



realtimepublishers.com[™]

The Definitive Guide[™] To

Windows Installer
Technology
for System Administrators



Darwin Sanoy and Jeremy Moskowitz

Chapter 3: Windows Installer Internals	45
Application Management Meta Data	45
MSI File Format.....	46
Three Streams	46
The Database.....	46
“Open” File Format.....	48
How Packages Describe Software Applications and Installation Procedures	48
Software Application Information	49
Identification in Windows Installer	49
Component Structure and Attributes	50
Component Name	51
Component Codes.....	52
Keypaths	52
Entry Points and Advertisements.....	53
Typical Components	55
Features.....	56
Package Execution Information.....	57
Standard Actions.....	57
Custom Actions.....	58
Sequences.....	59
Properties	60
Notable Properties.....	62
Self-Healing Overview	62
Summary of Package Structure Concepts.....	63
Customizing Packages	65
Managed Application Settings.....	67
Creating Transforms for Application Settings.....	68
Using Transforms.....	69
Administrative Installs	70
Building and Using Administrative Installs.....	71
Installing from an Administrative Share.....	72
Serving Applications.....	73
Security and Policies.....	74

Windows Installer Policies	74
Elevated Privileges Implementation	75
Managed Applications	76
Always Install with Elevated Privileges (AlwaysInstallElevated) Policy	76
AlwaysInstallElevated Hacking.....	77
Disable Windows Installer (DisableMSI) Policy.....	77
Cache Transforms in Secure Location on Workstation (TransformsSecure)	78
Other Security-Oriented Policies	78
Non-Security Policies	78
Excess Recovery Options	78
Logging Policy.....	79
Software Restriction Policies	80
Certificate Rules.....	80
Hash Rules	80
Path Rules	81
Zone Rules	81
Combining Rules	81
Summary.....	81

Copyright Statement

© 2002 Realtimepublishers.com, Inc. All rights reserved. This site contains materials that have been created, developed, or commissioned by, and published with the permission of, Realtimepublishers.com, Inc. (the “Materials”) and this site and any such Materials are protected by international copyright and trademark laws.

THE MATERIALS ARE PROVIDED “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. The Materials are subject to change without notice and do not represent a commitment on the part of Realtimepublishers.com, Inc or its web site sponsors. In no event shall Realtimepublishers.com, Inc. or its web site sponsors be held liable for technical or editorial errors or omissions contained in the Materials, including without limitation, for any direct, indirect, incidental, special, exemplary or consequential damages whatsoever resulting from the use of any information contained in the Materials.

The Materials (including but not limited to the text, images, audio, and/or video) may not be copied, reproduced, republished, uploaded, posted, transmitted, or distributed in any way, in whole or in part, except that one copy may be downloaded for your personal, non-commercial use on a single computer. In connection with such use, you may not modify or obscure any copyright or other proprietary notice.


The Materials may contain trademarks, services marks and logos that are the property of third parties. You are not permitted to use these trademarks, services marks or logos without prior written consent of such third parties.

If you have any questions about these terms, or if you would like information about licensing materials from Realtimepublishers.com, please contact us via e-mail at info@realtimepublishers.com.

Chapter 3: Windows Installer Internals

by Darwin Sanoy

Why does an administrator need to be concerned with the internals of Windows Installer? When Windows Installer works correctly, it provides you with some sophisticated features that save you time and enhance your users' productivity. However, when things go wrong, finding the problem will depend heavily on your understanding of how the internals of packages work. This idea is applied equally to debugging vendor supplied packages as well as ones from your own internal packaging operations or in-house developers. A good framework for understanding Windows Installer internals will also give you the foundation for building good packages.

 The Windows Installer SDK and its tools will be referenced throughout this chapter. If you do not have access to the Microsoft Platform SDK, you can visit the SDK online using the shortcut URL <http://WindowsInstallerTraining.com/msisdk>. This URL has been set up because the URLs for MSDN online are long, cryptic, and frequently move (as the MSI SDK recently did!).

You can also install the SDK over the Web if you want to get the tools and the documentation in Help file format. Visit <http://www.microsoft.com/msdownload/platformsdk/sdkupdate/> and click Windows Installer SDK on the left navigation bar.

Application Management Meta Data

As we've already seen and discussed, the Windows Installer technology has many valuable new features such as self healing, improved uninstalls, and customization capabilities. A key element to enabling these new features is recording and referencing information that tracks how software applications should be installed. This information can be thought of as application management *meta data*, that is, data that references or describes other data.


There are two distinct storage areas for management meta data about Windows Installer packages. The first of these locations is in an MSI package file. The internal database in this file stores all the information required to install a software application. The second location in which Windows Installer package data is stored is the Windows Installer (MSI) repository on each computer. The Windows Installer repository is made up of a database within the registry and some cached files on the hard disk.

The Windows Installer repository gives Windows Installer intelligence when performing installations on demand and when self-healing. This information describes to Windows Installer which files, registry keys, and other configuration changes must be installed for an application to work correctly.

The meta data stored in the MSI repository contains a pointer back to the original MSI file. This pointer is generally used to retrieve source files. The data contained in the repository (such as installed product codes, upgrade codes, and so on) is not retrieved from the MSI file; it is stored directly in the repository on client computers.

MSI File Format


Most of the files used by MSI utilize a special Microsoft file technology called *COM Structured Storage*. This storage technology basically creates multiple spaces, called *streams*, within a file. You can think of these streams as files within files, not unlike a Visio diagram embedded in a Word document.

 COM Structured Storage does not use Alternative Data Streams. Alternative Data Streams are a file–system–level technology available on NTFS systems that allows multiple data storage areas in the same physical file.

Three Streams

An MSI file usually has three streams: one for summary information, one for the MSI database, and one for storing the installation files. (Installation files can also be stored externally.) Other streams (such as the AdminProperties stream) might be created by various Windows Installer activities, but these three are the main three to start with. The many other file formats utilized by Windows Installer are generally variants on the MSI file format, such as


- .MST—Transform file
- .MSP—Patch file
- .CUB—Validation file

 Msiinfo.exe is a Windows Installer SDK tool that allows the summary information stream to be queried and updated from the command line.

The Database


The Windows Installer database stream contains the fundamental information required to perform the installation of the software application. Only items inside the database can be customized using transforms. Transforms are essentially database overlays that are used at installation time.

The Windows Installer database is normalized. In database language, this means that to represent any given entity (say all the information to install a file) there may be several linked tables involved.


 Although you will periodically need to view a table, a complete understanding of every nuance of the relationships between these tables should not be necessary if you are investing in good Windows Installer authoring tools.

There are two important types of information that are contained within the database:

- Information about the software application to be installed. This information includes which files, registry keys, and shortcuts should be installed. It also includes information about how the developer organized the package within the Windows Installer rules. Following the Windows Installer rules for package structures results in the software application being represented as Features and Components, which we will be discussing in a bit.
- Information about the actual execution of the package. Figure 3.1 illustrates that the package execution logic is stored in the database along with the tables describing the software application. Windows Installer was not designed as a monolithic script processing engine that can only have a list of files and registry keys fed to it. Instead, many of the subroutines within Windows Installer are configurable by the package developer. A package developer can configure whether the subroutine is called at all and what order it is called in, and the developer can apply if-then statements to these subroutines to have them run only if certain conditions are true.

 When the term *software application* is used throughout this book, it refers specifically to the actual files and registry keys built by developers that a Windows Installer package is designed to deploy.

It is important to understand that most of the package processing logic is in the database because this allows it to be customized. As would be expected, an administrator can customize which files and registry keys are copied during a package, however, the administrator can also customize the original package logic built-in by the vendor. This capability provides an unprecedented level of customization of vendor-provided software.

 Windows Installer actually has a small subset of SQL within it. It is used for package processing, and can be used to retrieve and write data to and from the database tables. To learn more, consult the Windows Installer SDK document “Examples of Database Queries Using SQL and Script.” A Windows Installer SDK script called `wirunsql.vbs` allows you to easily run arbitrary SQL commands on an MSI file.

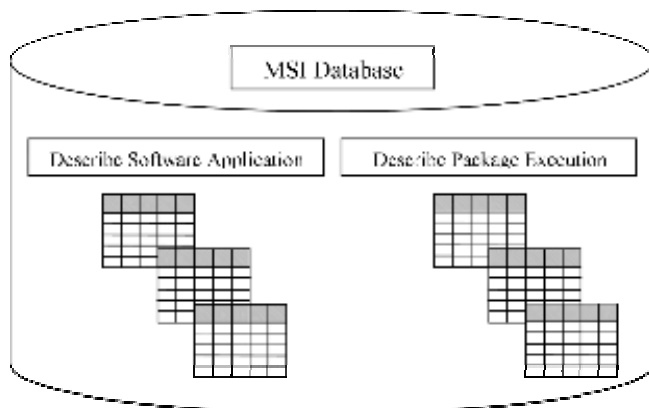



Figure 3.1: MSI database tables.

“Open” File Format

One of the greatest strengths to the MSI file format is that it is “open.” This does not mean that the MSI format is tied to the increasingly popular open systems movement; rather there is a specific standard for the format, and it uses existing, well-documented Microsoft file structures. Microsoft provides the APIs to read, open, and change these files. Because no particular tool vendor owns or controls the format, many tools can read and write the same MSI files. Additionally, a package created by a software application vendor is able to be opened by any IT professional. This openness allows administrators to customize vendor packages and gives vendors the flexibility to allow customization.

 Package developers can still insert custom functionality and proprietary approaches through the use of custom actions and custom tables—an open file format does not force package developers to expose everything they must do to ensure their software is installed and properly licensed.

How Packages Describe Software Applications and Installation Procedures


Windows Installer logically describes software application and installation procedures with the relational database mentioned earlier. The following section attempts to describe this schema in skeletal detail as a way of providing enough information to proceed on to more advanced topics. Every package engineer and administrator will have widely varying needs for more detailed study of this topic based on their individual experiences and company requirements for package building and troubleshooting.

