Make Access Security a Joy

COM+'s role-based security approach enormously simplifies implementing and maintaining your application's security.

by Juval Lowy

What You Need

- Windows 2000
- Visual C++ 6.0
- Microsoft Platform SDK

Perhaps nothing epitomizes the difference between developing a distributed enterprise-wide system using COM+ and developing one using DCOM like the COM+ security service. COM+ provides an easy-to-use administrative security infrastructure that makes configuring and enforcing security a joy. COM+ security is based on a new, intuitive security concept called role-based security, which enormously simplifies managing and configuring your application's security. COM+ security enables you to leave all security-related functionality outside the scope of your components and configure security administratively. Roles are used for access control, and declarative attributes are used for the rest of the security settings.

A *role* is composed of a symbolic category of users who share the same security privileges. When you assign a role to a component, you are granting access to that component to whomever is a member of that role. When a user who was not granted access (is not a member of the role) tries to invoke a method on that component, the method invocation will fail with the error code E_ACCESSDENIED ("Permission Denied" in Visual Basic). COM+ lets you assign more than one role to a component, and in fact, you can assign roles to interfaces and methods as well. You configure your access security policy administratively using the COM+ Explorer. This article explains how to use role-based security, both declaratively and programmatically, and how to design role-based security effectively. In future articles, I will address other security aspects, such as security pitfalls, application-level security settings, and how security ties in with other COM+ services.

The best way to explain role-based security is simply to demonstrate it. Suppose you have a COM+ bank application, which contains one component—the bank component. The bank component supports two interfaces that allow users to manage bank accounts and loans (download Listing 1 from the *VCDJ* Web site; see the Go Online box for details).

During the requirements-gathering phase of product development, you discover not every user of the application can access every method. In fact, there are four kinds of users: a bank customer, a bank teller, a bank loan consultant, and a bank manager. The bank manager is the most powerful user and can access every method on every interface of the component. The bank teller can access all the methods of the IAccountsManager interface but is not authorized to deal with loans and cannot access any method of the ILoansManager interface. Similarly, the loan consultant can access any method of the ILoansManager interface, but cannot access any method of the IAccountsManager interface. A consultant is never trained to be a teller.

Finally, the bank customer can access some of the methods on both the interfaces. The customer can transfer funds between accounts and determine the balance on a specified account, but cannot open a new account or close an existing one. The customer can make a loan payment, but cannot apply for a loan or calculate the payment.

If you enforced this set of security requirements on your own, you would face a programming nightmare. You would have to remember who is allowed to access what and couple the objects to the security policy. The objects would have to verify who the caller is and

whether the caller has the credentials to access the objects. The resulting solution would be fragile—imagine the work you would have to do if these requirements changed.

Fortunately, COM+ makes managing such security-access policy a joy. After importing the bank component to a COM+ application (be it server or a library application), you need to define the appropriate roles for this application. Every COM+ application has a folder called Roles. To add a new role, expand the Roles folder, right-click on it, and select New from the context menu. Type Bank Manager in the dialog box that comes up and click on OK. In the Roles folder, you should see a new item called Bank Manager. Add the remaining roles: Customer, Teller, and Loan Consultant. The application should look now like Figure 1.

You can now add users to each role. Every role has a Users folder, which lets you add registered users from your domain or work group. For example, navigate to the Users folder of the Customer role, right-click on the folder, and select New from the context menu. This will bring up the standard Windows dialog box for selecting users. Select the users that are part of the Customer role, such as "Joe Customer." You can populate this role and the remaining roles in the bank application with their users.

The next step is to grant access to components, interfaces, and methods for the various roles in the application, according to the bank application's requirements. Display the bank component properties page, and select the Security tab. It contains a list of every role defined for this application. Check the Manager role to allow a manager access to all the interfaces and methods on this component. Ensure that the "Enforce component level access check" check box under Authorization is selected. This check box is your component-access security master switch, instructing COM+ to verify participation in roles before accessing this component. Next, configure security for the interface level. Display the IAccountsManager interface properties page, and select the Security tab.

Select the Teller role to grant this role access to all the methods of this interface. The upper portion of the interface Security tab contains *inherited roles*, which are roles that were granted access at the component level and therefore have access to this interface, as well. Even if the Bank Manager role is not checked at the IAccountsManager interface level, that role can still access the interface.

Similarly, you can configure the ILoansManager interface to grant access to the Loan Consultant role. The Bank Manager should also be inherited in that interface. Note that the loan consultant cannot access any method

THE BANK APPLICATION'S ROLES FOLDER

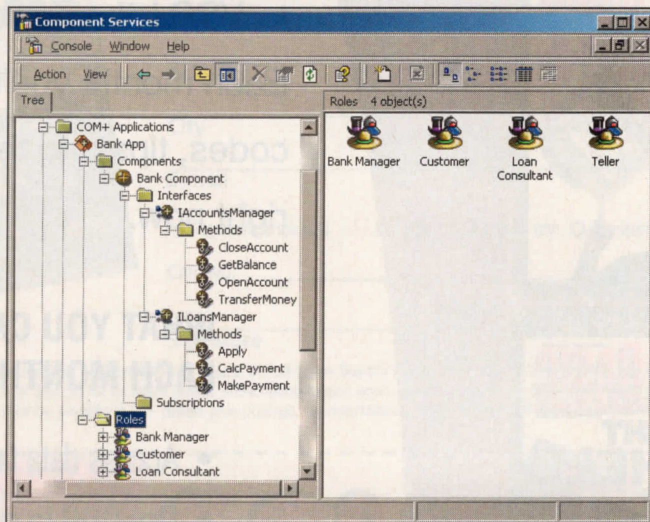

Figure 1 | There are four roles—Bank Manager, Customer, Loan Consultant, and Teller—listed in the right pane. Each role represents a group of users directly from the application domain. The role name is just a string. The developer is responsible for assigning meaningful role names and configuring the role as semantic at the component, interface, and method levels.

on the IAccountsManager interface, and the teller cannot access any method on the IAccountsManager interface, just as the requirements stipulate.

