8 Tips for Avoiding COM+ Security Pitfalls

COM+ security is powerful, but not perfect. These tips can help you make the most of it.

by Juval Lowy

WHAT YOU NEED

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

In last month's Productive COM+ column, you learned how COM+ security enables you to leave almost all security-related functionality outside your components. You learned to use roles for access control and declarative attributes for the rest of the security settings. Configuring security through an administrative tool makes it easier to manage and maintain your application's security policy.

COM+ security solves some classic distributed-computing problems difficult to solve on your own or requiring a lot of development and testing effort. Even with a single-machine application, COM+ security can provide elegant solutions for administration and configuration issues. All you have to do is understand a few simple security concepts and configure your application—COM+ does the rest. But no service is flawless, and security is no exception. In this article you'll learn about eight COM+ security pitfalls, and whenever possible, you'll find out how to avoid them.

This article assumes you're familiar with COM+ security and that you understand terms such as role-based security, authentication, and authorization. See Resources for articles that cover COM+ security basics. If you're unfamiliar with these terms, read those articles first.

CHECK THE SECURITY LEVEL

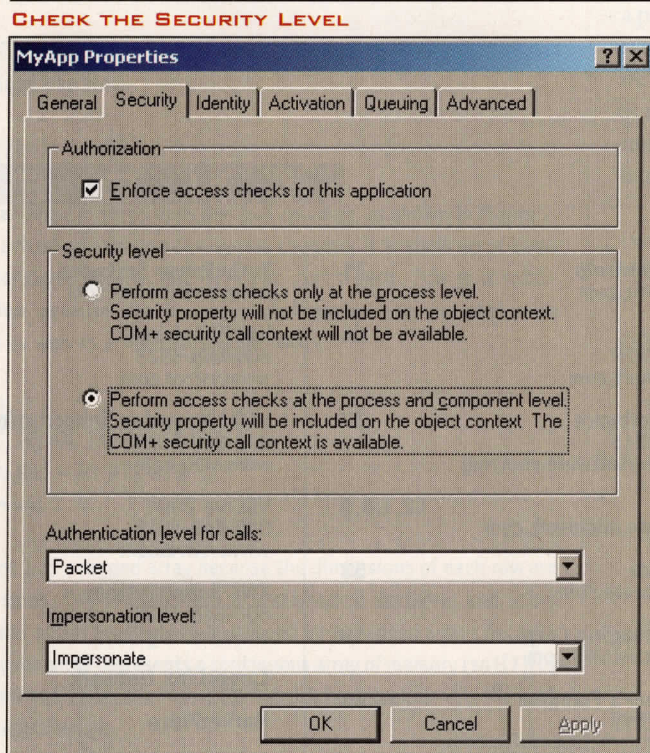

Figure 1 | On the application Security tab, always enforce access checks at the application level and set the security level to perform access checks at the application and component levels.

1. Don't Call CoInitializeSecurity()

If you're familiar with DCOM security, calling CoInitializeSecurity() is second nature. In the old DCOM days, CoInitializeSecurity() opened the door to manageable security, and any properly written DCOM server called CoInitializeSecurity() to ensure the required security levels. A configured COM+ component, however, can't call CoInitializeSecurity() because COM+ loads any configured component in a hosting process. If the component is part of a server application, COM+ calls CoInitializeSecurity() when COM+ creates the process, with the application global security settings as parameters. If the component is

part of a library application, the hosting process had to call CoInitializeSecurity() before doing anything else with COM, or else COM would have called CoInitializeSecurity() for the hosting process.

CoInitializeSecurity() might be an issue when porting an existing DCOM server to COM+. If the ported server used CoInitializeSecurity(), you'll have to remove the call from the code, look at the parameters for CoInitializeSecurity(), and configure the global application security settings accordingly.

2. Don't Perform Access Checks Only at The Process Level

The Security level properties group is in the center of every application's Security tab (see Figure 1). This Security level is the role-based security master switch for that application's components. Setting the master switch to the upper position ("Perform access checks only at the process level") disables all role-based security configurations at lower levels, including component, interface, and method (see Figure 2). Performing access checks only at the process level allows all calls through, regardless of the lower levels' settings, as long as the calls pass the generic application-level authorization check, which I'll discuss next.

A side effect of performing the security checks only at the process level is you won't be able to make any programmatic role-based security checks inside your components because the security information won't be part of the call object. You won't be able to access interfaces such as ISecurityCallContext. When you create new objects, COM+ ignores their security requirements in deciding which context to activate them in.

