# Services in Windows

October 18, 2010


## Abstract

This paper provides information about changes to Windows® services that were introduced in Windows Vista®. It also provides some best-practices guidelines for developers who intend to implement services for these versions of Windows.

This information applies to the following operating systems:

- Windows Server® 2008 R2
- Windows 7
- Windows Server 2008
- Windows Vista

References and resources discussed here are listed at the end of this paper.

The current version of this paper is maintained on the Web at:
http://www.microsoft.com/whdc/system/sysinternals/windows-services.mspx

**Document History**

| Date | Change |
|---|---|
| October 14, 2010 | Changed title to remove “Vista”, because paper also applies to later versions of Windows. Minor cleanup to clarify that statements made about Windows Vista also apply to later versions of Windows. |
| November 30, 2009 | Updated UI requirements and removed unimplemented SC commands. |
| August 15, 2006 | First publication |

# Contents

# Overview of Windows Services

Windows® services are applications that typically start when the computer is booted and run quietly in the background until it is shut down. Strictly speaking, a service is any Windows application that is implemented with the services API. However, services normally handle low-level tasks that require little or no user interaction.

Although services are often effectively invisible to users, Windows cannot function normally without them. A number of essential operating system functions are handled by services, including the following:

- **Networking.** A number of system services support networking. For example, the Dynamic Host Configuration Protocol (DHCP) client service manages network configuration by registering and updating Internet Protocol (IP) addresses for the system.
- **Hardware.** The Plug and Play service enables a computer to recognize and respond to changes in its hardware configuration, such as a user adding or removing hardware.
- **Remote access.** Terminal Services allow users to log on to a computer from a remote location.

In addition to services that are part of Windows, most computers also have several third-party applications that run partly or wholly as services. Common examples of third-party services include firewalls and antivirus applications.

This white paper provides an overview of the changes to the services model that were introduced beginning with Windows Vista. It also provides some best-practices guidelines for developers who intend to implement services for these versions of Windows. For the convenience of readers who might not be familiar with services, the remainder of this section provides a brief overview of how services work and how they are implemented. For more complete information, see the Windows Services documentation in the MSDN® Library.

## How Services Work

The primary difference between services and normal applications is that services are managed by the Service Control Manager (SCM). Services are implemented with the services API, which handles the interaction between the SCM and services. The SCM maintains a database of installed services and provides a unified way to control them, including:

- Starting and stopping services.
- Managing running services.
- Maintaining service-related state information.

Services exist in one of three states: *started*, *stopped*, or *paused*.

- **Started** is the normal running state for a service.
- A **stopped** service has been completely shut down and must go through a normal startup procedure to enter the started state again.

- A **paused** service suspends normal processing, but remains in memory and continues to respond to control requests. Paused services can therefore return to the started state without going through the startup procedure.

A key characteristic of a service is how it is started. The SCM has a database that includes information on how each service should be started. The following are the service startup types:

- **Automatic.** The SCM automatically starts these services during the system's boot process. They are often called *auto-start* services.
- **Manual.** These services must be started manually with the Control Panel Administrative Tools application, with the sc.exe command-line tool, or programmatically with the **StartService** function. They are often called *demand-start* services. Windows 7 introduces trigger-start services. For more information, see “Developing Efficient Background Processes for Windows” on the WHDC Web site.
- **Disabled.** These services cannot be started. To start a disabled service, the user must first change the startup type to automatic or manual.

After a service has started, the SCM uses *control requests* to manage the service's state. For example, the SCM sends control requests to notify a service that it is pausing, is resuming operation, or should be preparing to shut down. The SCM's database also contains the security settings for each service. These settings control how much access a service has to system resources and enable system administrators to control access to each service.

## Changes to the Windows Services Model

Services have been an essential part of Windows for many years. They enable developers to create long-running executable applications that:

- Can be automatically started when the computer starts.
- Can be paused and restarted.
- Can function whether or not an interactive user is logged on.
- Can run in the context of a user account that is different from either the logged-on user or the default computer account.

These features make services ideal for scenarios where there is a need for long-running functionality that does not interfere with users who are working on the computer. However, since services were introduced, the environment in which they function has changed substantially. These changes have created various issues with security, reliability, performance, management, and administration.