Your key to mastering Windows Installer is to understand its language. The essence of this language is the framework provided by Windows Installers management meta data. The concepts you learn in this section will continually crop up in the following areas:

- Windows event logs
- Windows Installer logs
- Authoring tools
- The Windows Installer SDK
- Application deployment kits (such as the Office 2000 Custom Installation Wizard)
- Windows Installer command lines
- Group Policy

Software Application Information

Some of the tables in an MSI file store data about how the software application is structured. There are tables that deal with files, registry entries, INI file entries and shortcuts. Windows Installer also introduces a schema that describes the internals of the software application to Windows Installer. This schema defines two main logical entities known as Features and Components. Features and Components are the fundamental constructs that organize all the configuration details of a software application that is installed by the package.

 The handy term *component* has been severely overused in the technology industry. When used in the context of Windows Installer, the term *Component* has a very specific meaning. The term *COM Component* refers to compiled executable software that is registered in the Windows registry so that it can be located by many different programs. To confuse matters more, most COM Components will have a dedicated Windows Installer Component to define them in Windows Installer.

Previous setup technologies did not have a way for the OS to know the details of how elements of software relate. (For more information about the benefits of Windows Installer compared with early application management technology, see the sidebar “Application Management Before Windows Installer.”) The developer might know that three registry keys, four DLLs, and two INI settings are required for the database view feature to work, but there has not been a way to encode this *management meta data* in the packaging technology or the OS to facilitate intelligent application management.

Application Management Before Windows Installer

Long before Windows Installer, several innovative companies built intelligent application management technology for Windows that included self-healing and other benefits. Understandably, these technologies are expensive and heavily proprietary—sometimes taking a framework approach that requires usage of proprietary distribution mechanisms to take advantage of the packaging engines. Windows Installer has advantages over these approaches in that: it is free, it decouples distribution from packaging (which allows flexibility when building deployment solutions from different technologies), it generally makes packages more resilient for use in many deployment scenarios, and it is owned by Microsoft (which means all newer versions of Windows ship with Windows Installer).

Identification in Windows Installer

For many administrators, this section might be your first encounter with a programming concept known as a Globally Unique Identifier. GUIDs pre-date Windows Installer and have been used in many areas of programming as a result of their ability to create unique identities. You may have seen them in the registry in the CLSID subkey or HKEY_CLASSES_ROOT.



GUIDs are used throughout Windows Installer to identify most elements of a software package. A GUID is a 128-bit integer (“2 to the power of 128” possible values). GUIDs allow *unique identities* to be assigned to objects by many independent developers without a requirement for central coordination.

To understand how GUIDs work, think of 10,000 administrators using a packaging tool to generate 100 product codes each. None of the 1,000,000 products codes would be the same. GUID generation uses a special algorithm with many different seed values to ensure an extremely low probability of identical GUIDs being generated.

Here are some of the package elements that GUIDs are used to identify in Windows Installer:

- Package files
- Products
- Components
- Patch files

GUIDs are utilized directly during many Windows Installer activities. For instance, you might want to trigger a reinstall of an installed package and have the target computer determine from where the original package file was sourced. You can do so using the following command line, which uses the /f switch and product GUID to perform a re-install:

```
MSIEXEC /f {869A369E-6BD5-42e1-B9E9-B3543A46D5F6}
```

Component Structure and Attributes

As Figure 3.2 illustrates, Windows Installer Components are the fundamental unit that define the functionality of the software application. Components can have many types of associated resources. Some resource types include files, registry keys, shortcuts, and INI file settings. Some new attributes that are specific to Windows Installer can also be a part of a Component, including entry points, keypaths, and Component Codes. Although a Component can contain these items, it is not required to contain all of them.



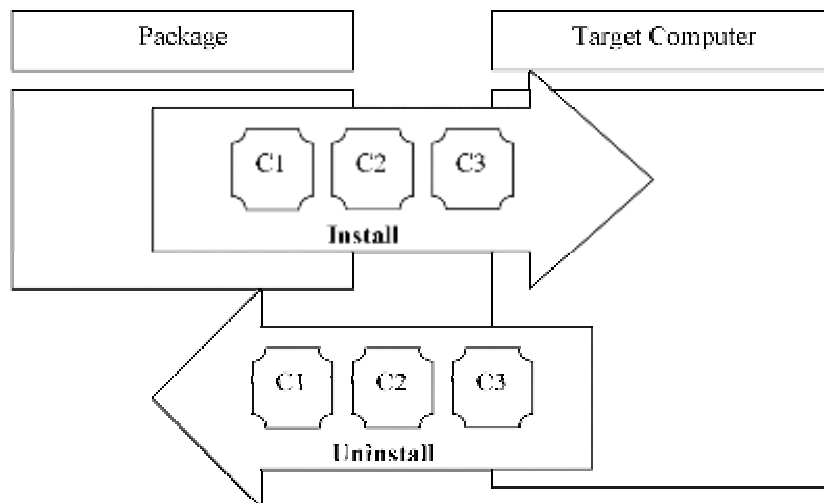


Figure 3.2: Windows Installer operates on lists of Components.

Components are the fundamental unit that Windows Installer manages. Any operation such as installation, maintenance installation, self-healing, and uninstall result in a list of Components that must be operated on to achieve the desired result. Components are also reference counted to prevent uninstallation when more than one application is using a shared piece of software. (For more information about reference counts, see the sidebar “A Brief History of Reference Counts.”)

A Brief History of Reference Counts

Reference counts (refcounts) were introduced with Windows 95. All installation programs that follow Microsoft’s installation guidelines increment a counter in the registry whenever a DLL is installed to a shared location, such as the system directory.

For example, if four applications had installed abc.dll to the System32 directory, that DLL would have a refcount of 4 in the registry. If one of the applications is uninstalled (again assuming installation guidelines are followed), the uninstall program would simply change the refcount to 3 and leave the file in place because other applications are obviously using the DLL. If the refcount for a file is 1, the uninstall program is free to remove the file because it can assume that the program being uninstalled is the only program using the file. Occasionally, uninstall programs will break other software because they remove shared registry keys required for a DLL to work properly. Windows Installer improves on refcounts by putting them at the Component level. Because a Component contains all the various system resources required for a DLL to operate properly, these related resources will remain on the system if other software is still using the DLL.

Component Name

As Figure 3.3 shows, Components have a friendly name. This friendly name displays in most authoring tools. The friendly name, however, is not how a Component is ultimately identified. A Component is identified by its Component Code. Component names make the processes of authoring and updating packages easier so that we do not have to remember 128-bit hexadecimal integers.

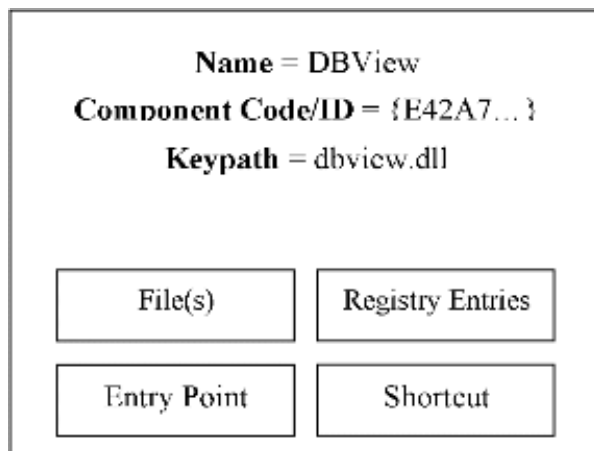



Figure 3.3: Component definition.

Component Codes

Component Codes (or Component IDs) are the identifying attribute for a Component. Component Codes are GUIDs that uniquely identify a Component across the world. In theory, a Component Code should be unique among all Components in the world.

 For more detailed information about Component structure and identity rules, refer to the Windows Installer SDK document titled “Organizing Applications Into Components” and its sub-documents.

Keypaths


Through the concept of a Component, Windows Installer uses meta data to model a functional unit of the application software it is describing. This Component definition is placed in the repository of any machine on to which it is installed. However, if the Component becomes broken, how does Windows Installer tell that the Component is not installed as defined in the repository? This is where the keypath comes in.

For each Component that is installed on a computer, Windows Installer checks the existence of a specially tagged resource (known as a keypath) within the Component to determine whether the Component is healthy or in need of repair. If this tagged resource is missing, the entire Component is re-installed. A keypath can be a directory, a file, a registry key, or an ODBC data source.

The reason that Microsoft Word still works when winword.exe is deleted is because winword.exe is the keypath of a Windows Installer Component. A computer with Office XP installed would have a Component definition in its Windows Installer repository that describes winword.exe. When a user attempts to use Microsoft Word, Windows Installer checks to see that the keypath of this Component (winword.exe) exists. If it does not exist, self-healing would be invoked to fix the problem. There are other details to how self-healing works that will be covered later.

Among the uses of a keypath, three are very relevant to administrators:


- Self-healing detection
- Advertising/Install-on-Demand detection
- User profile fix-up detection (special case of self healing)

 Although not a formal term, *user profile fix-up* refers to a lesser known feature of Windows Installer that lets installed packages properly set up a user profile when the user has not previously used the application. This functionality works even when the user has previously logged on to the computer.

When a user starts an application, standard self-healing checks are performed. If the package is structured correctly, Windows Installer will perceive the lack of user information for the application as being “broken” (even though they never existed) and self-heal the user portions of the package.

Entry Points and Advertisements


Ever wonder how Windows Installer knows to get involved with repairing or installing an application? Entry points allow Windows Installer to proxy the startup of an application and perform application management tasks before the user is allowed to access the application. In other words, when you double-click the icon for a Windows Installer packaged software application, it does not actually attempt to start the application directly. The icon is a special icon that asks Windows Installer to find the software application and start it. This is when Windows Installer can use the MSI repository information, the installed application resources (files, registry keys, and so on), and the original package file to perform the magic of self-healing and install on demand.

 For entry points to work correctly in Windows NT 4.0, you must have a newer version of the shell installed. To update the shell, install Internet Explorer (IE) 4.01, SP1 with the Active Desktop or IE 5.x with the Windows Desktop Update. You can also update the shell when deploying a customized Office XP installation. See the Office XP NT deployment Web site for more details (<http://www.microsoft.com/office/ork/xp/one/depd01.htm>).


