



Application security and directory services

13.1	The relationship between security and directories	254
13.2	Storing key and certificate data	259
13.3	Using digital certificates	262
13.4	Managing authorization information	268
13.5	Encrypting LDAP sessions using JNDI and SSL	270
13.6	Summary	271

Security is a vast topic that means many things to many people. In this chapter, we will look at how applications can leverage directories to provide security services. We'll also explore the relationship between directories and security services.

By the end of this chapter, you will have learned the answers to these questions:

- What is security, and how does it relate to directories?
- How can directories most effectively enable different types of security?
- How can an LDAP server be used for authentication?
- What are the limitations of private keys when used for authentication? How can certificates help?
- How can certificates be generated and stored in the directory?
- What is necessary to enable session encryption in JNDI?

13.1 THE RELATIONSHIP BETWEEN SECURITY AND DIRECTORIES

There are many misconceptions about the role directories play in securing an environment. To avoid these pitfalls, we will briefly look at what security really entails and then explore the relationship between security and directory services.

13.1.1 What is security?

In computing, *security* is a broad term. It tends to imply that information and functionality are available to those who should be able to access them and unavailable to those who shouldn't have access. Although this is true, application security can be summed up as comprising several components: authentication, authorization, privacy, availability, and integrity. Table 13.1 briefly summarizes these security-related components.

Table 13.1 Security involves a number of important components

Security component	Definition
Authentication	The process of identifying who is attempting to access a particular set of resources. When you connect to the directory and do a bind, you let the server know who you are by passing an identity and credentials that prove your identity.
Authorization	The process of evaluating whether the authenticated entity is authorized to do something to a particular resource under a defined set of circumstances. For example, after you authenticate to a bank ATM using a plastic card and matching PIN number, the machine uses a set of predefined rules to determine that you can only access the two accounts associated with your authenticated identity.
Privacy	The anonymity and secrecy of information. When you fill out a form on the Internet with personal information, you expect that data to remain private. Similarly, when you're transmitting a credit card number or password over the Internet, you need to ensure that it remains a secret from anyone who might be spying on Internet data.
Availability	Continuous, uninterrupted service. When hackers deface a web site or use denial-of-service (DOS) attacks to make a service unreachable, the availability of the information provided by that service is affected while the site is repaired.
Integrity	The assurance that information has not been changed or corrupted by an unauthorized party. Viruses can affect the integrity of a system by adding information to a document that can produce undesired, potentially damaging, results.

Figure 13.1 shows how all these components come together to form a secure computing environment.

Note that from a user's perspective, the end goal is to get access to the desired application functionality or information. What happens in between is merely overhead that allows access to happen in a way that reduces the risk of unauthorized individuals gaining access to inappropriate resources.

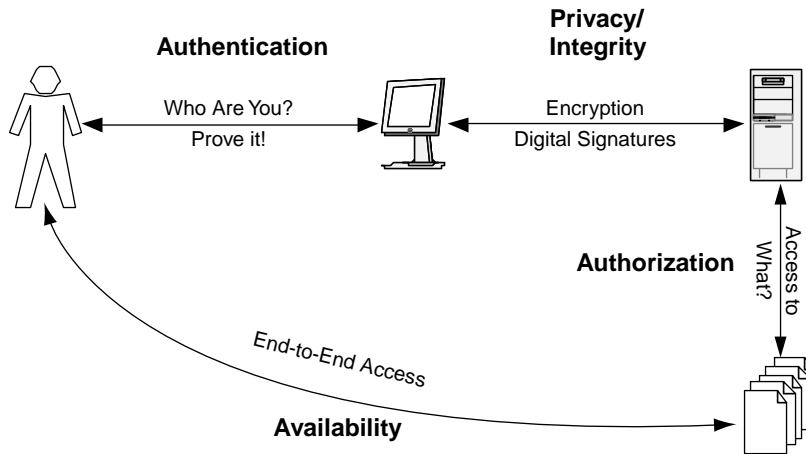


Figure 13.1 An environment that includes all the security elements

Assessing security risks to your environment

Although security includes many components, it's important to keep in mind that security is rarely perfect and should never be viewed as an all-or-nothing proposition. For example, an application that allows people to look at general-purpose news on a subscription basis will suffer little damage if a hacker finds a password and looks at the site for free. On the other hand, an application that allows someone to make financial transactions is a more enticing target and susceptible to greater damage.

The differences noted are linked directly to risk. The single most important aspect of security is risk evaluation and reduction. Different environments have different levels of need for the types of security we've defined. In the two examples we just gave, the first situation might need simple logins and passwords with few control processes, and the second might require biometric authentication mechanisms for high value transactions.

