalued. As we know from the section titled Optimizing Search Performance, in Chapter 5, we generally want to try to avoid searches on nonindexed attributes, so using `objectClass` alone in a filter might not be the most efficient solution. This is why we see a lot of samples that search for `user` objects using an indexed filter like this:

```
(&(objectCategory=person)(objectClass=user))
```

This will get the job done, but we can do even better than this.

NOTE Behavior Change in Windows Server 2003

Windows Server 2003 Active Directory indexes the `objectClass` attribute by default, so this rule really only applies to Windows 2000 Active Directory now. Additionally, the schema administrators for your domain may have already indexed it, so check the schema before making assumptions.

One key difference between `contact` objects and `user` objects in Active Directory is that `user` objects have a `sAMAccountName` attribute that is indexed by default. Thus, we could build a filter like this:

```
(&(objectCategory=person)(sAMAccountName=*))
```

This will separate the contacts from the users effectively. However, another approach is available that can find `user` objects directly, and it may be the most efficient technique for Active Directory:

```
(sAMAccountType=805306368)
```

Using the `sAMAccountType` attribute with a value of `805306368` accesses a single-valued, indexed attribute that uniquely defines `user` objects. The only downside here is that this attribute is not well documented, so it may not be recommended by Microsoft. However, in our investigations, it is effective.

One piece of good news is that all of these attributes are included in the global catalog, so we can use all of these filters there as well.

This should provide a foundation to build on for various `user` object searches. We may also wish to combine these filters with other attributes to find different subsets of `user` objects matching specific criteria. We will see some examples of that in the rest of this section.

Finding Users in ADAM

Things change a little bit for ADAM, because we don't have things like `sAMAccountName` or `sAMAccountType` on an ADAM user. These attributes are actually from the auxiliary class called `securityPrincipal`. This class happens to be slightly different depending on whether it comes from

Active Directory or ADAM. In this case, the securityPrincipal class does not contain sAMAccountName or related attributes in ADAM.

This means that we need to adjust our filters slightly to find our users in ADAM. It turns out that we can use some other indexed attributes, such as userPrincipalName, to find our users. However, since this attribute is not required, we have to be careful and ensure that all of our user objects have set a value for userPrincipalName.

As mentioned in Chapter 8, ADAM also includes the idea of a userProxy object that we often will want to return as well. To do so, we should look for common attributes found on both classes that would filter them for our needs. In this case, we know that both the user and userProxy classes contain userPrincipalName, so it is a good candidate for this purpose. We could use a simple filter such as (userPrincipalName=*) if we knew that all of our objects would have a value set for this attribute. However, since not all user or userProxy objects might have a value set for userPrincipalName, we would have to use objectCategory to be safe. For example:

```
(|(objectCategory=person)(objectCategory=userProxy))
```

The implications here are that if any other class in the directory also shared the same objectCategory of person or userProxy, they would also be returned. In default ADAM instances, this does not occur, but we should always check with our particular ADAM instance. Now, if we want to separate the two classes and treat them differently, we can use objectClass, as the classes are different for each type:

```
(&(objectClass=user)(objectCategory=person))

(&(objectClass=userProxy)(objectCategory=userProxy))
```

The first filter will return only user objects using objectCategory as an index, and the second will return only userProxy objects in the same manner. It is important to remember that we can customize ADAM heavily with application-specific classes. We should never rely on the fact that any particular class will be there or even be indexed. As such, these are only guidelines, and we really need to check the particular ADAM instance's schema ourselves to determine what is there.

NOTE New Versions of ADAM Include the userProxyFull Class

With later versions of ADAM, a new class called `userProxyFull` can be found that essentially mirrors the `user` class with the addition of the `msDS-BindProxy` auxiliary class. This means that we cannot use `objectCategory` to distinguish between `user` and `userProxyFull` classes as we can with `userProxy` classes. We must instead rely upon the `objectClass` to distinguish between the two. This is another example of where we need to be cognizant of what our schema is for ADAM.

Creating Users

Creating users in Active Directory is actually just like creating any other object in the directory. We simply create an instance of the class we want and set any required attributes on the object. However, to get a fully functional user object that is "enabled," has a password, and has other useful attributes takes a bit more work.

But first, the basics! Let's create a `user` object in an organizational unit called `people` that exists in the root of our domain (see Listing 10.1).

LISTING 10.1: Creating an Active Directory or ADAM User

```
DirectoryEntry parent = new DirectoryEntry(
    "LDAP://OU=people,DC=mydomain,DC=com",
    null,
    null,
    AuthenticationTypes.Secure
    );

DirectoryEntry user =
    parent.Children.Add("CN=test.user", "user");

using (user)
{
    //sAMAccountName is required for W2k AD, we would not use
    //this for ADAM, however.
    user.Properties["sAMAccountName"].Value = "test.user";

    //userPrincipalName is not required, but recommended
    //for ADAM. AD also contains this, so we can use it.
    user.Properties["userPrincipalName"].Value = "test.user";
    user.CommitChanges();
}
```

As Listing 10.1 demonstrates, creating a user is pretty simple. We just need to know the distinguished name (DN) of the object that will serve as the user's parent container. This is typically an organizational unit, but it could be a container or a lot of different things in ADAM. We should also note that ADAM is slightly different from Active Directory in that `sAMAccountName` does not exist in ADAM, so there are actually no required attributes! Instead, we chose to populate `userPrincipalName` (which is a good idea in both Active Directory and ADAM). The `userPrincipalName` attribute is one of the accepted names used for binding to the directory.

If we remember from Chapter 3, we know that creating an object requires supplying a relative distinguished name (RDN) value. The prefix that we should use is defined in the schema by the `rDNAttId` attribute on the class. Active Directory and ADAM tend to use the `CN` RDN prefix for a number of object types; however, it is not uncommon for third-party LDAP directories from other vendors to use something different (e.g., `UID`). The RDN prefix (e.g., `CN`, `OU`, `O`, etc.) will differ slightly by object, but for `user` objects, we should generally choose the `CN` prefix. Essentially, we must always supply the name using the correct RDN attribute name for the class being created.

In Listing 10.1, we set the `sAMAccountName` attribute. This is not strictly required in the Windows 2003 version of Active Directory, but it is in the Windows 2000 version. Windows 2003 Active Directory will pick a random name for us, though, so it is probably a good idea to set one intentionally, especially since this is the user's login name in the Windows NT format (`domain\login`).

While the code in Listing 10.1 will create a `user` object in Active Directory (and with slight modification, in ADAM) it will not be very useful. The user account will not have a password, it will contain very few attributes, and it will not even be enabled by default. While some of these attributes are simple strings that do not require anything particularly complicated to set, passwords and account properties are more complicated. The sections that follow detail how to deal with them and make user accounts behave like the user accounts we have come to expect.

Managing User Account Features

Creating `user` objects in Active Directory is not difficult. As we have seen, with the right permissions, we can create a basic user account in just a few lines of code.

However, getting `user` objects to behave like Windows user accounts is a bit more challenging. Windows accounts have many features, such as enabled/disabled status, names and identifications used for security and email, password management, and expiration and lockout status, all of which require more-intimate knowledge of how things work under the hood. The next several sections explore this in detail.

Managing Basic User Account Properties in Active Directory

Many of the important behaviors associated with a Windows account in Active Directory, such as enabled/disabled status, are controlled by an attribute called `userAccountControl`. This attribute contains a 32-bit integer that represents a bitwise enumeration of various flags that control account behavior.

These flags are represented in ADSI by an enumerated constant called `ADS_USER_FLAG`. Because this enumeration is so important in terms of working with user objects in `System.DirectoryServices` (SDS), we will convert the ADSI enumeration into a .NET-style enumeration, as shown in Listing 10.2.

LISTING 10.2: User Account Control Flags

```
[Flags]
public enum AdsUserFlags
{
    Script = 1,                                      // 0x1
    AccountDisabled = 2,                             // 0x2
    HomeDirectoryRequired = 8,                       // 0x8
    AccountLockedOut = 16,                           // 0x10
    PasswordNotRequired = 32,                        // 0x20
    PasswordCannotChange = 64,                       // 0x40
    EncryptedTextPasswordAllowed = 128,              // 0x80
    TempDuplicateAccount = 256,                      // 0x100
    NormalAccount = 512,                             // 0x200
    InterDomainTrustAccount = 2048,                  // 0x800
    WorkstationTrustAccount = 4096,                  // 0x1000
    ServerTrustAccount = 8192,                       // 0x2000
    PasswordDoesNotExpire = 65536,                   // 0x10000
```

```
        MnsLogonAccount = 131072,                        // 0x20000
        SmartCardRequired = 262144,                      // 0x40000
        TrustedForDelegation = 524288,                   // 0x80000
        AccountNotDelegated = 1048576,                   // 0x100000
        UseDesKeyOnly= 2097152,                          // 0x200000
        DontRequirePreauth= 4194304,                     // 0x400000
        PasswordExpired = 8388608,                       // 0x800000
        TrustedToAuthenticateForDelegation = 16777216,  // 0x1000000
        NoAuthDataRequired = 33554432                    // 0x2000000
    }
```

As we look through the members of this enumeration, we see a variety of words we associate with Windows accounts, such as `AccountDisabled` and `PasswordNotRequired` (the last one we hope you never use!). We also see some flags that we probably do not recognize, such as `MnsLogonAccount` and `UseDesKeyOnly`. For the most part, the esoteric flags are not important in daily account management tasks, so we can ignore them. Chances are, if we need these flags we are probably quite aware of them already.