An entry point turns into an *advertised interface* when any Feature that its Component belongs to is advertised or installed on a target computer system. When a Windows Installer package is advertised, advertised interfaces make it appear as though the application is installed and ready to use. When a Windows Installer package is installed, advertised interfaces trigger Windows Installer for self-healing and user profile fix-up checking. An entry point/advertised interface can be:

- A shortcut (special Windows Installer shortcut)
- A document extension (association)
- A MIME type (Internet document types)
- A Class ID (CLSID)—Programmatic identities used for sharing software within and between various applications



 For Windows Installer functionality to work as expected, users must launch applications from Windows Installer shortcuts. If users in your organization are accustomed to creating their own shortcuts by right dragging and dropping application executables, these shortcuts will not trigger self-healing or any other Windows Installer functionality. Unfortunately, it is not easy to prevent users from doing this—it will be necessary to re-culture them through Help desk interaction and other types of communication. Windows Installer shortcuts created by the installation package (on the Start menu or desktop) can be copied to new locations. Windows 2000 (Win2K) and later allow right-dragging shortcuts right out of the Start menu. One drawback is that they are not upgraded when the underlying package is upgraded, so they may not work after a major upgrade to the software application.

Advertisement of document extensions, MIME types, and CLSIDs are all accomplished by configuring the registry on the target computer; however, Windows Installer does not internally store this information as registry keys. Advertising data is stored in special tables and does not become registry entries until the package is installed on the target computer.

 When first starting with Windows Installer, it can be easy to confuse advertised interfaces with advertising an application to users. Even if you never plan to advertise applications (make them appear as installed, but they actually install on first use), you will still need advertised interfaces in your package if you require self-healing or user profile fix-up to work properly.

The following list provides a summary of information we have covered about Component structure and attributes:

- File resources—Components can contain file resources. If a file resource is the keypath to the Component, it is known as the key file. If a file is not the keypath, it is known as a companion file. There is no practical limit on the number of files or file types that can be in a Component. There are, however, rules about Component structure that define when certain types of files should have an entirely dedicated Component.
- Registry resources—Registry resources are registry keys that are required by the Component.
- Shortcut resources (entry point)—Shortcuts are defined within a Component and must point to a file *within the Component*. Shortcuts can be advertised (entry points) or standard Windows shortcuts.
- Document extension mappings and MIME types (entry points)—Document extensions and MIME types are configured at the Component level and point to a file *within the Component*.

- Additional resources and attributes—Components can have many resources and configuration items associated with them. Some of these include:
 - Controlling and installing services
 - Making INI file entries
 - Creating directories
 - Setting environment variables
 - Configuring ODBC

Typical Components

When I first heard Components described, I thought they would be something like spell check and contain 3 executables, 14 DLLs, 10 registry keys, and so on. It turns out that this type of item (spell check, for example) would be a Feature that contains multiple Components. One of the things that helped me understand Components was learning what typical Components are like, as Table 3.1, Table 3.2, and Table 3.3 illustrate.

Component Item	Typical Configuration
Keypath	The code file.
File Resources	Only the code file and any required data files.
Registry Resources	COM Registration (CLSID) keys and data keys.
Advertisements	Any registry entries, extension mappings, CLSIDs, or ODBC data sources associated with the file (if any).
Service Settings	Any service control and installation items associated with the code file.

Table 3.1: Executable Code Component (EXE, DLL, OCX).

In repackaged applications, most of the registry keys for an application may be contained in a couple of Components (one for HKEY_CURRENT_USER and one for HKEY_CURRENT_MACHINE) except if they are explicitly required for a component to operate correctly. For packages received from a software vendor, most of the registry keys may be with the primary application executable.

Component Item	Typical Configuration
Keypath	A registry key in the relevant hive that should always be present if the Component is installed.
File Resources	None.
Registry Resources	Registry keys for HKEY_LOCAL_MACHINE or HKEY_CURRENT_USER.
Advertisements	Any registry entries, extension mappings, CLSIDs or ODBC data sources associated with the file (if any).
Service Settings	None.

Table 3.2: Registry key Component (HKEY_LOCAL_MACHINE, HKEY_CURRENT_USER).



Package developers have more flexibility in areas such as Components that contain templates for the software application. If templates were critical to this application, each one could be a dedicated component.

Component Item	Typical Configuration
Keypath	The template directory or a single template file.
File Resources	All templates.
Registry Resources	COM Registration (CLSID) keys and data keys.
Advertisements	Any registry entries, extension mappings, CLSIDs or ODBC data sources associated with the file (if any).
Service Settings	None.

Table 3.3: Templates Component (template files for software application).

Features

After you have a basic understanding of Components, Features are quite easy to understand. Features are buckets (container objects) for Components. Features have very few attributes assigned directly to them, they are actually the sum total of the Components contained within them.

Although Features are simply buckets for Components, many of the configuration capabilities of Windows Installer operate on Features. For instance, you can advertise a Feature, but not Components. If you advertise a Feature and 3 advertised interfaces appear, you know that among the Components that make up that feature, there are 3 entry points defined. You can find out exactly which Components contain these items by examining the Components that make up the Feature. Features have some unique attributes. These include:

- Windows Installer configuration commands operate on Features (installing, advertising, uninstalling, and so on)
- Self-healing, install-on-demand and user profile fix-up (discussed in an earlier note) operate at the Feature level
- Features can contain other Features
- Features can be arranged in hierarchical relationships (by being contained by other Features)
- Features contain Components
- Multiple Features can contain the same Component
- Features are **NOT** identified by GUIDs but rather by a Feature Identifier, which is a text string

By contrast, Components do not have these attributes—they cannot contain Features or other Components, they cannot be arranged in hierarchies, and they are not addressed directly through the command line to accomplish installation and configuration activities.

Earlier, we talked about how Windows Installer essentially operates on a list of Components. We can modify this concept by understanding that we specify that list of Components to Windows Installer by using convenient buckets called Features, as Figure 3.4 illustrates.

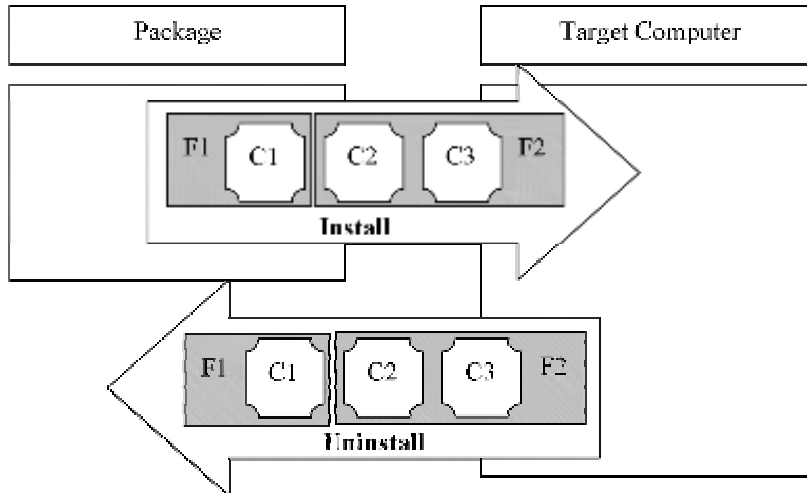


Figure 3.4: Windows Installer operates on lists of Components that are grouped by Features.

Most of the attributes assigned directly to Features are concerned with how these Features are displayed in the Feature selection dialog box presented by Windows Installer during an interactive install. Every package will have a root Feature that is always installed.

Package Execution Information


Even though the package processing engine is built into the OS, much of the engine's functionality can be controlled from within a package file. This allows administrators to customize the actual logic used to install packages, even when packages come from software vendors or in-house programmers. Previous to Windows Installer, setup program processing logic was inaccessible because it was compiled into binary executable files (EXEs) and could not be altered.

Standard Actions

As mentioned earlier, Windows Installer is not a huge block of code that simply processes a package. There are many subroutines within Windows Installer that are called during package installation, configuration, and uninstall. These subroutines are partially configurable through the Windows Installer database in a package. These subroutines are called *Standard Actions*. Standard Actions can be configured in three ways, they can

- Be included or not included
- Be reordered
- Have if-then statements (conditions) placed on them to control their execution

Although Standard Actions are configurable in these ways, there are still many rules about which Standard Actions should be included as well as ordering dependencies on other Standard Actions. The SDK's Standard Action reference should be studied before attempting to reorder any of them.


 For more information about rules for reordering Standard Actions, refer to the Windows Installer SDK document titled "Standard Actions Reference" and all of its sub-documents. The Windows Installer SDK files also include a template for the default set of actions that would be expected in a generic package. This file is called Sequence.msi and can be found in the MSI SDK directory \Program Files\Microsoft SDK\Samples\SysMgmt\Msi\Database.


Custom Actions

Custom Actions allow package developers to extend Windows Installer with just about any functionality they desire. Custom Actions have information available to them about the running installation. Only certain types of items can be called as a Custom Action. Some of the most relevant are:

- Calling DLLs
- Calling EXEs
- Calling a VBScript
- Calling a JScript
- Setting a property

VBScript tends to be the popular choice among administrators who need to create Custom Actions primarily because VBScript can be used for many diverse administrative scripting needs. In addition, VBScript is similar to other scripting languages administrators might already use.


 Setup tool vendors also allow you to use their proprietary scripting languages as Custom Actions. For example, Wise Package Studio allows compiled Wise Script to be used as a Custom Action and InstallShield allows InstallScript to be used.

 Windows Installer 2.0 has new error logging features for scripted Custom Actions. Previous versions simply reported that a scripted Custom Action had failed and gave the Custom Action name. Windows Installer 2.0 (shipped with Windows XP and Win2K SP3 and is downloadable) will log the actual error and the script line number where it occurred.

Like Standard Actions, Custom Actions can have their sequence controlled and conditions placed on them.

Sequences

We have been discussing how the order of Standard Actions and Custom Actions can be controlled. Windows Installer also supports the ability to have multiple sets of ordered actions. These ordered sets are called *sequences*. Sequences help organize installations. There are two sequences involved in an interactive installation, as Figure 3.5 shows. The Install UI sequence contains all the actions (including dialog boxes) required to gather information from the user during an interactive installation. The Install Execute sequence handles changes to the system such as copying files and updating registry entries. This two-sequence approach is also used for silent installs—the entire Install UI sequence is simply skipped when an installation is run completely silent.

 Silent installations are utilized heavily in automated software deployment. Most administrator-authored Custom Actions will need to be placed in the Install Execute sequence to ensure that they are executed during silent installations.

