
AWS End-of-Support Migration Program (EMP) for Windows Server

User Guide



AWS End-of-Support Migration Program (EMP) for Windows Server: User Guide

Copyright © Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

Table of Contents

What Is AWS End-of-Support Migration Program (EMP) for Windows Server?	1
Features	1
Concepts	2
Supported operating systems and requirements	2
Accessing	2
Pricing	3
How it works	4
The EMP process	5
Discovery	6
Packaging preparation	6
Packaging	6
Deployment	6
System requirements	6
Limitations	7
Data collection	7
Compatibility Package Builder components	7
Get started	10
Decision tree	10
Planning a migration	11
High-level discovery	11
Install Compatibility Package Builder	13
Package an application	13
Standard packaging	15
Reverse packaging	17
Package an IIS-based application	26
Discovery	26
Migrate	27
Troubleshooting	29
Package contents	30
Source package contents	30
Deployed package contents	35
Deploy an application	35
Requirements	36
Run deployment tool	37
Working with EMP packages	38
Best practices	38
Compatibility package features	39
ForceExternalManifest	40
RegClassesMerging	41
DoNotHideDebugger	41
HandleInvalidHandle	42
NotWow64Process	42
NetworkRedirection	42
LocalMappedObjectShim	44
DEPOptOut	45
COMVirtualization	45
ForceWindowsVersion	46
RedirectX64PackagedRegistry	46
LoadSystemResources	47
Edit, upgrade, and maintain packages	47
Edit	47
Upgrade	48
Maintain	48
Optimize Process Monitor	49

Update a deployed package	50
Uninstall a package	51
Enable logging	52
Manage custom configurations	54
Link packages	56
ODBC drivers	57
Enable out-of-process COM	58
Add SXS assemblies	59
Exclude or detach a process	60
Run cmd.exe as a child process	61
Security	64
Data collected	65
Troubleshooting	66
Versions	68
Document History	69

What Is AWS End-of-Support Migration Program (EMP) for Windows Server?

The AWS End-of-Support Migration Program (EMP) for Windows Server provides the technology and guidance to migrate your applications running on Windows Server 2003, Windows Server 2008, and Windows Server 2008 R2 to the latest, supported versions of Windows Server running on Amazon Web Services (AWS). Using EMP technology, you can decouple critical applications from their underlying operating system so that they can be migrated to a supported version of Windows Server on AWS without code changes.

Topic contents

- [Features of AWS End-of-Support Migration Program \(EMP\) for Windows Server \(p. 1\)](#)
- [AWS End-of-Support Migration Program \(EMP\) for Windows Server concepts \(p. 2\)](#)
- [Supported operating systems and requirements \(p. 2\)](#)
- [Accessing AWS End-of-Support Migration Program \(EMP\) for Windows Server \(p. 2\)](#)
- [Pricing for EMP \(p. 3\)](#)

Features of AWS End-of-Support Migration Program (EMP) for Windows Server

AWS End-of-Support Migration Program (EMP) for Windows Server offers the following features to help you migrate your applications running on legacy Windows Server operating systems to supported versions.

Install capture

During packaging, a snapshot is taken of the current state of the operating system and a capture process starts recording all changes made during the application installation.

Runtime analysis

File and registry access, along with changes that occur when an application is launched from this section of the packing wizard, are recorded and made available for addition to the package being created. This feature allows EMP to include first-launch changes by the application to the system.

Legacy OS support

An EMP package is created on the operating system that is currently supported by the application being packaged. This compatibility packaging allows you to migrate an application from an operating system with which it is compatible to one with which it is incompatible, while preserving application functionality.

Cross-platform support

You can deploy a single package across multiple supported operating systems.

Pre- and post- deployment scripts

You can include custom scripts in the package to deploy MSI (Microsoft Installer) files or to run other tasks that must be performed as part of the package/application deployment process.

Fast editing with no package recompilation

The time it takes to package, test, and update applications and configurations is reduced with fast editing, which requires no package recompilation. EMP packages can be updated to include application updates or the latest EMP components.

AWS End-of-Support Migration Program (EMP) for Windows Server concepts

The following concept is central to your understanding and use of AWS End-of-Support Migration Program (EMP) for Windows Server.

Packaging

The process of capturing a legacy application using the EMP package builder on the source operating system.

Clean packaging server

A server instance of the Windows operating system version that is supported by the application to be packaged.

Compatibility package

When the EMP compatibility packaging process is complete, the output of the package builder is called an EMP compatibility package.

Supported operating systems and requirements

For supported operating systems and requirements to successfully migrate your application with AWS End-of-Support Migration Program (EMP) for Windows Server, see [AWS End-of-Support Migration Program \(EMP\) for Windows Server System requirements \(p. 6\)](#).

Accessing AWS End-of-Support Migration Program (EMP) for Windows Server

AWS End-of-Support Migration Program (EMP) for Windows Server offers standalone tools that you download and install on your developer workstation. You can download the EMP tools from the [End-of-Support Migration Program for Windows Server product page](#).

New releases of the AWS End-of-Support Migration Program (EMP) for Windows Server Compatibility Package Builder are provided in MSI format. To upgrade to a new version from a previous version, uninstall the previous version by using the **Add or Remove Programs** feature from legacy Windows operating systems, or from **Programs and Features** in the **Control Panel** for later operating systems. Then, reinstall the package with the latest MSI.

Pricing for EMP

AWS End-of-Support Migration Program (EMP) for Windows Server is available for use at no cost.

How AWS End-of-Support Migration Program (EMP) for Windows Server works

AWS End-of-Support Migration Program (EMP) for Windows Server technology identifies the dependencies that your application has on a legacy operating system, and creates a package that includes the resources necessary for the application to run on a new version of Windows Server.

The EMP Compatibility Package Builder creates packages for legacy Windows applications to run on any supported target operating system. It uses an install-and-capture snapshot process to create a package that includes all of the changes made during application installation. After the package has been created, application and runtime analyses detect additional configuration changes that occur when the application first runs. The EMP package can be manually deployed using a command passed to its deployment program. Deployment can be automated with scripts or managed by enterprise tools, such as AWS Systems Manager.

Application compatibility packages include everything an application requires to run on a modern operating system. This includes all of the application files, runtimes, components, deployment tools, and the embedded redirection and compatibility engine. The EMP Compatibility Package Builder creates a package on the existing operating system that runs the legacy application (for example, Windows Server 2003 or Windows Server 2008) so that it can determine the application dependencies for the out-of-support operating system and set the required redirections. The redirection and compatibility engine is designed to run in user mode. When the package is deployed to the target machine, the file type associations are registered and shortcuts are created to run the application.

Compatibility packaging follows these three principles:

- Packaging is performed on the operating system supported by the application.
- Runtime analysis is performed to detect and review first-run activity of the application after installation.
- The redirection and compatibility engine, along with the package deployment program, are included in the package.

The package does not include the legacy operating system, which means you never run any part of the legacy Windows Server version on the new Windows Server to which the application is upgraded. The redirection engine intercepts the API calls that the application makes to the underlying Windows Server operating system, and redirects them to the files and registry within the created package. As a result, the request by the application for resources that are present within the package are redirected to the package so that the application runs successfully on the new operating system. This lightweight strategy means that the target application can see a combination of the redirected and local file system and registry with minimal impact on the performance of the application or local machine. The approximate RAM overhead per application is 3 to 5 MB, with no measurable CPU impact after the application has started.

Common redirections (not including file and registry redirections) supported by the EMP compatibility engine include:

- **The application uses a fixed port.** The EMP engine redirects to an appropriate and available port on the new version of the operating system.

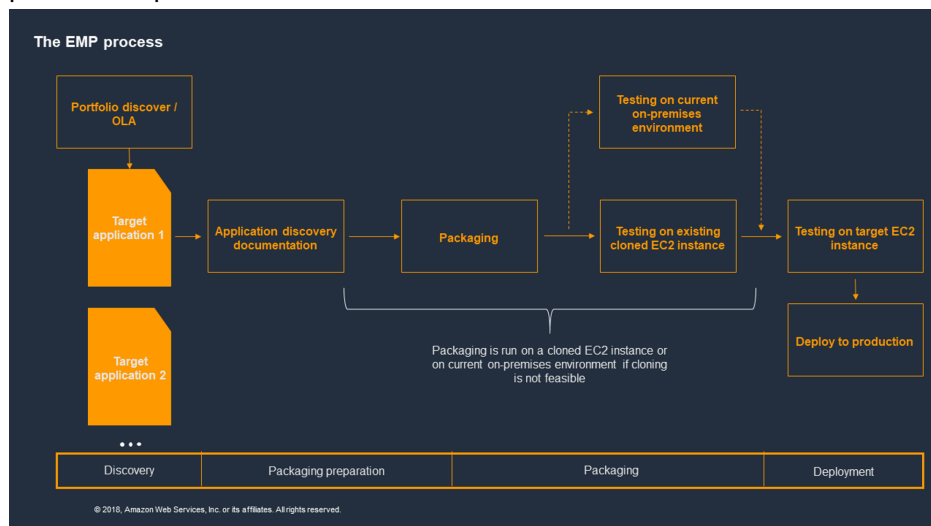
- **The application accesses data stored in a fixed location that is not available on the new OS version.** The EMP engine redirects these requests to the appropriate location on the new version of the operating system.
- **The application is hardcoded to c:\windows.** The EMP engine redirects the application to c:\Windows.
- **The application requires Java version 1.3.** EMP packages and isolates the application with Java 1.3 so that it does not make calls to newer Java runtimes in the new version of the operating system.

Topics

- [The AWS End-of-Support Migration Program \(EMP\) for Windows Server process \(p. 5\)](#)
- [AWS End-of-Support Migration Program \(EMP\) for Windows Server System requirements \(p. 6\)](#)
- [AWS End-of-Support Migration Program \(EMP\) for Windows Server limitations \(p. 7\)](#)
- [Data collection \(p. 7\)](#)
- [Components of the EMP Compatibility Package Builder \(p. 7\)](#)

The AWS End-of-Support Migration Program (EMP) for Windows Server process

This section provides a high-level overview of the EMP process, including the steps involved in each phase of the process.



The EMP process:

- [Discovery \(p. 6\)](#)
- [Packaging preparation \(p. 6\)](#)
- [Packaging \(p. 6\)](#)
- [Deployment \(p. 6\)](#)

Discovery

During discovery, target applications are recognized using AWS Optimization and Licensing Assessment (AWS OLA). Discovery includes identifying legacy applications that require EMP packaging for migration to later versions of the Windows Server operating system.

Packaging preparation

Packaging preparation includes the following steps.

1. Clone the server image to the AWS instance, if possible.
2. Complete application discovery.
3. Plan user acceptance testing (UAT).
4. Define the target environment.

Packaging

Packaging includes the following steps.

1. EMP package builder packages the application on the source server version, guided by application discovery. The source server version is the Windows Server operating system that currently hosts the application.
2. The package is tested on a clean source server version and any required remediation is performed.
3. The package is tested on the target server version instance following the UAT plan. Any necessary remediation is performed.

Deployment

Deployment includes the following steps.

1. Deployment and testing on the target environment.
2. Production deployment.

AWS End-of-Support Migration Program (EMP) for Windows Server System requirements

AWS End-of-Support Migration Program (EMP) for Windows Server supports the following operating systems:

- Windows Server 2016 (64-bit)
- Windows Server 2012 R2 (64-bit)
- Windows Server 2008 R2 (64-bit)
- Windows Server 2008 (32-bit and 64-bit)
- Windows Server 2003 SP2 (32-bit)

The following system requirements must be met to migrate your application with AWS End-of-Support Migration Program (EMP) for Windows Server.

- **.NET.** Microsoft .NET 4.0 Client Profile, or later.
- **Memory.** As required by the packaged applications. Minimum 2 GB.
- **Processor.** As required by the packaged applications. Two CPUs recommended.
- **Disk space.** 10 GB required to create the snapshots. The size of the EMP package depends on the size of the application. It will be 10 to 50 percent larger than the application being packaged.

AWS End-of-Support Migration Program (EMP) for Windows Server limitations

The following application and component types cannot be migrated using AWS End-of-Support Migration Program (EMP) for Windows Server.

- Applications that do not run on the Windows operating system.
- 16-bit applications. If the target operating system is a 64-bit Windows operating system, the NT Virtual DOS Machine (NTVDM) required to run these applications is available on 32-bit Windows operating systems only.
- Kernel-mode drivers that are a different bitness than the target operating system. Device drivers are not virtualized with EMP and therefore must be compatible with the target operating system. Compatible drivers can be deployed with the package. For example, if you are moving to a 64-bit operating system, you must have a 64-bit driver that is compatible with the new operating system.
- Low-level applications. For example, antivirus, firewall, and VPN applications.
- Explorer Shell Extensions.
- Microsoft BizTalk and Microsoft Transaction Server (MTS)-based systems.
- Desktop applications.

Data collection

AWS collects usage information through the EMP telemetry module during the deployment and subsequent use of EMP packages. The telemetry module sends the collected data to an application modernization metrics service running on AWS. To view the data collected by the telemetry module, see [Data collected by the AWS End-of-Support Migration Program \(EMP\) for Windows Server](#) (p. 65) in the Security section of this guide.

Components of the EMP Compatibility Package Builder

The installation directory of the Compatibility Package Builder includes the following files and folders.

Files

- **Compatibility.Package.Builder.exe** — the Package Builder program.
- **Compatibility.Package.Builder.cfg** — Used to configure packager settings, such as file scan root directory.
- **Compatibility.Package.Builder.exe.config** — Contains packager program settings, such as packager event logging level and dependencies, such as the .NET runtime version.
- **Compatibility.Package.Builder.log** — Logs package builder events during packaging.

- **Compatibility.Package.CmdLineBuilder.exe** — The CLI version of the Package Builder. Uses the `PackageScript.xml` as its response file.
- **Compatibility.Package.CmdLineBuilder.exe.config** — Includes the CLI program settings, such as CLI event logging level and dependencies, such as the .NET runtime version.
- **EULA.rtf** and **eula.base** — The end-user-license-agreement files. the content of the `.rtf` file is copied into the `.base` file, which is saved as HTML in the package root folder.
- **ExclusionList.json** — A configuration file that contains the list of files, folders, and registry keys that are ignored during scanning.
- **Open Source Licenses.txt** — Contains the license for the open-source components in the Package Builder.

Folders

- **Images** — Contains the graphic files used by the Package Builder application.
- **x64 (Engine Binaries)** — Contains the EMP runtime files for packages to be deployed on a 64-bit system. When packaging is performed on a 64-bit machine, the contents of this folder are automatically copied into the root folder of the EMP package during the package build. An EMP package that was built on a 64-bit machine cannot be run on a 32-bit machine.
- **x86 (Engine Binaries)** — Contains the EMP runtime files for packages to be deployed on a 32-bit system. When packaging is performed on a 32-bit machine, the contents of this folder are automatically copied into the root folder of the EMP package during the package build. An EMP package that was built on a 32-bit machine will run on a 64-bit machine, however, we recommend that you use the appropriate runtimes for the destination platform. This is automatically handled during the deployment process.
- **Tools** — Contains three sets of tools that are available to EMP packages.
 - **DiscoveryTool** — A command line tool that can perform a limited discovery of a server.

Commands

- `Compatibility.Package.DiscoveryTool.exe -d "<InstallDirectory>` — Writes a list of loaded COM servers and drivers in `<Currentdirectory>\Report.json`.
- `Compatibility.Package.DiscoveryTool.exe -l` — Lists features and subfeatures of the tool in the command console.
- **Editor** — Used to edit existing packages. This tool has a shortcut in the **Start** menu, where it goes by **Compatibility Package Editor**.
- **ReversePackagingTools** — A command line tool set used in the reverse packaging process, a method of compatibility packaging used when application installation media is not available. This tool set is used during the first two stages of reverse packaging:
 1. **Install media reverse engineering** — A reverse engineered installation media, called a package source is generated from a working instance of the application on the server.
 2. **EMP package build** — The package source installation is then captured on a clean packaging server using the Compatibility Package Builder.

Commands

- `type <procmoncapture.CSV> | GeneratePackageSource.exe | RemoveKnownFolders.exe > <outputfilename.json>` — Generates a manifest for the package source from the CSV file using `RemoveKnownFolders.exe` to remove common known system registry keys, files, and folders.
- `type <filteredoutputfilename.json> | ExportFromSystem.exe > <logfilefilename.txt>` — Extracts the listed files, directories, and registry keys from the operating system and compresses them along with the remaining contents of the `ReversePackagingTools` folder into `PackageSource.zip`. This file is a reverse-engineered installation media for the application.

- `type <filteredoutputfilename.json> | DeployToSystem.exe >`
`<logfilefilename.txt>` — Installs Package Source during the package build on the packaging server using the package builder.
- **IISTools** — A set of command line scripts used to enable the automatic migration of legacy Windows IIS applications to a modern, supported version of Windows Server on AWS. For more information, see [Package an IIS-based application \(p. 26\)](#).

