Global Information Assurance Certification Paper

Adventures in COM Security

Ryan Kelly

March 01, 2002

SANS GSEC

RPC at a glance

Logon Sessions, profiles, and Network authentication make up the infrastructure that lies beneath COM and COM+. Microsoft's RPC model sits between COM and other protocols such as TCP, UDP, SSL, NTLM, etc.[1] MSRPC doesn't care which network protocol is being used, but the client and server must agree on TCP, UDP, and so on. The same goes for authentication protocols such as SSL, Kerberos, and the like. With the server being able to listen for multiple protocols simultaneously, the client must choose a protocol that the server will respond to.

MSRPC uses two different types tracking, static and dynamic. Static is used by default, this is important to note. Dynamic tracking causes RPC to inherent the way the operating system is working. Anonymous authentication is supported by MSRPC when local communication is taking place. Non-local communication does not.

Overview of COM Security

COM offers both programmatic and declarative security. Declarative puts the responsibility for deciding who or what has access to an application on the administrator. This is beneficial because there is no need for the code of an application to change as security policies do.

The administrator is also responsible for configuring security settings in a Windows NT environment. The administrator knows the user and group accounts as well as the physical location of all the computers in the network. Learning how to administer security in a COM-based application is inherently similar to the Windows NT security model.

The Windows NT security model is based on user accounts that may or may not belong to a particular group. A SID, or Security ID, is unique for each user and group account. A security token is created when a user logs on to a Windows NT domain. The security token contains the SID of the user's account and the SID of any group account the user belongs to. Each process that the user runs carries the security token. The system checks the security token to make sure the user is authorized each time a user attempts to access a particular resource.

A list of SIDs associated with a resource that tells the system which users have access rights, is known as a discretionary access control list (DACL). An administrator manages this list with a DACL editor. A good DACL editor for configuring access rights to a distributed application is DCOMCnfg.exe.

Configuring Distributed COM Security

Distributed COM is enabled by default. If you disable it, all activation requests from a remote client will be denied by the SCM of the local machine. A system-wide authentication level is also configurable. As you know, authentication is the process of verifying the user's identity. The COM security layer on the server examines the users

credentials when a user attempts to activate an object across the network. This is to prevent impersonation.

A loadable security module called a security support provider (SSP) and the RPC layer performs the authentication. All SSPs are written to a standard API called the Security Support Provider Interface (SSPI). Using the SSPI, third-party vendors can create custom SSPs for a Windows NT environment. The Windows NTLM SSP comes with Windows NT 4. Windows NT 5 includes the NTLM SSP as well as the Kerberos SSP.

Both the client computer and the server computer have their own default authentication levels. The RPC layer uses the higher authentication level when two computers aren't configured to use the same scheme. Higher levels of authentication will degrade performance.

Developers often do not write application code that address authentication in higher-level programming languages. This is good for application code because it can be maintained easier. After an application has been put into production, a system administrator can easily change to a different authentication level or a new security service provider. Sometimes programmatically changing the authentication level can be beneficial. The code to talk to COM's API must been written in a low-level language such as C++. The COM library exposes an API that allows you to adjust the authentication level and other security settings on a component-by-component basis. Creating a DLL written in C++ will provide you with more flexibility to the servers and the client applications that are programmed using a high-level language, such as Visual Basic.

The impersonation level protects the client, while the authentication level allows a component to determine the identity of the caller. Since authentication allows a client to effectively pass it's security credentials across the network, the impersonation level determines how the server can use the client's token. Higher levels of impersonation allow the server to do more with the client's security credentials.

A high-level programming language cannot easily change the impersonation level at run time, as with the authentication level. There are three different configuration options for authentication and impersonation. The first option is to use the default levels, which are Connect for authentication and Identify for impersonation. The second approach is to configure the machine-wide levels on each computer for some other level or levels. The third option typically requires some help from another language, such as C++.

Configuring the Server's Application ID

Each AppID represents a COM application. A COM application is a collection of CLSIDs whose objects are all loaded into a single-server process. When you configure the AppID for a server, you can specify which users can access the application and Windows NT user account will be used to serve as the identity of the server process. The user account is known as the server's security principal.

To configure the settings for an AppID with DCOMCnfg.exe, you must select the AppID on the Applications tab. Each entry in the Applications list box is a local AppID. Remember that DCOMCnfg.exe often creates AppIDs using information from the first CLSID it finds in the Registry. This practice can result in confusion because the

application's description looks as if it pertains to the CLSID, not to the server application. When DCOMCnfg.exe creates an AppID, uses the same GUID that defines the first CLSID. However, once you find the correct COM application in the list, you can select Properties. This opens another dialog box, in which you can modify the attributes for the AppID.