Standard packages (built according to Microsoft templates and guidelines) also have four other sequences. The Advertising UI and Advertising Execute sequences are used when a package is advertised using MSIEXEC or Group Policy deployment. The Admin UI and Admin Execute sequences are used when a package is used to build an administrative install location.

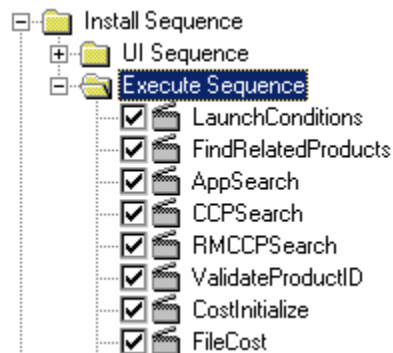


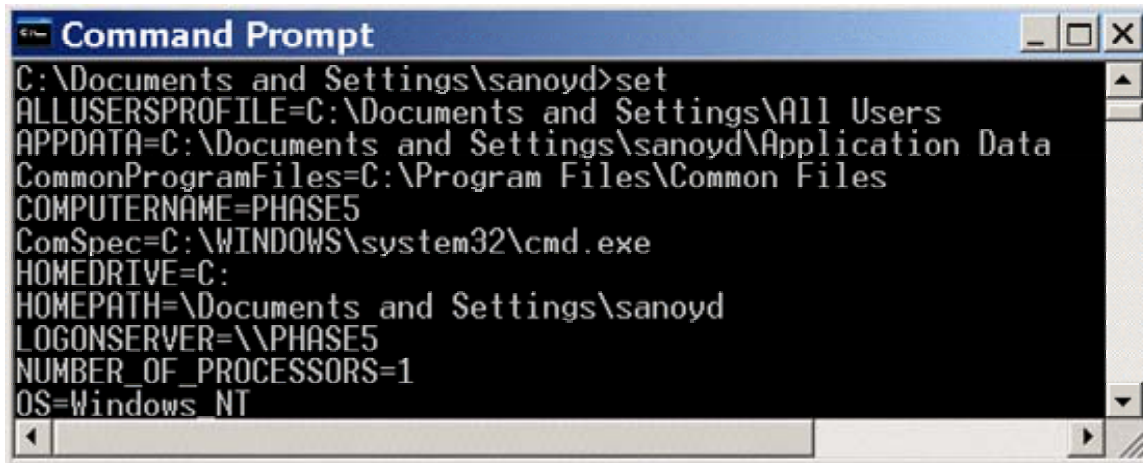
Figure 3.5: Sequences and actions.

Uninstalls and maintenance installs are handled by the Install UI and Install Execute sequences. When specific actions are only relevant to a specific install type, such as uninstall, conditions are used to ensure that those actions only execute when appropriate.

Authoring tools will represent sequences in different ways, but essentially they are interpreting a table that simply has the action name and an associated sequence number. There is a separate table for each sequence. Although a rare occurrence, package developers can create their own custom sequences if desired.

Properties

Windows Installer uses Properties to store package data before and during package processing. They are the equivalent of a variable in a scripting or programming language. Properties are similar to environment variables. As Figure 3.6 illustrates, environment variables provide system information (such as computer name and OS). They can also be used to store data in batch file scripts. For instance, a script might prompt the user to choose a menu item—an environment variable could be used to store that choice for later use.



```

C:\Documents and Settings\sanojd>set
ALLUSERSPROFILE=C:\Documents and Settings\All Users
APPDATA=C:\Documents and Settings\sanojd\Application Data
CommonProgramFiles=C:\Program Files\Common Files
COMPUTERTNAME=PHASE5
ComSpec=C:\WINDOWS\system32\cmd.exe
HOMEDRIVE=C:
HOMEPATH=\Documents and Settings\sanojd
LOGONSERVER=\\PHASE5
NUMBER_OF_PROCESSORS=1
OS=Windows NT
  
```


Figure 3.6: Set command output.

Properties behave like environment variables and scripting variables in other ways as well:

- Properties do not have data types, they can store numeric or string data
- Properties do not need to be declared before use—they can be created on the command line, in transforms or by custom actions
- Properties are used to store data about the system

Properties are used store all kinds of data and control parameters. They store data and control parameters such as:

- Installation progress
- Data collected by locator tables (such as registry keys)
- Type of installation activity (such as install, uninstall, rollback, and so on)
- Data about the target system (such as OS version and user profile location)
- Current date and time
- Control information for installation activities (such as the list of features to install or advertise)

 Properties can be created on the fly, so do not assume that the property table in a package is a comprehensive list of all properties used or created by the package.

Properties have several classes that determine how they can be manipulated during package operations. The class of a property is determined by the text case of the property name and whether it is in the SecureCustomProperties property or one of the built-in Restricted Public Properties.


- Private Properties can only be changed by transforms and custom actions—they cannot be changed on the command line. Private Properties must have at least one lowercase letter.
- Public Properties can be changed on the command line or in the installation UI in addition to transforms and Custom Actions. Public Properties must contain only upper-case characters.
- Restricted Public Properties can only be changed by administrators or if the EnableUserControl policy is turned on. Restricted Public Properties must contain only upper-case characters AND be either on the list of built-in Restricted Public Properties or added to the SecureCustomProperties property if they are a custom property.

Any of these property types can be built into Windows Installer (known as default) or defined by the developer (known as custom).

Properties are also used by MSIEXEC as command-line arguments. This can be a little hard to get used to because MSIEXEC also uses switches that start with the forward slash character. The following command line shows that applying a transform during package installation is done using the TRANSFORMS property rather than a special command-line switch:

```
MSIEXEC /I package.msi TRANSFORMS=custom.mst
```


In this example, the /I is an MSIEXEC switch and TRANSFORMS is a property.

 When starting out with Windows Installer, it is important to familiarize yourself with all the built-in properties and the information they communicate or the functions they control. Consider reading through all the information in the “Properties” section of the Windows Installer SDK as a good primer.

Notable Properties

There are several notable properties that will be used many, many times. Most of them control how a package is installed:

- TRANSFORMS—Specifies a list of transforms to apply to an MSI during package installation.
- ADDLOCAL—Lists features to install on the local computer.
- ALLUSERS—Controls whether installations are performed for all users of the computer or just the user running the installation.
- ROOTDRIVE—Controls which drive Windows Installer installs packages on—by default packages are installed on the local drive that has the most free space.
- INSTALLDIR—Controls the exact directory to which a package must be installed.
- REBOOT—Controls whether the package requests a reboot after installation.

 When properties are specified in multiple places, Windows Installer has a method for determining which value should be used. Examine the MSI SDK document titled “Order of Property Precedence” for more information.

Self-Healing Overview

Self-healing is the ability of Windows Installer to detect and repair any critical resources that are required for the user to successfully launch and use the application. Every resource of a package is not checked during self-healing. Because self-healing occurs as the application is launched, exhaustive checking of every resource would lead to excessive wait times.

Earlier we discussed how Windows Installer performs basic actions (install, uninstall, and so on) on lists of Components. We also discussed how these lists of Components were specified by a list of Features. Self-healing follows this approach as well.

Self-healing, install-on-demand, and user profile fix-up are all variations on the same functionality provided by Windows Installer. Windows Installer is asked to find the appropriate software application when an entry point is activated by a user (usually double-clicking a shortcut or document type). If Windows Installer finds the software is not yet installed, it will immediately install it. If the software is installed, it will be verified by self-healing. In both cases, this happens at the Feature level.

As Figure 3.7 illustrates, when an entry point is activated, the Component to which the entry point belongs is checked for which Feature it is attached to. *Every* component in that Feature is checked for non-existence of the keypaths. If *any* single keypath is missing, the entire feature is reinstalled.



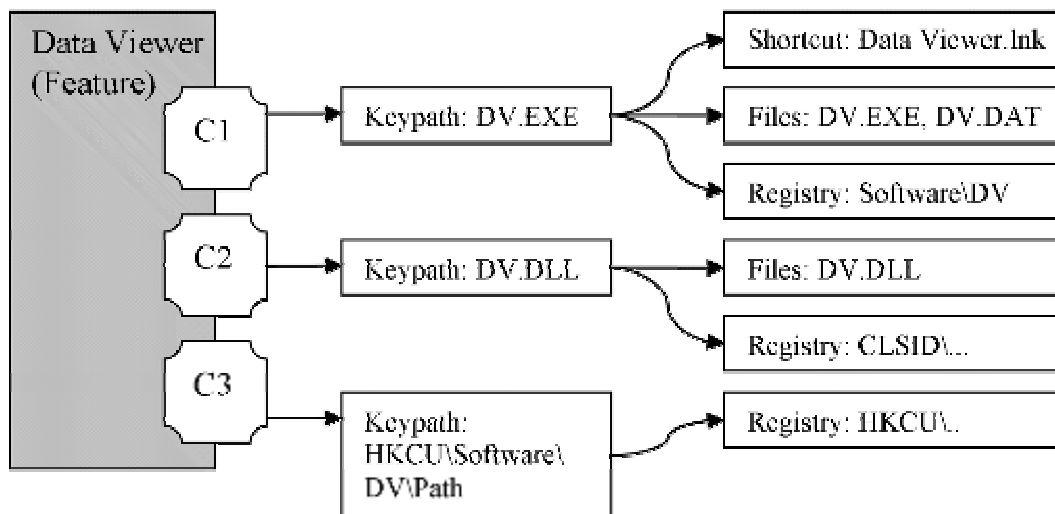


Figure 3.7: Self-healing component structure.

For example, say the Component in Figure 3.7 was installed on a computer. After a couple of months, someone accidentally deletes the file DV.DLL. The next time the user launched the shortcut Data Viewer.lnk, the files DV.EXE, DV.DLL, and the registry key HKEY_CURRENT_USER\Software\DV\Path would be checked for existence. If any of these three resources were missing, the entire Feature (which is made up of the Components C1, C2, and C3) would be reinstalled. This is why self-healing results in much more installation activity than a single component re-installation.