Get started with AWS End-of-Support Migration Program (EMP) for Windows Server

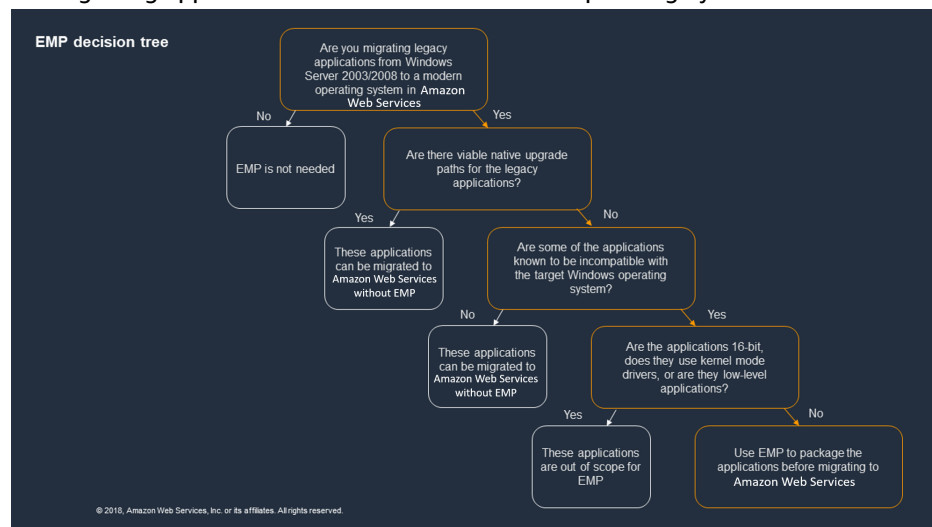
This section helps you get started with creating an EMP package to help you migrate your legacy application to a modern Windows Server operating system. It includes a decision tree to help you determine whether the AWS End-of-Support Migration Program (EMP) for Windows Server is the right solution for your migration, considerations for planning your migration, high-level discovery steps, Compatibility Package Builder installation steps, procedures for packaging an application with or without installation media, and steps for deploying a packaged application.

Getting started topics

- [AWS End-of-Support Migration Program \(EMP\) for Windows Server decision tree \(p. 10\)](#)
- [Planning an AWS End-of-Support Migration Program \(EMP\) for Windows Server migration \(p. 11\)](#)
- [High-level AWS End-of-Support Migration Program \(EMP\) for Windows Server application discovery exercise \(p. 11\)](#)
- [Install AWS EMP Compatibility Package Builder \(p. 13\)](#)
- [AWS End-of-Support Migration Program \(EMP\) for Windows Server application packaging model \(p. 13\)](#)
- [Package an IIS-based application \(p. 26\)](#)
- [EMP compatibility package contents \(p. 30\)](#)
- [Deploy an EMP package \(p. 35\)](#)

AWS End-of-Support Migration Program (EMP) for Windows Server decision tree

The EMP decision tree is a set of questions to assist you in determining whether EMP is the right solution for migrating applications onto modern Windows Operating Systems in AWS.



Planning an AWS End-of-Support Migration Program (EMP) for Windows Server migration

When you plan an AWS End-of-Support Migration Program (EMP) for Windows Server migration, consider the following:

- **Application environment — legacy to target**

It is important to understand the existing architecture of the application that is being migrated so that the target environment can be correctly set up. In an n-tier application model, you should identify each server that makes up the model and understand which of the servers require migration and which of them require EMP for the migration process. Some of the servers may already be migrated onto a modern platform in AWS and using a step-by-step migration approach ensures a smooth transition to AWS.

For example, if you attempt to migrate a three-tier application that consists of an Internet Information Server (IIS) web server, application server, and SQL Server 2008, all running on Windows Server 2008 on premises, the migration plan might look like this: the IIS Web Server migrates to a Windows Server 2019 in AWS without the need for EMP migration. The application running on the application server is captured into an EMP package and migrated onto a Windows Server 2019 running on AWS, and SQL Server is migrated to Amazon Relational Database Service (Amazon RDS). If the application works only on the legacy version of SQL Server, then it can be captured into an EMP package and migrated to a modern operating system.

- **Standard vs. reverse packaging**

The [application discovery \(p. 11\)](#) exercise determines whether the [standard \(p. 15\)](#) or [reverse \(p. 17\)](#) packaging model should be applied.

Note

If the criteria for standard packaging is met, we recommend that you follow this methodology rather than the reverse packaging methodology.

- **Packaging and testing environment**

Set up your packaging and testing environments according to the results of the application discovery exercise. For more information about each packaging scenario, see [AWS End-of-Support Migration Program \(EMP\) for Windows Server application packaging model \(p. 13\)](#).

High-level AWS End-of-Support Migration Program (EMP) for Windows Server application discovery exercise

After you identify the applications to migrate using EMP, we recommend that you perform an application discovery exercise for each application. EMP application discovery refers to the process of gathering and documenting all of the information about a legacy application that is required to inform the creation of a functioning EMP package and its deployment onto a modern, supported target environment. The information required to complete application discovery comes from both the legacy and target environments, and typically includes configuration details, installation instructions, topology, customizations, security requirements, users, and more.

When the application discovery is complete, analyze the information against the [EMP limitations \(p. 7\)](#) to determine EMP eligibility.

We recommend that an application subject matter expert (SME) is available to assist with the discovery process. The application SME is a representative who knows how the application works for the business and is familiar with the business workflows within the application. The SME can demonstrate all business workflows and also define or agree upon an acceptance testing plan for the application if one does not exist. The acceptance testing plan can be used during the testing of the application in the EMP package.

The following table shows a discovery checklist to guide you through the discovery process.

Checklist item	Details
Application name and version	The name and version of the application that you want to migrate.
Application subject matter expert (SME)	The name and contact details of the application SME to assist with the migration.
Software prerequisites required by the application	The software required to be installed before or with the application. For example, .NET 2.0 SP1, earlier versions of Java, Visual C++ runtimes, and more. The dependencies can be included in the same package as the application.
Source operating system	The operating system on which the application currently runs.
Target operating system	The operating system to which you want to migrate the application.
Are the install media and install instructions available?	The answer to this question determines the EMP packaging model to be follow for migration (standard (p. 15) or reverse (p. 17) packaging). For standard packaging, gather install media, instructions, and customizations required to carry out a complete setup of the application on the legacy OS.
Application topology and external dependencies	For example, three-tier topology with an application/desktop tier, an IIS-based web tier, and a database tier all hosted on different servers requiring connectivity between one another.
Does the application require specific drivers?	If so, is a 64-bit compatible version available? (Check the EMP limitations (p. 7)).
Other application details	What is the bit rate of the application: 32- or 64-bit? Does it use COM+ or DCOM components? Are Windows Services installed or used by the application? Do any services require domain or local service accounts? Is the application subject to Data Execution Prevention (DEP)? Are there any firewall ports that should be opened?
Windows features and roles required	The Windows features and roles required by the application to set up on the modern operating system.
Known issues	Any known issues of the application to account for during the testing phases.

Checklist item	Details
Cross-check against EMP limitations	Verify whether the application is in scope for EMP packaging.

Install AWS EMP Compatibility Package Builder

The AWS EMP Compatibility Package Builder is the primary tool used to create EMP compatibility packages to deploy functioning legacy Windows applications on modern Windows Server operating systems on which they would otherwise be incompatible. Package Builder is provided in an MSI installer file that installs the tool on a clean packaging server. A *clean packaging server* is a server instance of the Windows operating system version that is supported by the application to be packaged.

Perform the following steps to install the AWS EMP Compatibility Package Builder.

1. If you have not done so already, download AWS EMP Compatibility Package Builder from the [AWS End-of-Support Migration Program \(EMP\) for Windows Server product page](#).

New releases of the AWS End-of-Support Migration Program (EMP) for Windows Server Compatibility Package Builder are provided in MSI format. To upgrade to a new version from a previous version, uninstall the previous version by using the **Add or Remove Programs** feature from legacy Windows operating systems, or from **Programs and Features** in the **Control Panel** for later operating systems. Then, reinstall the package with the latest MSI.

2. After you have downloaded the EMP tools, double-click the Compatibility Package Builder file to run it.
3. On the **Welcome to the Compatibility Package Builder Setup Wizard** pop-up, choose **Next**.
4. In the **End-User License Agreement**, select the terms agreement, and choose **Next**.
5. Under **EMP Telemetry**, select the check box to enable telemetry (optional), and choose **Next**.
6. Accept the default **Destination Folder**, or modify it, and choose **Next**.
7. Choose **Install**.
8. When the application installation completes, choose **Finish**.

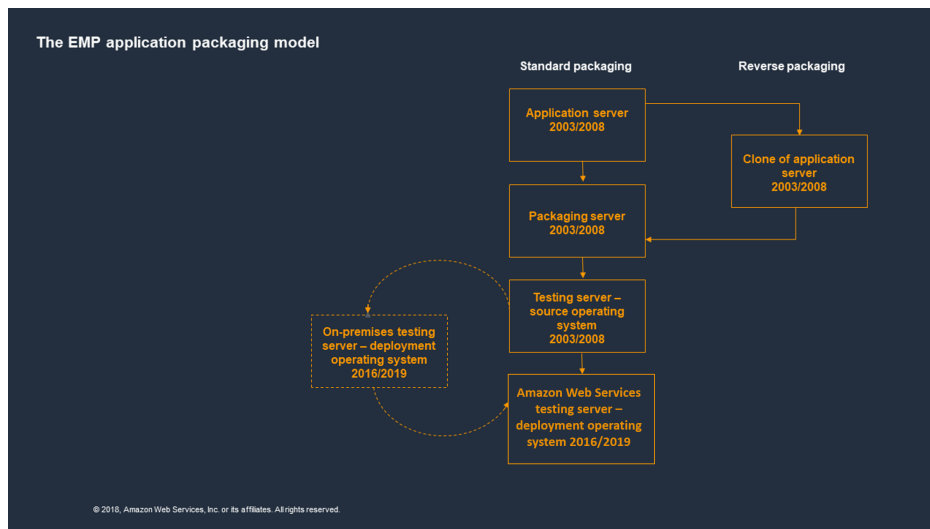
AWS End-of-Support Migration Program (EMP) for Windows Server application packaging model

The following model and descriptions show the process of packaging a legacy application using AWS End-of-Support Migration Program (EMP) for Windows Server. There are two packaging scenarios:

- **Standard packaging.** Use when the application media and install instructions are available to recreate an up-to-date installation of a legacy application in its current supported operating system.
- **Reverse packaging.** Use when the application media or the install instructions are not available to recreate an up-to-date installation of a legacy application in its current supported operating system.

Note

If the criteria for standard packaging is met, then we recommend that you apply this packaging method instead of the reverse packaging method.



Application server

The legacy application server on which the legacy application is installed and runs.

Clone of the application server

The clone of the application server that is used for the process capture phase of the reverse packaging process. For information about the reverse packaging process, see [How to package an application when installation media is not available \(reverse packaging\)](#) (p. 17).

If you do not want to clone the production servers, then you can skip this step and follow the initial steps of the reverse packaging process on the production application server. We recommend that you follow standard best practices for working on a production system.

Packaging server

The server is an original build, including service packs, of the server operating system where the application is installed and runs. The EMP product is installed and used to create the EMP application package on this server.

Testing Server—source operating system

This testing server is an original build, including service packs, of the server operating system where the application is installed and runs. It is used to validate the EMP package on the operating system on which it was created.

AWS Testing Server—deployment operating system

This testing server is the target operating system on which the EMP package must run. This server is used to validate the application in an EMP package on the target operating system within the AWS environment.

On-premises testing server—deployment operating system

This server is the target operating system on which the EMP packages must run. This server is used to validate the application in an EMP package in the target, on-premises operating system.

Note

This step is required only if there is a benefit to test the package in an on-premises environment to validate that it works as expected before migrating the application to AWS, or if it is required to troubleshoot any issues. It helps identify whether issues are the result of the AWS environment setup, the EMP package, or the target operating system.

If you want someone at AWS or one of our partners to perform the application packaging for you, then submit a request to [AWS IQ](#).

Packaging instructions:

- [How to package an application when installation media is available \(standard packaging\) \(p. 15\)](#)
- [How to package an application when installation media is not available \(reverse packaging\) \(p. 17\)](#)

How to package an application when installation media is available (standard packaging)

When installation media is available, an application is packaged with the EMP Compatibility Package Builder. The package builder uses installation snapshot-based packaging along with runtime analysis to create compatibility packages for applications.

Perform the steps in the following stages to create a compatibility package using the package builder when installation media is available:

- [Stage 1: Install capture \(Required\) \(p. 15\)](#)
- [Stage 2: Runtime analysis \(Optional\) \(p. 15\)](#)
- [Stage 3: Edit package contents \(Optional\) \(p. 16\)](#)
- [Stage 4: Package Finalization \(Required\) \(p. 16\)](#)

Stage 1: Install capture (Required)

1. After you install the tools from the [End-of-Support Migration Program for Windows Server](#) product page, launch the package builder from your desktop.
2. On the **Select Package Folder** page, choose **Select Folder**. Select a package folder to specify where the package will be created, then choose **OK**.
3. Choose **Next**.
4. On the **Start Capture** page, choose **Start Capture** to capture the state of the system before the application is installed.
5. Choose **Next** when the capture is complete.
6. Install the application, components, and any required dependencies.
7. When all of the installations have completed, reboot **ONLY** if required by the application installer. After the reboot completes, you can restart the package builder from your desktop.
8. Return to the package builder, and on the **Install Application** page, choose **Next**.
9. If required, proceed to [Stage 2: Runtime analysis \(Optional\) \(p. 15\)](#); otherwise, choose **Next** to proceed to [Stage 3: Edit package contents \(Optional\) \(p. 16\)](#) .

Note

When it is launched from the desktop shortcut only, runtime analysis reviews file and registry operations (for example, read, modification, and creation) performed by processes. This is not a required step, but can be used to review and include changes to the registry and file system during the first run of the application. It can also be helpful to review what the application processes do when running under the EMP engine.

Stage 2: Runtime analysis (Optional)

The package builder detects all files, registry keys, and shortcuts created or modified when the application runs for the first time. Shortcuts are displayed on the **Run Installed Applications** screen. If no shortcuts are displayed, proceed to [Stage 3: Edit package contents \(Optional\) \(p. 16\)](#).

Important

Do not use any desktop or **Start** menu shortcuts. Start the application using only the shortcuts displayed in the package builder. If you use shortcuts other than those displayed in the package builder, configuration changes will not be captured by the package builder.

The following steps enable application files and registry entries that are created when the application is configured for the first time to be captured in the package.

1. Start the application by using the shortcut in the package builder. Do not use shortcuts on the desktop or **Start** menu.
2. Choose each shortcut to load the shortcuts.
3. Perform any required configuration changes to ensure that the application is configured for users the first time it starts. If the packaged application fails during a particular workflow, or an issue is identified during user acceptance testing (UAT), try to repeat the workflow during the packaging process.
4. Close the programs, and choose **Next** to continue.
5. Choose **Complete Capture** to record the final state of the system.
6. After the capture successfully completes, the following message displays `Capture completed. Please click "next" to continue.` Choose **Next**.

Important

Do not uninstall the application during runtime analysis or the package builder will capture this change and create the package without the application.

Stage 3: Edit package contents (Optional)

Files, registry keys, and redirects can be added to, removed from, or modified in the package depending on what was captured during the install capture and runtime analysis.

The following steps show how to modify the contents of the package.

1. On the **Captured Files** page, you can use the left-hand pane **Files in package** to view or remove files, or to add redirections for files captured in the package by the install capture process.
2. Navigate to the file or folder, open the context (right-click) menu, then choose **Redirect** or **Remove Item**, as required. If you choose a folder and want to redirect all subfolders, then choose **Redirect Children**.

If you redirect or remove an item, the available options on the context menu changes to **Remove Redirect** or **Add Item**, which allows you to reverse your changes.

3. To include the files detected by runtime analysis, use the right-hand pane **Files requested at runtime**
4. Navigate to the file, open the context (right-click) menu, and choose **Include in package**.
5. Choose **Next**, which displays the **Captured Registry Keys** page. From this page, you can view and manage registry keys in the same way as files.
6. Choose **Next** when you have made the required changes for registry keys.

Stage 4: Package Finalization (Required)

1. On the **Package** page, in the **App Name** box, enter a unique name for the application, which automatically populates the **App ID** box.
2. Optionally, enter a description for the application in the **Description** box.
3. From the **Run** drop-down list, optionally select the executable that is used to load the application.

4. Choose **Package App** to create the package.
5. Choose **Next** when the Press "Next" to continue message is displayed.
6. To view the contents of the package, choose **Open Package**.
7. To make changes, choose **Edit this Package** to modify the contents of the package or to change the name, description, and initial executable in the package.
8. When you are finished, close the package builder by choosing the **X** in the top right-hand corner.

How to package an application when installation media is not available (reverse packaging)

The following steps must be performed to create a compatibility package using the package builder when installation media is not available:

- [Prerequisites for reverse packaging \(p. 17\)](#)
- [Use Process Monitor to discover the files and registry keys used by the application \(p. 18\)](#)
- [Filter the capture \(p. 19\)](#)
- [Choose which processes to save in the capture \(p. 19\)](#)
- [Capture the files and registries \(p. 19\)](#)
- [Create the export bundle of the files and registry keys required for the application \(p. 21\)](#)
- [Create the new packaged version of the application \(p. 21\)](#)
- [Package applications on multiple drives \(p. 22\)](#)
- [Integrate COM+ applications into EMP packages \(p. 23\)](#)

Prerequisites for reverse packaging

The following prerequisites must be met to successfully package an application when the installation media is not available.

Requirements

To successfully migrate server applications when the installation media is unavailable, you must have access to a virtual server or instance with a working version of the application.

Software prerequisites

The following software is required to package an application without the installation media.

- **Windows Sysinternals Process Monitor (Procmon.exe)**. Used to trace files, processes, and registry keys of a running application. Copy the `procmon.exe` tool to a folder on the instance. In the packaging procedure described in this topic, we copy it to the `ReversePackagingTools` folder of the EMP Compatibility Package Builder installation .
- **The EMP package builder**. To install the EMP Compatibility Package Builder, see [Install AWS EMP Compatibility Package Builder \(p. 13\)](#). You can find the reverse packaging tools in the **Tools** folder of the installation.
 - On a 32-bit instance, the default installation folder path is `C:\Program Files\AWS\EMP`.
 - On a 64-bit instance, the default installation folder path is `C:\Program Files (x86)\AWS\EMP`.
- A **text editor** with syntax highlighting that supports JSON and XML. [Notepad++](#) is one free option.