The actual level to which the various components of application security are required is also dependent on risk. Consider the following scenarios:

- A public web site that provides general-purpose news for free may only be interested in knowing that someone accessed the site, not who that person is. In this instance, authentication and authorization are fairly low needs; a much higher need exists for ensuring availability and integrity of the news service to the public.
- Two CEOs communicating about a possible merger need privacy and integrity, as well as basic levels of availability. Privacy ensures that competitors and investors cannot eavesdrop on this information, and integrity assures each CEO that they are not speaking with an impostor.

Certain top-secret information may be so sensitive that all other factors are given higher priority than availability. In some cases, the risk associated with the unauthorized access is so high that complete service interruption or self-destruction is preferable.

13.1.2 How LDAP provides security

LDAP is not an authentication service, nor is it an authorization service. LDAP also does not directly affect availability, provide any layer of privacy, or ensure data integrity outside the directory itself. Why is it, then, that people so often find their way to directory services when trying to implement solutions related to security?

Although LDAP is not an authentication service, it stores the identities and credentials used by those services, and many applications can use LDAP as an authentication service if one is not available. LDAP aids in authorization because it acts as a network-accessible data store for access control lists and policies that can be used to authorize access to resources.

Public key cryptography allows applications and people to authenticate and secure information without either party having previously shared the secret code that is used in the process. Strong data privacy and integrity are made possible by public key cryptography, and directories play a vital role in distributing the information needed to make many aspects of public key cryptography work on a large scale. We will discuss public key cryptography and the critical role directories play with this technology in section 13.2.

Finally, a properly distributed directory, when used for storing and providing access to information, can increase overall availability of the applications that depend on it.

Figure 13.2 shows how some off-the-shelf authentication and authorization services use directories to store credentials, policies, and access controls.

It is important to note that it is not necessary to have those services if the application itself is sufficiently aware of how to use that information. Using the information in the directory this way is perfectly acceptable, and applications that do so in a few

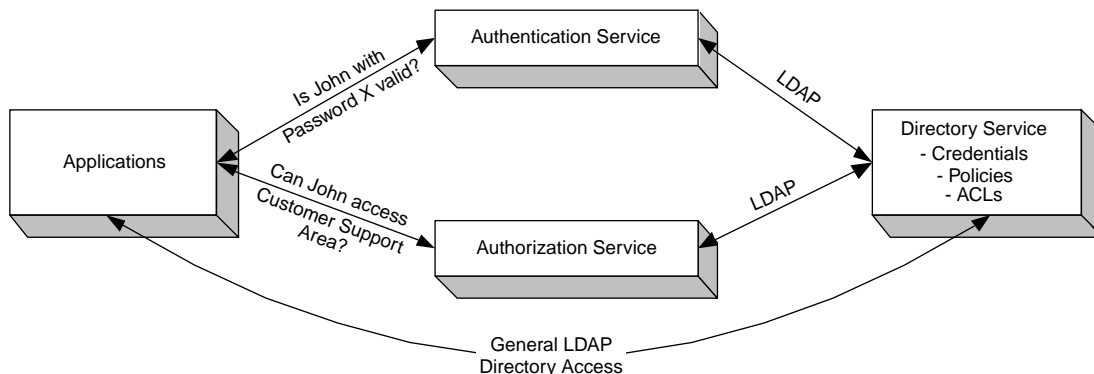


Figure 13.2 Directories provide information to external authentication and authorization services.

standard ways will be compatible with off-the-shelf variants to a great degree. We will walk through using the directory directly from your applications later in the chapter.

LDAP authentication support

Just because LDAP directories are not external authentication services does not mean they do not perform authentication. In fact, much of the recent work related to LDAP has been to strengthen the authentication mechanisms available.

The bind operation allows an LDAP client to authenticate. This in turn allows the LDAP server to authorize access to a set of entries and attributes based on any access control lists it may be using. Figure 13.3 shows this process.

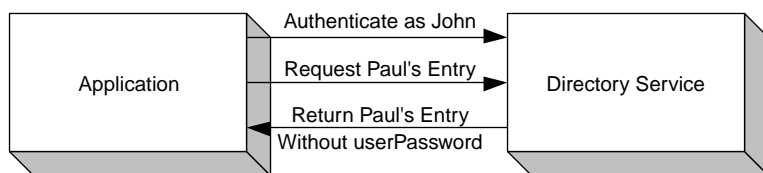


Figure 13.3 The directory service knows that the application user is John, based on an LDAP bind request. It determines that John is not allowed to see Paul's password.

LDAP has long supported anonymous and simple authentication. *Anonymous* means exactly what it sounds like, and *simple authentication* means that passwords are transmitted in the clear over the network.

Earlier LDAP vendors added a level of security in the form of SSL encryption, which allows passwords to be encrypted by the virtue of having encryption for the entire session. SSL also adds the ability to authenticate the server using certificates.