To configure the server's security settings, you can use the Security tab, as shown in Figure 1.2. This tab offers three security configuration choices; you can set and modify each one using a standard Windows NT DACL editor. All access permissions are set in terms of Windows NT user and group accounts. A user must have access permission to activate and call methods on objects. A user must have these permissions explicitly or through a group in order to use the server. The SYSTEM account must be included in the access control DACL (at least in Windows NT 4). If you forget this, you're application won't function properly.

A user must have launch permissions to launch the application isn't already running. This configurable option lets an administrator restrict the times when the application can be used. For example, users who have access rights but not launch rights can't start the server after the administrator has shut it down for the day.

The last set of options on the security tab allows you to set the configuration permissions. This setting tells the system which users can configure the AppID. Configuration permissions are typically given only to the system administrator and simply set the DACL on the physical Registry key of the AppID, thus preventing unauthorized users from changing the more interesting Access and Launch permission DACLs or other AppID related settings.

Anything that doesn't have a dedicated AppID uses a machine-wide default profile. The system retrieves this default profile from the registry when it fails to find a valid AppID. You can use the Default Security tab in DCOMCnfg.exe to modify these default settings.

Configuring the Server's Identity

Each COM application assumes the identity of one of the identity of one of Windows NT user accounts when it's launched. This means that objects loaded into the server process run under the identity of this security principal. The setting for a server's identity is stored in the AppID key under the *RundAs* named value. You can modify which user account is used on the Identity tab for a specific AppID in DCOMCnfg.exe, as shown in Figure 1.3. You have three choices when configuring your server's *RunAs* identity. You can select The Interactive User, The Launching User, or a dedicated Windows NT user account.

The default setting in Windows NT 4 is The Launching User, but this is almost always the wrong choice for a distributed application. With this setting, the activating client's user account serves as the security principal for the server's process. This can cause multiple instances of the server to be launched on one computer. For example, consider what happens if you use this setting and two remote users, Alice and Bob, both activate objects. This causes two separate instances of the server process to run as Bob. It's unfortunate that this is the default for Windows NT 4. Future versions of COM will remove this configuration altogether.

The most common approach is to create a dedicated Windows NT user account that serves as the security principal for distributed applications. Once you create a local domain account, you can configure the AppID by selecting This User and entering the user name and password, as shown in Figure 1.5. (You must know the password of the user account.) DCOMCnfg.exe does two other important tasks when you select a user in this manner. It grants the *Logon as Batch Job* right to the user account, and it writes the account's password to a special place in the Registry. These two tasks are required to properly assign a user account to the server's identity. Simply adding a *RunAs* named value to an AppID by hand (Using RegEdit.exe, for example) is insufficient.

Once you configure your application to run using a specific user account, all objects activated from the AppID run in a single-server process. This also makes it easy to configure other areas of security in a distributed application. If your business objects access a file server or use SQL Server's integrated security model, the administrator can grant permissions to a single business object account. There's no need to grant access permissions on a user-by-user basis. The administrator simply has to grant permissions to individual users as they attempt to access the distributed application.

The remaining option you have for server identity is to run the server process as an interactive user. This isn't a good idea for a distributed server application in a production environment. When you configure a server to run as the interactive user, the SCM launches the server's process using the user account of whoever happens to be logged on to the console of the server's computer at the time. If the administrator is logged on, you get one set of permissions. If nobody is logged on, the distributed application can't be launched. As you can see, this choice can be problematic.

There are a few reasons why you might run a server as the interactive user. If the server process will run on the same computer as the client application, you probably want both processes to run under the same account. You can see that the interactive user is the same as the launching user when a user activates an object from a local server. Only the interactive user account can display windows on the desktop of the local machine. When you run objects under a dedicated user account, you can't display windows on the screen. Actually displaying a window will succeed, but the window will be displayed in a "non-interactive window station," which means no human will be there to see it. This situation means that the interactive user account can be useful during debugging, when you want to display a message from an out-of-process server. This also means that invoking a message box from a business object running under a dedicated user account will hang the object indefinitely, since nobody will ever press any buttons to dismiss it. This is a good reason to create your servers using Visual Basic's Unattended Execution option.

MTS Security Model

As you'll recall, when a user tries to access a distributed application from across the network, the system authenticates the user's credentials and checks whether the user has the proper authorization. Administrators authorize access to users in a distributed application by granting *access permissions* and optionally *launch permissions* to Windows NT user accounts and group accounts, using a tool such as DCOMCnfg.exe. This tool's editor lets you modify the discretionary access control list (DACL) associated with the application's AppID.