Self-healing will not repair resources (mainly files and registry keys) if the keypath of the component they belong to is properly installed on the system. For example, if DV.DAT in Figure 3.7 was missing, it would not be self-healed if DV.EXE was present on the system. To compensate for this, users can be taught to use the Repair option in Add/Remove Programs. This option does a full re-install of the application and will fix problems with missing resources that are not fixed by self-healing.

Summary of Package Structure Concepts

Windows Installer introduces an entire level of application management meta data that is fundamental to creating the many new features and capabilities Windows Installer is famous for. Although it is not a simple task to become familiar with the structure, rules, and terminology of this meta data, doing so unlocks many secrets!

Here are some of the highlights:

- Windows Installer describes software applications using a set of database tables.
- Major aspects of how the package is processed are also described in this database.
- To accommodate the new paradigm for installations, new logical entities are defined by Windows Installer to break down the software application into manageable sub-parts.

- These logical entities are known as Components and Features. Components and Features allow Windows Installer to map the relationships between specific software application resources (such as files and registry entries) for use in management activities such as self-healing, sharing of application resources, and install-on-demand.
- Windows Installer defines additional entities for managing package processing—these entities, known as Actions and Sequences, control the behavior of an installation package while it is performing installation, configuration, and uninstall activities.
- The design of Windows Installer allows package developers to have a large degree of control over Windows Installer’s internal functions. This same design allows administrators the same level of control even after a software vendor has built its completed installation package—something not possible with previous setup technologies.
- Variables in Windows Installer are known as properties; they store all types of control information and data for packages. Custom properties can be created by package developers.

Figure 3.8 illustrates how all the internals of a package are utilized to accomplish installations. The numbers correspond to the following discussion.

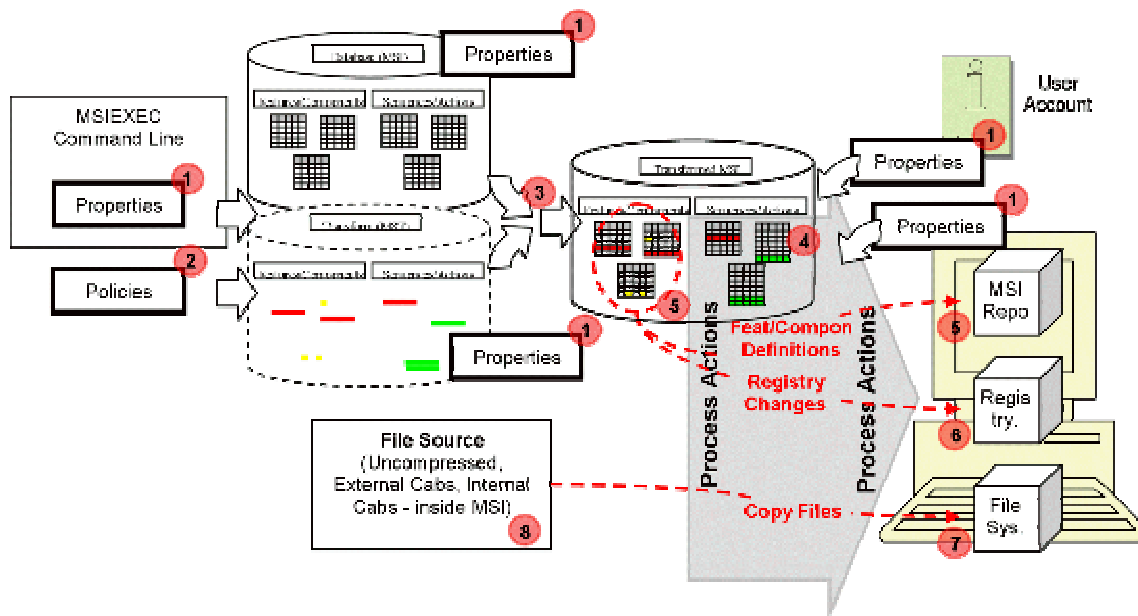


Figure 3.8: Package processing internals.

Properties **1** are a dominant element because they are used to control installations, gather information from the target computer and user account, and for custom functionality. Policies **2** (which are not stored as properties) are read by Windows Installer as needed; some are enforced before any package processing begins. Policies control many behaviors of Windows Installer—they are covered later in this chapter. If


transforms exist, they are read and applied to the package file **3**. Windows Installer processes the actions contained in the relevant sequence(s) to install the package **4**. Package processing causes the package Features and Components to be installed **5**. This step includes copying the Feature and Component definitions into the target computer's MSI repository. When Features and Components are processed, all changes are made to the target system, including creating registry entries **6** and copying files **7**. Files are copied from the source **8**, which can be stored as uncompressed, compressed CAB files, or internal CAB files (inside the MSI file).

Customizing Packages

One of the most powerful benefits of Windows Installer is the ability for administrators to customize installation packages regardless of who built them. Previous to Windows Installer, most software could only be effectively customized through the use of manual installation or repackaging. The following list highlights some of the difficulties in software deployment that result from the inability to customize software installation packages:

- Manual installation of software has very high cost.
- User installation of software creates higher costs due to misconfiguration and end user self-support.
- Repackaging introduces quality risks due to incorrectly installed software applications.
- Repackaging might violate some software vendor's support agreements.
- Enterprise-wide repackaging creates additional costs and requires disciplined processes.

Fortunately, Windows Installer has been designed with these challenges in mind. The primary method for customizing an installation package is known as a *transform*. Transforms are a separate file with an .MST extension. They are specified during the installation of a package.

 We briefly explored transforms in Chapter 1.

After our entire installation is modeled in a database, customization is easily accomplished by adding, modifying, or eliminating database rows and cells from various tables. Eliminating a Component from an installation can simply require removal of three rows from three tables and a change to the value of one cell in a row of a fourth table. To add a Feature with two existing Components might only require new rows in two tables.

Transforms use a concept called *overlay* to accomplish customization. Instead of permanently changing a package file's database, overlays are done using a temporary copy of the database created during installation. This allows for many different customizations to be done from the same MSI file on disk because customizations can be picked at install time. Overlays are an extremely flexible method of customization because more than one transform can be used at once.

Transform files are *deltas*, which cannot be used standalone because they only contain the changes you want to make to an MSI file. Anytime you want to create, edit, or apply an MST file, the MSI file it is based on is required to work with the transform.

Because transforms can include changes to the package logic, they cannot be applied to a package that is already installed on the workstation, but must be specified with the initial package installation or advertisement.

Figure 3.9 shows that a transform is just a delta of information that requires the original package to form a complete customized installation. It also demonstrates the overlay concept, whereby loading the transform on top of the MSI file gives the complete picture.

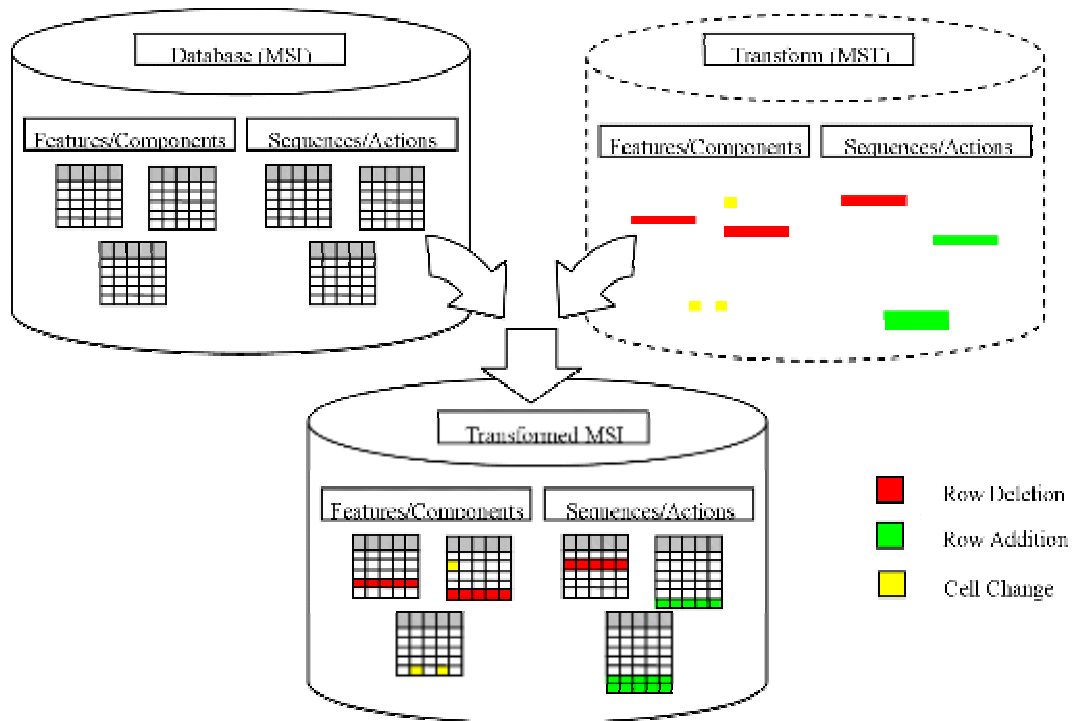


Figure 3.9: How transforms work.

When transforms are applied to a package installation, they are copied locally into a cache. These cached copies are applied to any subsequent reconfigurations of the application so that customizations stay intact.

For packages built by administrators, transforms might be less useful because customizations can be integrated directly into the MSI packages. However, in large enterprises and for repackaged installs that have many possible configurations, transforms are an effective means of customizing repackaged software.

Transforms should be used for customizing all MSI packages received from software vendors—vendor MSI packages should not be directly edited. This is not simply a best practice, but an expectation and assumption of software vendors, Microsoft, and the Windows Installer SDK.

Managed Application Settings

Windows Installer was released with the suite of Win2K technologies known as IntelliMirror. Group Policy is the companion technology that provides deployment and application settings management by way of dynamic policy settings. Compared with Windows Installer, Group Policy provides superior capabilities for managing application settings because they are applied when users log on and at regular intervals afterward.

Since the introduction of Win2K, many organizations have been hampered in deploying Active Directory (AD). Those who implement it rarely burden the directory with exhaustive settings management for all applications in the enterprise. This raises some challenges for package deployment with regard to settings that must be actively managed.

When an application setting is made in a package file or transform, the setting usually consists of one or more registry entries. After the software is installed, Windows Installer will ensure that the same settings are made if self-healing is required or if a new user logs on and uses the application.