More recent standards have required support for the SASL, which allows client and server to negotiate and use different mechanisms to authenticate each other.

Using LDAP as an authentication service

Applications presenting valid credentials to an LDAP server via the LDAP bind operation are informed that the credentials passed were valid. This bit of functionality has long been used to allow applications to check the password presented by a user against the one stored in the directory. Thus, although LDAP is not a true authentication service, applications can use it to provide this service as shown in figure 13.4.

Most applications that offer to use LDAP as an authentication service do so exactly as we've just described. By using the bind operation instead of comparing the value stored in the `userPassword` attribute or other locations, the application can check the validity of credentials provided by a user without needing to worry about the lack of password encryption standards.

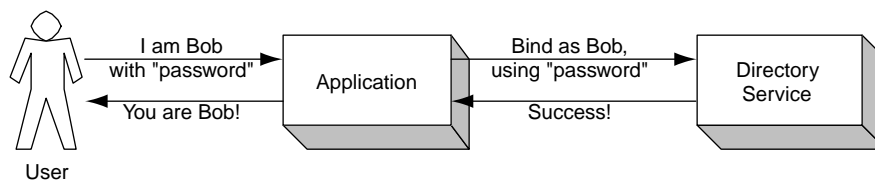


Figure 13.4 Using LDAP as an authentication service

The example in listing 13.1 shows how LDAP can be used to check the validity of a password.

Listing 13.1 LDAPLogin.java

```

import javax.naming.directory.DirContext;
import javax.naming.NamingException;
import java.util.Vector;

public class LDAPLogin {

    private static final String USERID_ATTRIBUTE = "uid";
    private static void main(String args[]) {
        if (args.length != 2) {
            System.err.println("Must give userid and password");
        }
        boolean result = LDAPLogin.login(args[0],args[1]);

        if (result) {
            System.out.println("Login as " +
                args[0] + " was successful!");
        } else {
            System.out.println("Login as " +
                args[0] + " was incorrect!");
        }
    }

    public static boolean login(String username, String password) {
        LDAPConnection lc = new LDAPConnection();
        String dn = null;
        //Filter construction for search on UID
        String filter = new String("(" + USERID_ATTRIBUTE + "=" +
            username + ")");

        try {
            DirContext dc = lc.open();
            // Perform the search
            Vector results = LDAPSearch.search("", "sub", filter,
                new String[0]);

            // If we get more than one result, the userid given was
            // not unique. If no result is returned, the userid
            // given was invalid.
            if (results.size() != 1) {
                return false;
            }
        }
    }
}

```

Set attribute for UserID comparison

Take two arguments: login and password

```

    }

    Entry entry = (Entry)results.elementAt(0);
    dn = entry.getDN();
    lc.close(dc);
} catch (NamingException ne) {
    // Error getting initial anonymous connection
    System.err.println("Error connecting to server.");
    return false;
}

lc = new LDAPConnection(dn,password);
try {
    DirContext dc = lc.open();
    lc.close(dc);
} catch (NamingException ne) {
    // Error Logging in
    return false;
}
return true;
}
}

```

Get DN from only returned entry

Open connection using DN and password

Failed connection indicates incorrect password; successful connection indicates correct password

This technique allows passwords to be checked against the contents of a Novell directory or an IBM directory, even though both of these directories use different password encryption techniques. The limitation of this technique is that many authentication mechanisms cannot be supported this way. For example, a web server does not have a person's private key, so it is unable to bind to the directory server on the user's behalf if it wants to authenticate using public key infrastructure. So, it is up to the web server or a trusted authentication service to perform the validation.

13.2 STORING KEY AND CERTIFICATE DATA

Storage of user credentials is one of the most popular uses of directory services. As we discussed earlier in the chapter, these credentials form the basis for many aspects of security, but are particularly important for authentication and authorization. Many types of credentials can be stored in an LDAP-enabled directory. The types we will discuss here include preshared secret keys and the digital certificates used in public key cryptography.

13.2.1 Preshared secret keys

The most common credentials used today are preshared secret keys. These keys include passwords that both parties need to know in advance for successful authentication to occur.

The `userPassword` attribute type is used to associate a password with an entry. However, the value of this attribute may be in any format, because the syntax is simply defined as being a binary string.

```

userPassword: my_password

userPassword: {crypt}Bf8YuicZ5nloP

```

Figure 13.5 Encrypted passwords are prefixed by the algorithm used.

Such ambiguity makes it difficult for an external authentication service to compare the password provided by an end user with an encrypted password in the directory. To mitigate this potential issue, convention dictates that a password is stored in plain text unless it is prefixed by the encryption algorithm used. Figure 13.5 shows how this process works.