Identity is obviously an important idea in Windows NT as well as COM security. An administrator should configure the AppID for each distributed application using a RunAs setting so that the system knows which Windows NT user account to use as the application's security principal. An application that hasn't been explicitly assigned a RunAs setting runs under the identity of the launching user. When the Service Control Manager (SCM) launches an application, it uses the RunAs setting to assign an identity to the application's process. Objects created inside the application run with the same credentials as the application's security principal. Identity is also known as activation security because it's set up when the application is activated.

As you know, COM security is built on top of Windows NT security and the Remote Procedure Call (RPC) layer. COM can leverage the accounts database and SID management as well as authentication and authorization checking from the underlying system.[2] It's important to understand how all this works because MTS security layered on top of COM security. MTS uses COM authentication to validate users. MTS also uses COM's activation security to control which Windows NT user account serves as the identity for a server package.

However, MTS doesn't use the same authorization scheme as COM. Unlike COM+, the access control provided by COM lacks granularity. You can configure only two types of permissions for an application: access permissions and launch permissions. This means that access control is an all-or-nothing proposition. You can either grant or deny access permissions on an application-wide basis for each Windows NT user or group. Once you let a user in the door, you can't control what the user can do. MTS replaces COM's access scheme with its own to provide a greater degree of granularity and control.

In Windows NT 4, every COM application should be configured with its own AppID. An MTS application is no exception. An MTS server package is an MTS application. MTS creates and manages a hidden AppID for every server package. Using the MTS Explorer, you can easily change the authentication level and identity settings for a server package. When you change these settings, MTS modifies the hidden AppID for you.

The designers of MTS decided to replace COM access control with their own version. The AppID behind an MTS server package is configured to use the default machinewide launch permissions; however, when the package is launched, MTS grants access permissions to all users. In effect, MTS turns off COM's access control. All users must be able to get past the security checkpoints of COM in order to reach those of MTS.

MTS Role Management

MTS extends COM security with its own security model using the concept of a *role*. A given user can be assigned to multiple roles. A role is a set of users within an MTS application that have the same security profile. Whenever you design a secured MTS application, you should define an appropriately named role for each type of user. For example, a systems designer can design an application with three roles: Reader, Writer, and Manager.[2] In MTS, both declarative and programmatic security checks are performed on the roles you define instead of on SIDs, as in COM. The MTS security model thus offers more control and flexibility than does COM.

Let's look at declarative security first. When a developer creates a new server package, declarative authorization checking is turned off by default. The creator can turn it on by selecting the Enable Authorization Checking option on the Security tab of the server package's Properties dialog box. They can then configure which roles (and therefore which users) have permission to invoke methods on objects running inside the application. There are four steps to configuring declarative security checks with roles in an MTS server package. They are, enable authorization checking for the package, create a set of roles inside the server package, add a role to the role membership of each component you want to permit access to (access permissions can also be set at the interface level of a component), and associate Windows NT user accounts and group accounts with the roles you created.

The ability to preconfigure the permissions for the entire application in the development environment is one of the largest benefits of roles. Permissions in MTS makes the role-based model much more flexible than the COM security model, in which access permissions are assigned directly to the SIDs. The concept of roles is more natural for developers, and the use of roles eases deployment of a single package in multiple production environments.

Another benefit of this model is that you can configure access through permissions on a component-by-component as well as an interface-by-interface basis. This declarative security scheme provides much more control that COM security.

For example, you can create a *CCustomers* component that implements two user-defined interfaces, *IqueryCustomer* and *IUpdateCustomer*. When you set up the permissions for your application, you can configure every role in the package to have access to the *IQueryCustomer* interface but restrict access to the *IUpdateCustomer* interface for every role except Writer. Stop and think about the implications of this example. I simply can't overemphasize how powerful declarative security checks are at the interface level.

Developers always have the ability to create their own user-defined interfaces and implement them in their components. The key point is that when developer designs his or her components in terms of user-defined interfaces, they have created far more control for their administrators at deployment time.

Securing MTS Applications

A role is defined within the scope of an MTS server package. That is, each role exists within a single MTS application (running process). When a server package is launched, the MTS container application (mtx.exe) and the MTS executive (mtxex.dll) work together to set up the role-base authorization-checking scheme.[2]

Package security isn't available with library packages. Library packages aren't secured because they're loaded into the address space of an unknown client application. You can't be sure that the client application will be a secured MTS application. Furthermore, you must set up role-based authorization checking when the hosting process is launched. You can't do this with library packages because of their passive nature.

What this boils down to is that roles and role membership have no meaning within a library package. When you attempt to create a role or assign role membership in a library package, the MTS Explorer displays a dialog box indicating that the package security isn't available. You must be careful because a client application can defeat the

MTS security model by creating objects directly from a library package. When a non-MTS client creates objects from a library package, the application is unsecured and the role-based security features of MTS won't function properly.