This section discusses enhancements that were introduced in Windows Vista to address service-related issues.

## Security Enhancements

In recent years, services have been an attractive target for virus writers who want to attack Windows. Recent examples include Blaster, Sasser, and Code Red. This situation occurs for a number of reasons:

- Services are typically long running. Often, they start when the system boots up and stop when it shuts down.
- Services are often network facing, which makes them especially vulnerable to remote attacks.
- Services typically run in a high-privilege account such as LocalSystem.

This section discusses the enhancements that were introduced in Windows Vista to mitigate the security issues that are associated with services. These changes address two important goals:

- Limiting access to services by user applications. *Session 0 isolation* requires services and user applications to run in separate sessions.
- ”Hardening” services to limit the ability of a compromised service to damage a system. There are two complementary ways to accomplish this goal:
  - *Running with least privilege* allows services to run with only those privileges that they need, and nothing more.
  - *Service isolation* allows services to isolate themselves from other services or applications by using a unique *service identity*. A service can use this identity to restrict access to its resources by other services or applications. A service can also use its identity to restrict the service's access to the resources of other services or applications. For example, service isolation allows an antivirus service to maintain exclusive access to its signature definition files.

### Running with Least Privilege

Windows services commonly run in the LocalSystem account, the most powerful account on the system. This makes such services attractive targets for virus writers. Ideally, services should limit their damage potential by running in a lower-privilege account such as LocalService or NetworkService. However, many services require at least some privileges that only LocalSystem supports. The all-or-nothing model that was used in versions of Windows prior to Windows Vista meant that a service that required any LocalSystem privileges had to also include all other LocalSystem privileges. This often meant including privileges that the service did not require, which created an unnecessarily high damage potential.

Beginning with Windows Vista, services can *run with least privilege*. Services are no longer restricted to the default set of privileges that are supported by a standard account. Instead, services can select an account that has the privileges that they require and then remove all other unnecessary privileges. This feature can be used for any type of service account: LocalService, NetworkService, LocalSystem, a domain, or a local account.

Services specify their required privileges by one of the mechanisms discussed later. When the SCM starts the service:

- For stand-alone services, the SCM checks the list of required privileges against the process token. Any privileges that were not specified as required are removed from the token.
- For shared-process services, such as services hosted in svchost, the list of privileges is the union of the required privileges for all services in the group. The only privileges that the SCM removes from the process token are those that no member of the group has specified as required.
- If a service does not specify a required set of privileges, the SCM assumes by default that the service requires all privileges that are associated with the account. This assures backward compatibility. However, if a service group contains even one such service, the entire group runs with the account's default set of privileges.

If a service requires privileges that are not in the process token, the SCM does not start the service. For example, a process that is running in the NetworkService account could specify SeTcbPrivilege as a required privilege. However, SeTcbPrivilege is not supported by a NetworkService process token, so the start attempt fails. Administrators must understand this issue when they make configuration changes such as changing the service image of a shared-process service while the target service process is running. The service can start only if the target service process supports the specified privileges.

### How to Specify Required Privileges

Sc.exe—a command-line tool for managing services—has two new commands that support specifying required privileges for a service:

- **privs**. This command sets the required privileges for a service.

  The syntax for the **privs** command is:

  ```
  sc <server> privs [Privileges]
  ```

- **qprivs**. This command queries for the required privileges of a service. The syntax for the **qprivs** command is:

  ```
  sc <server> qprivs [service name] buffersize
  ```

*Privileges* is a string that contains a list of privileges that are separated by a forward slash (/). For example, to specify backup and restore privileges, set *Privileges* to SeBackupPrivilege/SeRestorePrivilege.

To specify required privileges programmatically, call **ChangeServiceConfig2** with the following parameter values:

- Set the *dwInfoLevel* parameter to SERVICE_CONFIG_REQUIRED_PRIVILEGES_INFO.
- Set the *lpInfo* buffer to point to a SERVICE_REQUIRED_PRIVILEGES_INFO structure. This structure contains a single multistring that lists the required privileges.

Use **QueryServiceConfig2** to query for a service's required privileges.