A problem arises when one of these registry keys needs to be changed to a different value. Previous to Windows Installer, most administrators would run a simple script to fix up the registry keys on all existing machines. This can still be done, but it leaves out several important scenarios that Group Policies would catch:

- Some self-healing scenarios can set the application setting to the older value contained in the package file.
- New installations of the package after the fix has been set will use the older value.
- Multi-user machines will not have the older value for all existing users.
- New user logons to multi-user machines will have the older value.

To handle these situations purely with Windows Installer technology would require that the application setting be updated in a transform. Either all computers would need to uninstall and re-install the software, or an upgrade package would need to be created and deployed. The later only works if the package is not from a software vendor because you should not create upgrade packages for MSI packages received from a software vendor.

The importance of catching every one of the exceptions is relative to how critical the fix is and how it affects specific user communities. If the problem is blue screening computers on a stock trading floor, it is essential to eliminate any possibility that the old setting is put on any computer. If it creates a minor annoyance to users, it might not be as critical to prevent every case of the old setting being installed. Without some type of policies mechanism, there is no clear path for how to handle this issue, but it is important to be knowledgeable of it and discuss it early in the design of application management and packaging processes.

☞ This is more of a hack than a tip. Windows Installer does not validate that a cached transform file is the exact same file that was used during the original install. For computer-based installations, the location of these files is easy to determine. Administrators can replace the cached transform with an updated copy and simply trigger a maintenance installation or re-install to change application settings *after* deployment. However, there must have been a transform deployed with the original installation for this hack to work.

If an organization has policy setting capabilities of any type—such as NT or Windows 9x System Policies or third-party policy management systems—this problem can be resolved by using the policy mechanism. To prevent overburdening the policy mechanism with application settings, it is prudent to only use it for settings that absolutely must be managed.

Creating Transforms for Application Settings

There are essentially three common types of transform-creation tools. In the previous chapters, we discussed vendor-supplied tools and third-party tools that step through the user installation interface and essentially automate the choices a user would make during a package installation.

In many cases, administrators will need to automate settings that are not exposed using these tools. Detailed settings such as the application data directory or back-end database server are usually only exposed in the most advanced customization tools such as the Office Custom Installation Wizard.

Tier-1 packaging tools generally include a lower-level tool for creating transforms. With this type of tool, the package developer loads the MSI to be customized, then uses the authoring tool as though editing the MSI itself. All the capabilities of editing an MSI are available, but the changes are saved in a transform and the original MSI is left changed.


Although these tools are very powerful, they do not assist in discovering the system changes (registry and INI files) that equate to configuration changes performed from within the software application. A configuration monitoring tool must be used to actually discover the required settings. You might be able to use the repackaging portion of your MSI packaging tool or select any tool that can effectively monitor and report system changes.


📄 Wise Package Studio Professional has a very helpful HTML-formatted “change detection report” that is automatically created in the same directory as the package. The report does not automatically display, so you have to know its there to use it. This report is a great source of information for finding which registry keys and INI settings were changed while configuring a software application.

Here is a method for creating transforms for advanced settings using the tools and skills you are already familiar with:

1. Start up the software application you want to customize.
2. Start your configuration change monitoring tool (such as your repackager).
3. Proceed to make the desired configuration changes to your software application.

4. Ensure that these changes are “committed”. This step may require some experimentation due to the differences in how software applications are programmed. Many applications will save configuration changes when you click Apply or OK in the configuration dialog box. Keep an eye out for software applications that delay the saving of configuration changes until a later time or until you exit the application.
5. Stop your change monitoring tool.
6. Examine the output of the change monitoring tool.
7. If your transform tool allows importing of registry data directly from the computer on which it is running, open the transform tool and create a new transform. Use the output from the configuration monitoring tool to determine which registry entries to copy from the current workstation into the transform.

 The last step in this procedure is counterintuitive to many packaging processes. This is because extremely clean packaging processes usually call for the package editing tool to be run on a separate computer from the repackaging tool due to the changes made to a workstation by the package editing tool. In this case, the package editing tool is being used to copy only selected items from the repackaging workstation, not to generate the initial package as it is with repackaging.

 There are several interesting transform tools and scripts in the Windows Installer SDK: *Wigenxfm.vbs* and *Msitran.exe* can generate a transform by comparing two MSI files. For scripting, this is the only way to actually create a transform.

Wiuexfm.vbs and *Msitran.exe* can permanently apply the contents of a transform to a database.

Wilstxfm.vbs lists the contents of a transform in a command window.

Using Transforms

Transforms must be specified during installation or advertisement of an application. The switches used for each of these scenarios is quite different. In Chapter 1, we briefly explored how to apply transforms to an existing MSI package. To review, here is the command line for applying a transform when installing a package:

```
MSIEXEC /i mypackage.msi TRANSFORMS=mycust.mst
```

The special public property TRANSFORMS causes the transform to be applied. Here is the command line for applying a transform when advertising a package:

```
MSIEXEC /j[u,m] mypackage.msi /t mycust.mst
```

For advertising, a special switch and sub-switch are used for transforms. The */j* switch indicates that the package will be advertised. The */j* switch is directly followed by a *u* for a user-based advertisement or an *m* for a computer-based advertisement. The */t* switch is a sub-switch of the */j* switch, and can only be used in conjunction with the */j* switch.





It can be easy to get confused and attempt to use the /t switch with the /i switch.

Administrative Installs


Chapter 1 briefly discussed administrative installs. In this section, we will dive a little deeper. Administrative installs are not really installations of a package, but rather a special preparation of your package to allow it to be installed from a network. A client installation must still be done for each computer that needs to run the software application.

The following is a list of the main uses for an administrative installation. Knowing this list will help you understand whether they can play a useful role in your environment.

- Pre-setting properties with the AdminProperties property—With normal installations, double-clicking an MSI file results in the full installation interface and no special command-line properties are applied. This can result in leaving out the TRANSFORMS property, which might cause important customizations to be omitted. Administrative installations allow the administrator to specify a list of properties (*not* MSIEXEC switches) to be used when the MSI is double-clicked. This is useful if you have distribution scenarios in which users are either directed to run MSIs from the network or they can easily find them on the network.
- Served applications—Windows Installer natively supports running applications executables from a server rather than from the local hard drive. This is known as *Run from source* in Windows Installer. To do this, an administrative installation must be created. A unique aspect about Windows Installer served applications is that they can be fault tolerant. If multiple administrative installations are made available and specified as sources for the applications files, Windows Installer will check the list of sources until it finds one that is available.
- Pre-activation of Microsoft products—Microsoft Products that require activation can only be pre-activated if they are setup as an administrative install. There are other ways to ensure that activation does not require user intervention.
- Reduce back-end replication—In large networks, the amount of data passed over the network to distribution servers can cause network load problems. Because administrative installs can be patched, using administrative installs can reduce back-end (server to server) replication. Patches have the potential to dramatically reduce the amount of data transfer required because they only contain binary deltas of files that change between two versions of a package. The benefits of reduced bandwidth for maintenance must be balanced with the uncompressed format of administrative installs—which use up to twice as much disk space as an MSI that has the application files compressed within internal or external CAB files.



- Extract only needed files from a software CD-ROM—Many software vendors send out a single, large CD-ROM that has all or most of their software packages—especially if their software is under site or blanket licenses. Some CD-ROMs may contain files required to deploy applications in multiple spoken languages. Such a CD-ROM might contain more than 400MB when all that is needed is a 10MB application. Performing an administrative installation with the desired MSI file will extract only the needed files to the network. This approach only works if the individual applications on the CD-ROM have their own MSI files rather than one large MSI file.

 Administrative installations are also required to create patches. Most authoring tools will automatically create the administrative installations for you if they do not exist before you start the patch tool.

Building and Using Administrative Installs

During an administrative install, the package is prepared to be installed from the network. Figure 3.10 illustrate that this process extracts all files into a directory structure and copies the MSI (without embedded files) to the root of the administrative install location. If used, the ADMINPROPERTIES value is also embedded as an additional stream at this time. An administrative install share is created by using the /a command-line switch with MSIEXEC. When performing an administrative install, there is usually only one dialog box requesting a network location for the install. Windows Installer does not check whether the location is actually on the network, so this location can be local if you are simply testing a package. Here are a couple samples command lines for setting up administrative install shares:

- `MSIEXEC /a mypackage.msi`
Prepares the package in the directory specified on the wizard dialog box that appears after this command line is run.
- `MSIEXEC /a mypackage.msi ADMINPROPERTIES = "TRANSFORMS=mytrans.mst"`
Prepares the package in the specified directory and embeds the special property ADMINPROPERTIES to be used upon client installation.
- `MSIEXEC /a mypackage.msi /p myfix.msp`
Applies a patch to an existing administrative installation.

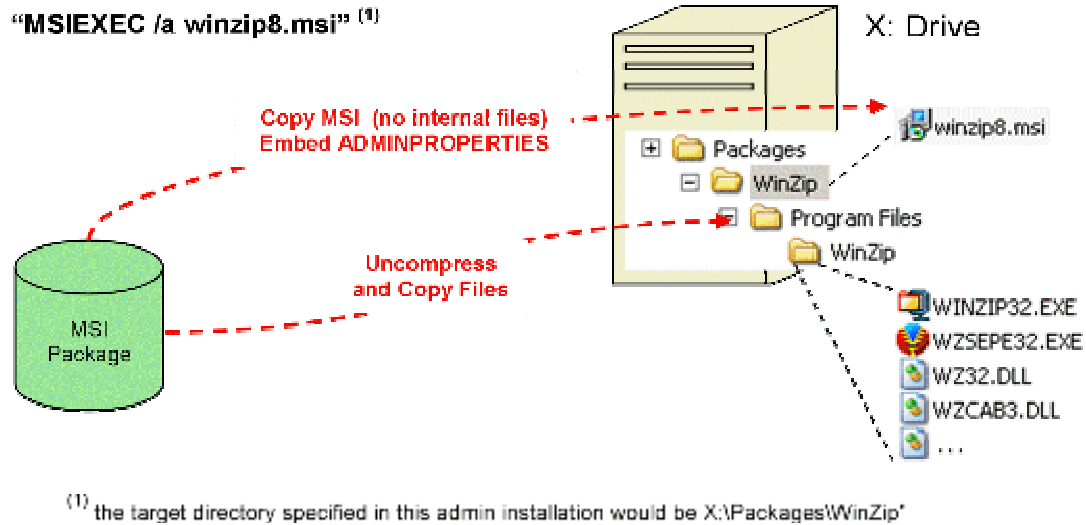


Figure 3.10: Creating an administrative install.