Of course, the application may not be familiar with the type of encryption algorithm used to store the password. In other instances, a comparison cannot be performed using the LDAP compare operation because the client does not have access to the entire context necessary to generate the encrypted version of the shared secret. For instance, in the “crypt” example in figure 13.5, the first two characters are a salt—random characters—that impacts the final encrypted value.

Problems with secret keys

The problem with preshared secret keys is that both parties need to know the secret. Consider a situation involving two applications. The same users participate in both applications and want to use the same password for every application they use. Doing so may be fine if both applications are written and operated by the same group, but such an environment cannot be guaranteed in most instances. Where such a closed environment is not possible, trust becomes an issue.

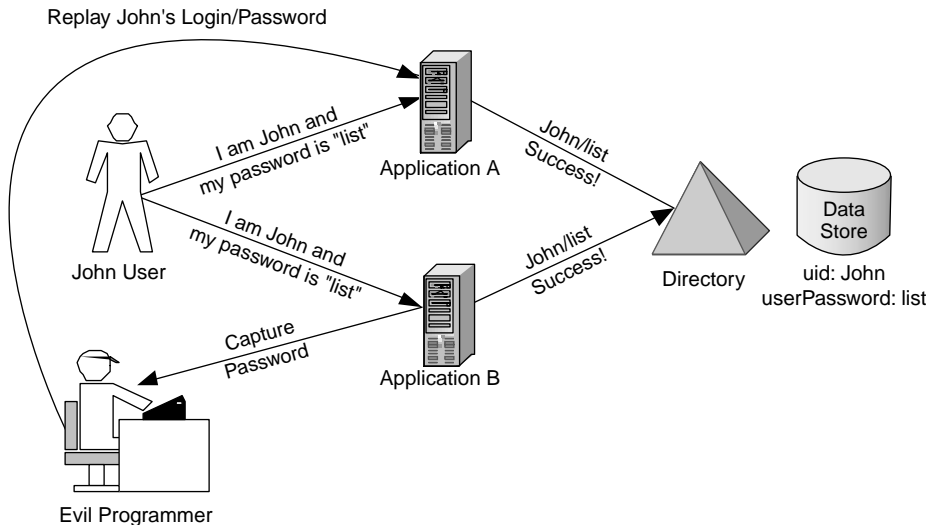


Figure 13.6 An evil developer of application B knows that he can replay passwords given to his application to application A.

It is possible that the developer of one of the two applications may decide to capture passwords provided to users of that application and relay them to the second application, gaining access via the captured accounts. Figure 13.6 shows this risk.

Although this risk has existed for some time, it has been amplified as directory environments have become more integrated. Directories that are integrated beyond organizational boundaries, such as those directory environments used in extranets, are at a higher degree of risk from these kinds of attacks. True authentication services, such as Kerberos, can mitigate some of these risks, but they are often difficult to deploy across organizational boundaries.

13.2.2 Public/private key pairs

Public key cryptography solves many of the problems that secret keys pose. Unlike secret keys, public key cryptography removes the need to preshare any private information used in the authentication process.

Instead, a user generates a pair of keys: one private, the other public. The private key is secret; only the user should ever know what it is. The public key, on the other hand, can and should be shared with the world.

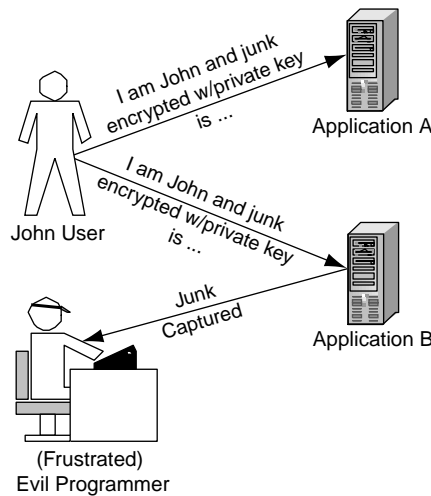


Figure 13.7 With public key cryptography, an evil programmer cannot use information sent to his application to access another application.

Only the private key can decrypt information the public key encrypts. Similarly, only the public key can decrypt information the private key encrypts. The latter instance may seem odd, considering that anyone can have access to the public key. However, if the public key is guaranteed to have been generated by John User, you can also guarantee that only John User has encrypted a piece of data. As a result, you know with certainty that it is in fact John User who is attempting to access your application.

Figure 13.7 shows that even though John is using the same private key as part of the authentication process, the information that goes over the wire and is received by the application is useless in the type of replay attack shown in the previous section.

Because it is theoretically possible for anyone to have the public key, it is now possible for anyone to authenticate John User without divulging secret information.

Problems with key pairs

The problem with this scenario is that it is often difficult to guarantee that John User really owns a particular public key. After all, if Evil Hacker forges an email to Sally Jones using John User's email address, he can tell Sally about his "new public key."