The change in privileges takes effect the next time the service is started. Note that both the command-line tool and the function check whether the privileges in the list are valid, but they do not determine whether the service can support the specified privileges. That task is handled by the SCM when it attempts to start the service.

**Note:** The list of required privileges can be modified only by callers that have the SERVICE_CHANGE_CONFIG access right. By default, only local administrators, power users, and server operators on a domain controller can get this right remotely or locally. Callers can query for the list of required privileges if they have SERVICE_QUERY_CONFIG access right. By default, only local administrators, power users, and server operators on a domain controller get this right remotely. Services and interactive users can get the SERVICE_QUERY_CONFIG access right locally.

## Service Isolation

Many services require access to certain objects that are available only to high-privilege accounts. For example, a service might be required to write to a registry key that provides write access only to administrators. In versions of Windows prior to Windows Vista, services typically gained access to such objects by running in a high-privilege account such as LocalSystem. An alternative approach was to weaken the security on the objects to allow access by services that are running in a generic lower-privilege account.

Both approaches increased the risk that an attacker or malware could gain control of the system. The only way for an administrator to mitigate this risk was to create an account specifically for the service and allow access to the objects only for that account. However, this approach created manageability problems, most notably password management, because the administrator no longer had the advantages of using built-in operating system accounts.

To mitigate this problem, Windows Vista introduced service isolation, which provides services a way to access specific objects without being required to either run in a high-privilege account or weaken the objects' security protection. For example, service isolation allows an antivirus service to run in a lower-privilege account than LocalSystem, but still maintain complete access to its signature definition files or registry keys that would normally be accessible only to administrators.

A service isolates an object for its exclusive use by securing the resource—such as file or registry key access—with an access control entry that contains a service security ID (SID). This ID is referred to as a *per-service SID*. A per-service SID is derived from the service's name and is unique to that service.

After a SID has been assigned to a service, the service owner can then modify the required objects' access control lists (ACLs) to allow access to the SID. For example, a registry key in HKEY_LOCAL_MACHINE\SOFTWARE would normally be accessible only to services with administrative privileges. By adding the per-service SID to the key's ACL, the service can run in a lower-privilege account, but still have access to the key.

If a per-service SID is enabled, it is added to the service's process token. Note that a per-service SID must be enabled at the time the service's process is started to be added to the process token. If a process hosts multiple services with enabled SIDs, they are all added to the process token. As discussed later, per-service SIDs also allow a process token to be converted to a restricted token by adding one or more SIDs to the restricted token list.

### Reducing the Damage Potential by Using Restricted SIDs

Using a per-service SID, as described in the previous section, provides a good degree of isolation and allows the service to run in a lower-privilege account. However, it does not prevent the service from accessing other resources that are accessible to the account because the process token also contains the SID for the account. Consider the following scenario:

> Service X runs in the LocalService account and has a service SID enabled. In addition to having access to objects that have specifically granted this service access—by using the per-service SID—it also has access to all objects that grant access to LocalService. As a result, if this service were to be compromised, the attacker could cause damage by accessing the resources that are not related to the service.

To mitigate this problem and reduce the damage potential of compromised services, Windows Vista and later versions of Windows use a hybrid approach that combines write-restricted tokens and per-service SIDs to introduce restricted SIDs for services.

When a service enables a restricted SID, the per-service SID of that service is added to both the normal and restricted SID list of the write-restricted service process token. This guarantees that the service can write only to objects that have explicitly granted write access to one of the SIDs in the restricted list. Returning to the preceding example, by enabling restricted SIDs, Service X can no longer write to any objects that grants write access to the LocalService account because those objects do not explicitly grant write access to the per-service SID of Service X.

### How to Specify Per-Service SIDs

To specify a per-service SID, the SID type must be set to SERVICE_SID_TYPE_UNRESTRICTED. Services that do not require SIDs can either refrain from setting the SID type or set it to SERVICE_SID_TYPE_NONE. After the type has been set, the SID is added to the process token the next time that the process is created. The SID has the following two attributes enabled:

- SE_GROUP_ENABLED_BY_DEFAULT
- SE_GROUP_OWNER

The service's process token is also augmented with an Allow access control entry (ACE) that provides GENERIC_ALL access for the service logon SID. This allows the service to enable or disable service SIDs in the process token when the service starts or stops.