Because you can turn off role-based security at the component level (see Figure 3), you shouldn't disable role-based security at the application level. Instead, disable security only for those components that don't require it. As a rule, always enable security at the highest level possible and disable it at the lowest level possible.

3. Don't Disable Application-Level Authorization

At the top of your application's Security tab is the Authorization checkbox (see Figure 1). When you check the "Enable access checks for this application" checkbox, COM+ verifies that the calling client is a member of at least one role defined for the application (remember, you define roles at the application level). This step accelerates access denials to callers because if the caller isn't a member of any role defined for the application, it's pointless to proceed and check at lower levels to determine whether the caller should be granted access.

If you don't check the authorization checkbox—even if a component is set to use and enforce role-based security—COM+ allows in all calls to that component, regardless of the caller's identity and role membership. This is dangerous because your components might require declarative access control, and they don't have another mechanism in place to implement access-control requirements.

In addition, when you leave the authorization checkbox unchecked, the component security tab

Set the Process Level Security Check

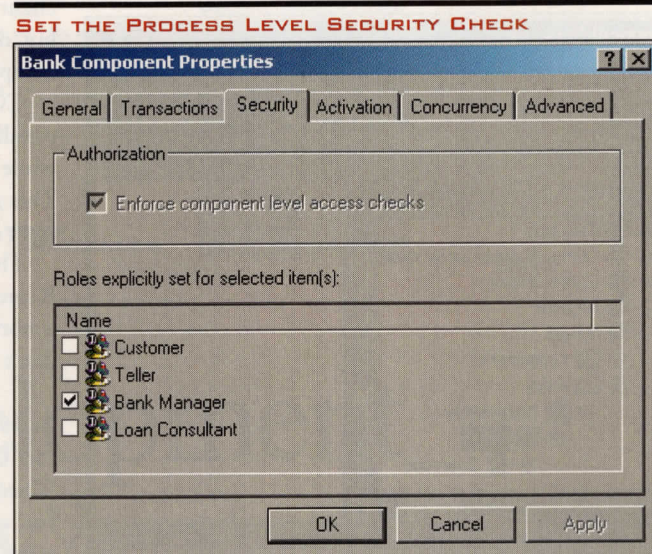

Figure 2 | When you set the security access check at the process level, you disable and ignore all security configurations at the component, interface, and method levels.

Deactivate Role-Based Security

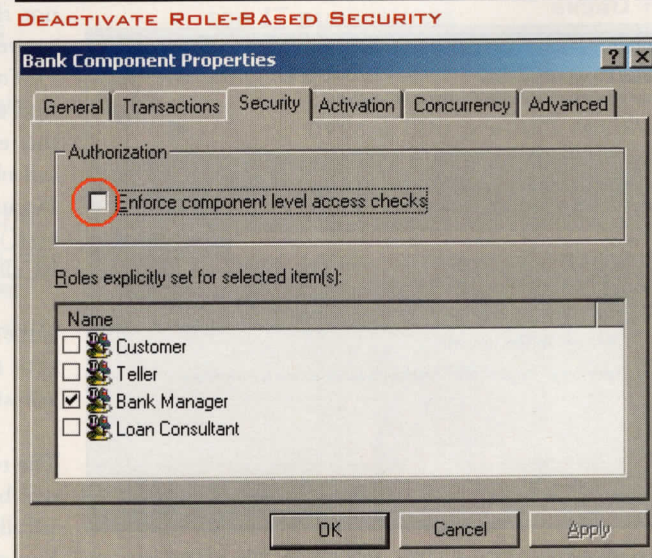

Figure 3 | Because you can disable role-based security at the component level, you shouldn't disable role-based security at the application level.

won't be grayed out. The component, interface, and method level Security tab will appear to be functioning, but they are ignored. Always leave application-level authorization checked.

4. Enable Application-Level Authorization Carefully

As you just learned, you should always enable application-level authorization. But what happens if your application has a number of components requiring role-based security and a few other components that don't? The components not requiring access control might be serving a different set of clients altogether. The problem with application-level authorization is when a call comes into an application, COM+ verifies that the caller is a member of at least one role defined for this application. If the caller isn't a member, COM+ denies the caller access—even if the caller is trying to access a component not requiring access control.