Installing from an Administrative Share

Client installations operate the same as installing from any other installation source. Clients install from the administrative share using the standard command line or by double-clicking the MSI file located on the administrative share, as Figure 3.11 illustrates.

"MSIEXEC /i x:\packages\winzip\winzip8.msi /qn"

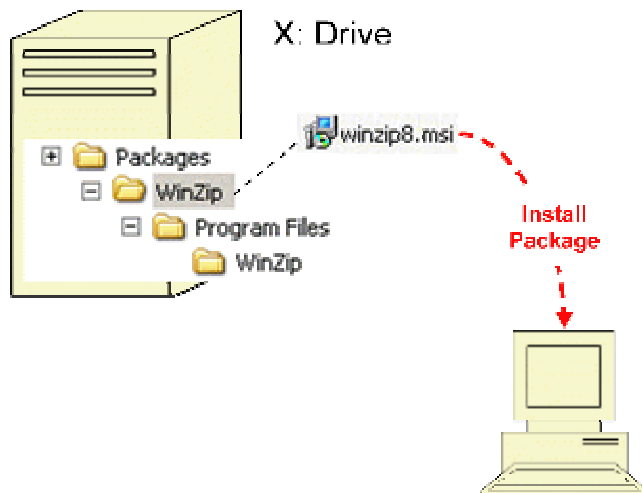


Figure 3.11: Client install from an administrative install.

Serving Applications

Over time, there have been many terms used for the concept of leaving the software application files on the server and having the client execute them from there. For our discussion, we will refer to this as *served applications*. One of the most notable uses for served applications is for implementations that require high availability. Having an application installed on multiple servers allows for fault tolerance when a server fails. Windows Installer supports fault-tolerant served applications.


 Some packagers attempt to build Windows Installer packages for legacy served applications without using Windows Installer's native support for served applications. The legacy approach is to point icons to *existing* binaries located on the network. However, Windows Installer does not allow a shortcut to point to files that are not contained in the current package. A popular workaround is to copy traditional .LNK shortcuts to clients. Although this approach works to a limited degree, the shortcuts will not trigger any Windows Installer activities such as self-healing and install-on-demand. The native support must be used to avoid extensive workarounds and enable the full Windows Installer feature set.

Figure 3.12 shows two key properties used to configure Windows Installer packages for fault-tolerant served applications. The ADDSOURCE property causes the Windows Installer shortcuts to look for the application files at the administrative install location. ADDSOURCE takes a list of features as its value, the special value ALL indicates that all features should remain on the server. The SOURCELIST parameter causes the package installation to include a list of additional locations at which the software application files can be found.

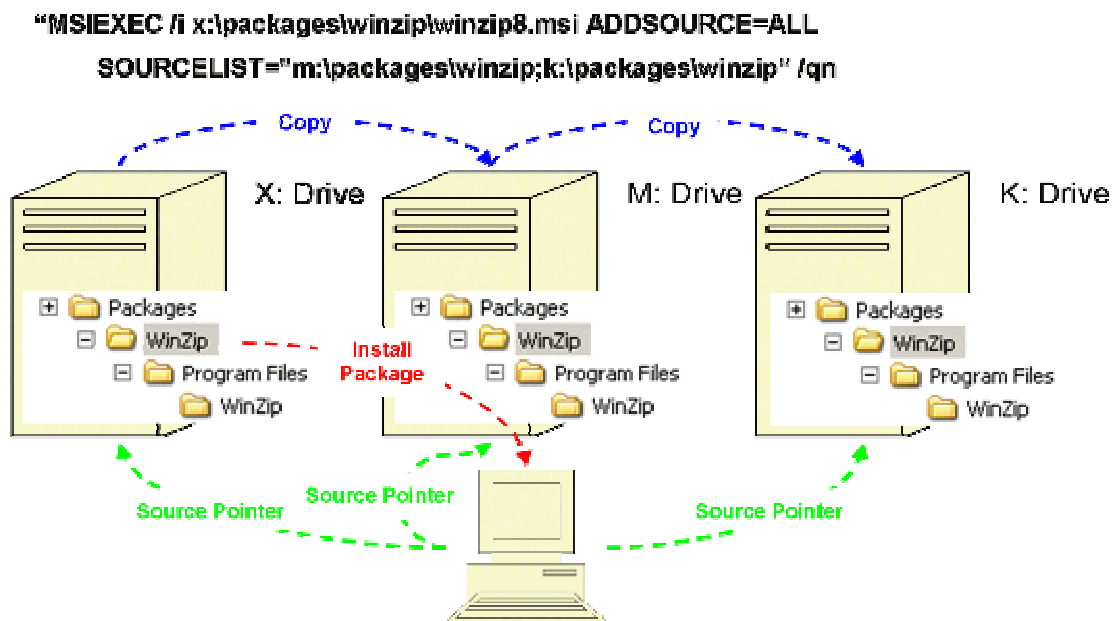



Figure 3.12: Served application configuration (fault tolerant).

Here is the basic process for setting up fault-tolerant served applications:

1. Create an administrative installation of the application.
2. Replicate the administrative installation to multiple locations or perform additional administrative installations with the EXACT same package (package codes should match).
3. Install clients using the ADDLOCAL and SOURCELIST properties.

Windows Installer does not perform load balancing between the various sources for the application. If this is desired, a load-balancing file system technology such as Win2K's Distributed File System (DFS) should be used. Manual load balancing can be accomplished by ensuring that an equal number of clients perform the initial install from each server location.


 Chapter 5 will contain a more in-depth look at Windows Installer source lists.

Security and Policies

Windows Installer security and policies is an area of great interest to administrators. Security and policies give some of the flexibility required to design an application deployment approach that is secure from viruses and end-user abuse. Proper attention to Windows Installer security and policies helps address the following significant risks:

- Viruses that take advantage of MSI security capabilities
- Security exploits by users, administrators, or hackers
- Unauthorized software installs on corporate machines
- Software piracy


This section will focus on key policies in Windows Installer and new Windows XP policies for controlling which applications can be installed. As a point of clarification, this section discusses how to configure Windows Installer service settings using policies, not how to deploy Windows Installer packages using Group Policy-based application deployment.

 If you work in a very large organization, it is important to consider that Help desk and first-level administrators might have the technical savvy and physical access needed to abuse elevated privileges. Protecting against these types of exploits requires a different perspective than just having to consider internal end users and external hackers.


Windows Installer Policies

As with most technologies introduced with Win2K, Windows Installer is configurable through policies. However, unlike many Win2K Group Policies, Windows Installer security policies are registry-based. In practical terms this means that AD and special policy processing agents are not required to manage these policies. Any mechanism currently used to mass deploy registry tweaks can be used to effectively configure MSI

policies. This includes initial computer build, distribution of .REG files, third-party policy management systems (such as those provided by NetWare), and Windows 9x and NT System Policies.


 An updated System Policy template (.ADM file) is available for download at WindowsInstallerTraining.com. This .ADM file includes two new MSI 2.0 policies as well as the debug policy. This updated policy file can be downloaded from <http://windowsinstallertraining.com/msiebook>.

The following discussion will focus on the essential Windows Installer policies that should be considered by administrators. These policies generally deal with securing Windows Installer's elevated privileges capabilities.

 For an exhaustive list of Windows Installer Policies, refer to the Windows Installer SDK document titled "System Policy" and all its sub-documents.

Elevated Privileges Implementation

There are some basic concepts of elevated privileges that should be understood before diving into the policies that configure them. Whenever an MSI package is installed, an instance of MSIEXEC.EXE is started in the user's context. This occurs, as Figure 3.13 shows, whether the package is started by double-clicking an MSI or if MSIEXEC.EXE is called via a batch file, logon script, or software distribution system.

 We will be discussing elevated privileges and software distribution systems in more detail in Chapter 5.

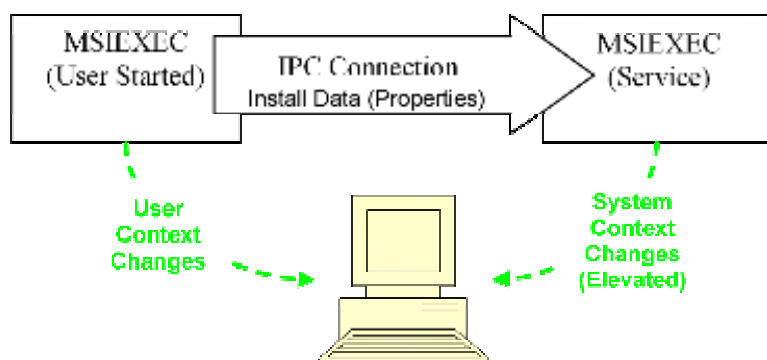



Figure 3.13: Elevated privileges implementation.

An elevated installation is one that uses administrative rights for a portion of the installation. If elevated privileges are requested and approved, an inter-process communication occurs between the instance of msiexec.exe that is started in the user context and the instance running as a Windows service. If elevated privileges are granted, the security rights of the system account are utilized for the activities performed by the service. Windows Installer enforces strict rules about the data that is allowed to cross the

IPC connection and what types of commands can be performed on the service instance of MSIEXEC.EXE. This approach is more secure than the user context switching approach provided by tools such as the NT Switch User utility or Windows XP's RunAs functionality.


 It might be tempting to change the account used by the Windows Installer service as a method of preventing abuse of the System Account. This is likely to create difficulties for your installations and should be unnecessary given the built-in and policy-based security controls in MSI.

Managed Applications

Windows Installer gives selected applications Managed Application status depending on how they are installed. Packages that come from any of the following sources are considered Managed:


- Assigned through Group Policy to users (Advertised) or computers (Full Install)
- Assigned using the MSIEXEC command line by an account that has local administrative privileges on the target computer (Advertised or Installed).
- Deployed through SMS 2003 (beta name was Topaz)

Managed Application status gives a software installation elevated privileges during the initial installation and for all subsequent installer operations such as self-heal, install-on-demand, maintenance installs (adding/removing features), and uninstalls. That is to say, packages that are tagged as Managed on a specific computer continue to have elevated privileges for subsequent installation activities on that computer. These elevated privileges continue to operate independent of the original reason that granted the package Managed status.