After a SID has been added to a process token, it cannot be removed. The SID type must be changed, and then the process must be recycled. The change takes place only when the process is restarted.

To make the process token write-restricted, the SID must be set to SERVICE_SID_TYPE_RESTRICTED. Several issues must be considered when using SERVICE_SID_TYPE_RESTRICTED:

- If a process hosts multiple services, all of them must be set to SERVICE_SID_TYPE_RESTRICTED. Otherwise, any service that is set to SERVICE_SID_TYPE_RESTRICTED fails to start.
- Three SIDs are automatically added to the restricted list:
  - World SID (S-1-1-0). This SID provides the service with write access to any objects whose ACLs support the World SID. In particular, the World SID provides access to some DLLs in the load path.
  - Service logon SID. This SID supports write access to the named pipe that connects the service process to the SCM.
  - Write-restricted SID (S-1-5-33). This SID allows objects to have an ACL that lets any write-restricted service process write to the object. One common example is event tracing for Windows (ETW) objects.

Sc.exe has two new commands that support per-service SIDs:

- **sidtype**. This command changes a service's SID.

  The syntax for the **sidtype** command is:

  ```
  sc <server> sidtype [service name] [type]
  ```

- **qsidtype**. This command retrieves the setting for service's SID. The syntax for the **qsidtype** command is:

  ```
  sc <server> qsidtype [service name]
  ```

To set the flag programmatically, call **ChangeServiceConfig2** with the following parameter values. The change takes effect the next time the system is booted:

- Set the *dwInfoLevel* parameter to SERVICE_CONFIG_SERVICE_SID_INFO.
- Set the *lpInfo* buffer to point to a SERVICE_SID_INFO structure. This structure contains a single DWORD member that contains the SID type.

Two related public functions are also useful to service owners:

- **LookupAccountName** takes a SID and returns the associated service name.
- **LookupAccountSID** takes a service name and returns the associated SID.

Sc.exe can also be used to get a specified service's service SID. The following command returns the SID for a service that is named *service name*:

```
sc showsid [service name]
```

**Note:** A caller must have the SERVICE_CHANGE_CONFIG access right to change this setting. By default, only administrators, power users, and server operators on a domain controller can get this right remotely or locally. A caller must have the SERVICE_QUERY_CONFIG access right to query the setting. By default, only administrators, power users, and server operators on a domain controller can obtain this right remotely. In addition, services and interactive users can obtain SERVICE_QUERY_CONFIG access rights locally.

## Restricted Network Access

Many services are network facing, which makes them vulnerable to remote attacks. Beginning with Windows Vista, service hardening allows developers to limit a service's access to network resources such as ports, protocols, and direction of network traffic. For example, a DHCP service—which renews a system's IP address—can restrict itself to local port 68 and inbound User Datagram Protocol (UDP) traffic on remote port 67. Attempts to open or listen on other ports will be blocked by the Windows firewall.

Windows Vista and later versions of Windows support the following service network restriction scenarios:

| Scenario | Example | Restrictions |
|---|---|---|
| No network access | The shell hardware detection service (ShellHWDetection) | The service cannot listen or connect to the network. |
| Listens on static TCP or UDP ports. | Remote procedure call services (Rpcss) registered with the Internet Assigned Numbers Authority (IANA), on port 135 | The service can listen only on specified endpoints. |
| Listens on configurable TCP or UPD ports | Domain name service (DNS) | The service can listen on configured endpoints. |

Service network restrictions are implemented with per-service SIDS. The mechanism is similar to that used to restrict a service's file or registry key access, as discussed in "Service Isolation" earlier in this paper. Beginning with Windows Vista, enhancements to the Windows firewall API provide the necessary support. The **INetFwServiceRestriction** interface includes the methods that are used to restrict a service's network access. For further information, see "Internet Connection Sharing and Internet Connection Firewall" on MSDN.