The important thing to note is that even though 21 flags are currently defined for use with the `userAccountControl` attribute, Active Directory does not actually use all of them! Specifically, the ones that are not meaningful to Active Directory are

- `AccountLockedOut`
- `PasswordCannotChange`
- `PasswordExpired`

Active Directory actually uses different mechanisms to control these account properties, so do not try to read them from `userAccountControl`! We discuss how to deal with the special cases in the upcoming sections.

Reading User Account Properties

Reading the `userAccountControl` attribute is simple. Listing 10.3 uses an enumeration we defined in Listing 10.2.

Listing 10.3: Reading the userAccountControl Attribute

```
DirectoryEntry user = new DirectoryEntry(
    "LDAP://CN=User1,CN=users,DC=domain,DC=com",
    null,
           null,
           AuthenticationTypes.Secure
           );

       AdsUserFlags userFlags = (AdsUserFlags)
           user.Properties["userAccountControl"].Value;

       Console.WriteLine(
           "AdsUserFlags for {0}: {1}",
           user.Path,
           userFlags
           );
```

This will generally write `NormalAccount` to the console for a typical user and may include other flags as well, depending on our specific deployment. If the account is disabled, `AccountDisabled` will be displayed in addition.

For Windows Server 2003 Active Directory, we could also use the `msDS-User-Account-Control-Computed` attribute in place of `userAccountControl`. Since it is constructed, we would need to use our `RefreshCache` technique from Chapter 3. However, the one benefit of using this attribute is that the three flags we previously mentioned as not being used with `userAccountControl` would actually be used and be accurate. This is not an option with Windows 2000 Server Active Directory installations—they will always need to use `userAccountControl` for reading user account properties.

Writing User Account Properties

Writing values is equally as easy as reading them. We just create an integer value representing the proper combination of flags and overwrite the existing `userAccountControl` value, as shown in Listing 10.4.

LISTING 10.4: Writing Account Values

```
    DirectoryEntry entry = new DirectoryEntry(
        "LDAP://CN=some user,CN=users,DC=mydomain,DC=com",
        null,
        null,
        AuthenticationTypes.Secure
        );

    AdsUserFlags newValue = AdsUserFlags.NormalAccount
```

```
        | AdsUserFlags.DontExpirePassword;

    entry.Properties["userAccountControl"].Value = newValue;
    entry.CommitChanges()
```

The trick here is that we must use valid combinations of flag values. In addition, other aspects of the account or the policies in effect may prevent certain values from being set.

A classic example that can trip up new developers happens when enabling an account by "unsetting" the `AccountDisabled` flag. In many domains, a minimum password length is required for all user accounts (and hopefully this is what you use as well). An account cannot be enabled unless it has a password. Therefore, we must set a valid password before enabling the account.

As a result, many typical provisioning processes that create accounts follow this protocol.

1. Create the account with initial values and commit changes.
2. Use the `SetPassword` operation to set an initial password.
3. Enable the account and commit changes again.

Keep this in mind when creating and provisioning user accounts if errors occur.

Delegation Settings

Three flags in the enumeration relate to delegation, which we discussed in Chapter 8. Specifically, `TrustedForDelegation` and `TrustedToAuthenticateForDelegation` are used by service accounts that will be allowed to delegate users' credentials to other machines. The difference between them is that `TrustedForDelegation` is the "unconstrained" delegation setting that works with Windows 2000 Server, and it represents the flag used when only Kerberos authentication is allowed in constrained delegation. `TrustedToAuthenticateForDelegation` is new with Windows Server 2003 and is used when delegation from any protocol is allowed. This is known as the "Protocol Transition" setting.

Finally, `AccountNotDelegated` is used to flag an account as “sensitive and cannot be delegated.” This is typically used on highly privileged accounts such as those used by directory administrators, where we would probably not want their account to be delegated by another service due to the security risk it poses.

Managing Basic User Account Properties in ADAM

ADAM works differently than Active Directory in that it does not rely on the `userAccountControl` attribute to maintain important account properties. Instead, Microsoft introduced a number of attributes prefixed with `ms-DS-` or `msDS` to hold this information:

- `ms-DS-UserAccountAutoLocked`*
- `msDS-User-Account-Control-Computed`*
- `msDS-UserAccountDisabled`
- `msDS-UserDontExpirePassword`
- `ms-DS-UserEncryptedTextPasswordAllowed`
- `msDS-UserPasswordExpired`*
- `ms-DS-UserPasswordNotRequired`

The * in the preceding list indicates that the attribute is constructed.

With the exception of the integer-valued `msDS-User-Account-Control-Computed`, these attributes are Boolean values in the directory. Some of these attributes are also constructed attributes and as such, they cannot be written. We wish we could say there was some method behind the slightly different `ldapDisplayName` prefix values, but it just appears that it was overlooked.

Reading User Account Properties

The constructed attribute called `msDS-User-Account-Control-Computed` takes the place of `userAccountControl` in ADAM. This attribute is new to Windows Server 2003 Active Directory and ADAM and we can use it rather than `userAccountControl` to read account properties. However, given that this is a constructed attribute, we can neither search on any values held within it nor set any values on this attribute. Listing 10.5 demonstrates how similar it is to read this attribute compared to using the `userAccountControl` attribute in Listing 10.3.

LISTING 10.5: Reading the msDS-User-Account-Control-Computed Attribute

```
DirectoryEntry user = new DirectoryEntry(
    "LDAP://CN=User1,CN=users,DC=domain,DC=com",
    null,
    null,
    AuthenticationTypes.Secure
    );

//this is a pain to type a lot :)
string msDS = "msDS-User-Account-Control-Computed";

using (user)
{
    //this is constructed attribute
    user.RefreshCache(
        new string[]{msDS}
        );

    AdsUserFlags userFlags =
        (AdsUserFlags)user.Properties[msDS].Value;

    Console.WriteLine(
        "AdsUserFlags for {0}: {1}",
        user.Path,
        userFlags
        );
}
```

We should note that the `msDS-User-Account-Control-Computed` attribute will accurately hold values like `AccountLockedOut`, `PasswordCannotChange`, and `PasswordExpired`. This departs from the `userAccountControl` attribute, where these values are not represented accurately because the flags are not used. We also have the option of using the special "alias" attributes such as `ms-DS-UserAccountAutoLocked` here. We simply read the Boolean value they return. In many cases, this may be more straightforward.

Writing User Account Properties

Since the `userAccountControl` attribute is not used with ADAM, and its equivalent but constructed attribute cannot be written, we need to use the other `msDS` and `ms-DS` attributes to actually set values. Listing 10.6 shows one such example where we can enable or disable an ADAM account.

LISTING 10.6: Writing Account Values

```
string adsPath =
    "LDAP://localhost:389/"
    + "CN=User1,OU=Users,O=dunnry,C=US";

DirectoryEntry user = new DirectoryEntry(
    adsPath,
    null,
    null,
    AuthenticationTypes.Secure
    );

string attrib = "msDS-UserAccountDisabled";

using (user)
{
    //disable the account
    user.Properties[attrib].Value = true;
    user.CommitChanges();
}
```

As writing each of the other nonconstructed Boolean account properties is exactly the same, we will not demonstrate further examples. The key point to take away here is that we need to look to these attributes in lieu of using the `userAccountControl` attribute for ADAM.

NOTE Boolean Attributes Can Accept String Values as Well

Boolean syntax attributes can accept both the .NET Boolean `true` and `false` as well as the LDAP string equivalent `TRUE` and `FALSE` (note the case). This is because the underlying value is held as a string in the LDAP directory and only marshaled as a Boolean for us to use in .NET. If we remember that searching for Boolean attributes requires using `TRUE` and `FALSE`, all of this stuff starts to make sense. Since using the actual Boolean type in .NET tends to be easier, we only mention it in passing as a fun factoid.

Determining Domain-Wide Account Policies

When working with user accounts in Active Directory, it is common to need to refer to domain-wide account policies. For example, policies such as the minimum and maximum password age and the minimum password length, as well as lockout policy, are determined at the domain level and apply to each `user` object in the domain.

All of the values are stored directly in the domain root object (not in `RootDSE`, but in the object pointed to by the `defaultNamingContext` attribute in `RootDSE`) as a set of attributes such as `maxPwdAge`, `minPwdLength`, and `lockoutThreshold`. Additionally, the password complexity rules are encoded in an enumerated value in the `pwdProperties` attribute.

These values tend to be quite static in most domains, so we would typically want to read these values only once per program execution. To make the policy values easy to consume, we show in Listing 10.7 a wrapper class for the domain account policies that converts all of the values into convenient .NET data types, such as `TimeSpan`. A special .NET enumeration type for the types of the password policy is also included. We won't be able to include all of the class properties in the book, as that would take too much space, but we will have the full class available on the book's web site.