The next phase of this process is to use Process Monitor to discover the files and registry keys that are used by the application on the source machine.

To optimize the setup of Process Monitor for reverse packaging, see [Optimize Process Monitor for Reverse Packaging \(p. 49\)](#).

Use Process Monitor to discover the files and registry keys used by the application

The next phase of the process is to use Process Monitor to discover the files and registry keys that are used by the application on the source machine.

To set up Process Monitor for application discovery, see [Optimize Process Monitor for Reverse Packaging \(p. 49\)](#).

Reverse packaging steps

- [Prepare to discover the application \(p. 18\)](#)
- [Discover the files and registry keys used by the application \(p. 18\)](#)

Prepare to discover the application

To prepare for discovery, complete the following steps.

1. Verify that no applications or processes other than those required by Windows are running. This improves the quality of the capture.
2. Navigate to the `ReversePackagingTools` folder.
3. Open (double-click) `procmon.exe` to launch Process Monitor.
4. On the **Process Monitor License Agreement** pop-up, choose **Agree** to load Process Monitor.
5. To ensure that Process Monitor captures everything that happens on the machine, you must enable advanced output by selecting **Enable Advanced Output** from the Filter dropdown menu.

Discover the files and registry keys used by the application

After Process Monitor is installed and configured, complete the following steps to discover the application:

1. Load the application that you want to discover.
2. Use the application to perform typical operations.

To get a high-quality discovery of the application files and registry keys, use the common workflows performed by your users and any workflows performed at the end of the month or quarter, such as reporting. If you load and close the application only, you are more likely to generate a package that fails during user acceptance testing. We recommend that you use the application as it is typically used in your production environment, and for a longer period of time. This prevents the necessity of repeating the discovery process because of missed files and registry keys.

If you are not sure which workflows are relevant to the application, ask an end user of the application to perform their workflows on the application.

3. When you have finished using all of the features that you want to discover, close the application.
4. Return to the Process Monitor application.
5. Choose the **Capture** icon (image of a magnifying glass) on the Process Monitor toolbar. The **Process Monitor** dialog box will display, stating `Disconnecting from Event Tracking for Windows (ETW)`. This can take up to a minute. When the disconnection completes, the **Capture** icon will appear as crossed out by a red line.

Filter the capture

After you create the capture, you must filter the output and determine which items to safely discard.

To filter the capture, perform the following steps:

1. Choose the **Show Process and Thread Activity** icon on the Process Monitor toolbar to clear the selection. This removes these operations from the Process Monitor trace because they are not required for this process.
2. Choose the **Show Network Activity** icon on the Process Monitor toolbar to clear the selection. This removes these operations from the Process Monitor trace because they are not required for this process.
3. Choose the **Save** icon on the toolbar.
4. On the **Save to File** pop-up, verify that both **Events displayed using current filter** and **Also include profiling events** are selected, then choose **OK**.

Note

Later in this procedure, we will save only the processes that we believe should be included in the capture as the basis for capturing and creating the packaged version of the application. The reason we save the capture at this point in the overall process is to have the option to reload this version of the capture if we later realized we have omitted something from the final capture. This prevents the necessity of repeating the capture from scratch if we omitted something in error.

5. Display the process tree by entering **CTRL+T** or by selecting **Process Tree** from the **Tools** menu. The process tree displays, showing all of the processes captured by the Process Monitor.

Choose which processes to save in the capture

After saving the capture, you must choose which processes to save. Choose only the processes required by the application to run successfully.

To save processes in the capture, complete the following steps.

1. Scroll through the Process Tree until you locate the executable used to load the application.
2. Choose the executable.
3. Choose **Include Process** to save this process in the final capture.
4. Review the Process Tree for additional processes to include in the final capture. Choose **Include Process** or **Include Subtree** as required.
5. When you are finished reviewing the Process Tree, choose **Close**.
6. On the toolbar, choose the **Save** icon.
7. On the **Save to File** pop-up, verify that both **Events displayed using current filter** and **Also include profiling events** are selected.
8. Under the **Format** section, select **Comma-Separated Values (CSV)**, and then choose **OK**.
9. Open Windows Explorer and verify that the log file has been created in the .CSV format, and that the file size is greater than 0.
10. Close the Process Monitor.

Capture the files and registries

After creating the log file, you must run the EMP reverse packaging tools to process the .CSV capture file created by Process Monitor. The following two tools are used in this process:

- **GeneratePackageSource.** This command compiles a list of the files and registry keys accessed by the application from the Process Monitor capture. These are exported to a .JSON file one line at a time.
- **RemoveKnownFolders.** This command removes known files, folders, and registry keys that are not required from the Process Monitor capture. Examples include background Windows services and components that are already present on the target server.

To process the capture file with EMP tools, complete the following steps.

1. Open a command prompt as an administrator.
2. Navigate to the folder that contains the Process Monitor capture files, for example, C:\Program Files\AWS\EMP\Tools\ReversePackagingTools.
3. Enter the following command, where <procmoncapture.csv> is the name of the .CSV file that you created and <outputfilename.json> is the name of output file that you want to create:

```
type <procmoncapture.csv> | GeneratePackageSource.exe | RemoveKnownFolders.exe  
> <outputfilename.json>
```

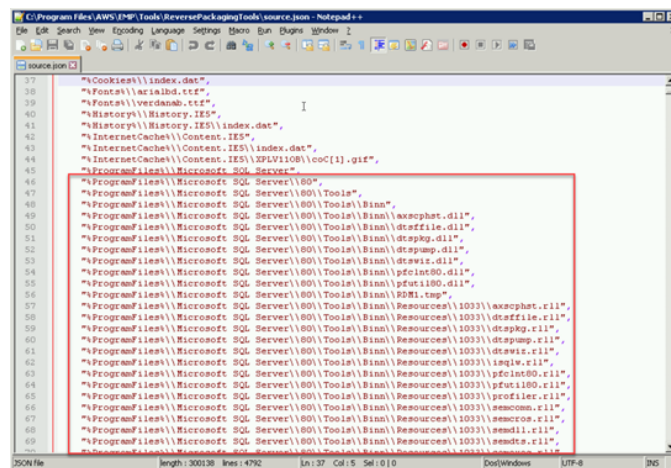
For example:

```
type Logfile.csv | GeneratePackageSource.exe | RemoveKnownFolders.exe > source.json
```

Note

The GeneratePackageSource and RemoveKnownFolders commands can be run one at a time by first piping the logfile.csv into GeneratePackageSource.exe, and then running RemoveKnownFolders.exe to generate the source.json. If the GeneratePackageSource.exe hangs, the Process Monitor may still have a lock on the CSV file. If this is the case, resolve by rebooting the machine.

4. Navigate to Windows Explorer and verify that the .JSON file has been created and that its size is greater than 0.
5. Edit the .JSON file in a text editor, such as Notepad++.
6. Remove the lines where the path name is the same because a folder includes any subfolders and contents.



In this example, the files listed in the selected area of the .JSON file can be updated to "%ProgramFiles%\Microsoft SQL Server", because it includes the folders and contents of this folder.

7. Remove other items that are not relevant to the application and change entries, such as hard-coded drive letters, to variables.

8. Save the file with a different file name, such as `FilteredSource.json`. This allows you to revert to the original file if you encounter problems later in the packaging process.

Create the export bundle of the files and registry keys required for the application

After you edit and save the filtered source .JSON file, you must run the `ExportFromSystem` EMP tool. This tool reads the .JSON file and compiles copies of files and extracts copies of registry keys required to run the application. It adds them to the `Files` and `Registry` folders in the `ReversePackagingTools` folder. When the export is complete, these folders and their contents are compressed into a file called `PackageSource.zip`.

Note

Before you run this command, verify that all application services and processes are stopped so that all of the required resources are successfully exported from the system. Any processes and services that are in use will prevent a successful export.

Follow these steps to run the `ExportFromSystem` EMP tool.

1. Return to the administrator command prompt that you opened previously. If you closed it, open a new command prompt as an administrator and navigate to `C:\Program Files\AWS\EMP\Tools\ReversePackagingTools`.
2. Enter the following command. `<filterfile.JSON>` is the name of the .JSON file that you created when you captured the files and registry keys. `<logfile.txt>` is the name of the log file that you want to create to capture the output of running the `ExportFromSystem` command.

```
type <filterfile.JSON> | ExportFromSystem.exe > <logfile.txt>
```

For example:

```
type FilteredSource.json | ExportFromSystem.exe > exportfromsystem.txt
```

We recommend that you pipe the output to a file to check for errors.

Note

It takes a while before the `C:\Program Files\AWS\EMP\Tools\ReversePackagingTools` prompt displays. This is because of the number of actions being performed by the `ExportFromSystem` tool.

3. Open Windows Explorer and verify that the `PackageSource.zip` has been created and that its size is greater than 0. When the `PackageSource.zip` is created, you can use this to create an EMP package for the application.

Create the new packaged version of the application

The final step in the reverse packaging process is to create a new packaged version of the application.

To package an application, the virtual machine on which the application is to be packaged must be running the same version and service packs as the source machine of the application.

You must first copy the `PackageSource.zip` that you previously created to a new virtual machine.

1. Log on to the new virtual machine.
2. Copy and extract the `PackageSource.zip` to an empty folder.

The contents of the `PackageSource.zip` should be the same as the `C:\Program Files\AWS\EMP\Tools\ReversePackagingTools` folder on the source machine.

3. Follow the steps to create a new package using the EMP Compatibility Package Builder. When **Install Application** is displayed, you can install the application.

You can now install the application using the EMP `DeployToSystem` tool, which automates the installation process. The application must be installed with this tool, and not with the application's native installation system.

1. As an administrator, open a command prompt.
2. Go to the folder that contains the files extracted from the `PackageSource.zip`.
3. Enter the following command, where `<filterfile.JSON>` is the name of the .JSON file that you previously created of the final list of files and registry keys used by the application, and `<logfile.txt>` is the name of the log file that you want to create to capture the output of the `DeployToSystem` command.

```
type <filterfile.JSON> | DeployToSystem.exe > <logfile.txt>
```

For example:

```
type FilteredSource.json | DeployToSystem.exe > deploytosystem.txt
```

We recommend that you pipe the output to a file to check for errors.

It will take a while before your folder is redisplayed. If you are prompted to reboot the instance, you should do so only after the application is installed. If you reboot, run the Compatibility Package Builder using the desktop shortcut.

When the command prompt is displayed, return to the EMP Compatibility Package Builder to complete the packaging process.

Package applications on multiple drives

This section shows the additional configuration required to capture applications that are set up on multiple drives. For this example, a typical install of SQL Server 2000 is installed on the C drive, and the data is stored on the D drive.

Application directories

- **Data directory** — `D:\Program Files\Microsoft SQL Server`
- **Application directory** — `C:\Program Files\Microsoft SQL Server`

The Process Monitor backing file should be used if process monitoring will take several minutes or hours. Create this on the non-system drive with additional filters.

Example: package applications on multiple drives for SQL Server 2000

1. Expand `PackageSource.zip` to the C drive.
2. Run the `DeployToSystem` command from the C drive location. File copy errors may occur if you run this command from the D drive.
3. Package files in the C drive may contain references to D drive locations. These locations must be redirected in the `AppRegistry.xml` file.

```
<Write>
  <KeyName>HKEY_CURRENT_USER\Software\AWSEMP\Compatibility.Package|%GUID%\HKLM
\SOFTWARE\Microsoft\MSSQLServer\Replication</KeyName>
  <ValueName>WorkingDirectory</ValueName>
  <Value ValueType="String">D:\ProgramFiles\Microsoft SQL Server\MSSQL\REPLDATA</
Value>
</Write>
```

Many applications also have an internal configuration file that contains important file or directory paths. For example, SQL 2000 includes a `sqlsunin.ini` file with important file paths. If redirection is not implemented in this file for calls to the D drive file locations, the SQL Server fails to start.

The `procmon` and compatibility engine logs will show several non-redirection and path not found errors, especially in the `ERRORLOG` file.

4. Add the following folder match redirection to `Redirections.xml` to redirect calls from the D drive to the package.

```
<FolderMatch>
  <From>D:\</From>
  <To>ProgData</To>
</FolderMatch>
```

Integrate COM+ applications into EMP packages

This topic contains information for scripted integration of COM+ applications into EMP packages. This includes how to detect whether an application includes COM+ applications and how to include COM+ applications in EMP packages.

COM+ application topics

- [Detect whether an application includes COM+ applications \(p. 23\)](#)
- [Include COM+ applications in an EMP package \(p. 24\)](#)

Detect whether an application includes COM+ applications

Detect COM+ applications

1. Open the Component Services console using one of the following methods.
 - Run `dcomcnfg` in the command line or using PowerShell and expand Component Services.
 - Open Component Services from the **Start** menu: **Start>All Programs>Administrative Tools>Component Services**.
 - Enter `Component Services` in the **Search** box.
2. Expand Component Services to list the COM+ applications. The following are the default COM+ applications that are included in Windows Server 2003 R2 and 2008 R2, with IIS and Application Server roles enabled. .NET Utilities is included only in Windows Server 2003 R2. COM+ Utilities (32-bit) is only included on Windows Server 2008 R2. The rest of the applications are included on both operating system versions.
 - .NET Utilities (Windows Server 2003 R2)
 - COM+ Explorer
 - COM+ QC Dead Letter Queue Listener
 - COM+ Utilities

- COM+ Utilities 32-bit (Windows Server 2008 R2)
- System Application

The names of the COM+ applications typically indicate to which applications they belong.

Include COM+ applications in an EMP package

To include COM+ applications in EMP packages, perform an EMP package build using the standard or reverse packaging models. In this example, the Source Server is the server instance upon which standard packaging was performed, or upon which the process monitoring was performed if the package is built using reverse packaging.

Export the COM+ applications from the source server

1. Open the Component Services console using one of the following methods.
 - Run `dcomcnfg` in the command line or using PowerShell and expand Component Services.
 - Open Component Services from the **Start** menu: **Start>All Programs>Administrative Tools>Component Services**.
 - Enter `Component Services` in the **Search** box.
2. Expand COM+ Applications and identify the COM+ applications that are installed according to the specified applications.
3. Right-click on the first COM+ application that you want to export to open the context menu, and select **Properties**.
4. Choose the **Activation** tab to display the **Activation type** details.
5. Change the activation type to **Server application**. Choose **OK** on the two warning messages, and then choose the **Advanced** tab.
6. Under **Debugging**, select **Launch in debugger** so that you can edit the **Debugger path**. The default debugger path is `C:\Windows\system32\dllhost.exe /ProcessID:{GUID}`. For example, `C:\Windows\system32\dllhost.exe /ProcessID:{0481F901-E8DC-446C-B82F-7746E380214D}`.
7. Add the following string to the path, where `<DeployedPackagePath>` is the path to the deployed EMP package (`%DefaultDir%`). Ensure that there is one space character between this string and the default path:

```
"<DeployedPackagePath>\Compatibility.Package.Engine.exe" /f
```

The debugger path should be set to:

```
"<DeployedPackagePath>\Compatibility.Package.Engine.exe" /f C:\Windows\system32\dllhost.exe /ProcessID:{GUID}
```

For example:

```
"C:\ProgramData\EMP\SQL2005STDSP4_7807\Compatibility.Package.Engine.exe" /f C:\Windows\system32\dllhost.exe /ProcessID:{0481F901-E8DC-446C-B82F-7746E380214D}
```

8. Return to the **Activation** tab and set the **Activation type** back to **Library application**. Choose **Apply** and **OK** to close the COM+ Properties window.
9. Repeat steps three through eight for all of the other COM+ applications you discovered.
10. Right-click on the first COM+ application to open the context menu and select **Export**.

11. Choose **Next** on the Export Wizard.
12. Enter the path to the exported MSI file and select **Export user identities with roles**. Choose **Next**.
13. Choose **Finish** to complete the export.
14. The exported COM+ application MSI and .cab files are exported to the specified location.
15. Repeat steps ten through fourteen for all of the COM+ applications you discovered.
16. Copy all of the exported COM+ applications into the root folder of the EMP package.

Discover missing dependent files

1. On a clean instance running the same operating system as the source server, verify that the required IIS and Application Server roles are enabled.
2. Deploy the EMP package. Do not install the COM+ export at this time.

The exported COM+ applications in the MSI file often do not include all of the dependencies required to register the COM+ applications on a new server. In this case, the dependent libraries would have been captured into the EMP package. However, when the COM+ MSI installation runs, the installations fails because the libraries cannot be found.

3. Open (double-click) the first COM+ MSI to start the installation. If dependencies are not missing and the installations successfully completes, the COM+ application should appear in the Component Services. Run the other COM+ MSI files. If all of the COM+ MSI installations complete successfully, then proceed to the next procedure (Integrate the COM+ application installation into the package deployment). If any installations fail with the error message `Error registering COM+ Application`. Contact your support personnel for more information, proceed with the next steps to discover and add the missing libraries.
4. Launch SysInternals Process Monitor (procmon), clear the procmon window, start monitoring, and attempt the first failed MSI installation again.
5. Stop the monitoring when the error appears.
6. Include a filter with a path that contains `C:\Program Files\COMPlus Applications` (applicable to 32-bit COM+ applications installed on Windows Server 2003 and 64-bit COM+ applications installed on Windows Server 2008 R2). Missing DLLs and possibly TLB files are displayed. If you are running Windows Server 2008 R2 and no missing libraries are displayed, modify your filter to include `C:\Program Files (x86)\COMPlus Applications` to discover any 32-bit COM+ libraries required by your application.
7. Search the ProgData directory for the missing files. When you have found them, create the native directory `C:\Program Files\COMPlus Applications\GUID` and copy the files from their package location into the GUID folder.
8. Clear the procmon screen and attempt the installation again. If it completes, then all of the dependencies have been found. If it does not complete, repeat step 7 to discover the missing files.
9. Repeat this process until all of the COM+ applications have been successfully installed. You may have to modify your filter to contain `C:\Program Files (x86)\COMPlus Applications` to pick up any missing 32-bit COM+ application libraries. The COMPlus Applications folders should now include all of the libraries required to register all of the COM+ applications.
10. Copy the COMPlus Applications folder from the `C:\Program Files` into the corresponding location in the EMP package (`ProgData\Program Files`). Repeat the same process for `C:\Program Files (x86)` if you have discovered any 32-bit COM+ libraries on a 64-bit machine.