The following Microsoft® Visual Basic® script example uses the firewall API to restrict the 6to4 service to only inbound TCP 500 traffic.

```
'require variable declarations
option explicit
'handle errors
on error resume next
'direction
Const NET_FW_DIRECTION_IN = 1
' Create the FwPolicy2 object.
Dim fwPolicy2
Set fwPolicy2 = CreateObject("HNetCfg.FwPolicy2")
Dim FwSvcRestr
Set FwSvcRestr = fwPolicy2.ServiceRestriction
'restrict service
FwSvcRestr.RestrictService "6to4", "c:\windows\system32\svchost.exe",
TRUE, TRUE
```

```
'Create a new restriction rule
Dim NewRule
set NewRule = CreateObject("HNetCfg.FwRule")
NewRule.Name = "6to4 500"
NewRule.Description = "Allow 6to4 to receive TCP traffic on port 500"
NewRule.ApplicationName = "c:\windows\system32\svchost.exe"
NewRule.ServiceName = "6to4"
NewRule.Protocol = 6
NewRule.LocalPorts = "500"
NewRule.Direction = NET_FW_DIRECTION_IN
NewRule.Enabled = TRUE
'Add the behavior rule to the WSH store.
fwPolicy2.ServiceRestriction.Rules.Add NewRule
```

## Session 0 Isolation

Windows accommodates multiple logged-on users by putting each user in a separate *session*. Session 0 is created during startup, with additional sessions added as required. Services have always run in Session 0. In versions of Windows prior to Windows Vista, user applications have also been able to run in Session 0. For example, for Windows XP with Fast User Switching (FUS) enabled, the first user to log on is assigned to Session 0 and that user's applications run in that session. Session 1 is assigned to the second user to log on, and so on.

Running services and user applications in the same session creates several security issues. To address those issues, Windows Vista introduced two significant changes to Session 0:

- Session 0 is reserved exclusively for services and other applications that are not associated with an interactive user session. User applications must run in Session 1 or higher.
- Session 0 does not support user interfaces. In particular, processes that are running in Session 0 do not have access to the graphics hardware and cannot directly display any UI on the monitor.

Session 0 isolation has a number of implications for services, including the following:

- Services cannot use **PostMessage** or **SendMessage** to send messages to user applications. All applications run in Session 1 or higher and have a different message queue. For the same reason, applications cannot send Windows messages to a service. Services that send messages to applications must use mechanisms such as remote procedure calls (RPCs) or named pipes.
- Services that have a UI, such as a dialog box, cannot display it directly in Windows Vista and later versions of Windows. Services that require user interaction must handle it indirectly. For simple interactions, services can display a message box in the user's session by calling **WTSSendMessage.**
- For more complex interactions, developers should move their UI code into an agent that runs in the user's session and handles all UI requirements. The agent communicates with the service through RPC or named pipes. If the user initiates the UI interaction by using Control Panel, Internet Explorer, or a similar UI experience, that UI experience should start the agent. The agent then handles all UI interactions. If UI is required but is not initiated by the user, the service must request the agent to start any required UI, instead of attempting to launch that

UI by itself. In the rare situation where the service must initiate a user interaction and the agent is not already running, the service should call the **CreateProcessAsUser** API to start the agent. The agent can then initiate all UI interactions. It is important for developers to carefully review all possible usage scenarios and consider moving all UI code into an agent that runs in the user session.

For more information, see “Impact of Session 0 Isolation on Services and Drivers in Windows” on the WHDC Web site.

## Performance Enhancements

The long-running nature of services means that they can have a significant impact on performance. This section discusses several enhancements that reduce the impact of services on system performance.

### Delayed Auto-Start

In versions of Windows prior to Windows Vista, there were two ways to start a service: auto-start and demand-start. Auto-start services start automatically during the system's boot process. Demand-start services can be started in several ways, but they all require a client to manually restart the service.

There are two primary reasons for defining a service as auto-start:

- The service must be started early in the boot process because other services depend on it.
- Administrators often want an unattended start for their service that does not require the presence of an interactive user. That ensures that the service is available when it is needed.

The problem with auto-start services is that their growing numbers increases the time it takes to boot the system. However, many auto-start services belong in the second of the two categories. They are not required to be part of the boot sequence; they just require an unattended start so that they are available reasonably soon after the system is ready for use.

Windows Vista addressed the impact of services on boot performance by introducing a variant of auto-start services that are called *delayed auto-start*. Services that are designated as delayed auto-start still start automatically when the system is started up. However, they are not started during the boot sequence. Instead, they are started shortly after the system has booted. This improves boot performance while still providing these services with an unattended start.