We will refer to this sample in future discussions when demonstrating how to determine an account's lockout status and for finding accounts with expiring passwords. It is also worthy to note that any `LargeInteger` values in these policy attributes are stored as negative values. We chose to invert them back to positive values because it is easier to think about them in this way. Developers choosing to use these attributes should keep this in mind, as it will throw off calculations later if not accounted for.

Listing 10.7: Determining Domain Policies

```
[Flags]
public enum PasswordPolicy
{
    DOMAIN_PASSWORD_COMPLEX=1,
    DOMAIN_PASSWORD_NO_ANON_CHANGE=2,
    DOMAIN_PASSWORD_NO_CLEAR_CHANGE=4,
    DOMAIN_LOCKOUT_ADMINS=8,
    DOMAIN_PASSWORD_STORE_CLEARTEXT=16,
    DOMAIN_REFUSE_PASSWORD_CHANGE=32
}
```

```
public class DomainPolicy
{
    ResultPropertyCollection attribs;

    public DomainPolicy(DirectoryEntry domainRoot)
    {
        string[] policyAttributes = new string[] {
            "maxPwdAge", "minPwdAge", "minPwdLength",
            "lockoutDuration", "lockOutObservationWindow",
            "lockoutThreshold", "pwdProperties",
            "pwdHistoryLength", "objectClass",
            "distinguishedName"
            };

        //we take advantage of the marshaling with
        //DirectorySearcher for LargeInteger values...
        DirectorySearcher ds = new DirectorySearcher(
            domainRoot,
            "(objectClass=domainDNS)",
            policyAttributes,
            SearchScope.Base
            );

        SearchResult result = ds.FindOne();

        //do some quick validation...
        if (result == null)
        {
            throw new ArgumentException(
                "domainRoot is not a domainDNS object."
                );
        }

        this.attribs = result.Properties;
    }

    //for some odd reason, the intervals are all stored
    //as negative numbers.  We use this to "invert" them
    private long GetAbsValue(object longInt)
    {
        return Math.Abs((long)longInt);
    }

    public TimeSpan MaxPasswordAge
    {
        get
        {
            string val = "maxPwdAge";
            if (this.attribs.Contains(val))
            {
                long ticks = GetAbsValue(
                    this.attribs[val][0]
                    );

                if (ticks > 0)
                    return TimeSpan.FromTicks(ticks);
            }

            return TimeSpan.MaxValue;
        }
    }

    public PasswordPolicy PasswordProperties
    {
        get
        {
            string val = "pwdProperties";
            //this should fail if not found
            return (PasswordPolicy)this.attribs[val][0];
        }
    }

    //truncated for book space
}
```

Listing 10.7 is meant to run on an Active Directory domain. Where does this leave ADAM instances? By default, ADAM will assume any local or domain policies on the Windows 2003 server where it is running. This means that if our Windows 2003 server is a member of the domain, we can simply use code similar to that in Listing 10.7. If, however, the server is running in a workgroup configuration, the policy will be determined locally. Therefore, Listing 10.7 would not be appropriate. Instead, we would need to know our local policy or attempt to discover it using Windows Management Instrumentation (WMI) classes.

Determining Password Expiration

Earlier in this chapter, we mentioned that accounts could have passwords that expire. Most Active Directory domains and many ADAM instances force passwords to expire periodically to improve security. As such, we often need to know when a user's password will expire.

Determining password expiration on user accounts in Active Directory and ADAM might appear tricky, but really, it is a simple matter of calculation.

Password expiration is determined based on when an individual password was last changed, and on the domain-wide password expiration policy, which we detailed in the previous section. The algorithm is essentially this:

```
if "password change date" + "max password age" >= "now"
    "password is expired"
```

Typically, Windows monitors password expiration and will inform a user that her password is expiring soon when she logs on locally to Windows. It then provides a mechanism to change the password. As long as the user changes her password before it expires, she can continue to log in to the domain and all is good. However, if the password expires, then the user cannot log in again until an administrator resets it.

This situation is not as straightforward for ADAM users, as there is no natural "login" process that informs users of pending password expiration and prompts them for a password change. Instead, it is completely up to the developer to supply both a notification and a means by which to change a password when using ADAM.

Programmatic LDAP binds to either directory must be handled explicitly by the developer, as we will not be warned of pending password expiration. Once a password has expired, all LDAP binds will fail until the password is reset by the user or an administrator.

How Password Modification Dates Are Stored

Active Directory and ADAM use the `pwdLastSet` attribute to record when a password was last changed, via either an end-user password change or an administrative reset. Like most time-based Windows data in the directory, the attribute uses the 2.5.5.16 `LargeInteger` attribute syntax, which essentially holds a Windows `FILETIME` structure as an 8-byte integer. We already discussed how to read and write these attributes in Chapter 6, as well as build LDAP search filters based on them in Chapter 4, so that knowledge should be easy to apply to this problem.

There is one edge case that developers should be aware of when dealing with this attribute. Namely, the `pwdLastSet` attribute can be set to zero (`0`), which implies that the password is automatically expired and must be changed at next login.

NOTE Zero Is Not a Valid FILETIME Value in .NET

The `0` value for `pwdLastSet` is especially important to remember because it is not a valid `FILETIME` value. If we pass it to a .NET function that converts between .NET `DateTime` structures and `FILETIME`, it will throw an error. Additionally, if `pwdLastSet` is `0`, the user cannot bind to the directory via LDAP. This makes it impossible to do programmatic password changes via LDAP. Only administrative resets with different credentials are possible in this state.

Determining a Single User's Password Expiration Date

Now that we know the details on how this mechanism works, we are ready to write some code to check this. The first thing we need is a user's `pwdLastSet` value as a .NET `Int64`, or long integer. As per Chapter 6, we can do this using `DirectorySearcher` and its built-in marshaling of the data, or we can use one of the conversion functions we described for use with `DirectoryEntry`. For our purposes, we will use `DirectorySearcher` for converting the `LargeInteger` values in conjunction with Listing 10.7 to obtain domain policies.

We will step through a larger class we have chosen to name `PasswordExpires`, explaining as we go the thought process that surrounds what we are trying to accomplish. As such, we might have to refer to previous listings to see any member variables. This is a complete class and it requires a number of lines, but don't worry about needing to copy it verbatim. We will include it as a sample on the book's web site under its listing number.

The first part of determining password expiration is to determine our domain policy for the maximum password age (`MaxPwdAge`). Listing 10.8 shows how we can easily accomplish this using the `DomainPolicy` class we introduced in Listing 10.7 along with some tricks we learned in Chapter 9 using `System.DirectoryServices.ActiveDirectory` (SDS.AD) classes.

LISTING 10.8: PasswordExpires, Part I

```
public class PasswordExpires
{
    DomainPolicy policy;

    const int UF_DONT_EXPIRE_PASSWD = 0x10000;
```

```
        public PasswordExpires()
        {
            //get our current domain policy
            Domain domain = Domain.GetCurrentDomain();
            DirectoryEntry root = domain.GetDirectoryEntry();

            using (domain)
            using (root)
            {
                this.policy = new DomainPolicy(root);
            }
        }
```

In Listing 10.8, we are simply using the `Domain` class from SDS.AD to get a `DirectoryEntry` object bound to the current domain's default naming context. We need the root partition of the domain in order to determine our domain policies. At this point, we simply load our `DomainPolicy` object with our root `domainDNS` object. Next, we need to calculate the actual `DateTime` when a user's password would expire. Listing 10.9 shows how we can accomplish this.

LISTING 10.9: PasswordExpires, Part II

```
        public DateTime GetExpiration(DirectoryEntry user)
        {
            int flags =
                (int)user.Properties["userAccountControl"][0];

            //check to see if password is set to expire
            if(Convert.ToBoolean(flags & UF_DONT_EXPIRE_PASSWD))
            {
                //the user's password will never expire
                return DateTime.MaxValue;
            }

            long ticks = GetInt64(user, "pwdLastSet");

            //user must change password at next login
            if (ticks == 0)
                return DateTime.MinValue;

            //password has never been set
            if (ticks == -1)
            {
                throw new InvalidOperationException(
                    "User does not have a password"
                    );
            }
     //get when the user last set their password;
     DateTime pwdLastSet = DateTime.FromFileTime(
         ticks
         );

     //use our policy class to determine when
     //it will expire
     return pwdLastSet.Add(
         this.policy.MaxPasswordAge
         );
 }
```

The first thing we do in Listing 10.9 is check to see if the user's account is set so that the password never expires. We will use the convention that `DateTime.MaxValue` means it will never expire. Next, we are using a helper function called `GetInt64` (see Listing 10.10). This function marshals the user's `pwdLastSet` attribute into an `Int64` for us using a `DirectorySearcher` object. Notice that one of three conditions can arise out of this check. First, the `pwdLastSet` attribute might be `null` (if it is not set on the object), in which case the user account has no password. We chose to treat the situation where a user does not have a password as an error condition, but this can differ by application. Second, the attribute might be `0`, which means that the user must change her password at the next logon. We chose to use the convention that `DateTime.MinValue` meant that the password must be changed at next log on. Lastly, the attribute might contain some value that we can interpret as a `FILETIME` structure and can convert using `DateTime.FromFileTime`. The calculation for determining when a user's password will expire is simple. We just add the `DateTime` value of when the user last changed her password to the `TimeSpan` value of the domain's `MaxPwdAge` policy. If the user's password has already expired, we will still get a `DateTime` value, but it will be in the past.