Without proper verification, Sally will have the false sense that the information sent by Evil Hacker is from John. In the next section, we look at how digital certificates can solve this problem, particularly when used with a directory.

13.3 USING DIGITAL CERTIFICATES

Digital certificates attempt to solve part of the public key distribution problem mentioned in the previous section. Together with the rest of public key cryptography, digital certificates greatly enhance many of the security components we discussed at the beginning of the chapter.

A digital certificate is basically a signed public key. By *signed*, we mean that some entity has encrypted a public key with its private key. This signing process is a guarantee by a third party that the signed public key is valid.

Consider the scenario depicted in figure 13.8. Paul knows John. Additionally, Sally knows Paul's public key. Rather than John simply sending his public key to Sally, he instead asks Paul to sign the key and generate a certificate. Now it is possible for John, Paul, or anyone else to forward this certificate to Sally. Sally can verify that it has not been changed because she trusts the validity of the signing key (Paul).

Although digital certificates solve much of the trust problem associated with using public/private key pairs for authentication, they still involve a few issues:

- Certificates require wide distribution.
- Certificates need to be associated with identity information.
- Private keys may be compromised.

Directories can be used to help resolve or mitigate all of these issues.

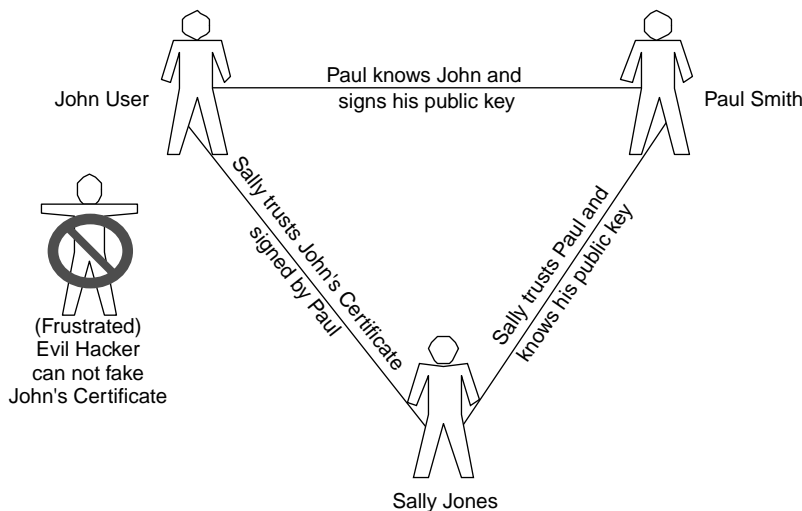


Figure 13.8 Sally trusts Paul and knows his public key. She also trusts certificates signed by Paul, including John's.

13.3.1 Creating a digital certificate in Java

In Java, a standard tool called `keytool` exists for creating public/private key pairs and managing certificates. The following command generates a public/private key pair:

```
$ keytool -genkey -alias jdoe -keyalg rsa
      -dname "cn=John Doe,dc=manning,dc=com"
Enter keystore password: password
Enter key password for <jdoe>
      (RETURN if same as keystore password): password
```

That's it. You now have a set of keys associated with `jdoe`. You can see them using `keytool`'s `list` command:

```
$ keytool -list
Enter keystore password: password

Keystore type: jks
Keystore provider: SUN

Your keystore contains 1 entry:

jdoe, Fri Dec 07 07:34:23 CST 2001, keyEntry,
Certificate fingerprint (MD5):
46:1D:C8:96:95:C4:E1:B5:07:C8:48:4F:EA:7F:C6:62
```

However, you still don't have a certificate. In order to generate a certificate, you need to do one of two things: self-sign or submit for signing. Because much of the point of having a certificate is to be able to share it and help people authenticate you, it doesn't help to self-sign unless everyone plans to trust you explicitly. The exception is during testing.

Self-signing a certificate

To self-sign a certificate, use the `selfcert` argument to `keytool`, as follows:

```
$ keytool -selfcert -alias jdoe -keyalg rsa
      -dname "cn=John Doe,dc=manning,dc=com"
      -validity 365
Enter keystore password: password
```

Now notice that when you list the certificates, the fingerprint is different:

```
$ keytool -list
Enter keystore password: password

Keystore type: jks
Keystore provider: SUN

Your keystore contains 1 entry:

jdoe, Fri Dec 07 07:46:28 CST 2001, keyEntry,
Certificate fingerprint (MD5):
65:30:7E:43:EE:DD:19:91:33:91:F0:96:F5:1D:CD:10
```

Submitting to a signing authority

If, instead of self-signing, you want to submit your key to a signing authority such as Thawte or VeriSign, you can do the following:

```
$ keytool -certreq -alias jdoe -file jdoe.csr
Enter keystore password: password
```

You can now submit the file `jdoe.csr` to a signing authority (the CSR extension identifies it as a certificate signing request). That authority will return a fully signed certificate. You store that certificate in a file called `jdoe.cer`, and import it into your Java keystore using the following command:

```
$ keytool -import -alias jdoe -file jdoe.cer -v
Enter keystore password: password
```

As you can see, working with certificates using the Java command-line tools isn't too difficult. These certificates can, in turn, be used by the Java Cryptography Extensions (JCE). With the few exceptions that we'll discuss in a moment, JCE is out of the scope of this book.