Fortunately, you can outsmart COM+ by defining a new role in your application, called Place Holder, and adding an Everyone user to it (see Figure 4). Now all the callers are members of at least one role, and components that don't require role-based security can accept calls from any user.

OUTSMART COM+

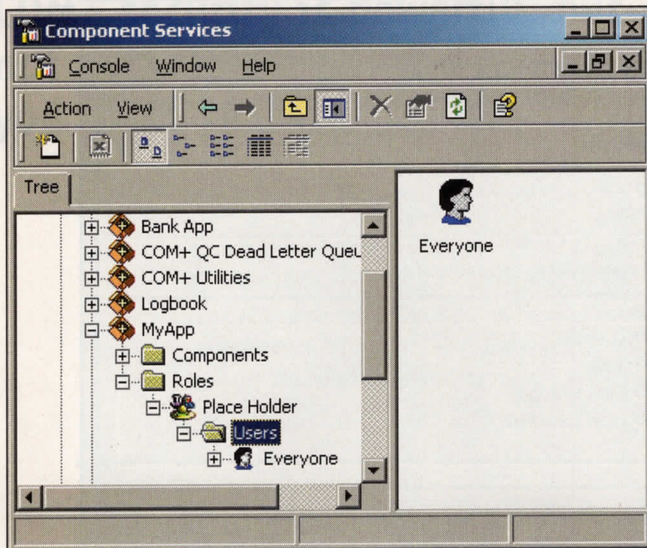

Figure 4 | By adding a role as a placeholder for the "Everyone" user, you allow calls from users that are not part of any other role to access components that don't require access security.

EXPORT USERS

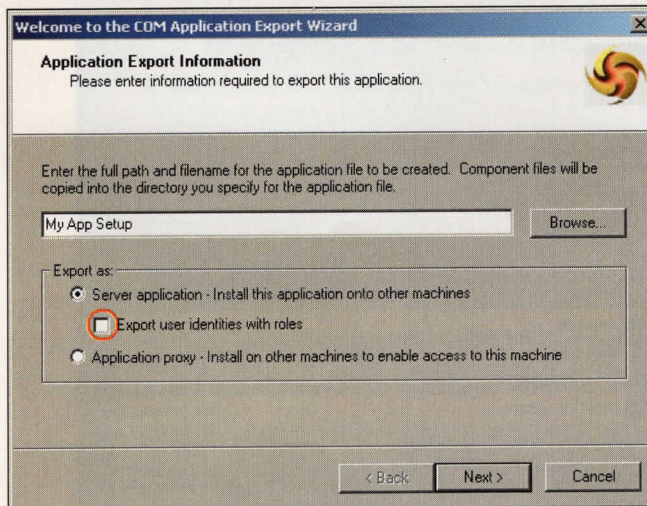

Figure 5 | When exporting a COM+ application, the wizard lets you export the users associated with the roles. Use this setting with care, as users are deployment-site specific.

5. Avoid Sensitive Work at the Object Constructor

Imagine a situation where, using role-based security, you grant a client access to one component in your application, component A, but not to another component, B. When the client tries to create component B, COM+ creates the object, but only lets the client access the IUnknown methods of component B and denies access to any other interface.

COM+ does this by design to avoid a common DCOM pitfall: allowing a client to create a new object in a new process while forgetting to grant the client access to the object inside. This mistake results in a zombie process because the client can't even call IUnknown::Release() on the object it created.

The preceding COM+ scenario implies that the component B constructor is actually executing code on behalf of a client that is not allowed to access the component. To avoid this pitfall, don't do any sensitive security work in the object constructor.

6. IsCallerInRole() Returns TRUE When Security Isn't Enabled

When administrative role-based security isn't granular enough, you can use programmatic role-based security to verify a caller's membership in a particular role. For example, imagine a bank component where one requirement is that a customer can only transfer money if the sum involved is less than $5,000, while a bank teller can transfer any amount. Declarative role-based security goes down only to the method level—not the parameter level—so role-based security can only assure you that the caller is either a teller or a customer.

To implement the preceding requirement, you must find out the caller's role programmatically. COM+ makes this easy. 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(), defined as:

```
HRESULT IsCallerInRole(
BSTR bstrRole,
VARIANT_BOOL* pbInRole);
```

The role is indicated by the string bstrRole, and *pbInRole will be true if the caller is a member and false otherwise. IsCallerInRole() lets you verify the caller's role membership. You must enable properly role-based security, however, for ISecurityCallContext::IsCallerInRole() to return accurate results. In both the following two scenarios, IsCallerInRole()

always returns TRUE regardless of the actual caller's role membership. Scenario 1: You enable role-based security at the application level, but you don't enforce it at the component level (see Figure 3); calls to ISecurityCallContext::IsCallerInRole() from within the component always return TRUE. Scenario 2: At the application level, you don't enforce authorization; all calls to ISecurityCallContext::IsCallerInRole() always return TRUE, even if you enforce component-level access checks.