Service developers and administrators should consider the following issues before designating a service as delayed auto-start:

- Understand the service's dependencies. If another service that must be started during boot depends on this service, then there is no reason to set its delayed auto-start flag. The SCM will start the service during boot, regardless of the flag setting

- There is no specific time guarantee for when a delayed auto-start service starts. If a client application attempts to use a delayed auto-start service before it has been started, the attempt fails. Clients that depend on delayed auto-start services must handle the failure gracefully and either retry the attempt later or call **StartService** to demand-start the service. If such failures happen frequently, the service is probably not a good candidate for delayed auto-start.
- Delayed auto-start services cannot belong to a load-order group. They can belong to a stand-alone group.

#### How to Designate a Service as Delayed Auto-Start

To create a delayed auto-start service, set the *delayed auto-start* flag. The flag can be set for any start type, but takes effect only for auto-start services. If it is set for an auto-start service, startup is delayed until the boot sequence has finished. With the other start types, the flag is simply ignored.

To set the delayed auto-start flag programmatically, call **ChangeServiceConfig2** with the following parameter values. The change takes effect the next time the system is booted:

- Set the *dwInfoLevel* parameter to SERVICE_CONFIG_DELAYED_AUTO_START_INFO.
- Set the *lpInfo* buffer to point to a SERVICE_DELAYED_AUTO_START_INFO structure. This structure contains a single BOOL member that sets or clears the delayed auto-start flag.

The delayed auto-start flag can be modified only by callers with the SERVICE_CHANGE_CONFIG access right. By default, only local administrators, power users, and server operators on a domain controller receive this right remotely. Services and interactive users can obtain the SERVICE_CHANGE_CONFIG access right locally.

For information about service startup features in Windows 7, see “Developing Efficient Background Processes for Windows” on the WHDC Web site.

## Service State Change Notifications

In versions of Windows prior to Windows Vista, the only way for a client to determine whether a service had changed its status or been created or deleted was to use the service query API—such as the **QueryServiceStatusEx** function—and poll the status of the service. This was not the best approach because these polling loops reduced system performance. In addition, polling loops have historically been a significant source of bugs.

Windows Vista introduced a new function, **NotifyServiceStatusChange**, which allows the SCM to notify a client when a specified service is created, is deleted, or changes its status.

### How to Have a Client Notified When a Service's State Changes

To register for state change notifications, a client calls **NotifyServiceStatusChange** to specify the service and change for which it requires notification. The client also provides the SCM with a pointer to a callback function. When the specified change occurs, the SCM calls the callback function to notify the client.

The following list contains several notes on the use of **NotifyServiceStatusChange.** For more information, see the function's reference page on MSDN.

- **NotifyServiceStatusChange** can be used by local or remote clients.
- The callback function is called only once. If the client wants a subsequent notification for this change, the client must call **NotifyServiceStatusChange** again and reregister the callback function.
- The callback function is normally called after a transition to the specified state has occurred. There is an exception to this rule for the first time that the caller invokes **NotifyServiceStatusChange**. In that case, if the service is already in the specified state, SCM calls the callback function.
- Clients can cancel the notification by calling **CloseServiceHandle** to close the service handle.
- Clients should not exit the thread that they used to call **NotifyServiceStatusChange** unless the callback function has been called or they have canceled the notification. Otherwise, they will create a memory leak.
- If one or more services hold open handles to a service, the service is not deleted until the next time the system is booted. If this is the case, no delete notification is sent.

**Note:** A caller must have the SERVICE_QUERY_STATUS access right to call **NotifyServiceStatusChange**. By default, only administrators, power users, and server operators on a domain controller can obtain this right remotely. Services and interactive users can obtain this right locally.

## Other Enhancements

Windows Vista and later versions of Windows include several other services enhancements, most of which improve reliability or ease of administration.

### Preshutdown Notifications and Shutdown Ordering

In versions of Windows prior to Windows Vista, service shutdown starts when the system notifies the SCM that the computer is shutting down. The SCM then has a default time of approximately 20 seconds to shut down all running services before the system terminates the SCM process (Services.exe). This procedure does not always allow services to gracefully shut themselves down.