Integrate the COM+ application installation into the package deployment

When your COM+ applications are in the package root folder and, if there were any missing COM+ registration dependencies, the discovered libraries in the COMPlus Applications folders are in the package, add the COM+ application installation to the deployment using the `DeploymentScript.xml`

feature. The following example tasks are required to add the COM+ application installation to the deployment.

The first tasks copy the COMPlus Applications folders into their corresponding native locations. This is required only if there are any missing COM+ registration dependencies. Add another task to copy the ProgData\Program Files(x86)\COMPlus Applications if you added this to your package. The second task, or set of tasks, runs the MSI installations.

```
<Install>
  <Programs>
    <Program Order="0" PreInstall="true">
      <ProcessWindowStyle>Normal</ProcessWindowStyle>
      <Path>%SystemX86%\CMD.exe</Path>
      <Args>/c XCOPY /E /H /I /R /Q /Y "C:\ProgramData\EMP\SQL2005STDSP4_7807\ProgData\Program Files\COMPlus Applications" "C:\Program Files\COMPlus Applications"</Args>
      <WaitCondition TimeoutInSeconds="0">Exit</WaitCondition>
    </Program>
    <Program Order="1" PostInstall="true">
      <ProcessWindowStyle>Hidden</ProcessWindowStyle>
      <Path>%SystemX86%\msiexec.exe</Path>
      <Args>/i "%DefaultDir%\Microsoft.SqlServer.MSMQTask.MSI" /L*v "%DefaultDir%\Microsoft.SqlServer.MSMQTask.MSIInstallLog.txt</Args>
      <WaitCondition TimeoutInSeconds="0">Exit</WaitCondition>
    </Program>
  </Programs>
</Install>
```

Your deployed package can now be used as a source package for a successful deployment of your package, which includes COM+ applications. To test the package, copy it to another server instance. The next time the package is deployed, it should automatically install the COM+ applications. The debugger setting in the COM+ applications allows them to be virtualized by the package engine.

Package an IIS-based application

EMP supports packaging and migrating legacy IIS-based applications that run on Windows Server 2003, Windows Server 2008, and Windows Server 2008 R2 to the latest, supported versions of Windows Server running on AWS.

Topics

- [Discovery \(p. 26\)](#)
- [Migrate your IIS-based application \(p. 27\)](#)
- [Troubleshooting packaging an IIS-based application \(p. 29\)](#)

Discovery

The first step of a migration plan is to identify additional IIS-based application dependencies that are installed on the same server as the application you are migrating. For example, an IIS-based application can be dependent on a third-party application that generates reports, such as *Crystal Reports*.

If the dependency information is not available, try the following steps:

1. Navigate to and inspect the configuration files of the web application for dependencies.

The following example `web.config` file, found in `C:\inetpub\wwwroot` of an IIS-based application, shows a dependency on *Crystal Reports* assemblies:


```
PS C:\> Export-IISWebSiteWithDependentFeatures.ps1 -Name Website1,Website2 -  
OutputDirectory C:\DestinationFolder
```

-Name — specify the name of one or more website(s) identified in the **Sites** node.

-OutputDirectory — specify the folder to which the website contents and configuration will be saved.

-DisableContent — optional argument to export the configuration of the legacy IIS websites without exporting the web application files. This command is useful when the web application files are stored on a network drive that is mapped on the server, and there is no requirement to migrate them to the modern Windows Server.

When you run this command, the installation of [MSDeploy](#) provided with the EMP release and included in the `IISTools` folder will be silently installed if it is not already.

When the command completes, a folder is created in the output directory location that you specified. The folder is called `EMP-IIS`. In addition, this folder captures the Windows features that are installed on the legacy server.

Inspect the output of the `Export[WebsiteName].msdeploy.err` and `ExportGlobalConfig.msdeploy.err` files for runtime errors and remediate as required. An empty file indicates that no errors were recorded.

Copy the `EMP-IIS` folder to the target server.

4. Uninstall the EMP Compatibility Package Builder from the source server.
5. Run the PowerShell script `Uninstall-MSDeploy.ps1` provided in the `MSDeploy` folder to uninstall the Web Deploy application.

Target server steps

Apply the following steps to the target server to which the IIS-based web application will be migrated.

1. Install the EMP Compatibility Package Builder on the target server. The EMP IIS migration tools are located in the `Tools/IISTools` folder within the installation directory (64-bit system: `C:\Program Files (x86)\AWS\EMP\` or 32-bit system: `C:\Program Files\AWS\EMP\`).
2. Open PowerShell as an administrator and change the directory to the `IISTools` folder. Then, run the following command.

```
PS C:\> Import-IISWebSiteWithDependentFeatures.ps1 -Path C:\DestinationFolder\EMP-IIS
```

When you run this command, the installation of [MSDeploy](#) provided with the EMP release and included in the `IISTools` folder will be silently installed if it is not already.

The command will then install and set up the server with the Windows features identified from the source server. If a feature that was identified in the source server is deprecated and a replacement feature is not identified, a warning message is displayed. You can edit the `Config.xml` file located in the root of the `EMP-IIS` folder if a manual change to the list of features to install is required.

For Windows Server 2003 applications, a list of Windows features on the legacy server is not identified. Instead, a default list of Windows features is configured on the target operating system.

When the command completes, the web application configuration of the IIS-based web application is set up on the target server.

Inspect the output of the `Import[WebsiteName].msdeploy.err` and `ImportGlobalConfig.msdeploy.err` files for runtime errors and remediate as required. An empty file indicates that no errors were recorded.

3. Uninstall the EMP Compatibility Package Builder from the target operating system. If you discovered application dependencies and are moving to the next stage to capture them, do not uninstall the EMP Compatibility Package Builder.

Stage 2: Capture application dependencies (optional)

This procedure is necessary only if you identify application dependencies in the discovery phase.

1. Follow either the [standard \(p. 15\)](#) or [reverse \(p. 17\)](#) packaging process to capture the application dependencies in an EMP package. Move the package to the target server and [deploy \(p. 35\)](#) the package on the target server with an additional `/DeployAllRegistry` switch.

Example command

```
C:\EMP\Package0001\Compatibility.Package.Deployment.Exe /acceptEULA /deploydir "C:\Programdata\EMP" /DeployAllRegistry
```

The `/DeployAllRegistry` switch makes the EMP package accessible at the machine level and ensures that IIS-based Windows accounts such as IUSR can access the EMP package registry when required.

2. Open PowerShell as an administrator and navigate to the `IISTools` folder. Run the following command.

```
PS C:\> Set-IISEMPConfigurations.ps1 -WebSite WebSite -EMPPackagePath c:\EMPPackageDeployLocation
```

`-WebSite` — specify the name of the website(s) identified in the **Sites** node.

`-EMPPackagePath` — specify the folder to which the website contents and configuration will be saved. For example, `C:\Programdata\EMP\Package0001`.

This command will set the necessary IIS-related integration configurations required for the EMP package.

3. The IIS-based web application migration is complete and you can begin user testing.

Troubleshooting packaging an IIS-based application

The following actions can help you troubleshoot issues that can occur when you package an IIS-based application.

Set `Enable 32-bit application application pool setting to True`

Some applications require the `Enable 32-bit application application pool` to be set to `True` in order to work on a modern operating system. This is especially true for applications for which this setting is currently set to `True` in the legacy environment, or if the application has been ported from a 32-bit system. EMP does not set this option as part of the migration process.

Create IIS-based application migration log files

When you run the following PowerShell scripts, import and export log files are created in the `EMP-IIS` folder.

```
PS C:\> Export-IISWebSiteWithDependentFeatures.ps1
```

```
PS C:\> Import-IISWebSiteWithDependentFeatures.ps1
```

The `.err` files log errors when `.err` commands are run. The `.out` files create a descriptive log of the running of the command.

Sample log files

Export

```
Export[WebsiteName].msdeploy.err  
Export[WebsiteName].msdeploy.out  
ExportGlobalConfig.msdeploy.err  
ExportGlobalConfig.msdeploy.out
```

Import

```
Import[WebsiteName].msdeploy.err  
Import[WebsiteName].msdeploy.out  
ImportGlobalConfig.msdeploy.err  
ImportGlobalConfig.msdeploy.out
```

Use the scripts provided in the `IISTools` folder to help troubleshoot errors

The PowerShell scripts located in the `IISTools` folder support the `-confirm`, `-whatif`, `-verbose`, and `help` parameters.

EMP compatibility package contents

This section describes the folders and files that are included in an EMP compatibility package. When the EMP compatibility packaging process is complete, the output of the package builder is called an EMP compatibility package. The package contains both the file and registry data of the packaged application, and the EMP binaries and configuration files that are required to deploy and run the packaged application.

The EMP package, which is the product to deploy, is called the source package. The post-deployment package is called the deployed package. These packages are slightly different at the level of the root package folder. However, the packaged application content is the same in both packages.

Package contents

- [Source package contents \(p. 30\)](#)
- [Deployed package contents \(p. 35\)](#)

Source package contents

The source package root folder contains two folders, along with a list of files.

- **ProgData folder** — Contains the directory structure and files of the packaged application, captured during the EMP packaging process.

- **EngineBinaries folder** — Contains the EMP package engine binaries, which support both 32-bit and 64-bit Windows operating systems. During deployment, package deployment detects the bit rate of the operating system and deploys the appropriate engine binaries into the deployed package. Both 32-bit and 64-bit binaries are deployed into 64-bit Windows operating systems.

Files

- **Compatibility.Package.Engine.exe** — The compatibility package redirection (virtualization) engine. When the entry point of an application is invoked, the engine is launched with the arguments required to load the target application process as a child process of the engine. As a result, the engine is able to intercept and redirect calls from the process.
- **Compatibility.Package.Engine.Launcher.x64.exe** — Along with its 32-bit counterpart, this executable file is an out-of-process COM server launcher. For packages running on 32-bit Windows operating systems, only the x86 program is required. However, both programs are required on 64-bit Windows operating systems.
- **Compatibility.Package.Engine.x64.dll** — Along with its 32-bit counterpart, this .dll is a library that is used by the compatibility package engine. Both libraries are required for a package to run on a 64-bit Windows operating system. However, only the 32-bit library is required for 32-bit Windows operating systems.
- **HookYou.exe** and **HookMe.dll** — These files provide an alternative method to virtualize a process when it has not been started by the package engine.
- **Other source package files**
 - **_metadata.json** — Contains EMP package metadata.

```
{
  "PackageId": "SQL2005EXP_9482",
  "Icon": "%DefaultDir%\Compatibility.Package.Run.exe",
  "Name": "SQL2005EXP",
  "Version": "",
  "Publisher": "",
  "AWSProfileName": "default"
}
```

- **Compatibility.Package.Deployment.exe** — The package deployment program, called with arguments to deploy, update, or uninstall an EMP package. By default, it logs deployment events to a text file located at the root of the package folder. This program eliminates the need to install an agent on the target operating system.
- **Compatibility.Package.Deployment.exe.config** — Used to pass settings to the executable file, for example, the logging level. The default level is `INFO`.
- **DeploymentWorkflowLog.txt** — The log file created by the package deployment program. Contains a log of deployment events according to the log level in `Compatibility.Package.Deployment.exe.config`.
- **AppRegistry.xml** — The main registry file for the package. Contains most of the registry data of the packaged applications.

```
<RegistryOperations xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  NeedToBeDecoded="true" ValidateWrite="false">
  <Write>
    <KeyName>HKEY_CURRENT_USER\Software\AWSEMP\Compatibility.Package\%GUID%\HKLM\SYSTEM
\CurrentControlSet\services\eventlog\Application\Visual Studio - VsTemplate</KeyName>
    <ValueName>EventMessageFile</ValueName>
    <Value ValueType="ExpandString">%ProgramFilesX86%\Microsoft Visual Studio
8\Common7\IDE\msenv.dll</Value>
  </Write>
  <Write>
    <KeyName>HKEY_CURRENT_USER\Software\AWSEMP\Compatibility.Package\%GUID%\HKLM\SYSTEM
\CurrentControlSet\services\eventlog\Application\Visual Studio - VsTemplate</KeyName>
    <ValueName>TypesSupported</ValueName>
```

```
<Value ValueType="DWord">7</Value>
</Write>
```

- **ComDeployment.xml** — Contains instructions for deploying out-of-process COM servers, COM+, and DCOM components. The setting `Enabled="true"` denotes that the deployment of out-of-process COM servers, as well as COM+ and DCOM components, occurs during package deployment so that they are immediately available. You may want to set `Enabled="false"` to prevent this kind of deployment for some troubleshooting scenarios.

```
<COM xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" Enabled="true">
  <OutOfProcServers>
    <OutOfProcServer>
      <Path>%DefaultDir%\ProgData\Program Files (x86)\Microsoft Visual Studio
8\Common7\IDE\devenv.exe</Path>
    </OutOfProcServer>
  </OutOfProcServers>
</COM>
```

- **Compatibility.Package.Run.exe** — The program that serves as the entry point into the EMP package. When invoked, it verifies that the registry is loaded and consistent with the registry files contents. If not, it loads the package registry before launching the engine with arguments to launch the target application program. By default, it logs high-level run events to a text file located at the root of the package folder.
- **Compatibility.Package.Run.exe.config** — Used to pass settings to the package deployment executable, for example, the logging level.
- **Compatibility.Package.Engine.clc** — The package engine configuration file. Used to apply settings, such as compatibility features, links to other EMP packages, engine detach, and process exclusion from redirection.

The following is a sample of the default file that is included in each package.

```
<AAV PackageId="SQL2005_7206">
<Includes>
  <Include>Redirections.xml</Include>
</Includes>
<!--
<Excludes>
  <Exclude>SomeExecutable.exe</Exclude>
</Excludes>-->
<!--
<Detaches>
  <Detach>SomeExecutable.exe</Detach>
</Detaches>-->
<!--
<COM>
  <CLSID ID="" Excludes="false">
    <Registration ConnectionType = "MultiUse"/>
  </CLSID>
</COM>-->
<!--
<Features>
  <Feature>DEPOptOut</Feature>
  <Feature>HandleInvalidHandle</Feature>
  <Feature>NetworkRedirection</Feature>
  <Feature>LocalMappedObjectShim</Feature>
  <Feature>NotWow64Process</Feature>
  <Feature>ForceWindowsVersion</Feature>
  <Feature>COMVirtualization</Feature>
</Features>
-->
<Features>
  <Feature>RedirectX64PackagedRegistry</Feature>
```

```
</Features>  
</AAV>
```

Contents

- **Includes** — Element that specifies the `Redirections.xml` of the current package. No path is required because it exists in the root folder of the packages. Other packages can be linked to the current package by adding their `Redirections.xml` files here. For more information, see [Link EMP packages \(p. 56\)](#).
- **Excludes and Detaches** — For more information, see [Exclude or detach a process from the package redirection rules \(p. 60\)](#).
- **COM** — Element that specifies class IDs (CLSIDs) that must be excluded from `COMVirtualization`. For more information, see the steps for excluding CLSIDs from `COMVirtualization` under [Enable support for out-of-process Common Object Model \(COM\) in an EMP package \(p. 58\)](#).
- **Features** — See [EMP compatibility package features \(p. 39\)](#).
- **ComRegistryKeys.xml** — Contains the COM, DCOM, and COM+ registration data of the packaged application.
- **DeploymentScript.xml** — Introduces custom configurations into an EMP package. See [Managing EMP custom configurations \(p. 54\)](#).
- **EMP.TelemetryClient.exe** — Program that collects some basic operational information about the usage of EMP to improve the product. The ability of EMP to send telemetry data to AWS is mandatory to deploy packages. For more information, see [Deploy an EMP package \(p. 35\)](#). For data collected, see [Data collected by the AWS End-of-Support Migration Program \(EMP\) for Windows Server \(p. 65\)](#).
- **EMP.TelemetryClient.exe.config** — Passes settings to the Telemetry Client, for example, the `AWSProfileName`.
- **EnvironmentVariables.xml** — Contains the captured environment variables that are required to be available to the virtualized application.

```
<Variables xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">  
  <Variable Name="Path" Append="true" IsEncoded="false" Value="C:\Program Files  
  (x86)\Microsoft SQL Server\90\Tools\bin\ " />  
</Variables>
```

- **FileAssociations.xml** — Contains the file associations registration data for the packaged application.

```
<RegistryOperations xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
  NeedToBeDecoded="true" ValidateWrite="false">  <Write>  
  <KeyName>HKEY_LOCAL_MACHINE\SOFTWARE\Classes\.xdl</KeyName>  
  <ValueName></ValueName>  
  <Value ValueType="String">ssmsee.xdl.9.0_SQL2005_7206</Value>  
</Write>  
<Write>  
  <KeyName>HKEY_LOCAL_MACHINE\SOFTWARE\Classes\ssmsee.xdl.9.0_SQL2005_7206</KeyName>  
  <ValueName></ValueName>  
  <Value ValueType="String">Microsoft SQL Server Deadlock File</Value>  
</Write>
```

- **Programs.xml** — Specifies how EMP launches the programs of the packaged application. Each set of instructions for running a program is called a `RunCondition`. The `RunCondition` is an argument that is passed to `Compatibility.Package.Run.exe`. Each run condition (`run1`, `run2`, and so on) is a launch instruction to package run, which contains an argument to pass to the package engine. The package engine specifies what application program to launch, in what directory, for how long, and includes any arguments to be passed to it.