IsCallerInRole() misbehaves in both a library and a server application when either scenario takes place. To overcome this, you should always call another method of ISecurityCallContext to verify that security is enabled before checking role membership. This method, called IsSecurityEnabled(), is defined as:

```
HRESULT IsSecurityEnabled(VARIANT_BOOL* pbIsEnabled);
```

Download Listing 1 to learn how to use IsSecurityEnabled() in the bank example (see the Go Online Box for details).

7. Avoid Exporting Users With Roles

It's important to understand that roles are an integral part of your design, while allocating users to roles is part of your application deployment. Your application administrator should perform the final allocation of users to roles at the customer site. When deployed, your application should already have all the roles predefined in it, and the administrator should only allocate the users to roles. When you export a COM+ application, the application export wizard gives you the option of exporting the user identities with the roles (see Figure 5).

PROTECT YOUR SETTINGS

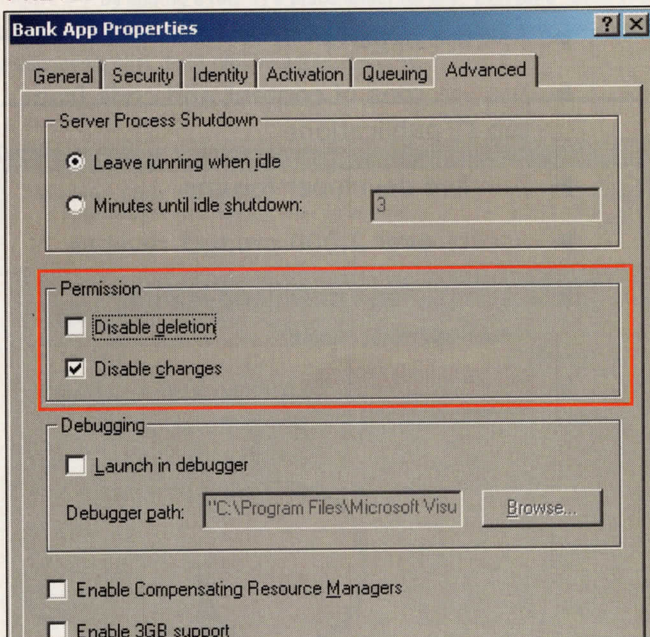

Figure 6 | Disabling and enabling changes to your application isn't password protected, which opens the door for those who aren't product administrators to change your security settings.

The application administrator should only use this option when making cloned installations at a particular customer site, from one machine to another. In fact, exporting user information from one deployment site to another might constitute a security breach. Customers don't want employee lists, user names, or the role they play in the organization available anywhere, let alone at another company's site. As a developer, exporting users with roles isn't useful to you.

RESOURCES

- COM+ Security under component services in the MSDN
- *VCDJ*'s Productive COM+ columns, including "Make Access Security a Joy" by Juval Lowy, February 2001

8. Disabling Changes to the Application Configuration Isn't Password Protected

Every COM+ application has a Permission properties group on the Advanced tab of its Properties page (see Figure 6). Selecting "Disable changes" prevents anybody from altering your application settings, as well as preventing people from making changes at the component, interface, and method levels. You might think this is a good way to preserve your security settings and access policy, but this checkbox isn't password protected—so anyone with administrative privileges can modify your precious security settings and introduce security gaps in your application. Customer-side administrators might be tempted to change your security settings to accommodate something else in the system or fool around with your application. This checkbox is there for a reason, and I don't understand why Microsoft didn't make it password protected as well. With this in mind, your company's product administrator should routinely verify that the security configurations haven't changed since the original deployment. VCDJ

About the Author

Juval Lowy is a seasoned software architect. He spends his time consulting, publishing, and conducting classes and conference talks 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 up-and-coming 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.

VC0103 Download all the code for this issue of *VCDJ*.

VC0103PC Download the code for this article separately.

VC0103PC_T Read this article online. DevX Premier Club membership is required.

Want to subscribe to the Premier Club? Go to www.devx.com.