When the SCM receives the shutdown notification, it in turn sends shutdown requests to every service serially, in random order. Each service has a wait hint that specifies how long the SCM should wait before it simply shuts down the service. In practice, the SCM waits for the longest wait hint to expire and then returns to its control handler, which shuts the services down. This shutdown model creates two types of problems for services:

- The shutdown order is essentially random. If a service expects other services to shut down in a particular order, it may fail to shut itself down gracefully.
- Some services might not have sufficient time to clean up adequately before shutdown. This cleanup often must be performed when the service is started again, which often results in long service startup times or data inconsistencies.

Windows Vista and later versions of Windows address these service shutdown issues in two ways:

- The SCM delivers a preshutdown notification before the actual shutdown notification. The preshutdown notification gives services that have a potentially lengthy shutdown procedure more time to shut down gracefully.
- Service owners that depend on the order in which other services shut down can specify any dependencies in a global dependency list.

Preshutdown notifications work much like shutdown notifications. The SCM sends preshutdown notifications to the registered services in an essentially random order. After sending a notification, the SCM waits a specified amount of time for the service to stop before assuming that the service will not respond. By default, the wait time is 3 minutes, but services can configure their wait time to match their particular requirements. For more information, see “Service Control Handler Function” on MSDN.

Services can specify shutdown dependencies in a global dependency list. Before shutting down a service, the SCM attempts to first shut down the dependent services. To take advantage of this feature, a service must be registered for a preshutdown notification.

If a service specifies dependencies, the shutdown process occurs synchronously. Suppose, for example, a service specifies a shutdown dependency on services “A,” “B,” and “C,” in that order. The SCM sends the notification to “A” first and waits for it to either stop or time-out. Then the SCM sends a notification to “B,” and so on. If any specified service fails to stop correctly or is not registered for preshutdown notifications, the SCM simply proceeds to the next service.

### How to Register for a Preshutdown Notification

A service registers for preshutdown notifications by setting SERVICE_ACCEPT_PRESHUTDOWN in its status block. Preshutdown notifications are sent only to running services that have registered for the notification. They are not sent to services that are in the SERVICE_STOPPED or SERVICE_STOP_PENDING states.

To set the time-out value, services call **ChangeServiceConfig2** with the following parameter settings:

- *dwInfoLevel*. Set this parameter to SERVICE_CONFIG_PRESHUTDOWN_INFO.
- *lpInfo*. Set this buffer to point to a SERVICE_PRESHUTDOWN_INFO structure with its **dwPreshutdownTimeout** member set to the time-out value, in milliseconds.

### How to Specify Shutdown Dependencies

To specify shutdown dependencies, create a multistring value that contains the service names in the order in which they should be shut down and assign it to the **Control** key's **PreshutdownOrder** value, as follows:

```
HKEY_LOCAL_MACHINE
  System
    CurrentControlSet
      Control
        PreshutdownOrder="Shutdown Order"
```

## Failure Detection and Recovery

If a service fails, the SCM can perform a *failure action*, such as restarting the service in an attempt to recover from that failure. In versions of Windows prior to Windows Vista, the definition of service failure was limited to the process crashing in any state other than SERVICE_STOPPED. In Windows Vista and later versions of Windows, a service is not required to crash to have the SCM initiate a failure action. Services can notify the SCM to initiate a failure action if they discover a nonfatal error condition such as a serious memory leak.

### How to Configure a Failure Action

The SCM initiates failure actions only for those service that have explicitly specified them. The most common failure action is “restart the service.” This action requires two values:

- The *recovery interval* is the time in milliseconds that the SCM waits before starting the recovery action.
- The *reset period* is the time in seconds after the last failure that the SCM waits before resetting the failure count to 0.

One way to configure failure actions is by using Sc.exe. The following example shows typical syntax:

```
sc.exe failure myservice reset=300
       actions=restart/60000/restart/120000/restart/none
```

This command notifies the SCM to:

- Wait 60 seconds following the first failure before restarting the service.
- Wait 120 seconds following the second failure before restarting the service.
- Not restart the service after the third failure. Another common action at this point would be to reboot the system. In that case, replace “restart/none” with “reboot/*reboot_interval*”, where *reboot_interval* specifies how long the SCM should wait before rebooting. 60000 is a typical value for *reboot_interval.*
- Reset the failure count to 0 after 300 seconds of successful operation. If reset is set to INFINITE, the failure count is never reset.