AWS End-of-Support Migration Program
(EMP) for Windows Server User Guide
Source package contents

```
<Programs xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Program>
    <RunCondition>run1</RunCondition>
    <ProcessWindowStyle>Normal</ProcessWindowStyle>
    <Path>%DefaultDir%\Compatibility.Package.Engine.exe</Path>
    <Args>/f "%DefaultDir%\ProgData\Program Files (x86)\Microsoft SQL Server\90\Tools
\Binn\VSShell\Common7\IDE\ssmsee.exe" %FILEARGS%</Args>
    <WorkingDirectory />
    <WaitCondition TimeoutInSeconds="0">None</WaitCondition>
  </Program>
```

- **Redirections.xml** — The redirection instruction set to the package engine, which specifies which file, folder, or registry key requests should be redirected and to where. It also contains a duplicate of some of the content of the CLC file, which allows it to be configured with compatibility package features, COM exclusions, process exclusions or detaches, as required, when it is used to link the package to another compatibility package.

The following are examples of file, folder, and registry redirection rules.

```
<ExactMatch>
  <From>%SystemX86%\SQLServerManager.msc</From>
  <To>ProgData\Windows\SysWOW64\SQLServerManager.msc</To>
</ExactMatch>
```

```
<FolderMatch>
  <From>%ProgramFiles%\Microsoft SQL Server</From>
  <To>ProgData\Program Files\Microsoft SQL Server</To>
</FolderMatch>
```

```
<KeyMatch>
  <From>HKLM\SYSTEM\CurrentControlSet\services\SQLWriter</From>
</KeyMatch>
```

- **Report.json** — Contains a report on any unsupported application features detected during the package build, for the attention of the packaging engineer.

The following is a sample report for an unsupported COM+ property.

```
[{
  "Level": "Warn",
  "Category": "COM+",
  "Message": "COM+ application DotNetTestMultiCOMPlusApp is using roles for managing the
the security. At the moment roles are not ported across in the compatibility package.
Please recreate all the roles on target machine manually after deploying the package.
Please ensure you configure the roles at component, interface and method level similar
to how they are configured at source machine."
}]
```

- **Services.xml** — Contains instructions to configure any Windows services captured during packaging. ImagePath specifies the package run program with a run condition. The run condition is detailed in the Program.xml file. So, although services are installed natively, the service image is virtualized.

```
<Services xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Service>
    <Name>MSSQL$SQLEXPRESS</Name>
    <ImagePath>%DefaultDir%\Compatibility.Package.Run.exe /RunConditions run5</
ImagePath>
```

```
<Description>Provides storage, processing and controlled access of data and rapid
transaction processing.</Description>
<DisplayName>SQL Server (SQLEXPRESS)</DisplayName>
<Startup>Automatic</Startup>
<StartOnDeploy>>true</StartOnDeploy>
<LogOnAs>
  <Username>NT AUTHORITY\NetworkService</Username>
</LogOnAs>
<Timeout>300000</Timeout>
</Service>
```

- **Shortcuts.xml** — Contains instructions to configure any program shortcuts captured during packaging. The <Target> is the package run program with a run condition as argument. The run condition is detailed in the Programs.xml file. When a program is launched from a shortcut, it runs as a child process of the package engine.

```
<Shortcuts xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Shortcut>
    <Path>%CommonPrograms%\Microsoft SQL Server 2005\SQL Server Management Studio
Express.lnk</Path>
    <Target>%DefaultDir%\Compatibility.Package.Run.exe</Target>
    <Args>/RunConditions run1</Args>
    <Description />
    <IconLocation>%DefaultDir%\ProgData\Program Files (x86)\Microsoft SQL Server
\90\Tools\Binn\VSShell\Common7\IDE\ssmsee.exe</IconLocation>
    <IconIndex>0</IconIndex>
    <WorkingDir />
  </Shortcut>
```

- **eula.html** — The end-user license agreement file.
- **Open Source Licenses.txt** — A license agreement that covers the open source components.

Deployed package contents

When the source package is deployed to the target Windows server, the contents of the .\EngineBinaries\x64 or .\EngineBinaries\x86 folders are copied into the package root folder during the package deployment depending on the bit rate of the operating system. The DeploymentWorkFlowLog.txt is populated with the logged package deployment events.

Once the package has been launched by the start of a service, when an application shortcut or other entry point into the package is launched, another log file is created within the deployed package root folder called RunWorkFlowLog.txt.

RunWorkFlowLog.txt

Contains a brief log of events that take place from package invocation to the launch of the package engine. Once the RunCondition is passed to the package engine that has successfully launched, package run exits after logging a successful launch event.

Deploy an EMP package

This topic contains information and steps to set up for and deploy an EMP package.

Topics

- [Requirements for deploying an EMP package \(p. 36\)](#)
- [Run the deployment tool \(p. 37\)](#)

Requirements for deploying an EMP package

When you deploy an EMP package on Windows Server 2012 or later, AWS credentials and connectivity to the AWS application modernization metrics service are required. The deployment fails if the package cannot validate the supplied AWS credentials or cannot send the mandatory telemetry to AWS.

There are two ways you can provide AWS credentials to the deployment package if they are not set up on the server.

- If your deployment server is not an Amazon EC2 instance, you can configure the AWS profile on the server and update the profile name in the `metadata.json` file in the root of the packaged folder.
- If your deployment server is an Amazon EC2 instance, you can assign an `execute-api:Invoke` IAM role to the server.

Configure the AWS profile on the server (server is not an EC2 instance)

You can configure the AWS profile on the server using the AWS CLI or AWS Tools for Windows PowerShell. You must set up an IAM user in your AWS environment. The IAM user must be configured to allow `execute-api:Invoke`. To configure this, assign the following IAM policy to the IAM user of the AWS profile on the server.

```
{ "Version": "2012-10-17", "Statement": [{ "Effect": "Allow", "Action": "execute-api:Invoke", "Resource": "*" }]}
```

AWS CLI

When you use the AWS CLI, profile information is stored in the `C:\Users\<username>\.aws` directory. Use the `aws configure` command to configure the profile for the IAM user you set up to allow for `execute-api:Invoke` permissions. The AWS CLI can be downloaded from the [Installing, updating, and uninstalling the AWS CLI](#) page. For more information about how to specify a profile using the AWS CLI, see [Named profiles](#).

AWS CLI example

```
C:\>aws configure
AWS Access Key ID [None]: <EXAMPLE-ACCESSKEY>
AWS Secret Access Key [None]: <EXAMPLE-SECRETKEY>
Default region name [None]:
Default output format [None]:

C:\>
```

AWS Tools for Windows PowerShell

When you use AWS Tools for Windows PowerShell, profile information is stored in `C:\Users\<username>\AppData\Local\AWSToolkit\RegisteredAccounts.json`. Use the `Set-AWSCredential` command to configure the profile. For more information about how to specify credentials using AWS Tools for PowerShell, see [Using AWS credentials](#) in the *AWS Tools for PowerShell User Guide*.

AWS Tools for Windows PowerShell example

```
PS C:\Program Files (x86)\AWS Tools\PowerShell\AWSPowerShell> Set-AWSCredential -
AccessKey <EXAMPLE-ACCESSKEY> -SecretKey <EXAMPLE-SECRETKEY> -StoreAs default
PS C:\Program Files (x86)\AWS Tools\PowerShell\AWSPowerShell>
```


Note

If you do not specify a name when you create a profile, it will default to `default` by both the AWS CLI and AWS Tools for PowerShell. You are not required to update the `metadata.json` file found in the root of the EMP package. If you specify a new name for the profile at a later time, update the `AWSProfileName` property in the `metadata.json` file.

Assign IAM role to the server (server is an EC2 instance)

Assign an IAM role to the deployment server and verify that the following IAM policy is applied to it. For more information about how to assign an IAM role, see [Creating IAM roles](#).

```
{ "Version": "2012-10-17", "Statement": [{ "Effect": "Allow", "Action": "execute-api:Invoke", "Resource": "*" }]}
```

Connectivity to the AWS application modernization metrics service

The EMP deployment server must have internet connectivity to create a secure `http` outbound connection to the AWS application modernization service.

Run the deployment tool

To run the deployment tool, perform the following steps.

1. Open a command prompt as an administrator.
2. Run the following command to deploy the package to all of the users of the server, where `<path-to-package>` is the path of the EMP package, and `<switches>` are the relevant command line switches you want to specify.

```
<path-to-package>\Compatibility.Package.Deployment.Exe /<switches>
```

For example:

```
C:\EMP\Package0001\Compatibility.Package.Deployment.Exe /acceptEULA /deploydir "C:\Programdata\EMP"
```

When you run this command, the following operations are performed.

- All files in the `<path_to_package>` folder are copied to the specified `/deploydir`.
- All shortcuts specified in the `shortcuts.xml` are written to the public profile for visibility to all users of the server.
- Shortcuts for a path in the user desktop or **Start** menu are translated to the equivalent of the directory of the public profile.
- Any file type associations specified in the `FileAssociation.xml` are created in HKLM root key of the registry.
- If the `/deploydir` switch is provided, the package is copied to the specified folder.

Note

The EMP package application registry is not written until a shortcut is first launched.

Working with EMP packages

This section includes information to help you manage EMP packages, including best practices, compatibility package features, how to edit, upgrade, and maintain an EMP package, how to update a deployed EMP package, and how to uninstall an EMP package.

Topics

- [Best practices for packaging applications with AWS End-of-Support Migration Program \(EMP\) for Windows Server \(p. 38\)](#)
- [EMP compatibility package features \(p. 39\)](#)
- [Edit, upgrade, and maintain an EMP package \(p. 47\)](#)
- [Optimize Process Monitor for Reverse Packaging \(p. 49\)](#)
- [Update a deployed EMP package \(p. 50\)](#)
- [Uninstall an EMP package \(p. 51\)](#)
- [Enable EMP compatibility package engine logging \(p. 52\)](#)
- [Managing EMP custom configurations \(p. 54\)](#)
- [Link EMP packages \(p. 56\)](#)
- [Applications using ODBC drivers \(p. 57\)](#)
- [Enable support for out-of-process Common Object Model \(COM\) in an EMP package \(p. 58\)](#)
- [Add side-by-side \(SXS\) assemblies to an EMP compatibility package \(p. 59\)](#)
- [Exclude or detach a process from the package redirection rules \(p. 60\)](#)
- [Run cmd.exe as a child process to the EMP compatibility package engine \(p. 61\)](#)

Best practices for packaging applications with AWS End-of-Support Migration Program (EMP) for Windows Server

Following the best practices in this section when creating EMP packages increases the likelihood of creating successful EMP packages and achieving faster migration lifecycles.

Note

Packaging refers specifically to the process of capturing a legacy application using the EMP Package Builder on the source operating system.

Always package on the source server

The [EMP Application Packaging Model \(p. 13\)](#) demonstrates that the application packaging step occurs on the source server. For example, if an application installs and functions on Windows Server 2008 R2, then the packaging step is performed on the same operating system. Technically, it is possible to package the application on the operating system that the application is migrating to, such as Windows Server 2019. However, packaging on the legacy OS version ensures that the application installs and functions as expected. The application may fail to install and function on the modern operating system and, as a result, the install capture process on the modern operating system can result in a package created in a non-working state.

Always package on a clean server

After an application has been installed or packaged on a server, we recommend that you do not repack the application on the same server in the same state. This is because the EMP Package Builder

compares the differences between snapshots taken before and after the application installation on the operating system. If any legacy application components are on the operating system before the snapshot is taken, the resulting EMP package will not include the application components.

Creating snapshots of your packaging machine before carrying out any packaging would mean you could always restore your system to a clean state if you needed to recapture the application.

Disable background noise

Turn off Microsoft Defender Antivirus, any other antivirus software, and automatic Windows Update to ensure that changes to the system by any of these background tasks are not captured in the package. If your security policy requires that you keep any antivirus software turned on, then you can unload them when performing the package builder process.

Application prerequisite software or dependencies

If an application requires prerequisite software, for example, a legacy version of Java, and this software is not included or installed natively in the target operating system, then we recommend that you include these components in the EMP package for the application.

If an application requires operating system dependencies, for example, specific Windows roles and features, or requires an application prerequisite software that will be included on the target operating system, then we recommend that you set up these dependencies on the packaging server before the application is captured using the EMP Package Builder.

Familiarize yourself with the application being packaged

We recommend that you document how the application installation is completed, and also the application workflows. This helps to ensure that you can refer back to the documentation during the packaging phases. This also helps to ensure that you can review the finalized list of steps required to capture a complete working state of the application. The document should list known issues to watch out for and help you plan how dependencies must be captured or migrated. We recommend that you perform a test install of the application before starting the EMP process.

One package per server

When possible, we recommend that you create a single EMP package for each server that is being migrated. This ensures a simple package design to help facilitate migration and future management of the application.

Discovery

We recommend that you understand how to structure an EMP package before you create it. To understand the package structure and how the application works in its environment, we recommend that you perform application discovery. For more information about the application discovery process, see [High-level AWS End-of-Support Migration Program \(EMP\) for Windows Server application discovery exercise \(p. 11\)](#). You can use this information to structure the EMP package.

EMP compatibility package features

The EMP compatibility package (`compatibility.package.engine.exe`) includes features that can be enabled to resolve compatibility issues with packaged applications. This topic defines these features and demonstrates how they work.

EMP compatibility package features

Feature name	Description
<code>ForceExternalManifest</code>	Forces the use of an external manifest file, which overrides the system default.

Feature name	Description
RegClassesMerging	Merges virtual and real registry values into a virtual key.
DoNotHideDebugger	Ensures that process debugging remains visible.
HandleInvalidHandle	Ignores invalid handle errors.
NotWow64Process	Forces 32-bit applications running on 64-bit Windows to be virtualized as if they are running on 32-bit Windows.
NetworkRedirection	Redirects hostnames, domain names, IPs, and ports.
LocalMappedObjectShim	Converts global file mappings to local.
DEPOptOut	Identifies and handles DEP exceptions.
ExcludeNativeWindows	Prevents a packaged application from using a native Windows application or process.
COMVirtualization	Virtualizes COM.
ForceWindowsVersion	Forces a Windows version check to a specific Windows version.
RedirectX64PackagedRegistry	Redirects 64-bit registry when running a 32-bit application.
LoadSystemResources	Loads architecture-independent resource files regardless of bitness.

Compatibility package features

- [ForceExternalManifest](#) (p. 40)
- [RegClassesMerging](#) (p. 41)
- [DoNotHideDebugger](#) (p. 41)
- [HandleInvalidHandle](#) (p. 42)
- [NotWow64Process](#) (p. 42)
- [NetworkRedirection](#) (p. 42)
- [LocalMappedObjectShim](#) (p. 44)
- [DEPOptOut](#) (p. 45)
- [COMVirtualization](#) (p. 45)
- [ForceWindowsVersion](#) (p. 46)
- [RedirectX64PackagedRegistry](#) (p. 46)
- [LoadSystemResources](#) (p. 47)

ForceExternalManifest

Windows includes a global setting in: `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\PreferExternalManifest` - `DWORD` that controls whether external manifests should be used when a process is launched (or DLL loaded). By default, this registry entry is missing, with

a value of 0 (disable). This means that Windows uses an embedded manifest from the `exe/dll`, if one exists.

The feature `ForceExternalManifest` allows the process launched by EMP to use an external manifest file.

An application manifest is an XML file that describes and identifies the shared and private side-by-side assemblies to which an application should bind at runtime. The name of an application manifest file is the name of the application's executable followed by `.manifest`.

The following is an example of an application manifest that disables Windows theming, which can cause compatibility issues with an earlier application version.

```
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0"
  xmlns:asmv3="urn:schemas-microsoft-com:asm.v3" >
  ...
  <asmv3:application>
    <asmv3:windowsSettings xmlns="http://schemas.microsoft.com/SMI/2011/WindowsSettings">
      <disableTheming>true</disableTheming>
    </asmv3:windowsSettings>
  </asmv3:application>
  ...
</assembly>
```

RegClassesMerging

If an application uses an `enumerate` key function to determine available keys, the compatibility package redirections may not account for this. Adding a high-level redirection to resolve this behavior can result in more issues and prevent your application from functioning. For example, if the application is enumerating `HKLM\Software\Classes` to determine available classes, adding a redirection for `HKLM\Software\Classes` will most likely result in a failure. The solution is to enable registry merging so that the redirected and local registry keys are seen as one registry.

The following compatibility engine configuration example enables the `RegClassesMerging` feature in the `Compatibility.Package.Engine.clc` file.

```
<AAV>
  <Features>
    <Feature>RegClassesMerging</Feature>
  </Features>
</AAV>
```

The following example adds a `KeyMatch` redirection rule for the registry key in the previous example to `Redirections.xml`.

```
<KeyMatch>
  <From>HKLM\SOFTWARE\Classes</From>
</KeyMatch>
```

DoNotHideDebugger

This feature ensures that process debugging remains visible. If not enabled, some applications will pass exceptions or breakpoints to the debugger (engine) and some applications can break. This feature is usually enabled to prevent tampering or for license protection, and is often not required.

The following compatibility engine configuration example enables the `DoNotHandleDebugger` feature in the `Compatibility.Package.Engine.clc` file.

```
<AAV>
  <Features>
    <Feature>DoNotHideDebugger</Feature>
  </Features>
</AAV>
```

HandleInvalidHandle

When invalid handles cause applications to fail to run when using `DoNotHideDebugger`, then the compatibility engine can be configured to ignore the invalid handles. If a thread uses a handle to a kernel object that is invalid (for example, because it is closed) Windows notifies the debugger. The EMP engine handles any invalid handles by default. However if `DoNotHideDebugger` is enabled, the EMP engine does not handle invalid handles. If the `HandleInvalidHandle` feature is set, then the exception is handled and invalid handles are ignored.