Knowing the date a user's password has expired is nice, but we might actually want to know how much time is left before the user's password expires. That is a very easy calculation, as Listing 10.10 demonstrates.

LISTING 10.10: PasswordExpires, Part III

```
     public TimeSpan GetTimeLeft(DirectoryEntry user)
    {
        DateTime willExpire = GetExpiration(user);
        if (willExpire == DateTime.MaxValue)
            return TimeSpan.MaxValue;

        if (willExpire == DateTime.MinValue)
            return TimeSpan.MinValue;

        if (willExpire.CompareTo(DateTime.Now) > 0)
        {
            //the password has not expired
            //(pwdLast + MaxPwdAge)- Now = Time Left
            return willExpire.Subtract(DateTime.Now);
        }

        //the password has already expired
        return TimeSpan.MinValue;
    }

    private Int64 GetInt64(DirectoryEntry entry, string attr)
    {
        //we will use the marshaling behavior of
        //the searcher
        DirectorySearcher ds = new DirectorySearcher(
            entry,
            String.Format("({0}=*)", attr),
            new string[] { attr },
            SearchScope.Base
            );

        SearchResult sr = ds.FindOne();

        if (sr != null)
        {
            if (sr.Properties.Contains(attr))
            {
                return (Int64)sr.Properties[attr][0];
            }
        }
        return -1;
    }
```

We chose to return a `TimeSpan` value representing the time left before a user's password would expire in Listing 10.10. If either `TimeSpan.MaxValue` or `TimeSpan.MinValue` is returned, it is meant to indicate that the user's password does not expire or has already expired, respectively. For completeness, we have also included our helper `GetInt64` in Listing 10.10, though by now we know that everyone is probably aware of how to marshal `LargeInteger` values from Chapter 6.

We typically want to do something based on when a password will expire, so it is important to know how much time is left. However, if we just want to know whether an account has expired, we can use the previously mentioned `msDS-User-Account-Control-Computed` attribute for Windows 2003 Active Directory and ADAM, or the aptly named `msDS-UserPasswordExpired` attribute for ADAM, to just give us a yes/no answer. Listing 10.11 shows one such example.

Listing 10.11: Checking Password Expiration

```
string adsPath = "LDAP://CN=User1,OU=Users,DC=domain,DC=com";

DirectoryEntry user = new DirectoryEntry(
    adsPath,
    null,
    null,
    AuthenticationTypes.Secure
    );

string attrib = "msDS-User-Account-Control-Computed";

using (user)
{
    user.RefreshCache(new string[] { attrib });

    int flags = (int)user.Properties[attrib].Value
        & (int)AdsUserFlags.PasswordExpired);

    if (Convert.ToBoolean(flags)
    {
        //password has expired
        Console.WriteLine("Expired");
    }
}
```

Of course, the problem with something like Listing 10.11 is that we don't know when the password actually expired or how much time is left before it does expire. Additionally, as we previously mentioned, a solution like this will work with only Windows 2003 Active Directory and ADAM. Windows 2000 Active Directory users must use a solution such as that shown in Listing 10.8.

Searching for Accounts with Expiring Passwords

Another thing we may wish to do is find all of the accounts with passwords expiring within a certain time range, perhaps to send an email notification directing users to a web-based portal where they can change their passwords. This is important for ADAM users and any Active Directory users that do not typically log in to Windows via the workstation.

The crux of this search is based on creating a search filter with the correct values. Let's say we want to find user accounts with passwords expiring between two dates. Since password expiration is based on the date the password was last changed and the maximum password age domain policy, we subtract the maximum password age from the two dates to get the values of `pwdLastSet` that will match. The code might look like that shown in Listing 10.12.

LISTING 10.12: Finding Expiring Passwords

```
public static string GetExpirationFilter(
    DateTime startDate,
    DateTime endDate,
    TimeSpan maxPwdAge
    )
{
    Int64 lowDate;
    Int64 highDate;
    string filterPattern = "(&(sAMAccountType=805306368)" +
        "(pwdLastSet>={0})(pwdLastSet<={1}))"

    lowDate = startDate.Subtract(maxPwdAge).ToFileTime();
    highDate = endDate.Subtract(maxPwdAge).ToFileTime();

    return String.Format(
        filterPattern,
        lowDate,
        highDate
        );
}
```

A complete sample that enumerates users with expiring passwords between two dates is available on this book's web site.

In both examples, we see that .NET makes this especially easy. The built-in support for dates, time spans, and Windows `FILETIME` structures simplifies much of the work. We can also easily construct variations on this, using similar techniques, to find accounts whose passwords have already expired, or to find all accounts that will expire before or after a certain date.

NOTE Performance Implications of Using pwdLastSet

The `pwdLastSet` attribute is not indexed, nor is it stored in the global catalog in Active Directory by default. As such, this search cannot be performed across an entire forest, and it can be slow, introducing the possibility of timeouts. Using a small page size for our `DirectorySearcher`, our results will be returned more quickly and we can help mitigate some of these risks. It is definitely much faster than enumerating all of the users in the domain and looking at each individual attribute value on the client side.

Determining Last Logon

The last time a user has logged onto the domain is held in an attribute on the user object. Called `lastLogon`, this attribute is a nonreplicated attribute, which means that each domain controller holds its own copy of the attribute, likely with different values. Checking the last time a user has logged onto the domain requires us to visit each domain controller and read the attribute. The value found for the latest `lastLogon` is the value we are after.

We covered in Chapter 9 how to use the Locator to enumerate all the domain controllers. We will use this technique again to iterate through each controller and retrieve the `lastLogon` attribute for each user. Listing 10.13 demonstrates how to accurately determine the last time a user has logged into the domain.

LISTING 10.13: Finding a User's Last Logon

```
string username = "user1";
string domain = "mydomain.com";

public static void LastLogon(string username, string domain)
{
    DirectoryContext context = new DirectoryContext(
        DirectoryContextType.Domain,
        domain
        );
        DateTime latestLogon = DateTime.MinValue;
        string servername = null;

        DomainControllerCollection dcc =
            DomainController.FindAll(context);

        foreach (DomainController dc in dcc)
        {
            DirectorySearcher ds;

            using (dc)
            using (ds = dc.GetDirectorySearcher())
            {
                ds.Filter = String.Format(
                    "(sAMAccountName={0})",
                    username
                    );
                ds.PropertiesToLoad.Add("lastLogon");
                ds.SizeLimit = 1;

                SearchResult sr = ds.FindOne();

                if (sr != null)
                {
                    DateTime lastLogon = DateTime.MinValue;
                    if (sr.Properties.Contains("lastLogon"))
                    {
                        lastLogon = DateTime.FromFileTime(
                            (long)sr.Properties["lastLogon"][0]
                            );
                    }

                    if (DateTime.Compare(lastLogon,latestLogon) > 0)
                    {
                        latestLogon = lastLogon;
                        servername = dc.Name;
                    }
                }
            }
        }

        Console.WriteLine(
            "Last Logon: {0} at {1}",
            servername,
            latestLogon.ToString()
            );
    }
```

We are using the SDS.AD namespace here to enumerate all of our domain controllers, and then we are using `DirectorySearcher` and `SearchResult` to marshal the `LargeInteger` syntax `lastLogon` attribute more easily. In domains with widely distributed domain controllers, we should be aware that network latency can slow this technique dramatically.

We should further keep in mind that this technique is relatively slow because it must bind to each domain controller in turn to find the user account and retrieve the `lastLogon` value. However, it is accurate, as each domain controller is searched and the latest logon is found.

Finding Stale Accounts

There is one more method we can use for finding old or unused accounts if we're running Windows Server 2003 Active Directory in full Windows 2003 mode. The `user` class schema has been updated to add a new attribute called `lastLogonTimestamp`. This is a replicated value that is updated periodically. How often the attribute is updated depends on a new domain policy attribute named `msDS-LogonTimeSyncInterval`. By default, this value is 14 days, but it is configurable. As such, this attribute is not accurate for purposes of determining exactly the last time a user logged into the domain. We must use the `lastLogon` attribute when accuracy matters.

The syntax of this attribute is `LargeInteger`, so it is similar to other techniques we have already demonstrated. We can simply create a filter using the `DateTime.ToFileTime` method appropriately:

```
//find all users and computers that
//have not been used in 30 days
String filter = String.Format(
    "(&(objectClass=user)(lastLogonTimestamp<={0}))",
    DateTime.Now.Subtract(TimeSpan.FromDays(30)).ToFileTime()
    );
```

At first blush, this seems like a pretty easy thing to do. Indeed it is, if we can live with the following limitations.