13.3.2 Storing and distributing digital certificates

Directories are a network storage point for digital certificates. Such a storage point offers applications a place to easily find a digital certificate for communications partners.

With shared secret keys, the risk rises as keys are used by more applications. In contrast, distributing digital certificates adds immense value by allowing credentials to be shared across administrative domains and even organizations without adding risk. Because LDAP-based directories can be highly distributed, storing certificates in such a directory lets you distribute certificates in a relatively standard way.

Digital certificates are commonly stored within LDAP servers in a format called X.509v3. Such a certificate includes the name of the certificate's owner, arbitrary key/value assertions, the public key of the owner, and an expiration time (see figure 13.9). Like LDAP, X.509v3 is defined using ASN.1, although it is encoded using a subset of Basic Encoding Rules (BERs) called Distinguished Encoding Rules (DERs). This encoding ensures that the same information is encoded identically each time, whereas BER allows for multiple encodings, particularly related to value length representations.

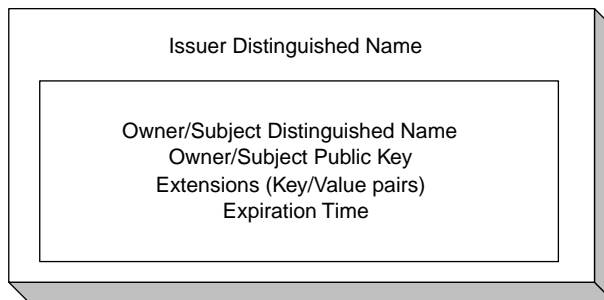


Figure 13.9
The contents of an X.509v3 digital certificate

The standard attribute type defined for storing certificates in an LDAP directory service is called `userCertificate`. The attribute is always stored in its DER-encoded, binary representation.

Let's look at what it takes to store the certificate you generated in the last section in the directory, rather than in Java's local keystore. To do so, you can use the JCE mentioned in the last section along with your existing knowledge of the JNDI.

Begin by exporting a certificate from your Java keystore—perhaps the `jdoe` certificate you issued in the last section, although any DER-formatted certificate will do. Place the exported certificate into a file called `test.cer`:

```
keytool -export -alias jdoe -file test.cer
```

Now that you have a certificate file, create the Java code in listing 13.2, which takes the contents of this file and publishes it to an appropriate entry within the directory. Note that it is possible with the JCE kit to read the certificate directly from the keystore, but this code deals with the more common occurrence of processing externally generated certificates. The output of your small program will look something like the following:

```
Found Certificate -
  Issuer : CN=John Doe, DC=manning, DC=com
  Subject: CN=John Doe, DC=manning, DC=com
  Expires: Sat Dec 07 07:46:27 CST 2002
Storing Certificate in Directory.
```

Listing 13.2 PublishCert.java

```
import java.io.InputStream;
import java.io.FileInputStream;
import java.security.cert.CertificateFactory;
import java.security.cert.X509Certificate;
import java.security.cert.CertificateEncodingException;
import java.util.Vector;

public class PublishCert {

    private static String certFile = "test.cer";

    public static void main(String[] args) {
        X509Certificate cert = null;
        try {
            InputStream inStream =
                new FileInputStream(certFile);
            CertificateFactory cf =
                CertificateFactory.getInstance("X.509");

            // Generate a certificate from the information
            // in the open file
            cert = (X509Certificate)cf
                .generateCertificate(inStream);

            inStream.close();
```