The following compatibility engine configuration example enables the `HandleInvalidHandle` feature in the `Compatibility.Package.Engine.clc` file.

```
<AAV>
  <Features>
    <Feature>DoNotHideDebugger</Feature>
    <Feature>HandleInvalidHandle</Feature>
  </Features>
</AAV>
```

NotWow64Process

When enabled, this feature prevents Windows from using WOW64 redirections and forces processes to run as 32-bit in the package by hooking the `IsWow64`, `PrintDlgExA`, `PrintDlgExW`, `PrintDlgA`, and `PrintDlgW` APIs.

The following compatibility engine configuration example enables the `NotWow64Process` feature in the `Compatibility.Package.Engine.clc` file.

```
<AAV>
  <Features>
    <Feature>NotWow64Process</Feature>
  </Features>
</AAV>
```

Note

This feature, when enabled, causes the printer driver host for 32-bit applications (`SPLWOW64.exe`) to fail, and must be excluded if the packaged application launches it.

NetworkRedirection

The `NetworkRedirection` feature enables network redirection for hostname, domain name, IP, and ports. This allows server applications running in packages to redirect their network requests to new names, IP addresses or ports, so that you can migrate applications to new servers without changing application source code.

Use cases

- Hostname virtualization with `<Network ThisComputer="legacy_machine_name">` applies to server applications for which you don't have the installation media, and the application has been extracted from the server on which it runs. This feature virtualizes the hostname of the server on which the application runs so that it behaves as if it runs on the original server.

- Domain Name redirection with <DomainName> is for server applications for which you don't have the installation media, and the application has been extracted from the server on which it runs. This feature redirects from the name a server application expects to find on the network to one that is present.
- IP address and port redirection with <Connect> enables applications to accept connections <From> one IP address or port <To> another.

Note

Apply the Microsoft server naming conventions described at <https://docs.microsoft.com/en-us/troubleshoot/windows-server/identity/naming-conventions-for-computer-domain-site-ou>. These are the only naming conventions supported.

The following compatibility engine configuration examples enable the NetworkRedirection feature in the Compatibility.Package.Engine.clc file.

Hostname virtualization

```
<AAV>
  <Features>
    <Feature>NetworkRedirection</Feature>
  </Features>
  <Network ThisComputer="legacy_machine_name">
  </Network>
</AAV>
```

Domain name redirection

```
<AAV>
  <Features>
    <Feature>NetworkRedirection</Feature>
  </Features>
  <Network>
    <DomainName>
      <From>legacy_machine</From>
      <To>new_machine</To>
    </DomainName>
  </Network>
</AAV>
```

IP and port redirection

The following example redirects network connections from 192.168.2.1 on port 13000 to 127.0.0.1 on port 12000.

```
<AAV>
  <Features>
    <Feature>NetworkRedirection</Feature>
  </Features>
  <Connect>
    <From>
      <IP>192.168.2.1</IP>
      <Port>13000</Port>
    </From>
    <To>
      <IP>127.0.0.1</IP>
      <Port>12000</Port>
    </To>
  </Connect>
</AAV>
```

Logging

When the <DomainName> feature is enabled, the compatibility engine logs the following.

```
Server redirection from %old% to %new%
```

Only the Microsoft server naming conventions described at <https://docs.microsoft.com/en-us/troubleshoot/windows-server/identity/naming-conventions-for-computer-domain-site-ou> are supported. The following messages are logged if a server name does not apply these naming conventions.

```
Server name cannot begin with a period (.) and will not be virtualized
```

```
Server name cannot be longer than 15 characters and will not be virtualized
```

```
Server cannot contain invalid character "invalid_character" and will not be virtualized
```

If <To> is not specified, the following message is logged.

```
Server redirection will not redirect from %old%
```

If <From> is not specified, the following message is logged.

```
Server redirection will not redirect to %new%
```

LocalMappedObjectShim

The name of the file mapping object can include a `Global` or `Local` prefix in order to create the object in the global or session namespace. If a service or system creates a file mapping object in the global namespace, any process running in any session can access that file mapping object if the caller has the required access rights. To ensure that the kernel object names created by your applications do not conflict with the names of any other applications, enable the `*LocalMappedObjectShim` feature. This feature converts all file mapping objects from the global to local namespace if no redirection rule is set for the object name.

Use cases

- Enable an application that requires administrator permissions to run on an account with lower permissions.
- Enable multiple instances of the desktop application to run on a server operating system when the use of global objects by the application prevents the application from installing.

The following compatibility engine configuration examples enable the `LocalMappedObjectShim` feature in the `Compatibility.Package.Engine.clc` file.

```
<AAV>  
  <Features>  
    <Feature>LocalMappedObjectShim</Feature>  
  </Features>  
</AAV>
```

File mapping exclusions can be applied for named file mapping objects so that they remain global objects by including the following tags.


```
<FileMappingExclusions>  
  <FileMappingExclusion>Global\string</FileMappingExclusion>  
</FileMappingExclusions>
```

DEPOptOut

Applications written using Visual Studio 2008, or earlier, are incompatible with operating systems enabled with Data Execution Prevention (DEP), this includes the following.

- Systems configured with Secure Boot.
- Default policies applied to the Windows operating system.
- Windows running the Enhanced Mitigation Experience Threat (EMET) toolkit.

This incompatibility is caused by the forcing of DEP enablement for an application.

Use cases

- Enable the application to opt out of DEP, so that it can run on the server or desktop without configuration changes for EMET, or the default policies that are applied to an application within its organization.
- For applications running in the compatibility package, the DEPOptOut feature resolves memory access violations by changing the memory address location to an executable part of memory.

Identify whether a failure is due to DEP

If a failure is caused by DEP, the application crashes with or without an error message. When a detailed error message is displayed, it shows the exception details, which include: Exception Code: c0000005, which means ACCESS VIOLATION, and Exception Data: 00000008.

On later versions of Windows, the message doesn't display the exception details. You must look at the Windows application event log. Error Event 1000 will report the exception code C0000005.

The following compatibility engine configuration example enables the DEPOptOut feature in the Compatibility.Package.Engine.clc file.

```
<AAV>  
  <Features>  
    <Feature>DEPOptOut</Feature>  
  </Features>  
</AAV>
```

COMVirtualization

The COMVirtualization feature is required to virtualize the out-of-process COM servers. If the ComDeployment.xml file in the compatibility package contains one or more OutOfProcServer entries, then you must enable the COMVirtualization feature.

The following compatibility engine configuration example enables the COMVirtualization feature in the Compatibility.Package.Engine.clc file.

```
<AAV>  
  <Features>  
    <Feature>COMVirtualization</Feature>  
  </Features>  
</AAV>
```

ForceWindowsVersion

If an application requires a particular version of the operating system in order to run, it queries the operating system to ensure that the expected version, build, service pack, or type (desktop or server) is returned. EMP compatibility packages can intercept the API requests and return values specified by the compatibility engine configuration file.

The following compatibility engine configuration example enables the `ForceWindowsVersion` feature in the `Compatibility.Package.Engine.clc` file.

Possible values for `ProductType` are:

- `Server`
- `DomainController`

```
<Features>
  <Feature>ForceWindowsVersion</Feature>
</Features>
<ForceWindowsVersion>
  <MajorVersion>INT</MajorVersion>
  <MinorVersion>INT</MinorVersion>
  <BuildNumber>INT</BuildNumber>
  <ProductType>TYPE</ProductType>
  <ServicePackText>INT</ServicePackText>
  <ServicePackMajor>INT</ServicePackMajor>
  <ServicePackMinor>INT</ServicePackMinor>
</ForceWindowsVersion>
```

Example scenario

An EMP-packaged application must run on Windows Server 2019. However, it fails with an error message stating that the application must be installed on Windows Server 2003. This is because it checks whether the operating system is Windows Server 2003 and finds Windows Server 2019. If the application does not require a specific service pack, `ForceWindowsVersion` can be configured as follows.

```
<Features>
  <Feature>ForceWindowsVersion</Feature>
</Features>
<ForceWindowsVersion>
  <MajorVersion>5</MajorVersion>
  <MinorVersion>2</MinorVersion>
  <BuildNumber>3790</BuildNumber>
</ForceWindowsVersion>
```

RedirectX64PackagedRegistry

The Compatibility Package Builder detects whether it is running on a 64-bit operating system and writes the `<Feature>RedirectX64PackagedRegistry</Feature>` configuration to the `clc` file so that the package knows which platform that it was created on. Packages created on a 32-bit operating system do not require this compatibility feature.

The following compatibility engine configuration example enables the `RedirectX64PackagedRegistry` feature in the `Compatibility.Package.Engine.clc` file.

```
<AAV>
  <Features>
    <Feature>RedirectX64PackagedRegistry</Feature>
```

```
</Features>  
</AAV>
```

LoadSystemResources

This feature loads architecture-independent resource files regardless of bitness. The `LoadSystemResources` feature is useful when 32-bit applications must access the resource-only DLL present in the native `system32` directory instead of `syswow64`. This is necessary when the wow64 file system redirection is enabled for a 32-bit application.

The following compatibility engine configuration example enables the `LoadSystemResources` feature in the `Compatibility.Package.Engine.clc` file.

```
<AAV>  
  <Features>  
    <Feature>LoadSystemResources</Feature>  
  </Features>  
</AAV>
```

Edit, upgrade, and maintain an EMP package

The EMP Package Editor is a tool that is used to modify existing EMP compatibility packages. You can use the Package Editor to apply upgrades, security updates, hot fixes, and service packs to the packaged application. You can also use the Editor to maintain the EMP components. The editor supports reboots so that you can apply an application update that requires a reboot during installation.

Important

You must use the Package Editor to update an EMP package on the same architecture on which the original package was created. For example, if the package was created on an x86 machine, then the Editor must update the package on an x86 machine.

The EMP Package Editor is installed with the EMP Compatibility Package Builder. A shortcut for the Editor is included in the same menu as the EMP Compatibility Package Builder.

Topics

- [Edit the application in an EMP package \(p. 47\)](#)
- [Upgrade the application in an EMP package \(p. 48\)](#)
- [Maintain an EMP package \(p. 48\)](#)

Edit the application in an EMP package

Perform the following steps to edit an application in an EMP package.

1. Verify that you are using the latest version of the EMP Compatibility Package Builder. For version history, see [AWS End-of-Support Migration Program \(EMP\) for Windows Server version history \(p. 68\)](#).
2. Copy the EMP package that you want to edit to the server.
3. Open the Amazon Web Services folder and launch the Compatibility Package Editor.
4. On the **Home** tab, choose **Open an existing Compatibility Package**.
5. Navigate to and choose the package that you want to update. Then choose **Select Folder**.
6. To make changes to files and folders, choose the **Files** tab and navigate to the file or folder you want to edit. You can review the files and folders added to or removed from the package. All changes are displayed by default. The following filters can be applied.

- **New** — Display all files added to the package.
- **Modified** — Display all files changed by the update.
- **Deleted** — Display all files removed during the application update.

When you open the context (right-click) menu on any folder, you can **Add Files**, **Add Folders**, or **Delete** files. When you right-click on a file, you can only **Delete** files.

7. To remove registry keys, open the **Registry** tab, navigate to the registry key that you want to remove, open the context (right-click) menu on the registry key, then choose **Delete**.
8. When you have finished editing files, folders, or registry keys, choose **Save** to build the changes into an updated package.
9. When the Package Editor updates a package, it creates a new package folder for the updated package. It appends the folder name with a version number. The original package is left unchanged for future reference. In addition, the package ID is not changed so that you can use the updated package to update a deployed instance of the original package.
10. To update a deployed EMP package with an updated version, run the following command on the server running the original deployed package.

```
<PathtoUpdatedPackage>\Compatibility.Package.Deployment.exe /update
```

Upgrade the application in an EMP package

Perform the following steps to upgrade an application in an EMP package.

1. Some patch installers check for the presence of an application before installing updates. So verify that you are using the Windows Server operating system version supported by the application and the update, and include the original application installed natively on the server.
2. Verify that the latest version of the EMP Compatibility Package Builder is installed on the server you want to update.
3. Copy the EMP package that you want to update to the server.
4. Open the Amazon Web Services folder and launch the Compatibility Package Editor.
5. On the **Home** tab, choose **Open an existing Compatibility Package**.
6. Navigate to and choose the package you want to update, then choose **Select Folder**.
7. To make changes to files and folders, choose the **Files** tab and navigate to the file or folder you want to edit. Choose **Update**.
8. Install the application patch and make any necessary configuration changes.
9. When the installation is complete, choose **Next** and the Package Editor merges the updates to the EMP package.
10. When the merge is complete, all of the updates made to the package by the patch installation can be viewed in the **Files** and **Registry** tabs. The upper right-hand corner displays the total number of files (**All**), as well as the number of **New**, **Modified**, **Deleted**, and **existing** files.
11. Choose **Save** to build the updates to a new package.

Maintain an EMP package

When a new version of EMP is released, you can update deployed EMP packages with new binaries. Perform the following steps to update a deployed EMP package with new EMP binaries.

1. Verify that the latest version of the EMP Compatibility Package Builder installed on the server.
2. Navigate to the Package Builder installation directory.
3. Navigate to the `Runtime.x64` or the `Runtime.x86` folder, depending on the package architecture.
4. Copy the files (or a subset of the files, depending on what requires updating) in this folder to the folder in the package that requires the update.

Optimize Process Monitor for Reverse Packaging

Process Monitor is an advanced monitoring tool for Windows that captures real-time file system, registry, process, and thread activity. The first step of the EMP reverse packaging process is to capture a Process Monitor (procmon) log of the entire functional running of the application on the source operating system. The log is used to create an EMP package consisting of all of the required components for the application to successfully function on a modern operating systems after it has been migrated. An incomplete capture can result in missing application components.

You can download the latest version of Process Monitor from Microsoft at: <https://docs.microsoft.com/en-us/sysinternals/downloads/procmon>.

Perform the following steps to set up capturing for Process Monitor. Use these steps only as a guide. The monitoring process requirements for each application can vary.

1. Download the tool on the system running the application that you want to discover.
2. As an administrator, open the command prompt, and launch Process Monitor.

```
Procmon.exe /AcceptEula /Noconnect
```

`/AcceptEula` automatically accepts the EULA license and bypasses the EULA dialog box.

`/Noconnect` This flag prevents Process Monitor from automatically starting log activity.

3. Configure Process Monitor to save captured logs to a backing file as opposed to virtual memory by navigating to **File>Backing Files** and choosing the **Use file name** option. Select the location and file to which you want to save the backing file.

Note

If the system used to capture the logs does not have sufficient storage capacity, you can store the data in another location, such as on a different server or external storage device.

4. Start the capture by choosing **Capture**. To stop the capture, choose **Capture** again.

The following optional steps reduce the size of the log file, where possible. To reduce the size of the log file, we recommend that you run procmon only when running the application windows to reduce capture of background noise and unrelated workflows.

1. Verify that the following options are not selected:
 - **Process and Thread Activity**
 - **Network Activity**
 - **Profiling Events**
2. Select **Drop Filtered Events** in the **Filter** menu. This prevents events that don't meet the filter criteria from being added to the log.
3. The following table contains common exclusion items related to the operating system that are not required for the application capture. You can add these exclusions to the Process Monitor application capture exclusion filter.

Exclusions in the `EMP_Procmon_Exclusions.PMF` filter

Exclusion item	Description
<code>Wininit.exe</code>	Windows Start-Up Application
<code>Wuauclt.exe</code>	Windows Update
<code>Dwm.exe</code>	Desktop window manager
<code>Spoolsv.exe</code>	Spooler SubSystem App
<code>Lsass.exe</code>	Local Security Authority Process
<code>Audiodg.exe</code>	Windows Audio Device Graph Isolation
<code>SearchIndexer.exe</code>	Windows Search Indexer
<code>Taskhostw.exe</code>	Host Process for Windows Tasks
<code>Ctfmon.exe</code>	CTF loader
<code>CcmExec.exe</code>	Host Process for Microsoft Endpoint Configuration Manager

You can further expand the number of exclusion items by performing the following steps.

1. Run procmon for a limited period of time or without the execution process.
2. Analyze the capture logs and note any processes that are not related to the application.
3. Add the unrelated processes as additional exclusion items.
4. For applications where a complete list of required processes are known, you can start a Process Monitor capture and include only these processes in the capture. If this method results in missed process applications, the final logs may not contain the required information to complete a working package.

Update a deployed EMP package

This topic contains information to guide you through the process of updating an already deployed EMP package.

To update the contents of an already deployed EMP package with a new version of the package, run the `Compatibility.Package.Deployment.exe` command with the `/update` switch.

```
Compatibility.Package.Deployment.exe /update
```

The package may include changes to the following:

- Files
- Registry settings
- Shortcuts
- Package configuration files
- File type associations

Important

If you attempt to use the `/deploydir` switch when a package has already been deployed, a `Failed to deploy' exit code -1` error will be returned. The `/update` switch must be used to update the package to the latest version, or the `/uninstall` switch must be used to remove the package first.

Each component is updated as follows.

File Associations

If the file type associations source file (`FileAssociations.xml`) in the new package is the same as the one in the currently deployed package, `/update` will recreate any missing file type associations and restore values or types to the original values and types specified during the initial deployment.

Note

The `/update` switch preserves any values that appear in the registry that are not specified in the source file. If the file type associations source file (`FileAssociations.xml`) in the new package is different from the one in the currently deployed package, `/update` deletes the registry values that do not appear in `FileAssociations.xml` and updates values and types that have changed.