- It is inaccurate up to the value of `msDS-LogonTimeSyncInterval` (14 days by default). This means we can be no better than the resolution of this attribute in terms of our accuracy in finding accounts.

- This is a Windows 2003 Active Directory–only feature, and even then, only when running in Windows 2003 mode.

Only NTLM and Kerberos logins update this attribute. Service Pack 1 must be applied to correct problems with NTLM as well. This means that any other type of operation that generates a login—certificates, custom Security Support Provider Interface (SSPI), Kerberos Service for User (S4U), and so on—will not update this attribute.

Additional caveats that depend on when the domain functional level was increased also affect the accuracy of this attribute. You can find further information about all the caveats at www.microsoft.com/technet/prodtechnol/windowsserver2003/library/TechRef/54094485-71f6-4be8-8ebf-faa45bc5db4c.mspx.

Determining Account Lockout

Determining whether an account is locked out is a strange odyssey. On the surface, it appears simple enough: We have a flag on `userAccountControl` called `UF_LOCKOUT`. Surely, that would mean that checking to see if the flag was flipped would tell us whether an account was locked out, right? Well, that really depends on which provider we use. As we mentioned earlier in this chapter, the `userAccountControl` attribute is inaccurate in terms of determining an account's lockout status for the `LDAP` provider. The situation is somewhat better if we use the `WinNT` provider, as this version of the `userAccountControl` attribute accurately reflects lockout status. Microsoft was aware of this flakey implementation and fixed it on Windows 2003 Active Directory and ADAM with the `msDS-User-Account-Control-Computed` attribute. This constructed attribute will accurately reflect the `UF_LOCKOUT` flag for the LDAP provider. Listing 10.14 shows a sample of how this would work.

LISTING 10.14: Determining Account Lockout

```
//user is a DirectoryEntry for our user account

string attrib = "msDS-User-Account-Control-Computed";

//this is a constructed attrib
user.RefreshCache(new string[]{attrib});
```

```
const int UF_LOCKOUT = 0x0010;

int flags =
    (int)user.Properties[attrib].Value;

if (Convert.ToBoolean(flags & UF_LOCKOUT))
{
    Console.WriteLine(
        "{0} is locked out",
        user.Name
        );
}
```

There are a couple problems with this method, of course.

- This works only for Windows 2003 Active Directory and ADAM, not for Windows 2000 Active Directory.
- Since `msDS-User-Account-Control-Computed` is a constructed attribute, it cannot be used in an LDAP search filter.

Unfortunately, since this attribute cannot be used in a search filter, we really cannot use this method to find locked accounts proactively. Luckily, we can actually compute whether an account is locked out fairly accurately, and search for it. Previously in this chapter, we showed how we could determine the domain's lockout duration policy. Used in conjunction with the `lockoutTime` attribute, we can accurately predict whether an account is locked out, and search for it. Listing 10.15 shows one such example.

LISTING 10.15: Searching for Locked-Out Accounts

```
class Lockout : IDisposable
{
    DirectoryContext context;
    DirectoryEntry root;
    DomainPolicy policy;

    public Lockout(string domainName)
    {
        this.context = new DirectoryContext(
            DirectoryContextType.Domain,
            domainName
            );

        //get our current domain policy
        Domain domain = Domain.GetDomain(this.context);
        this.root = domain.GetDirectoryEntry();
        this.policy = new DomainPolicy(this.root);
    }

    public void FindLockedAccounts()
    {
        //default for when accounts stay locked indefinitely
        string qry = "(lockoutTime>=1)";

        TimeSpan duration = this.policy.LockoutDuration;

        if (duration != TimeSpan.MaxValue)
        {
            DateTime lockoutThreshold =
                DateTime.Now.Subtract(duration);

            qry = String.Format(
                "(lockoutTime>={0})",
                lockoutThreshold.ToFileTime()
                );
        }

        DirectorySearcher ds = new DirectorySearcher(
            this.root,
            qry
            );

        using (SearchResultCollection src = ds.FindAll())
        {
            foreach (SearchResult sr in src)
            {
                long ticks =
                    (long)sr.Properties["lockoutTime"][0];

                Console.WriteLine(
                    "{0} locked out at {1}",
                    sr.Properties["name"][0],
                    DateTime.FromFileTime(ticks)
                    );
            }
        }
    }

    public void Dispose()
    {
        if (this.root != null)
        {
            this.root.Dispose();
        }
    }
}
```

Listing 10.15 gives us a fairly simple way of finding accounts that have been locked. We are using the `DomainPolicy` helper class we introduced earlier in this chapter to read the `lockoutDuration` attribute on the root domain. If the attribute is set to `TimeSpan.MaxValue`, it means that an account is to stay locked until an administrator unlocks it. We are accounting for this policy possibly by setting our search filter to designate that any account with a `lockoutTime` with a nonzero value is locked out. This filter is appropriate only when our domain policy tells us that accounts are to be locked out indefinitely, until an administrator unlocks them.

Managing Passwords for Active Directory Users

Unlike most Active Directory and ADAM user-management tasks, which we perform through simple manipulation of Active Directory objects and attributes via LDAP, managing passwords is a bit complex. Password changes require very special semantics that are enforced by the server, and developers need to understand these semantics for password management applications to be successful.

In order to try to facilitate the password management process, ADSI exposes two methods on the `IADsUser` interface: `SetPassword` and `ChangePassword`. `SetPassword` is used to perform an administrative reset of a user's password and is typically performed by an administrator. Knowledge of the previous password is not required. `ChangePassword` is used simply to change the password from one value to another and is typically performed only by the user represented by the directory object. It does require knowledge of the previous password, and thus it takes the old and new passwords as arguments.

Since the `DirectoryEntry` object does not directly expose the `IADsUser` ADSI interface, this is one case where we must use the `DirectoryEntry.Invoke` method to call these ADSI methods via late-bound reflection:

```
//given a DirectoryEntry "entry" that points to a user object
//this will reset the user's password
entry.Invoke("SetPassword", new object[] {"newpassword"});

//this will change the user's password
entry.Invoke("ChangePassword",
    new object[] {"oldpassword", "newpassword"});
```

Note that the parameters to the `SetPassword` and `ChangePassword` methods are passed in as an array of objects that contain strings.

Password Management Complications

This seems simple enough; just call the right method and pass in the right arguments. Unfortunately, in practice this turns out not to be the case, as password management functions tend to be the most troublesome aspect of programmatic user management beyond the normal issues related to security that we discussed in Chapter 8.

Password management can be hard, for a variety of reasons.

- Password policy and security can be complicated and must be understood.
- The underlying ADSI methods actually try a complicated series of different approaches to set or change the password, and all of them use different protocols and have different requirements and failure modes.
- The errors returned by ADSI are often not helpful and the resulting exception in .NET needs other special care.

Developers who have issues reliably using `SetPassword` and `ChangePassword` in all environments are unfortunately not in the minority. However, the following exploration of these three points will be immensely helpful.

Understanding Password Policy and Security

Active Directory enforces password policy at the domain level. Password policies that can be set include:

- The minimum allowed length
- The maximum allowed age
- The minimum allowed age (how often we can change passwords)
- Password complexity
- Password history (determines when we can reuse an old password)

Complex passwords are just passwords that require some mix of upper- and lowercase letters, numbers, and special characters. All of these policies are exposed via attributes set in the domain root object, which can easily be found via the `defaultNamingContext` attribute on `RootDSE`. We discussed how to find these domain-wide policy settings earlier in this chapter.

Knowing what these values are will help us predict how the server will behave when asked to modify any given password. All of the policies that are enabled are enforced for each password modification; however, administrative password resets (as opposed to end-user password changes) ignore all of them except for length and complexity. This makes sense, since an administrator who (presumably) has no knowledge of the user's previous passwords performs the reset.

Password security is relatively straightforward. Essentially, administrative users have rights to reset any account's password, and normal users have rights to change only their own passwords. The only trick here is that the right to reset passwords can be delegated beyond the scope of the default groups, such as `Domain Admins` and `Account Operators`, so it is possible that users outside of those groups have this right.

From the application developer's perspective, our job is to know which of the two tasks we are trying to perform and to make sure that we have the correct credentials to do so. This is pretty easy when building applications to change passwords, as the user has to give us her old password to make the change, so it is relatively easy to bind to the directory with her credentials directly and avoid any tricky impersonation scenarios (see Chapter 8). For password reset operations, our options can equally vary between using the current user's security context (if it's an admin application), to using a trusted service account (this is often done with helpdesk-type applications). There is not a clear-cut answer of which method to use here, and it will often depend on the type of application we are trying to build. Chapter 8 explains these scenarios in detail, especially for web applications.

Understanding the Underlying ADSI Methods

The `SetPassword` and `ChangePassword` methods may seem simple on the outside, but under the hood, it is just the opposite. There is a variety of different ways to modify passwords against Active Directory, and these methods pretty much try all of them. On the one hand, this is great for developers, as this approach gives developers the best possible chance that their operation will succeed. The downside of this approach is that each technique has different requirements and different ways to fail, so figuring out why things are not working can be especially difficult.