 Unlike traditional setup.exe installers, the Windows Installer engine is not only used during initial installation of a package. The Windows Installer engine is active in all phases of the application management lifecycle, including deployment, installation, configuration (adding/removing portions of a software application), self-healing, upgrades, and uninstalls.

Always Install with Elevated Privileges (AlwaysInstallElevated) Policy

The AlwaysInstallElevated policy is the most permissive configuration of elevated privileges and should be used sparingly. This policy must be set to 1 (Enabled) for the computer AND the user to be completely enabled. This policy allows all packages and installation activities to occur with elevated privileges regardless of their source or the user account that starts them. This policy is intended to permit all installation activities to complete normally for non-administrative users (as they would under Windows 9x) but do so without giving away local administrators rights that grant many more capabilities than application installation.

 Managed Application status is NOT given by using the AlwaysInstallElevated policy settings. If packages are installed with this policy turned on, and the policy is subsequently turned off, subsequent install activities are limited by user rights. This can hamper self-healing, application upgrades, and uninstalls.

AlwaysInstallElevated Hacking

Some organizations have used the two AlwaysInstallElevated keys as a method of programmatically controlling elevated privileges. Under this approach, security on these policy keys is configured to allow them to be changed by a wrapper script. The wrapper script will toggle the policy keys on, perform an MSI installation, then turn them off. Although this functionality is convenient, it has a couple downsides that should be taken into account. First, this approach might cause problems for self-healing or when the user attempts to reconfigure the application through Add/Remove Programs because the user will no longer have administrative rights to perform installation activities. Second, security exploits and viruses generally test for “security by ignorance” techniques such as these. There are probably valid scenarios in which using this method is acceptable—just make sure you are aware of the risks if you are considering it.


Disable Windows Installer (DisableMSI) Policy

The DisableMSI policy has three settings:

- 0 (Default) = Always Enabled
- 1 = For Non-Managed Packages
- 2 = Always Disabled

The value 0 means MSI is always enabled. The value 2 means that it is always disabled. There are very few circumstances in which completely disabling MSI is desirable. The value 1 restricts package installs to only be allowed from three sources: Group Policy, SMS 2003, or assignment by an administrator.

The *For Non-Managed Packages* value is usually of interest to organizations that want to restrict users from installing unauthorized software packages. This can be an effective approach for LAN-based environments, but it does create limiting situations for offline package deployment. If you have the luxury of deploying Windows XP you might want to consider software restriction policies (which will be discussed shortly).

 The Disable MSI policy overrides the more permissive AlwaysInstallElevated policy. If both are turned on, AlwaysInstallElevated is effectively disabled.

Cache Transforms in Secure Location on Workstation (TransformsSecure)

Whenever transforms are used for an installation, Windows Installer caches them on the local computer. This allows them to be applied to all subsequent installation activities. If a transform can be replaced by an end-user or IT personnel, their copy will be executed during any subsequent installation activities. If the application has Managed Application status, replacing cached transforms can allow malicious code to take advantage of local administrative rights.

For packages that are installed for users, transforms are cached in the user profile to support roaming profiles. When the TransformsSecure policy is used, it ensures that transforms are always cached in a secure location regardless of whether a user or computer installation is performed. For security sensitive implementations, this policy should be enabled.

Other Security-Oriented Policies

Most of the remaining security-oriented policies have their most restrictive setting by default. If an organization is deploying a new version of Windows, deploying software distribution or re-engineering application management, a full study of the security focused policies of Windows Installer will provide the background necessary to make wise design choices.

Non-Security Policies

There are several useful non-security policies in Windows Installer. The following section discusses these policies.

Excess Recovery Options


There are two policies that deal with how Windows Installer ensures that failed installation changes are backed out completely. Windows Installer has built-in support called rollback. This support is built-in to Windows Installer and works on all versions of Windows. Windows Installer also interfaces with system restore services on OS versions that have system restore (Windows XP and ME). When system restore is present, Windows Installer requests a restore point before performing installation activities.

There is one key difference between these two recovery technologies: The native rollback support is only used during an installation; if an installation completes normally, all roll back data is deleted. System restore allows the system to be arbitrarily returned to any restore point that is still in the system restore cache—this could be days after an installation.


For many organizations, having both of these options active just consumes extra disk space and extends package processing time. For production use, Windows Installer's built-in support should generally be left on. Windows Installer's usage of system restore could be turned off if desired. Each organization should test install and uninstall times with system restore both on and off and decide whether the impact is significant for typical software installation scenarios in their company.



Windows Installer's use of system restore is disabled using the `LimitSystemRestoreCheckpointing` computer policy. Setting it to 1 prevents Windows Installer from requesting a system restore checkpoint during installations.


 The `LimitSystemRestoreCheckpointing` policy only affects Windows Installer's usage of system restore. System restore will continue to be leveraged by the OS for all other non-Windows Installer activities.

Windows Installer rollback is disabled using the `DisableRollback` policy. It is configurable for both the computer or user—setting it to 1 in either location will cause rollback to be disabled.


 There is one situation in which you might want to disable both Windows Installer rollback and system restore for package installations. In some large scale deployments of Windows, an extra hour of workstation build time can be a critical cost and project management factor. In deployment scenarios in which computers are formatted and rebuilt, turning off these policies can reduce build time. Because a failed workstation build can be easily restarted, there are no risks to eliminating rollback capabilities.

Logging Policy

Windows Installer always logs information to the Windows event logs. In many cases, this information is sufficient for routine problem analysis. If more detailed data is required, the logging policy can provide it. To say that Windows Installer logging is exhaustive would be an understatement.

 The SDK document titled “Event Logging” lists all the messages that Windows Installer might record in the event log.

The logging policy is used to cause Windows Installer to create log files for all of its activities. Although the command-line logging options trigger logging for a specific package installation, the policy covers all installation activities, including self-healing, maintenance installs, and so on. There are 11 single-character switches that can be used to configure logging. Each of them logs specific types of information about the installation. When troubleshooting difficult packaging problems, it is a good idea to put the log in complete verbose mode so that no helpful information is missed.

 When configuring verbose logging, the 11 switches can be arranged to spell `voicewarmup`—this is an easy way to remember the switches, and they can be entered directly in this order.

When the logging policy is used to configure logging, no file location can be specified. All Windows Installer log file names have the following naming convention:

```
"MSI<randomcharacters>.LOG"
```

For user-initiated installs, the log is placed in the user's TEMP directory. For automated installs (such as GPO deployment), the log is written to the system TEMP directory.

 We will be discussing more details about logging in Chapter 4.

Software Restriction Policies

Software restriction policies are a new addition for Windows XP and .NET Server. Software restriction policies can enable or prevent execution of many types of files in Windows, including .MSIs and .MSTs. Because these policies are processed before Windows Installer is started, they are a very effective way of preventing unauthorized software installations. Software restriction policies are not a complete substitute for managing Windows Installer policies.

 Microsoft has a good white paper that summarizes software restriction policies at <http://www.microsoft.com/windowsxp/pro/techinfo/administration/restrictionpolicies/default.asp>.

Software restriction policies have four types of rules, discussed in the following sections. Each of these has different implementation considerations when used with Windows Installer.

Certificate Rules

Certificate rules allow restriction of software installations by requiring that MSI files and MST files are *code signed* with the specified certificate. If they are not signed, Windows will not allow them to be passed to Windows Installer for processing. Code signing is extremely powerful, but the following considerations should be taken into account when considering its usage:

- Administrative installs can change structure of the MSI file, so code signing must occur after the administrative install is made. In addition, the “master” administrative install needs to be replicated to preserve the code signing.
- Vendors might code sign their own installations. Removal of vendor code signing can cause problems if the vendor validates their own signing. The vendor’s certificate can be added to your software restriction policies if need be.
- Any changes to the package require that it be re-signed.
- Signing certificates are usually accessible by a very few people in the IT organization, which can inadvertently become a bottleneck to the packaging process if a large volume of packages and transforms are expected.

Hash Rules

Hash rules are very similar to certificate rules, except that hash rules do not alter the original file and they do not require a certificate to generate the cryptographic key used by the policies. Hashes can make it easier for administrators to restrict MSI execution without the elaboration of certificates and they may be just as effective at preventing users from installing unauthorized software. The MD5 hashes required for this type of restriction can be easily generated within the Group Policy interface. Hash rules would have the same limitations as certificate rules, except for the possible process bottlenecks. Hash rules would also leave vendor signed packages unchanged.

Path Rules

Path rules allow restriction of software installations by requiring that MSIs and MSTs run only from specific path locations. At first, this sounds limiting, however, path rules can be defined using wildcard characters, environment variables, and DFS share names, making this rule type very flexible. Here are some planning considerations if path rules sound like they will work for you:

- The repository strategy must be well defined to ensure that paths are consistent.
- A strategy for offline installs must be worked out to ensure that it fits with the use of path rules.
- Path rules that are too flexible may allow users or administrators to create a path that mimics the path rule and execute their own package from that location.

Zone Rules

Zone rules are only used for MSI files. They permit or restrict browser-based software installations from occurring based on the Internet zones in IE. The default zones include Internet, Intranet, Restricted Sites, Trusted Sites, and My Computer. These rules can be helpful for building a Web-based, self-service installation system.

Combining Rules

Multiple rules of all four types can be used in combination to create fine-grained control over software installations. Rules that are the most specific to the file being assessed take precedence over rules that are more general.



If you are using Windows XP with Win2K domain controllers, you must load the Windows Server Administration Tools from the Windows .NET Server CD-ROM onto a Windows XP workstation to configure software restriction policies in AD.

Summary

This chapter has laid the foundation for delving into the next level of Windows Installer technology. In addition to covering the basics of the internal structure of a package, we brought out some unique ways of building and utilizing transforms, administrative installs, and policies. Hopefully, the techniques you have learned will help you build more effective and secure packages.

In the next chapter, we will be discussing best practices for building packages. Get set to learn about repackaging, upgrades, and building processes!