Shortcuts

New shortcuts specified in `Shortcuts.xml` will be created. Shortcuts that do not exist in the XML file will be removed. Fields that are different between the currently deployed and the new version to be deployed will be updated to the latest version.

Registry

A registry update will be performed the next time the application runs. This does not happen as part of the update process. Updates are performed if the value of `Last Modified Date of Registry Added` in `HKCU\Software\AWSEMP\Compatibility.Package\{appid}` differs from value of the last modified date of the new application registry XML file (`AppRegistry.xml`).

The registry update removes all keys under `{appid}`, but not the values of the keys, and creates all of the entries specified in the `AppRegistry.xml` file. The update then sets the last modified time to the time of the new `AppRegistry.xml` file.

Subsequent application start events will not initiate registry updates because the modified time of the file will match the value stored in the registry. If the last modified time cannot be found in the registry, the registry will be created using the latest `AppRegistry.xml` file. If the `AppRegistry.xml` file is invalid, `/update` will report an error and will not remove any application registry.

Uninstall an EMP package

When you uninstall a package, all of the files, shortcuts, file type associations, and registry keys associated with the package are removed. Files in use, which cannot be deleted, are marked for deletion for the next reboot. After reboot, Windows removes the files marked for deletion along with the registry configuration for Out of Process COM virtualization.

To uninstall a package, perform the following steps.

1. Open a command prompt as an administrator.
2. Run the following command, where `<source-location-of-package>` is the location from which the package was initially deployed.

```
C:\<source-location-of-package>\Compatibility.Package.Deployment.exe /U
```

Note

If you run this command from the deployed location, the uninstall will be incomplete. Verify that the package is uninstalled from the correct source path.

Enable EMP compatibility package engine logging

This topic contains information to help you to enable logging for the EMP compatibility package engine. By default, logging is disabled. Logging helps you identify problems with applications running in an EMP package.

Logs generated by the compatibility engine are created in the following folders:

- On Windows Server 2008 and later versions: `*%LocalAppData%\AWSEMP\Logs*`
- On Windows Server 2003: `%UserProfile%\Local Settings\Application Data\AWSEMP\Logs`

Note

`LocalAppData` resolves to a special location for the `SYSTEM` account: `C:\Windows\System32\config\systemprofile\AppData\Local`.

You can update the default log file location, if required. To do this, see step four in the procedure at the end of this topic (**Enable compatibility package engine logging**).

Each time the package engine runs, two logs are created:

- **Log for package engine** — Outputs to `reversedate-time-PID-Compatibility.Package.Engine.log`. For example, `20190425-113000-000600-Compatibility.Package.Engine.log`.
- **Log for the child process(es) redirected by the package engine** — outputs to `reversedate-time-parentPID-PID-ExecutableName.log`. For example, `20190425-113000-000600-002820-MyApp.log`.

The package engine log for child processes includes the following columns:

Column	Description
TIMESTAMP	The date and time that the log file entry was written.
TID	The thread ID that generated the log item.
LOG TYPE	Defines the type of log entry: <ul style="list-style-type: none">• INFO — Contains general information, such as a file redirection.• ERROR — Generated when the Windows API function fails.• DEBUG — Contains details about how the compatibility package engine is running.
CATEGORY	The area to which each API belongs. For example, <code>File</code> or <code>Registry</code> . Categories also include subcategories, such as <code>Redirected</code> or <code>NotRedirected</code>

Column	Description
FUNCTION-LINE	The function that generates a log event message. This will often be a Windows API, although internal compatibility package engine functions generate their own log events.
FROM	Typically, the from/original name. For example, the filename, registry key, and so forth
TO	Typically, the virtualized name (filename, registry). It can also be empty to indicate that the original name is the same. Note For application logging, the FROM/TO fields are populated when virtualizing from one location to another. However, FROM can be used only to record the details of an API accessing a location that is not virtualized (for example, registry access to a non-virtualized path). The FROM/TO fields can also be empty when a location or object isn't virtualized, but for which a log event should be recorded (the message will contain details about the event). For example, the NotWow64Process virtualizes the Windows API PrintDlg, and the information about this is recorded
MESSAGE	Additional information related to the function that generates the log event (for example, other parameters). This means that the information is different for each function. For example, for FILE APIs, the information could be the desired access. For COM, it could be the class registration.
ERROR (CODE)	When an error occurs, the generated error code is displayed.
MESSAGE	The error message associated with the error code.

Enable compatibility package engine logging

1. Navigate to the package folder.
2. Open `Compatibility.Package.Engine.clc` in a text editor, such as Notepad++.
3. Create the tag using the following command and set the value for `Log` to one of the following, depending on the level of logging required.
 - **Off** — The default. Logging is disabled.
 - **On** — Only high-level APIs are logged. There is no nesting of log entries/logging of any lower-level API calls. For example, when you create a file, the `CreateFile` API is called. This, in turn, calls the lower level API `NTCreateFile`. When logging is set to `On`, you will not see entries for the lower-level APIs called, such as `NTCreateFile`.
 - **Verbose** — All entries, including both high-level and low-level API calls.

```
<AAV Log="<value>"
```

4. To customize the location where the log files are created, use an additional `LogPath` tag that points to the location where you want the log files to be created.

The following example enables log files to be created in the `C:\Temp` location.

```
<AAV Log="On" LogPath="C:\Temp">
```

Managing EMP custom configurations

You can add custom configurations to your EMP package using the `DeploymentScript.xml` file that is provided in the root folder of each created package. Each application works uniquely and implementation for the same application can work differently across environments.

The `DeploymentScript.xml` file can handle different methods for running Windows. This includes running scripted installations and Windows installer files, or it can use tools such as `CMD.exe`, `PowerShell.exe`, and `WSH` to achieve specific results.

The following example shows the structure of the `DeploymentScript.xml`.

```
<Deployment>
  <Install>
    <!--
      <Programs>
        <Program Order="0">
          <ProcessWindowStyle>Normal</ProcessWindowStyle>
          <Path>Program2</Path>
          <Args>arg2</Args>
          <WaitCondition TimeoutInSeconds="0">None</WaitCondition>
        </Program>
      </Programs>-->
    </Install>
    <Uninstall>
      <!--
        <Programs>
          <Program Order="0">
            <ProcessWindowStyle>Normal</ProcessWindowStyle>
            <Path>Program2</Path>
            <Args>arg2</Args>
            <WaitCondition TimeoutInSeconds="0">None</WaitCondition>
          </Program>
        </Programs>-->
      </Uninstall>
    </Deployment>
```

DeploymentScript.xml element and attribute descriptions:

- **Install** — Specifies what to run during the package deployment phase.
- **Programs** — Specifies the number of tasks to run.
- **Program** — Specifies the details of the task to run. To run more than one task, copy and paste the entire `Program` element and tailor it to the new action.
- **Order** — Specifies the sequence of the tasks as 0, 1, 2, and so on. The current release includes only the `Order` attribute.

- **PreInstall** and **PostInstall**— These attributes are not added by default. You can add them as needed.
- By default, the programs are run as `PostInstall` tasks. This means that they are run towards the end of the EMP package deployment, after the installation of services. The `PostInstall` attribute can be explicitly set as follows.

```
<Program Order="0" PostInstall="true">
```

Example scenario: deploy drivers silently as part of the EMP package deployment — The drivers will be deployed after the services have been installed and started, towards the end of the package deployment process. Tasks that install service dependencies must not be run as `PostInstall` tasks.

- You can run a `PreInstall` task by adding a `PreInstall="true"` attribute and value as follows. This runs the program before the services are installed.

```
<Program Order="0" PreInstall="true">
```

Example scenario: an application's Windows service requires a dependency in Windows features to be enabled on the server on which it is deployed before it can start — A `PreInstall` script can be used to enable the Windows feature so that it is set up and configured as part of the EMP deployment process before the service is started.

- **ProcessWindowStyle** — This attribute can be set to either `Normal` or `Hidden` depending on the requirements for the visibility of the process run.
- **Path** — The path to the program to be run.
- **Args** — This attribute allows for the passing of arguments to the program that is to be run.
- **WaitCondition** — The `TimeoutInSeconds` attribute can be set with either a `None` or `Exit` condition in the `<WaitConditions>` tag. A program configured with a timeout will display an error to the user if the process fails to close within the specified time. If the timeout is set to 0, it waits indefinitely.

If a dependency is installed prior to a service starting, you may need to add an `Exit` condition to ensure that the process completes and exits before `Compatibility.Package.Deployment.exe` moves to the next task, or commences services installation:

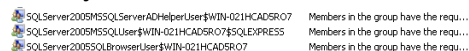
```
<WaitCondition TimeoutInSeconds="0">Exit</WaitCondition>
```

- **Uninstall** — Specifies what to run during the uninstall phase of an EMP package.

Example scenario: migrate local users and groups

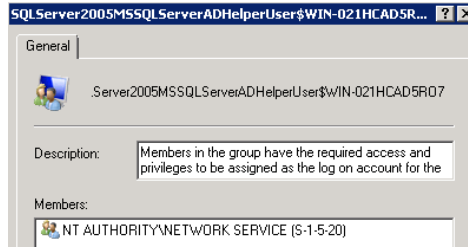
Applications that use local users and groups can be migrated by using the `DeploymentScript.xml` program tasks. Identify the local users and groups to be migrated from the legacy source machine. Take note of the users that have been added to these groups. Create a script that creates these local users and groups on the target system.

In this scenario, SQL Server 2005 Express creates the following local users and groups when installed natively on Windows Server 2008 R2.



SQLServer2005MSSQLServerADHelperUser\$WIN-021HCAD5R07	Members in the group have the requ...
SQLServer2005MSSQLUser\$WIN-021HCAD5R07\$SQLEXPRES	Members in the group have the requ...
SQLServer2005SQLBrowserUser\$WIN-021HCAD5R07	Members in the group have the requ...

The Network Service account is added to these groups.



The following PowerShell script creates these local users and groups. It is stored in a file called `CreateLocalUsersGroups.ps1`.

```
$groups = New-Object 'system.collections.generic.dictionary[string,string]'

$groups.Add('SQLServer2005MSSQLServerADHelperUser$' + $ENV:COMPUTERNAME, "Members in the
group have the required access and privileges to be assigned as the log on account for the
associated instance of SQL Server Active Directory Helper in SQL Server 2005.")

$groups.Add('SQLServer2005MSSQLUser$' + $ENV:COMPUTERNAME, "Members in the group have the
required access and privileges to be assigned as the log on account for the associated
instance of SQL Server and SQL Server FullText Search in SQL Server 2005.")

$groups.Add('SQLServer2005SQLBrowserUser$' + $ENV:COMPUTERNAME, "Members in the group have
the required access and privileges to be assigned as the log on account for the associated
instance of SQL Server Browser in SQL Server 2005.")

ForEach ($group in $groups.GetEnumerator())
{
    $newGroup = New-LocalGroup -Name $group.Key -Description ($group.Value.Substring(0,44) +
'...')
    Add-LocalGroupMember -Group $newGroup -Member "S-1-5-20"
}
```

To migrate local users and groups, add this file to the root of the EMP package and update the `DeploymentScript.xml` as follows.

```
<Install>
  </Program>
  <Program Order="0" PreInstall="true">
    <ProcessWindowStyle>Hidden</ProcessWindowStyle>
    <Path>Powershell.exe</Path>
    <Args>"%DefaultDir%\CreateLocalUsersGroups.ps1"</Args>
    <WorkingDirectory>%DefaultDir%</WorkingDirectory>
    <WaitCondition TimeoutInSeconds="0">Exit</WaitCondition>
  </Program>
</Install>
```

Link EMP packages

This topic contains an example `Redirections.xml` file to show you how to link EMP packages so that they can interact with each other. One scenario for which this process is useful is if an application requires access to files or registry keys from a different EMP package, for example, if you packaged a dependency in a separate EMP package and the main application requires access to the dependency application to function as required.

You can link packages by loading the `Redirections.xml` file from one package to another using the `<Includes>` tag in the EMP compatibility engine config file (`Compatibility.Package.Engine.clc`).

Example

In the following example, an `<Include>` tag entry has been added (C:\DeployDirForPackage2\ExamplePackage2\Redirections.xml). When a process from ExamplePackage1 starts, the engine that starts the process will load the Redirections.xml file from both ExamplePackage1 and ExamplePackage2.

```
<AAV PackageId="ExamplePackage1">
<Includes>
  <Include>Redirections.xml</Include>
  <Include>C:\DeployDirForPackage2\ExamplePackage2\Redirections.xml</Include>
</Includes>
```

When both Redirections.xml files are loaded, the process from ExamplePackage1 follows all of the redirections from ExamplePackage2. In this example, we could add an `<Include>` entry for the Redirections.xml for ExamplePackage1 in the Compatibility.Package.Engine.clc of ExamplePackage2 so that processes from either package can access each other.

In this example, a relative path is used. If ExamplePackage1 and ExamplePackage2 are deployed into the same DeployDir, you can enter the path as follows: `<Include>..\ExamplePackage2\Redirections.xml</Include>`.

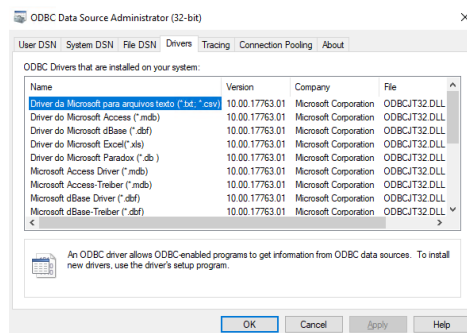
Applications using ODBC drivers

Migrate applications that require Open Database Connectivity (ODBC) drivers.

Microsoft ODBC drivers

Windows Server 2016 and Windows Server 2019 include several Microsoft ODBC drivers installed as part of the operating system. We recommend that you use these versions in the package instead of earlier versions.

To verify whether an ODBC driver is present on the target operating system, open the **Drivers** tab of the ODBC Data Source Administrator application. Note that there are separate 32-bit and 64-bit versions of the ODBC data sources and drivers.



You can also query for the installed ODBC drivers using PowerShell by running the `Get-ODBCDriver cmdlet`.

Third-party ODBC drivers

Applications that require ODBC drivers that are not included by default in Windows operating systems must be included within the EMP package so that the application has everything it requires to run. Applications that install third-party ODBC drivers as part of the application install process will include the ODBC drivers in the EMP package as part of the install capture process. Applications that require

ODBC drivers to be manually installed, either before or after the application is installed, must be installed during the install capture process so that they are included in the EMP package.

Enable support for out-of-process Common Object Model (COM) in an EMP package

This topic includes steps to enable support for out-of-process Common Model Objects (COM) in an EMP package. There are two main types of COM servers: in-process and out-of-process. In-process servers are implemented in a dynamic linked library (DLL). Out-of-process servers are implemented in an executable file (EXE).

Out-of-process servers can reside on either the local computer or a remote computer. In addition, COM provides a mechanism that allows an in-process server (DLL) to run in a surrogate EXE process to run the process on a remote computer. For more information about how to create DLL servers that can be loaded into a surrogate EXE process, see the Microsoft documentation at [DLL Surrogates](#).

Applications with components that run out-of-process require the `COMVirtualization` feature to be enabled to virtualize COM. Process Monitor (procmon) can be used to detect these components by monitoring `SVCHOST.exe`. Also, `APIMON` can be used, filtering for `CoCreateInstance`.

A failure results in either `CLSID not found` or `Component not registered` when the COM subsystem does not run within the virtual environment of the package.

Note

The `COMVirtualization` feature is not required if the application uses in-process COM objects. Applications that use in-process COM objects behave as expected without enabling the feature.

Enable the `COMVirtualization` feature

1. Navigate to the package folder of the application.
2. Edit the `Compatibility.Package.Engine.clc` in a text editor, such as Notepad++.
3. Enable the `COMVirtualization` feature by moving the feature tag out of the section of tags that are commented out.

```
<Features>
  <Feature>COMVirtualization</Feature>
</Features>
```

For a 64-bit package, add the `COMVirtualization` tag to the automatically added `<Features>` element that contains the `RedirectX64PackageRegistry` feature. If two `<Features>` elements are used in the CLC file, an error will result when you attempt to run the package.

```
<AAV>
  <Features>
    <Feature>RedirectX64PackagedRegistry</Feature>
    <Feature>COMVirtualization</Feature>
  </Features>
</AAV>
```

Debugging COM virtualization

- COM messages are written to the engine logs. Search for `COM` to find them.

- COM error messages are written to the engine logs. Search for `ERROR`, `COM` to find them.
- If COM instance creation fails, the engine log message will contain `Failed to create COM instance`.
- Resolving or troubleshooting problems with COM Virtualization may require temporary or permanent exclusion of CLSIDs from the process.

To exclude CLSIDs from COM virtualization

1. In the `Compatibility.Package.Engine.clc` configuration file, locate and edit the `<Features>` tag, which is used to enable COMVirtualization.

```
<AAV>
  <Features>
    <Feature>ComVirtualization</Feature>
  </Features>
</AAV>
```

2. To exclude CLSID `EF66E233-9F07-4E32-9119-FF40CDDD4DCF`, insert the following `<COM>` tag code outside of the `<Features>` tag to specify the CLSID you want to exclude.

```
<AAV>
  <Features>
    <Feature>ComVirtualization</Feature>
  </Features>
  <COM>
    <CLSID ID="{EF66E233-9F07-4E32-9119-FF40CDDD4DCF}" Excluded="true">
    </CLSID>
  </COM>
</AAV>
```

To include the CLSID, change the value of `Excluded` to `= "false"`, or remove the `<COM>` tags.

Add side-by-side (SXS) assemblies to an EMP compatibility package

A Windows side-by-side assembly is described by [manifests](#). A side-by-side assembly contains a collection of resources — a group of DLLs, Windows classes, COM servers, type libraries, or interfaces — that are always provided to applications together. These resources are described in the assembly manifest.