Create
certificate
from file

```

    } catch (Exception e) {
        System.err.println("Failed to Read Certificate: " +
            e.getMessage());
    }

    System.out.println("Found Certificate -");
    System.out.println("  Issuer : " + cert.getIssuerDN());
    System.out.println("  Subject: " + cert.getSubjectDN());
    System.out.println("  Expires: " + cert.getNotAfter());

    System.out.println("Storing Certificate in Directory.");

    // Get the certificate in its DER-encoded form
    // as a string of bytes.
    byte[] certbytes = null;
    try {
        certbytes = cert.getEncoded();
    } catch (CertificateEncodingException cee) {
        System.err.println("Unable to encode certificate: " +
            cee.getMessage());
    }

    // Use our Entry class from chapter 11
    Entry entry = new Entry();
    entry.setDN("cn=John Doe,dc=manning,dc=com");
    // Add the certificate value to the entry's userCertificate
    // using the ;binary attribute tag
    Vector certvals = new Vector();
    certvals.addElement(certbytes);
    entry.put("userCertificate",certvals);

    // We'll use the same value for CN and SN attributes
    Vector cnsnvals = new Vector();
    cnsnvals.addElement("John Doe");
    entry.put("cn",cnsnvals);
    entry.put("sn",cnsnvals);

    Vector ocvals = new Vector();
    ocvals.addElement("inetOrgPerson");
    entry.put("objectclass",ocvals);

    // Call the LDAPAdd.add() method from chapter 11
    LDAPConnection lc =
        new LDAPConnection("cn=admin","manager");
    LDAPAdd.add(lc,entry);
}
}

```

Set DN of
new entry

←

You can verify that the program added the certificate to the directory entry by using the following `ldapsearch` command:

```
> ldapsearch -b "cn=Test User,dc=manning,dc=com" -s
base objectclass=*
```

The returned entry in LDIF format looks something like this:

```
dn: cn=John Doe,dc=manning,dc=com
objectclass: inetOrgPerson
objectclass: organizationalPerson
objectclass: person
objectclass: top
userCertificate:: MIIB+DCCAWECEBDwQyDMwDQYJKoZIhvcNAQ...
sn: Doe
cn: John Doe
```

The `userCertificate` attribute can now easily be read using the search code you developed in chapter 11. You can use it to create a certificate from the directory, rather than from a file as you did in this example.

Associating certificates with related directory information

A digital certificate, once issued, cannot be changed. This means the certificate is a poor choice to store information about roles and other potentially dynamic information. Instead, the directory can be used to map the distinguished name of the certificate owner to additional information that can be used for authorization and other services.

Revoking compromised certificates

Digital certificates have a fixed expiration time. However, in some cases, you'll need to cancel a digital certificate before its expiration time. Certificates can be revoked for a number of reasons, such as the following:

- Someone was terminated or left a project where he or she is authorized to have a certificate from a particular issuer.
- The holder of the certificate changed his or her legal name and requires a new certificate to be issued with the new name.
- A private key has been compromised, in which case it must be revoked to prevent unauthorized users from gaining access with compromised credentials.

The need to revoke certificates is a real issue, because there is no way to know how far a particular certificate has propagated. Additionally, this need is not uncommon. Consider the number of people who have left your organization in the last year. Now add any cases in which a private key may have been compromised. Each of those situations would require that a certificate be expired prior to its noted expiration date.

You can resolve this problem a number of ways, including revocation lists and special services that focus on providing certificate validity information. Certificate authorities (the services that issue certificates) can create a CRL of certificate serial

numbers that have prematurely expired for one reason or another. Once created, this list is stored in a directory so that applications and validation services can verify the certificate being used continues to be valid.

More recently, protocols such as Online Certificate Status Protocol (OCSP) have played a more prominent role and have displaced direct lookup of CRLs in the directory. This change is the equivalent of merchants moving from using lists of bad or stolen credit card numbers to using real-time authorization systems when performing a typical credit card validation. Some of these OCSP services still perform queries against LDAP directories to locate expired certificates.

13.4 MANAGING AUTHORIZATION INFORMATION

Although we have touched on the need for directories as a storage point for authorization information, most of our focus has been on authentication. In this section, we take a closer look at authorization on two levels: directory and application.

13.4.1 Understanding access control rules

At its most basic level, authorization is the process of answering the question, “Does entity x have access to perform action y on resource z ?” The answer to this question is always yes or no, given enough context.

Servers and applications use access control rules to evaluate these authorization questions. Rules usually take this form: “Set of entities x has access to perform set of actions y on set of resources z when the following conditions are met.” The first three criteria map closely to the elements in the authorization question, but the final element is also important.

Take the following rule, for example: “Anyone who owns a record can update it.” The people allowed to update a record are not fulfilling a global role of “owner.” Instead, “owner” is a role that is associated with a particular resource.

Another example might say that “Managers can view salary information for any people they manage.” This common situation also requires that someone fulfill a role relative to the resource being changed. Just knowing that someone has the title “manager” does not mean he or she can view anyone’s salary: he or she must be the manager of the person whose information is being viewed.

If you’re familiar with basic Unix file permissions, you know that three basic roles are associated with each file:

- Owner
- Group owner
- Other

By being assigned one of these roles relative to a particular file, you can be authorized to read, write, and execute exactly the same way as others who might fulfill that role.

How these types of access control rules are evaluated depends on whether you are protecting a data store (such as a directory) or an application.