`SetPassword` and `ChangePassword` attempt password modifications using the following methods, in order:

- LDAP password modification over an SSL channel
- The Kerberos set password protocol over the Kerberos password port (`SetPassword` only)
- A Net* API remote procedure call (RPC)

As noted in the list, the Kerberos set password protocol is used only for `SetPassword`. `ChangePassword` does not seem to have an equivalent Kerberos protocol implementation. Let's take each technique in turn.

LDAP Password Modification

The first technique that is always attempted is an LDAP-based password modification. The core of this technique involves modifying the `unicodePwd` attribute directly. `SetPassword` does one modification with the `Replace` modification type specified, and `ChangePassword` does two modifications with a `Delete` and an `Add` specified, in that order. Active Directory enforces a restriction that any modification to the `unicodePwd` attribute must be made over an encrypted channel with a cipher strength of 128 bits. Otherwise, the server will reject the attempted modification. This helps ensure that the plaintext password is not intercepted on the network.

This is where SSL comes in. If we recall from Chapter 3, Active Directory supports two mechanisms for channel encryption: SSL and Kerberos. However, only SSL supports the minimum 128-bit cipher strength on all Active Directory platforms. Kerberos-based encryption has been strengthened to meet this requirement on Windows Server 2003, but not on Windows 2000 Server. Because the function attempts to work with either version of Active Directory, it always selects only SSL for the channel encryption technique.

This is unfortunate, because Kerberos-based encryption works out of the box with Active Directory, but SSL requires additional configuration steps including the acquisition of proper SSL certificates for each participating domain controller. Since SSL/LDAP is not required for normal Active Directory operation, many administrators do not bother to configure it. As a result, SSL is often not available for performing password modifications with `SetPassword` and `ChangePassword`.

Additionally, for an SSL/LDAP bind to succeed, proper DNS names must be used to connect to the domain controller. Typically, the server authentication aspect of the SSL handshake requires that the name used to access the server match the name of the server in the certificate, which is typically a DNS name. Therefore, it is important to remember to avoid using IP addresses and NetBIOS names when accessing a domain controller if SSL support is required.

A final thing to remember is that SSL/LDAP uses TCP port 636, not port 389, like typical LDAP traffic does. We must take this into account with any network firewall restrictions to make sure the proper ports are open.

Kerberos Set Password Protocol

The Kerberos specification includes a facility for setting user passwords. This facility does not use the original user's password, so it is used only for administrative resets (`SetPassword`) and not for end-user password changes using `ChangePassword`.

The Kerberos set password protocol also uses a different network port (TCP 441) than normal Kerberos traffic, which goes over port 88 UDP and TCP. Once again, we must consider potential firewall issues if we wish to use this protocol.

Kerberos set password is available when a secure bind was requested and the secure bind negotiates to Kerberos rather than to NTLM. Chapter 8 contains more information about this, including some sample code we can use to verify secure bind status on the server side.

Net API RPCs*

The final method attempted by the password modification methods are one of two RPC functions from the Net* family: `NetUserSetInfo` and `NetUserChangePassword`. They are used for `SetPassword` and `ChangePassword`, respectively.

The key thing to remember with these RPC functions is that they must use the current thread's security context. They do not directly support plaintext credentials to establish a security context, as LDAP does. With Windows 2000, this had the surprising effect of ignoring any credentials supplied to `DirectoryEntry` for `SetPassword` privileges. Developers found that if they switched environmental factors, identical code that previously worked would fail. There was also no way to know which of the three methods was being selected. So, if the test environment had SSL certificates installed correctly, `SetPassword` and `ChangePassword` would work just fine. Moving the identical code to production, where perhaps the SSL certificate was not installed correctly, would fail because unbeknownst to the developer, `SetPassword` had actually used `NetUserSetInfo` under the covers. This violated most developers' idea of obvious behavior. After all, if the previous two methods can use the credentials supplied to `DirectoryEntry` to affect password modifications with identical code, why wouldn't this one? Microsoft decided to fix this unexpected behavior with Windows 2003. Now, if the code specifies plaintext credentials in `DirectoryEntry`, these methods will use another API, `LogonUser`, to create a Windows login token for the code to impersonate while making the RPC function call. This has the effect of keeping the outward behavior of all three methods the same.

For Windows 2000 users, the only way to get this third method working correctly is to make sure our current thread's credentials are those of an account that has the proper privileges. From Chapter 8, we know this means that either our code must run under a trusted subsystem that holds these rights, or we must impersonate an account that holds these rights for the duration of the call. Please refer back to Chapter 8 for more details, and for additional references.

Finally, remember that a Windows RPC requires TCP port 135 (and potentially, other ports) to be open to the domain controller.

Error Handling with the Invoke Method in .NET

So far, all of the issues we have discussed apply to any API that calls `SetPassword` or `ChangePassword`. While error handling for these two methods is certainly relevant, it is a larger .NET topic that affects any code that uses reflection via the `Invoke` method. When the `Invoke` method is used to call any ADSI interface method, any exception thrown by the target method, including a COM error triggered via COM interop, will be wrapped in a `System.Reflection.TargetInvocationException` exception, with the actual exception in the `InnerException` property. Therefore, we will need to use a pattern like this with the `Invoke` method:

```
//given a DirectoryEntry "entry" that points to a user object
//this will reset the user's password
try
{
    entry.Invoke("SetPassword", new object[] {"newpassword"});
}
catch (TargetInvocationException ex)
{
    throw ex.InnerException;
}
```

Obviously, we may wish to do something different from simply rethrowing the `InnerException` property. The point here is that `Inner-Exception` contains the information we are interested in.

The other issue here is that the exceptions coming back from ADSI vary, from the vaguely helpful to the truly bewildering. We will not attempt to list all of them here, but here are a few hints.

- `System.UnauthorizedAccessException` always indicates a permissions problem.
- A `System.Runtime.InteropServices.COMException` exception with `ErrorCode 0x80072035 "Unwilling to perform"` generally means that an LDAP password modification failed due to a password policy issue.
- A `System.Runtime.InteropServices.COMException` exception from one of the Net*APIs is usually pretty specific about what the problem was.

Recommendations for Successful Password Modification Operations

In our experience, it is possible to get `SetPassword` and `ChangePassword` to work in just about any environment, as long as we understand what they are doing under the hood and we recognize any dependencies involved in each technique. However, the approach that yields the most consistent results is to enable SSL on our domain controllers so that LDAP password modifications will be used.

Administrators may complain that they do not want to enable SSL due to the potential expense and complexity of obtaining third-party certificates or configuring an internal Windows certificate authority. Given that this is not required for general-purpose Active Directory operation, they may regard this as a nice-to-have feature and try to insist that you find another way to get this to work. We suggest you do your best to try to convince them otherwise!

Why Can't We Do LDAP Password Modifications Directly in SDS?

You may be wondering why we cannot use SDS to modify the `unicodePwd` attribute directly. After all, it seems like this should be possible. However, even though we can create the correct value syntax to set `unicodePwd` correctly, the ADSI property cache prevents this from working. The fact that `unicodePwd` is a "write-only" attribute seems to interfere with our ability to modify it. It may be possible to reset passwords (depending on the version of the .NET Framework we are using), but it is definitely not possible to change passwords.

SDS.P to the Rescue

This is one area where we can do something with `System.DirectoryServices.Protocols` (SDS.P) that we cannot do with SDS. The lower-level access to direct LDAP modification operations available in SDS.P allows us to perform LDAP password modifications. There are three keys to this approach.

- A 128-bit encrypted channel must be established, either by SSL or by secure bind channel encryption.
- The `unicodePwd` attribute value is actually submitted as an octet string (byte array) that contains the Unicode encoding of the password value surrounded by double quotes (see Listing 10.16).
- The reset password takes a single modification operation with the `Replace` operation type, and the change password operation takes a `Delete` operation of the old value followed by an `Add` of the new value.

One interesting thing here is the secure channel requirement. If we have both client and server with operating systems more recent than Windows 2000, we can use the built-in Kerberos-based channel encryption that is available with a typical secure bind and we do not need SSL. This gives us some options that are not available with the ADSI methods.

Listing 10.16 demonstrates how we might apply SDS.P to password modification operations. It is implemented to allow both password changes and resets and it works with either type of channel encryption, although only one set of options is demonstrated.