Typically, a side-by-side assembly is a single DLL. For example, the Microsoft COMCTL32 assembly is a single DLL with a manifest, whereas the Microsoft Visual C++ development system run-time libraries assembly contains multiple files.

Manifests contain [metadata](#) that describes side-by-side assemblies and side-by-side assembly dependencies. Side-by-side assemblies are used by the operating system as fundamental units of naming, binding, versioning, deployment, and configuration.

Every side-by-side assembly has a unique identity. One of the attributes of the assembly identity is its version. For more information, see [Assembly Versions](#) in the Microsoft documentation.

The EMP Compatibility Package Builder does not configure support for side-by-side assemblies. If your application uses side-by-side assemblies that are distributed with installers, you must first attempt to install them natively on the target operating system. If this is successful, you can script the installation of

these assemblies into package deployment using the `DeploymentScript.xml`. For more information see [Managing EMP custom configurations \(p. 54\)](#). If native installation is not possible or does not address the issue, you can add the assemblies to the package as private assemblies.

The following error message will appear in the Windows Event Viewer, from which you can determine what assemblies you need, along with the version:

```
Activation context generation failed for "DemoApplication.exe". Dependent
AssemblyTowersWatson.Components.Licensing.ComSRM,publicKeyToken="97c62a3c455f5e0d",type="win32",version
not be found. Please use sxstrace.exe for detailed diagnosis.
```

Add side-by-side assemblies to an EMP package

1. Add the missing DLLs associated with the run-times to the same folder as the executable being run. Alternatively, use Process Monitor to monitor the application process along with `csrss.exe` to determine where the application is looking for these DLLs.
2. Locate the DLLs (usually located in `C:\Windows\Assembly` or `C:\Windows\WinSXS` folders).

Exclude or detach a process from the package redirection rules

This topic explains how to use the `Excludes` or `Detaches` features in the package redirection rules.

Excludes feature

We recommend the `Excludes` feature method to control the behavior of the package redirection and virtualization engine. It prevents child processes from being virtualized and redirected without affecting the virtualization of the parent processes.

This feature is useful when the packaged application is spawning local processes from locally installed applications that you do not want redirected. This is because the redirections change the behavior of the child processes.

Exclude a process from the package redirection rules

1. Open `Compatibility.Package.Engine.clc`.
2. Add the executable names to the `Exclude` tags.

```
<Excludes>
<Exclude>Excel.exe</Exclude>
<Exclude>Winword.exe</Exclude>
</Excludes>
```

Detaches feature

You can control the behavior of the virtualization and redirection engine using the `Detaches` feature. After an application has started, the virtualization engine exits when the initial create process function completes. All redirections and most virtualization features will be available to the parent process that started.

The `Detaches` feature is useful when child processes do not exit cleanly, and terminate with an exception.

Important

The virtualization and redirection engine of the package will detach from the parent process once virtualization is complete. Any child processes will not be virtualized, and the detached process will not benefit from `DEPOptOut` or `HandleInvalidHandle`, even if these features are enabled.

Detach a process from the package redirection rules

1. Open `Compatibility.Package.Engine.clc`.
2. Add the executable names to the `Detach` tags.

```
<Detaches>  
<Detach>Excel.exe</Detach>  
<Detach>Winword.exe</Detach>  
</Detaches>
```

Run cmd.exe as a child process to the EMP compatibility package engine

This topic explains how to test or troubleshoot an EMP package by launching `cmd.exe` as a child process to the EMP package engine. Doing this launches `cmd.exe` in the context of the EMP package to quickly run tests, validate application errors, and retest functionality.

Run cmd.exe as a child process

1. Deploy the EMP package following the usual deployment process.
2. Open a new `cmd.exe` window and navigate to the root of the EMP package. Run the following command.

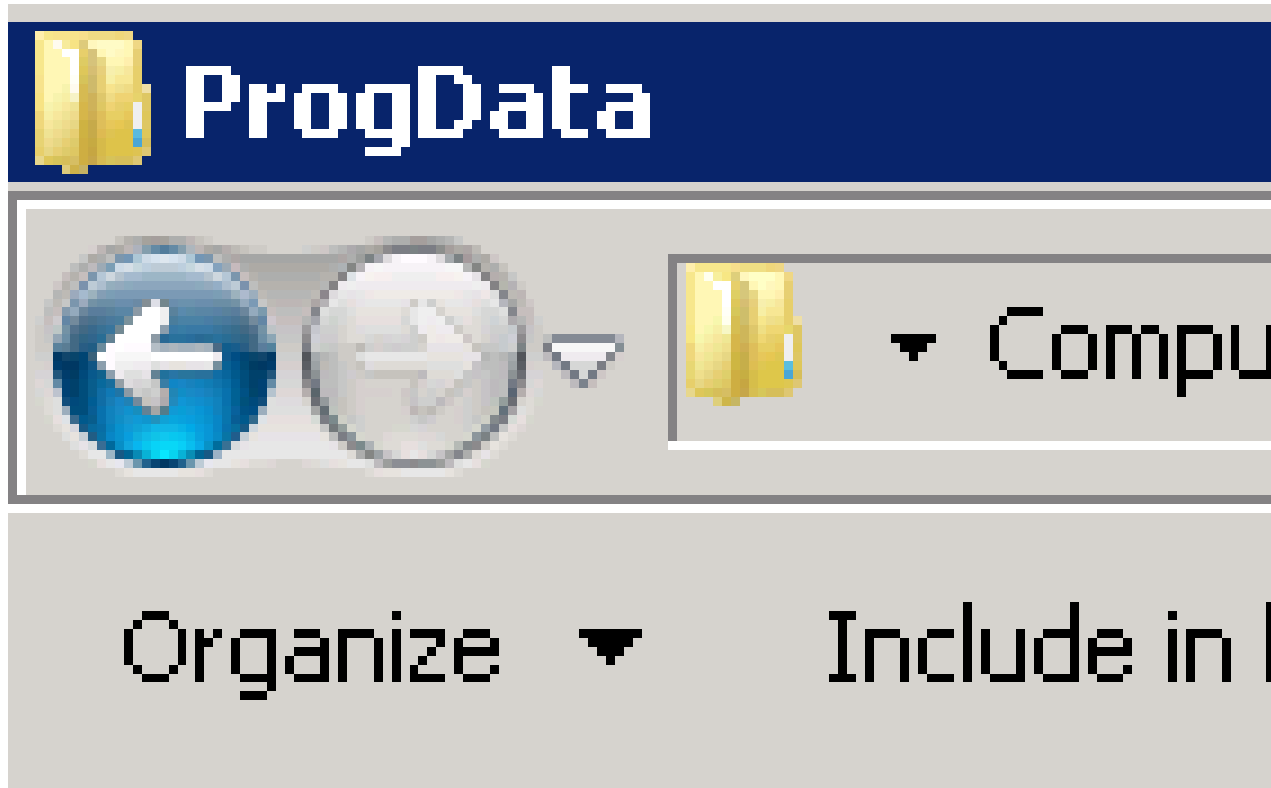
```
<pathtopackage>\Compatibility.Package.Engine.exe /F cmd.exe
```

This command launches the engine and runs the `cmd.exe` as a child process in the context of an EMP package. A new command window will open. All redirection rules and package configurations now apply to the new `cmd.exe` process. You can verify this using Process Explorer and looking for the `cmd.exe` under the parent `cmd.exe`.

Example: Verify that application can access its main installation folder


In this troubleshooting scenario, we want to verify that the `LegacyApp` application packaged in EMP can access its main installation folder, `LegacyAppFolder`.

1. Check the package to confirm that `LegacyAppFolder` exists.



Favorites

 Desktop

 Downloads

 Recent Places

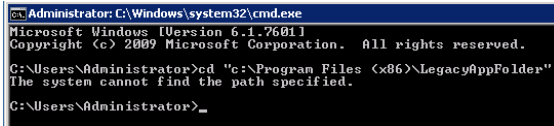
2. A corresponding `<FolderMatch>` redirection should exist in the `Redirections.xml` file of the package.

```
<FileSystem>
  <FolderMatch>
    <From>%ProgramFilesX86%\LegacyAppFolder</From>
```

AWS End-of-Support Migration Program
(EMP) for Windows Server User Guide
Run cmd.exe as a child process

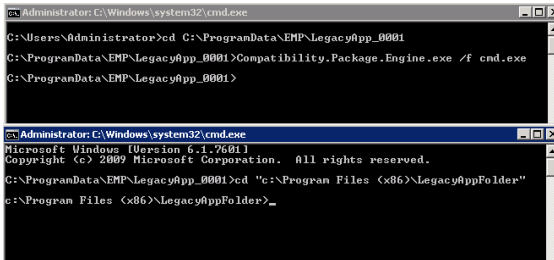
```
<To>ProgData\LegacyAppFolder</To>  
</FolderMatch>
```

3. Use cmd.exe to check for the file in the local system. The system should not be able to find the file.



```
Administrator: C:\Windows\system32\cmd.exe  
Microsoft Windows [Version 6.1.7601]  
Copyright (c) 2009 Microsoft Corporation. All rights reserved.  
C:\Users\Administrator>cd "c:\Program Files (x86)\LegacyAppFolder"  
The system cannot find the path specified.  
C:\Users\Administrator>_
```

4. Run cmd.exe as a child process to the package engine. This should confirm that the LegacyAppFolder is available in the context of the EMP package.



```
Administrator: C:\Windows\system32\cmd.exe  
C:\Users\Administrator>cd C:\ProgramData\EMP\LegacyApp_0001  
C:\ProgramData\EMP\LegacyApp_0001>Compatibility.Package.Engine.exe /f cmd.exe  
C:\ProgramData\EMP\LegacyApp_0001>
```

```
Administrator: C:\Windows\system32\cmd.exe  
Microsoft Windows [Version 6.1.7601]  
Copyright (c) 2009 Microsoft Corporation. All rights reserved.  
C:\ProgramData\EMP\LegacyApp_0001>cd "c:\Program Files (x86)\LegacyAppFolder"  
c:\Program Files (x86)\LegacyAppFolder>_
```

Security in AWS End-of-Support Migration Program (EMP) for Windows Server

Cloud security at AWS is the highest priority. As an AWS customer, you benefit from data centers and network architectures that are built to meet the requirements of the most security-sensitive organizations.

Security is a shared responsibility between AWS and you. The [shared responsibility model](#) describes this as security *of* the cloud and security *in* the cloud:

- **Security of the cloud** – AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services that you can use securely. Third-party auditors regularly test and verify the effectiveness of our security as part of the [AWS Compliance Programs](#). To learn about the compliance programs that apply to Porting Assistant for .NET, see [AWS Services in Scope by Compliance Program](#).
- **Security in the cloud** – Your responsibility is determined by the AWS service that you use. You are also responsible for other factors including the sensitivity of your data, your company's requirements, and applicable laws and regulations.

This documentation helps you understand how to apply the shared responsibility model when using AWS End-of-Support Migration Program (EMP) for Windows Server. The following list contains information about how AWS End-of-Support Migration Program (EMP) for Windows Server helps you to meet your security and compliance objectives.

- **Migrates legacy applications from insecure EOS Windows versions to the latest secure versions.** The EMP application packaging process captures only the legacy application components into an EMP package, not the legacy operating system components. The package can then be deployed and run in isolation, and benefits from the enhanced security and performance of the modern operating system. The modern operating system, in turn, runs on modern hardware, which also mitigates against hardware-based vulnerabilities. Because an EMP package is no longer dependent on the underlying operating system, it can run on a later version of Windows, including future semi-annual Windows releases. This ensures that you can take advantage of future security benefits and always run on a supported Windows version.
- **Application security updates.** Applications contained in an EMP package can be patched with security updates provided by the application vendors by using the [EMP Compatibility Package Editor \(p. 38\)](#). We recommend that you maintain packaged applications with their latest security updates to keep them as secure as possible in the AWS Cloud.
- **Application security vulnerabilities.** EMP does not resolve inherent security vulnerabilities in legacy applications. Because vulnerable applications benefit from the better security features of modern Windows servers, EMP reduces their exposure to exploitation. Regularly updating EMP-packaged applications with security updates where available is recommended to counter security vulnerabilities.

Data collected by the AWS End-of-Support Migration Program (EMP) for Windows Server

AWS collects usage information through the EMP telemetry module during the deployment and subsequent use of EMP packages. Collection of the telemetry collected during deployment is mandatory, whereas the runtime (usage) telemetry is optional. You can opt out of sending runtime telemetry when you install the EMP Compatibility Package Builder. The telemetry module sends the collected data to an application modernization metrics service running on AWS.

The following mandatory telemetry metrics are collected by AWS:

- AWS account ID
- Windows operating system version
- Infrastructure (AWS or not AWS)
- Hashed package ID

The following optional runtime telemetry metrics are collected by AWS:

- Windows operating system, for example, `Windows Server 2008`.
- Windows operating system architecture, for example, `x86`.
- Instance type. The instance type is the machine type on which the application is deployed. It can be either an Amazon EC2 instance or an on-premises machine.
- EMP compatibility engine version, for example, `1.0.0.4`.
- EMP compatibility engine architecture, for example `x86`.
- Virtualized executable name, for example, `Notepad++`.
- Virtualized Application Architecture, for example, `x86`.
- Namespace to identify whether the package run is a development or production run.

Troubleshooting AWS End-of-Support Migration Program (EMP) for Windows Server

When troubleshooting AWS End-of-Support Migration Program (EMP) for Windows Server issues, we recommend that you apply the following methodology. There are four major types of issues related to EMP packaging.

- **Environment** — The target system is running in production and may be subject to more stringent restrictions than the test server. Common examples include more restrictive group policies, antivirus restrictions, enabled controlled folders, and User Access Control restrictions.
- **Legacy application** — It is not uncommon to find pre-existing issues in the application that exist before packaging. For example, if a function fails in the packaged application you should test the functionality on the natively installed application on the legacy server.
- **Packaging** — These are issues related to the capture process and primarily an incomplete test plan. A common example would be functionality that should have been run during the capture, but was not included and therefore system calls to the registry or files was not captured.
- **EMP packaging tools** — These are issues related to the EMP product. If you suspect product issues, create a ticket so the product team can look into the issue.

To find the root cause of an issue, we recommend the following workflow, where you begin testing on the legacy operating system and then move towards testing on the target operating system.

1. Test the package on a clean machine running the source operating system and where the application has never been installed. A clean testing machine is a server instance of the Windows operating system version with as little else as possible installed on the machine.
2. Test the package on a machine running the target operating system where no other applications or policies are installed.
3. Test the package on a machine running the target operating system configured with other applications and policies that will exist in the target environment.

This testing process allows you to identify the step during which the package is failing.

When the package fails during testing, we recommend trying the following tools to remediate.

- **Check if any error messages appear when the application fails** — An application failure typically results in an error message. The error message may be from Windows and provide details about what has gone wrong, or it may provide a Windows error code, which can help to diagnose the problem. The message can also be from the application process that encountered the problem. This indicates that the process was successfully started then encountered a problem. Application error codes and other messages indicate that the error has been logged and assist with problem diagnoses.
- **Check the Windows Event Viewer for application errors** — Application errors are not always logged in Windows Event Logs; however, we recommend that you check them for useful diagnostic information.
- **Use the Sysinternals tool, Process Explorer** — Use Process Explorer to check the command line and environment variables used to load the application process.

1. Download Process Explorer from <https://docs.microsoft.com/en-us/sysinternals/downloads/process-explorer>.
 2. Launch your EMP packaged application or reproduce the error you want to investigate.
 3. Launch Process Explorer and look for the process you want to investigate.
 4. Open (double-click) the process that you want to investigate to open the **Properties** window.
 5. View the command line on the **Image** tab of the **Properties** window, in the **Command line** box.
 6. In the **Properties** window, choose the **Environment** tab to view the environment variables that are available to the process.
- **Use the Sysinternals tool, Process Monitor (procmon)** — Check Process Monitor to see if the application writes any log files to the drive to investigate. Also use Process Monitor to check if there are any registry keys or files the application is not able to find (the result returned for this scenario is `PATH NOT FOUND`). Or check to see if the application is failing to read or write from registry keys or files even when found (the returned result for this scenario is `ACCESS DENIED`). These results can be compared to the results obtained from a procmon log of a working installation of the application, which should show as `SUCCESS` for these operations.
 - **Enable the compatibility package engine logs**
 - To enable package engine logging, open the `Compatibility.Package.Engine.clc` file in a text editor, such as Notepad ++.
 - By default, the first line will contain the following: `<AAV PackageId="PackageID">`. For example, `AAV PackageId="SQL2005EXPSP4_8959">`.
 - Add the string `Log="on"` to the line so that it reads `<AAV PackageId="PackageID" Log="on">`. For example, `AAV PackageId="SQL2005EXPSP4_8959" Log="on">`.

AWS End-of-Support Migration Program (EMP) for Windows Server version history

The following table describes the released versions of AWS End-of-Support Migration Program (EMP) for Windows Server Compatibility Package Builder.

New releases of the AWS End-of-Support Migration Program (EMP) for Windows Server Compatibility Package Builder are provided in MSI format. To upgrade to a new version from a previous version, uninstall the previous version by using the **Add or Remove Programs** feature from legacy Windows operating systems, or from **Programs and Features** in the **Control Panel** for later operating systems. Then, reinstall the package with the latest MSI.

Version	Details	Release date
1.0.0	Initial release	October 15, 2020

Document History for User Guide

The following table describes the documentation for this release of AWS End-of-Support Migration Program (EMP) for Windows Server.

- **Latest documentation update:** October 15, 2020

update-history-change	update-history-description	update-history-date
Initial release (p. 69)	Initial release of the AWS End-of-Support Migration Program (EMP) for Windows Server User Guide.	October 15, 2020