13.4.2 Directory authorization

The need to protect information in the directory from being viewed or changed by unauthorized individuals is a type of authorization necessary in virtually every directory services environment. Unfortunately, no current standard exists for setting access control on information in an LDAP-enabled directory. There is also no current way for applications to determine in advance whether a desired operation will be allowed prior to an actual request.

Although no standard exists, most implementations provide some of the same types of access controls. These access controls include the ability to restrict certain parts of the directory tree to particular users and groups. Administrators can also restrict the attributes that directory users can access or update, in some cases based on their own relation to the entry being accessed or modified. For example, a user might authenticate to the server as a manager and be given special rights to manage entries in which the `manager` attribute is set to the authenticated user's distinguished name. Server documentation is the best source for server-specific access control information.

13.4.3 Application authorization

Applications use directory information for authorization as well as authentication. This is an excellent way to reuse information about people, groups, and accounts within the directory to enhance security. For example, if the directory entries about people contain a flag that indicates whether a person is a contractor or permanent employee, an application could allow or disallow access to parts of the application based on that information.

Such access control would be done within the application at some point following authentication by the user. With the distinguished name of the user known from the authentication process, it is only necessary to read that user's information to determine whether they are authorized to perform a particular action. If the test is to determine whether a user is a member of a group, you may need to search on the group to determine if the user is a member. Each of these options can be performed easily using the search techniques explored in this chapter and chapter 11.

It has become very popular to not only use the directory as an external source of authentication and authorization, but also to externalize considerable authentication and authorization logic from the application by using access management products. These products allow both user and policy information to be stored outside the application, often in a directory service, so that this information can be reused across multiple applications. The idea is to reduce the amount of security-related programming necessary in enterprise applications and portals.

Be aware that although these policies are often stored in a standards-based repository, such as an LDAP-enabled directory, the content of the policies is often

proprietary and usable only by a particular access management product. However, user and group information that follows standard schemas is not proprietary and usually reusable.

13.5 ENCRYPTING LDAP SESSIONS USING JNDI AND SSL

Encryption is important to protect the privacy of the data being transmitted. For the most part, SSL is the standard for encrypting data sessions over the Internet. Most directories support LDAPS, which is basically LDAP encapsulated within an SSL session. Using Sun's Java Secure Socket Extension (JSSE) toolkit, JNDI allows an LDAP server to connect securely over a network using SSL. Figure 13.10 shows how these pieces fit together.

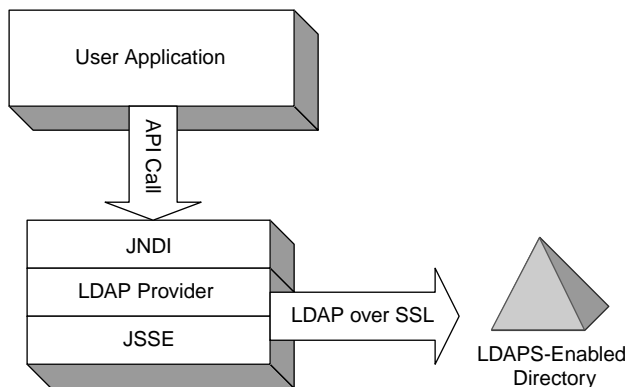


Figure 13.10
The various layers that fit together to enable LDAP over SSL in Java include JSSE and JNDI.

Wonderfully, you can do all this by simply changing a few lines of code when you create the initial `DirContext` object. Make the change shown in bold in listing 13.3 to the `LDAPConnection` class (from listing 11.1) to encrypt the sessions for all the examples in part 3 that use the class to open connections. No other code changes are necessary, because the fundamental LDAP operations are unchanged.

Listing 13.3 `LDAPConnection.java`

```
public DirContext open() throws NamingException {
    Properties env = new Properties();
    env.put(DirContext.INITIAL_CONTEXT_FACTORY,
           "com.sun.jndi.ldap.LdapCtxFactory");
    env.put(DirContext.PROVIDER_URL, "ldap://" + host +
           ":" + port);
    if (dn != null) {
        env.put(DirContext.SECURITY_PRINCIPAL, dn);
        env.put(DirContext.SECURITY_CREDENTIALS, password);
    }
}
```

```
env.put(Context.SECURITY_PROTOCOL, "ssl");
DirContext dirContext = new InitialDirContext(env);
return dirContext;
}
```

Note that all you have to do is add a line indicating that you want to use the `ssl` security protocol, and JNDI takes care of the rest. All your directory operations will now be private.

13.6 *SUMMARY*

In this chapter, you learned a considerable amount about application security and how it relates to directories. We discussed authentication using LDAP as a service for verifying passwords, and we presented an example that lets you validate a login and password against a server. You also saw how you can use Java to create and publish digital certificates. Finally, we explored how easy it is to enable SSL with Sun's LDAP provider for JNDI.