LISTING 10.16: Using .Protocols for Password Ops

```
using System;
using System.DirectoryServices.Protocols;
using System.Net;
using System.Text;

public class PasswordModifier
{
    public static void Main()
    {
        NetworkCredential credential = new NetworkCredential(
            "someuser",
            "Password1",
            "domain"
            );
        DirectoryConnection connection;

        try
        {
            //change these options to use Kerberos encryption
            connection = GetConnection(
                "domain.com:636",
                credential,
                true
                );

            ChangePassword(
                connection,
                "CN=someuser,CN=users,DC=domain,DC=com",
                "Password1",
                "Password2"
                );

            Console.WriteLine("Password modified!");
            IDisposable disposable = connection as IDisposable;
            if (disposable != null)
                disposable.Dispose();
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.ToString());
        }
    }

    private static DirectoryConnection GetConnection(
        string server,
        NetworkCredential credential,
        bool useSsl
        )
    {
        LdapConnection connection =
            new LdapConnection(server);

        if (useSsl)
        {
            connection.SessionOptions.SecureSocketLayer = true;
        }
        else
        {
            connection.SessionOptions.Sealing = true;
        }

        connection.Bind(credential);
        return connection;
    }

    private static void ChangePassword(
        DirectoryConnection connection,
        string userDN,
        string oldPassword,
        string newPassword
        )
    {
        DirectoryAttributeModification deleteMod =
            new DirectoryAttributeModification();
        deleteMod.Name = "unicodePwd";
        deleteMod.Add(GetPasswordData(oldPassword));
        deleteMod.Operation= DirectoryAttributeOperation.Delete;

        DirectoryAttributeModification addMod =
            new DirectoryAttributeModification();
        addMod.Name = "unicodePwd";
        addMod.Add(GetPasswordData(newPassword));
        addMod.Operation = DirectoryAttributeOperation.Add;
```

```
            ModifyRequest request = new ModifyRequest(
                userDN,
                deleteMod,
                addMod
                );

            DirectoryResponse response =
                connection.SendRequest(request);
        }

        private static void SetPassword(
            DirectoryConnection connection,
            string userDN,
            string password
            )
        {
            DirectoryAttributeModification pwdMod =
                new DirectoryAttributeModification();
            pwdMod.Name = "unicodePwd";
            pwdMod.Add(GetPasswordData(password));
            pwdMod.Operation = DirectoryAttributeOperation.Replace;

            ModifyRequest request = new ModifyRequest(
                userDN,
                pwdMod
                );

            DirectoryResponse response =
                connection.SendRequest(request);
        }

        private static byte[] GetPasswordData(string password)
        {
            string formattedPassword;
            formattedPassword = String.Format("\"{0}\"", password);
            return (Encoding.Unicode.GetBytes(formattedPassword));
        }
    }
```

Managing Passwords for ADAM Users

Managing passwords in ADAM is similar to managing passwords in Active Directory. However, there are a few important differences to be aware of. The primary difference is that the Kerberos password change protocol and the Net* APIs are not available for ADAM. This is because ADAM does not function as a Kerberos ticket granting service, nor does it expose the security account manager RPC interfaces that Active Directory does.

Because of this, the only technique available for modifying passwords on ADAM users is LDAP. None of the other techniques that the ADSI `IADsUser.SetPassword` and `ChangePassword` methods implement applies.

The other key difference is that ADAM allows us to relax the requirement on having a 128-bit secure channel for password modifications. This is helpful, because SSL is the only binding option available for ADAM users that allows encryption, and once again, SSL is not always an attractive option for administrators. SSL is notoriously more difficult to configure on ADAM than it is on Active Directory because of the extra complexity of associating the certificate with the correct service account.

To disable the requirement for a secure channel to be used for password modification operations, the thirteenth bit of the `dsHeuristics` attribute must be changed. The ADAM documentation contains more details on this. We mention this only because ADAM is often used for prototyping due to its portability and ease of deployment. For testing and development purposes, we often disable this requirement ourselves instead of wading through all the SSL muck. However, in production applications, we would never recommend to relax the security requirements around password management.

Programming Differences When Setting ADAM Passwords

When we relax the secure channel password requirements with ADAM, we need a way to specify that we will be sending plaintext passwords on the normal LDAP port instead of ciphertext on the SSL port. We use the `IADsObjectOptions` interface for this, using the `ADS_OPTION_PASSWORD_PORT_NUMBER` and `ADS_OPTION_PASSWORD_METHOD` flags in conjunction with the `SetOption` method. We have two ways to do this. In .NET 2.0, a new wrapper class, `DirectoryEntryConfiguration`, has strongly typed methods for setting these options. Listing 10.17 shows how we can accomplish this.

LISTING 10.17: Using DirectoryEntryConfiguration for ADAM

```
//.NET 2.0 sample for ADAM password changes
DirectoryEntry entry = new DirectoryEntry(
    "LDAP://adamserver.com/CN=someuser,OU=users,O=adamsample",
    "someuser@adam",
    "UserPassword1",
    AuthenticationTypes.None
    );
```

```
entry.Options.PasswordPort = 389;
entry.Options.PasswordEncoding =
    PasswordEncodingMethod.PasswordEncodingClear;

entry.Invoke(
    "ChangePassword",
    new object[] {"UserPassword1", "UserPassword2"}
    );
```

In .NET 1.x, we do not have the handy wrapper class for `IADsObjectOptions`, so instead we will use the `Invoke` method via reflection to accomplish the same thing. Listing 10.18 demonstrates the necessary operations.

LISTING 10.18: Setting IADsObjectOptions via Reflection

```
//.NET 1.x sample
const int ADS_OPTION_PASSWORD_PORTNUMBER = 6;
const int ADS_OPTION_PASSWORD_METHOD = 7;
const int ADS_PASSWORD_ENCODE_CLEAR = 1;

DirectoryEntry entry = new DirectoryEntry(
    "LDAP://adamserver.com/CN=someuser,OU=users,O=adamsample",
    "someuser@adam",
    "UserPassword1",
    AuthenticationTypes.None
    );
entry.Invoke(
    "SetOption",
    new object[] {ADS_OPTION_PASSWORD_PORTNUMBER, 389}
    );
entry.Invoke(
    "SetOption",
    new object[] {
    ADS_OPTION_PASSWORD_METHOD,
    ADS_PASSWORD_ENCODE_CLEAR
    }
    );
entry.Invoke(
    "ChangePassword",
    new object[] {"UserPassword1", "UserPassword2"}
    );
```

Even if we do not relax the secure channel password requirement for ADAM, it may still be necessary to change the password port number if our ADAM instance uses a different port for SSL traffic than the standard 636. Consequently, both of the techniques shown in Listings 10.17 and

10.18 still apply, though we will want to use the SSL password encoding option instead.

Additionally, it is possible to apply the LDAP password modification sample using SDS.P from the previous section on Active Directory password modification. There are two caveats.

- We may need to change the encryption method and port number as appropriate.
- When we are modifying passwords of ADAM users with an ADAM account, it will not be possible to use Kerberos channel encryption, as ADAM users cannot do Kerberos-based secure binds. That approach is not appropriate here. It is still possible to use this approach when using pass-through binding as a Windows user with a secure bind.

Sadly, all of this seems more complicated than it really needs to be, and it probably is. We hope that we have at least explained the topic thoroughly and have given you the tools you need to get the work done.

Determining User Group Membership in Active Directory and ADAM

We often need to know a user's group membership, especially when building applications that require role-based security. There are many cases when we cannot simply rely on Windows to do this expansion for us, and we need an LDAP-based approach instead. Unfortunately, many samples that attempt to show how to do this miss important details or make key mistakes that can lead to compromised security in our applications. We attempt to right these wrongs and show some proven techniques that have been effective for us.

Two linked multivalued attributes, called `member` and `memberOf`, control group membership. The group object always holds the `member` attribute. The `memberOf` attribute is a calculated back link held on the group member object itself. As such, group membership is always managed from the group object side (the forward link) of the relationship and the back link is updated by the system automatically. That is, we can read the `memberOf` attribute, but we cannot modify it directly. This multivalued attribute contains the user's direct group membership, with one exception: It does not contain what is called the primary group. This group receives special treatment, and we cover how to read it in the next chapter.

When we say that the `memberOf` attribute contains the user's direct membership, we mean that while we can view groups that directly contain the user object, we cannot view any group membership that is derived from the nesting of group memberships. We will have to use either a recursive technique or the `tokenGroups` attribute to expand a user's membership fully.

It turns out that using the `tokenGroups` attribute is typically what we are after. This attribute holds a security identifier (SID) for each security group (including the aforementioned primary group) for which the user is a member, including nested group membership. Recursive solutions can often be a little messy. As such, the only advantage that the recursive technique holds is that it will expand group membership in distribution lists, while the `tokenGroups` attribute contains only security group membership.

We will cover three techniques for reading group membership using the `tokenGroups` attribute. The first technique will use an LDAP search to find each SID in the `tokenGroups` attribute, and the second technique will use the `DsCrackNames` API to convert them in a single batch. The third technique will be a .NET 2.0-only solution using the new `IdentityReference`-based classes.

Our ultimate goal will be to convert the `tokenGroups` attribute into a collection of human-readable group names. A typical example of this is to build a `GenericPrincipal` object and fill it with roles for a custom ASP.NET Forms authentication mechanism.