To configure a failure action programmatically, call **ChangeServiceConfig2** with the following parameter values. The change takes effect the next time the system is booted:

- Set the *dwInfoLevel* parameter to SERVICE_CONFIG_FAILURE_ACTIONS.
- Set the *lpInfo* buffer to point to a SERVICE_FAILURE_ACTIONS structure. This structure contains several members that are used to specify the failure action.

### How to Notify the SCM to Initiate a Failure Action

Windows Vista introduced a new flag, **FailureActionsOnNonCrashFailures**, which services set if they want to be able to notify the SCM to initiate a failure action:

- Set **FailureActionsOnNonCrashFailures** to the default value of 0 to use the model that is in versions of Windows prior to Windows Vista. The SCM queues a failure action for the service only if the service's process crashes when the service is not in the SERVICE_STOPPED state.
- Set **FailureActionsOnNonCrashFailures** to 1 to enable a service to notify the SCM to initiate failure actions:
  - A service notifies the SCM to queue a failure action by entering the SERVICE_STOPPED state and setting the **SetServiceExitCode** function's *dwWin32ExitCode* parameter to anything other than ERROR_SUCCESS.
  - If the service's process crashes when the service is not in the SERVICE_STOPPED state, the SCM follows the failure model for versions of Windows prior to Windows Vista and queues a failure action for the service.

**Note:** For the **FailureActionsOnNonCrashFailures** flag to have any effect, the service must have failure actions configured.

Sc.exe has two new commands to enable configuration of the **FailureActionsOnNonCrashFailures** setting:

- **failureflag**. This command changes the setting of the **FailureActionsOnNonCrashFailures** flag. The following shows the syntax for the **failureflag** command:

```
sc server failureflag [service name] [flag]
```

- **qfailureflag**. This command retrieves the setting for the **FailureActionsOnNonCrashFailures** flag. The following shows the syntax for the **qfailureflag** command:

```
sc server qfailureflag [service name]
```

To set the **FailureActionsOnNonCrashFailures** flag programmatically, call **ChangeServiceConfig2** with the following parameter values. The change takes effect the next time the system is booted:

- Set the *dwInfoLevel* parameter to SERVICE_CONFIG_FAILURE_ACTIONS_FLAG.
- Set the *lpInfo* buffer to point to a SERVICE_FAILURE_ACTIONS_FLAG structure. This structure contains a single BOOL member, **fFailureActionsOnNonCrashFailures**, which sets or clears the failure-actions flag.

**Note:** The **FailureActionsOnNonCrashFailures** flag can be modified only by callers that have the SERVICE_CHANGE_CONFIG access right. By default, only local administrators, power users, and server operators on a domain controller can obtain this right remotely. Services and interactive users can obtain the SERVICE_CHANGE_CONFIG access right locally.

## Resources

The following links provide further information about Windows services and related topics.

### WHDC website

**Developing Efficient Background Processes for Windows**
http://www.microsoft.com/whdc/system/pnppwr/powermgmt/BackgroundProcs.mspx

**Impact of Session 0 Isolation on Services and Drivers in Windows**
http://www.microsoft.com/whdc/system/sysinternals/Session0Changes.mspx

### MSDN

**MSDN Windows Services documentation**
http://msdn.microsoft.com/en-us/library/ms685141(VS.85).aspx

**Internet Connection Sharing and Internet Connection Firewall**
http://msdn.microsoft.com/en-us/library/aa286579.aspx

**NotifyServiceStatusChange Function**
http://msdn.microsoft.com/en-us/library/ms684276(VS.85).aspx

**Service Control Handler Function**
http://msdn.microsoft.com/en-us/library/ms685149(VS.85).aspx

**Service Trigger Events**
http://msdn.microsoft.com/en-us/library/dd405513(VS.85).aspx

**Configuring a Service Using SC**
http://msdn.microsoft.com/en-us/library/ms682053(VS.85).aspx