Finally, you can configure access rights at the method level. A customer should be able to invoke the GetBalance() and TransferMoney() methods on the IAccountsManager interface and MakePayment() method on the ILoansManager interface. Granting access at the method level is very similar to the interface or component level. For example, to configure the GetBalance() method, display this method's properties page, select its Security tab, and check the Customer role. The method Security tab shows inherited roles from the interface and component levels. COM+ displays roles inherited from the component level using a component icon, and roles inherited from the interface level using an interface icon.

Because of the inherited nature of roles, you can use a simple guideline for configuring them: Put the more powerful roles upstream and the more restricted roles downstream.

Role-Based Security Benefits

COM+ role-based access control gives you (for most practical purposes) ultimate flexibility with zero coding because access control at the method level is usually granular enough. Role-based security offers a scalable solution that does not depend on the number of system users. Without it, you would have to assign

access rights for all objects and resources manually, and in some cases you'd have to impersonate users to find out whether they have the sufficient credentials. Configurable role-based security is an extensible solution that makes modifying a security policy easy. Like any other requirements, the security requirements of your application are likely to change and evolve over time, and now you have the right tool to handle it productively.

Roles map directly to terminology from your application domain. As part of the normal development life cycle, you should not only discover interfaces and classes during the requirements analysis phase, but also aspire to discern user roles and privileges. Focus your analysis efforts on discovering roles users play that distinguish them from one another. As you have seen in the bank example, roles work very well when you need to characterize groups of users based on the actions they can perform.

But roles don't work very well when access decision rests on the identity of a particular user (for example, only if the bank teller is Mary Smiling do you allow the teller to open an account), or on some special information regarding the nature of a particular piece of data (for example, bank customers cannot access accounts outside the country). Role-based security protects access to middle-tier objects. Middle-tier objects should be written to handle any client and access any data. Basing your object behavior on particular user identities makes your system inflexible and unscalable, and forcing your objects to know intimate details about the data does the same.

When you want to design effective roles, try to avoid a complex role-based access policy that has many roles and users allocated to multiple roles. Role-based security should be a straightforward solution, with crisp distinctions between roles. The simpler and clearer the solution, the more robust and maintainable it will be. For example, avoid defining roles with ambiguous criteria for who belongs to them. Your application administrator should be able to map users to roles instantly. Use meaningful, self-evident role names, borrowing as many terms and vocabulary directly from the application domain as possible. For example, Super User is a bad user name whereas Bank Manager is a good name, despite the fact that your application would function just fine with the former.

Occasionally, you will be tempted to define numerous roles, trying to model a real-life situation as closely as possible. Maybe different branches of the bank have different policies for what tellers are permitted to do. Try to collapse roles as much as possible. You can do this either by re-factoring your interfaces (deciding on what methods will be on what interface and which component supports which interface) or defining new interfaces and components. Avoiding numerous (more than a dozen) roles will also improve performance. If you have many roles, COM+ must scan the list to determine whether the caller is a member of a role that has access for each call.

Programmatic Role-Based Security

Sometimes administrative role-based security it not granular enough for the task at hand. Consider a situation where your application maintains a private resource (such as a database) that does not expose any public interfaces directly to the clients. You still want to allow only some callers of a method to access the resource while denying access to other callers who are not members of a specific role. The second (and more common) situation is when a method is invoked on your object and you want to know whether the caller is a member of a particular role so you can handle the call better.

For instance (consider the bank example), one of the requirements is that a customer can transfer money only if the sum involved is less than $5,000, whereas managers and tellers can transfer any amount. Declarative role-based security goes down only to the method level (not the parameter level), and can only assure you that the caller is a member of at least one of the roles you granted access to.

To implement the requirement, you must find out the caller's role programmatically. Fortunately, COM+ enables you to do just that. Every method call is represented by a COM+ call object. The call object implements an interface called ISecurityCallContext, obtained by calling CoGetCallContext(). ISecurityCallContext provides a method called IsCallerInRole(), which lets you verify the caller's role membership. Download Listing 2 to see how to implement the new requirement using the call object security interface. (Note that IsCallerInRole() is available on IObjectContext as well, a legacy from MTS.)

Security in a modern system is not an afterthought or a nice-to-have ("we will do it in the next release") feature, nor is it the realm of a knowledgeable few. You must design security into your COM+ application and components from day one, much the same way you design concurrency, threading model, factor out your interfaces, and allocate interfaces to components. VCDJ

RESOURCES

- COM+ Security under component services in the MSDN
- "Programming COM+ Security" by Yasser Shohoud (*VCDJ* Middle Tier column May 2000)

About the Author

Juval Lowy is a seasoned software architect. He spends his time publishing and conducting classes and conferences on component-oriented design and COM/COM+. He was an early adopter of COM, and has unique experience in COM design. This article is based on excerpts from his upcoming book *COM+ and .NET* (O'Reilly) scheduled for release in spring 2001. E-mail him at idesign@componentware.net.

GO ONLINE

Use these DevX Locator+ codes at www.vcdj.com to go directly to these related resources.

VC0102 Download all the code for this issue of *VCDJ*.
VC0102PC Download the code for this article separately. This article's code includes the sample bank application.
VC0102PC_T Read this article online. DevX Premier Club membership is required.
Want to subscribe to the Premier Club? Go to www.devx.com.