As you've seen, each class and interface can have one or more associated roles in its membership. If authorization checking is enabled, the caller must be in at least on role in order to successfully invoke a method. If MTS discovers that the caller doesn't belong to any role associated with the component or the interface in use, it fails the call and returns the well-known HRESULT E_ACCESSDENIED. If application that made the call was written in Visual Basic, the application will trap the error and raise a "Permission denied" error with a number of 70.

MTS performs authorization checks only on calls from outside the package, not on interpackage (intraprocess) calls. Interprocess calls are therefore much faster. However, because interprocess calls are assumed to be secure, you must protect your applications from unintended security holes.

Calls between server packages can also be tricky. For example, what happens if client A makes a call on server package B, which then makes a call on server package C? You might expect server package C to perform its security checks using client A's security credentials, but this isn't the case. MTS performs its security checks from hop to hop as opposed to performing them from end to end. This means you must understand the difference between the *direct caller* and the *original caller*. The direct caller is the party one hop up the call chain, while the original caller is the party at the top of the call chain. MTS always performs its security checks using the credentials of the direct caller.

Characteristics of COM+

The security model of COM+ is inherently the same as Microsoft's RPC model. An understanding of the COM Proxy architecture is necessary before we go any further. Each interface has a relationship to a particular object. The interface itself has no idea what security is or where the call is even being sent. Each object wraps MSRPC services and holds an RPC handle. The COM interface used to configure the binding handles is very similar to the RPC model. The *IClientSecurity* interface creates a security blanket around the objects pointed to by each of the other interfaces. *IClientSecurity* operates on interface pointers to proxies, where RPC uses binding handles. The three methods of *IClientSecurity* map directly to three corresponding methods of the MSRPC API.

Cloaking is a term used to describe control over identity tracking. In Windows NT 4, static tracking was the default used by RPC and the only identity-tracking option available. When applying the security blanket to an object, the blanket would cache the current identity of the calling thread. However, in COM+ we can use dynamic impersonation. MSRPC has supported dynamic identity tracking for years, but it was not implemented until Windows 2000. Windows 2000 does not only use dynamic identity tracking, it uses a third option by default that ignores the thread token. Instead, it uses the process token to gather credentials for outgoing calls.

The server gathers the caller's authentication settings when a call arrives, same as RPC. COM provides a pointer to *IServerSecurity* by calling *CoGetCallContext* upon the objects request. Similar to RPC, there are methods that save you a call to *CoGetCallContext*; such as, *CoQueryClientBlanket*, *ImpersonateClient*, and *RevertToSelf*. Here a COM interface is being called instead of an RPC API.

The server selects its SSPs (Security Service Providers) by making a single call to an API provided by COM that handles an array of data. To summarize the last few paragraphs: the client selects the authentication configuration, and the server detects those settings. The call to *IServerSecurity::CoQueryClientBlanket* will fail if the client has not chosen an authentication level. [3] It then verifies the level of authentication is appropriate, completes an audit of the request, and then performs the instructions defined in the method called.

As confusing as the past few paragraphs may have seemed, I will now described the features that COM adds to the RPC security model that make writing applications easier and more secure. COM and RPC servers must verify at each entry point whether the client is authenticated, before determining the type of authorization the client has. If the return value of *IServerSecurity::CoQueryClientBlanket* goes unchecked, an anonymous client would be authenticated. A safer way of taking care of this is to let the server-side object perform an authentication check before the client reaches the server-side code. This is known as COM Interception. [3]

A developer can also rely on COM to select the appropriate set of SSPs at runtime, depending on the version of operating system. In Windows 2000, the default security service providers include Kerberos, NTLM, and SPNEGO.

Conclusion

With a basic understanding of the Windows NT security model, COM security makes sense. It provides the ability to prevent unwanted users from executing middle-tier code that could potentially reveal sensitive information when called correctly. As shown, COM+ offers more granularity and control of permissions on top of RPC. I am interested to see what techniques of securing applications will be available in the future.

References

[1] Schultz, Eugene. *Windows NT/2000 Network Security*. MacMillan Technical Publishing, August 2000

[2] Eddon, Guy and Henry Eddon. *Inside Distributed COM*. Redmond, Wash.: Microsoft Press, 1998.

[3] Brown, Keith. *Programming Windows Security*, Wash.: Microsoft Press, 1998.

[4] Yasser Shohoud, "Programming COM+ Security", March 2002, http://security.devx.com/upload/free/features/vcdj/2000/05may00/mt0500/mt0500.asp

[5] Rogerson, Dale. *Inside COM*. Redmond, Wash.: Microsoft Press, 1997.