Retrieving the User's Token Groups

Regardless of the technique we choose to decode the `tokenGroups` attribute, we must first retrieve it. Since this is a constructed attribute, we must use the `RefreshCache` technique shown in Chapter 3 to first load the attribute into the property cache in a `DirectoryEntry` object. This is one of the few attributes that requires a `Base` search scope with `DirectorySearcher`, so we will generally choose to use a `DirectoryEntry` instance for this work instead:

```
//user is a DirectoryEntry
user.RefreshCache(
    new string[] {"tokenGroups"}
    );

//now the attribute will be available
int count = user.Properties["tokenGroups"].Count;

Console.WriteLine(
    "Found {0} Token Groups",
    count
    );
```

Technique #1: Using an LDAP Search

The big upshot to this approach is that this technique is pretty fast and we don't have to worry about using any P/Invoke code that can be intimidating to less-experienced developers. We simply iterate through the returned attribute and build a large LDAP filter that represents each security group. Once we build the filter, we can easily search the domain for the groups and return each one. Listing 10.19 shows how we can accomplish this.

LISTING 10.19: Retrieving Token Groups with an LDAP Search

```
StringBuilder sb = new StringBuilder();

//we are building an '|' clause
sb.Append("(|");

foreach (byte[] sid in user.Properties["tokenGroups"])
{
    //append each member into the filter
    sb.AppendFormat(
        "(objectSid={0})", BuildFilterOctetString(sid));
}

//end our initial filter
sb.Append(")");

DirectoryEntry searchRoot = new DirectoryEntry(
    "LDAP://DC=domain,DC=com",
    null,
    null,
    AuthenticationTypes.Secure
    );

using (searchRoot)
{
        //we now have our filter, we can just search for the groups
        DirectorySearcher ds = new DirectorySearcher(
            searchRoot,
            sb.ToString() //our filter
            );

        using (SearchResultCollection src = ds.FindAll())
        {
            foreach (SearchResult sr in src)
            {
                //Here is each group now...
                Console.WriteLine(
                    sr.Properties["samAccountName"][0]);
            }
        }
    }

    private string BuildFilterOctetString(byte[] bytes)
    {
        //see listing 4.2 for the complete code
    }
```

We rely on the helper method called `BuildFilterOctetString` from Listing 4.2 in Chapter 4 to format the binary SID correctly into a format we can use for our filter. This technique is fairly simple and relatively elegant. It is a great solution when we want to get more information about each group than just the name. The downside is that we don't directly have access to the `DOMAIN\GroupName` format from `SearchResult`. That would require string parsing, an additional search to find the NetBIOS name of the domain from the configuration partition, or a call to `DsCrackNames` to convert the name appropriately into our chosen format. Since `DOMAIN\GroupName` happens to be one of the most widely used formats, this tends to be its major drawback.

Notice that we use the `sAMAccountName` attribute to identify the group. This is important, as the `sAMAccountName` is used for security purposes and is unique in the domain. We often see samples that parse the DN to retrieve the group's `CN`. However, a `CN` can be duplicated in different containers in the same domain, so we can accidentally introduce security flaws by assuming it is unique. Always use a unique identifier intended for security purposes when making security decisions!

Technique #2: Using DsCrackNames

A more advanced technique exists that relies on the `DsCrackNames` API and forgoes searching the directory completely. The basic premise is that we will convert all of the byte-format SIDs into their string-readable Security Descriptor Description Language (SDDL)–format equivalents and pass an entire array of them into the `DsCrackNames` API, which can convert them into another format of our choosing (DN, NT Account format, etc.).

For .NET version 1.1, this requires using P/Invoke in order to convert the SID into SDDL format. It also involves wrapping the `DsCrackNames` API. Getting everything set up requires a bit of work, but it works well once it is done.

We have included all of the P/Invoke code and wrappers needed to use this functionality in the sample code included on this book's web site. For reference purposes, Listing 10.20 includes some of the important bits.

LISTING 10.20: Using DsCrackNames to Convert TokenGroups

```
//convert to array of string SIDs
int size = this.Properties["tokenGroups"].Count;
PropertyValueCollection pvc = this.Properties["tokenGroups"];

string[] sids = new string[size];

for (int i=0; i < size; i++)
{
    sids[i] = AdUtils.ConvertSidToSidString((byte[])pvc[i]);
}

//we want to pass in the SID format and retrieve
//the NT Format names.  This utility class is
//included in our web site library samples
//groupNames contains all the converted groups now
string[] groupNames = AdUtils.DsCrackNamesWrapper(
    sids,
    this.Context.Handle,
    DS_NAME_FORMAT.DS_SID_OR_SID_HISTORY_NAME,
    DS_NAME_FORMAT.DS_NT4_ACCOUNT_NAME
    );
```

Listing 10.20 uses two wrapper classes that help us convert a binary SID to the SDDL-format SID, and wraps our call to `DsCrackNames`. We are omitting this wrapper code because it would take several pages to present and it contains mostly P/Invoke declarations. We are also going to gloss over how we came to get the RPC handle necessary for `DsCrackNames`, for similar reasons. We wish we could dive into this code, as it is interesting, but it just takes too much book space and is irrelevant for this discussion. As usual, the complete listing is available on the book's web site. We should also note that developers more familiar with the `IADsNameTranslate` ADSI interface are free to substitute this method for `DsCrackNames`. They are actually one and the same.

For version 2.0, we no longer need to use P/Invoke for converting the SID, as we can do this using the `SecurityIdentifier` class. However, if we are already using version 2.0, then we should use technique #3 instead.

Technique #3: Using the SidIdentifier and IdentityReference Classes

This last technique uses the `SidIdentifier` and `IdentityReference` classes to convert between any of the `IdentityReference`-derived formats. These classes are available only in .NET 2.0. As demonstrated in Listing 10.21, this is the cleanest and simplest solution out of the three. As long as we are after only one of the `IdentityReference` formats (of which the widely used `NTAccount` format is one), we are in pretty good shape.

LISTING 10.21: Using SidIdentifier and IdentityReference

```
//we use the collection in order to
//batch the request for translation
IdentityReferenceCollection irc
    = ExpandTokenGroups(user).Translate(typeof(NTAccount));

foreach (NTAccount account in irc)
{
    Console.WriteLine(account);
}

//Sample Helper Function
private IdentityReferenceCollection ExpandTokenGroups(
    DirectoryEntry user)
{
    user.RefreshCache(new string[]{"tokenGroups"});

    IdentityReferenceCollection irc =
        new IdentityReferenceCollection();

    foreach (byte[] sidBytes in user.Properties["tokenGroups"])
    {
               irc.Add(new SecurityIdentifier(sidBytes, 0));
           }
           return irc;
       }
```

Each technique we presented has its own advantages and disadvantages. Depending on what information we require, we might choose one or more of the options. For instance, it is entirely plausible that we will want more information about each group, yet we also will want the group's NT format name. In this case, we might combine techniques #1 and #2 or #1 and #3.

Retrieving tokenGroups from ADAM

So far, the techniques we have described have applied to Active Directory. However, we may wish to expand an ADAM user's group membership as well. Additionally, if we are using ADAM in a pass-through authentication scenario where we are authenticating Windows users, we might like to know both their Windows and ADAM group memberships.

It turns out that ADAM also supports the `tokenGroups` attribute for ADAM users and bind proxy objects. We can use the same techniques we just described for them as well. The only caveat is that we cannot refer to ADAM groups by their `sAMAccountName`, as they do not have one. We also cannot use any Windows-based techniques for resolving SIDs into names, such as techniques #2 and #3 that we just described. We must use an LDAP search, as shown with technique #1.

ADAM has an additional trick up its sleeve, though. The `RootDSE` object supports the `tokenGroups` attribute as well and will provide both the Windows and ADAM group SIDs for the currently bound user. This is especially helpful with pass-through authentication, as there is no actual object representing the user in ADAM in this scenario. What object would we query to read the `tokenGroups` attribute? The code looks approximately like this:

```
DirectoryEntry entry = new DirectoryEntry(
    "GC://localhost:389",
    null,
    null,
    AuthenticationTypes.Secure);

entry.RefreshCache(new string[] {"tokenGroups"});
```

We should instantly notice that something is strange here. We are using the GC provider with ADAM and specifying a port of 389 (our ADAM instance's LDAP port in this case). What gives?

For some reason, the LDAP provider in ADSI cannot retrieve constructed attributes off of the RootDSE object, but the GC provider can. We also do not specify the RootDSE object name in the ADsPath in this case. We are uncertain of the reason, but we know this works.

At this point, the list of group SIDs we read from the tokenGroups attribute may contain both Windows and ADAM SIDs, so we may need to do two passes to resolve them into friendly names. However, we can still use techniques #1, #2, and #3 to do this.

SUMMARY

All of the topics in this chapter center on how to manage user objects in either Active Directory or ADAM. This tends to be the most common task that developers are asked to perform. We have covered how we can efficiently find our user objects and set account properties in conjunction with domain-wide policies.

We also touched upon some of the more difficult tasks of setting passwords, as well as determining last logons and password expirations. Finally, we covered how to determine a user's group membership. While we have covered some of the most common, yet troublesome, tasks in user management, we have not exhaustively covered every possible user management scenario. Unfortunately, there are far too many to do them justice. We hope that because we showed the most irksome of the tasks, developers can extrapolate from the examples and use previous chapters as leaping points to figure out each unique scenario.