***Assignment 8***

***Name- Harsh Kumar***

***Email-*** [***hs150harsh@gmail.com***](mailto:hs150harsh@gmail.com)

**Process Check and Deletion in PSAppDeployToolkit**

**Purpose**

This script demonstrates how to:

* Check if a specific process is running
* Stop (delete) the process if found
* Log the outcome using PowerShell and optionally PSAppDeployToolkit

**Key Cmdlets and Toolkit Functions**

**Get-Process**

* Retrieves process objects by name or ID.
* Example:
* Get-Process -Name "notepad.exe"
* Get-Process -Id 1234

**Stop-Process**

* Terminates a running process by name or ID.
* Example:
* Stop-Process -Name "chrome.exe" -Force
* Stop-Process -Id 1234

**Conditional Check**

* Verifies if a process is running before attempting to stop it:

if (Get-Process -Name "myProcess.exe") {

Stop-Process -Name "myProcess.exe" -NoConfirm

Write-Host "Process myProcess.exe has been stopped"

} else {

Write-Host "Process myProcess.exe was not running"

}

**⚙️ Toolkit Enhancements**

**🔹 Execute-Process**

* Launches or interacts with processes in a more controlled way.
* Supports logging, error handling, and return code capture.

**🔹 Get-PendingReboot**

* Checks if a system reboot is pending—useful before or after stopping critical processes.

**⚠️ Best Practices & Considerations**

* **Permissions**: Stopping processes may require admin rights.
* **Security**: Avoid terminating system-critical or user-sensitive processes.
* **Confirmation**: Use -NoConfirm to suppress prompts, especially in silent deployments.
* **Error Handling**: Wrap in Try-Catch blocks for robust logging and fallback logic.

**📝 Suggested Logging Integration**

If using PSAppDeployToolkit:

Write-Log -Message "Checking for process: myProcess.exe"

if (Get-Process -Name "myProcess.exe" -ErrorAction SilentlyContinue) {

Write-Log -Message "Process found. Attempting to stop."

Stop-Process -Name "myProcess.exe" -Force

Write-Log -Message "Process myProcess.exe stopped successfully."

} else {

Write-Log -Message "Process myProcess.exe not running."

}

**Set INI value in PSADT**

**Set-ADTIniValue in PSAppDeployToolkit**

**Purpose**

Sets or updates a value in a specific section/key of an INI file.

**Syntax**

Set-ADTIniValue -FilePath "<path to ini file>" -Section "<section name>" -Key "<key name>" -Value "<value>"

**✅ Example**

Set-ADTIniValue -FilePath "C:\Windows\Path\to\my.ini" `

-Section "MySection" `

-Key "MyKey" `

-Value "MyValue"

**📂 Result**

Updates MyKey=MyValue under [MySection] in the specified INI file.

**⚠️ Notes**

* INI file must exist; function does not create it.
* Automatically handles escaping and formatting.
* Useful for configuring legacy apps or installers.

**PSAppDeployToolkit Logging Location**

**📂 Default Log Path**

C:\Windows\Logs\Software

**⚙️ Configuration File**

* Located in: .\Config\Config.psd1
* Property to modify: LogPath

**🛠️ Customizing Log Path**

To change the log location, update the LogPath value in Config.psd1:

LogPath = 'C:\MyLogFolder'

**📄 Log File Naming Convention**

Typical format:

PSAppDeployToolkit\_<Version>\_<Lang>\_<Sequence>\_PSAppDeployToolkit\_<Action>.log

Example:

PSAppDeployToolkit\_4.0.6\_EN\_01\_PSAppDeployToolkit\_Install.log

**📌 Notes**

* Log files are auto-generated per deployment session.
* Useful for troubleshooting, auditing, and rollback analysis.
* Ensure the custom path has write permissions for the executing user.

**📝 Quick Note: MSI/MSP Logging in PSAppDeployToolkit**

**🧠 Purpose**

Enable detailed logging for MSI installations, MSP patches, and uninstallations using Execute-MSI.

**🔧 Key Parameters**

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| -LogName | Sets the log file name (auto-appends .log if missing) |
| -LogPath | Sets custom log file location (defaults to PSADT log directory) |
| -LogVerbosity | Controls log detail level: Verbose, Informational, Error |
| -ContinueOnError | Determines whether to continue or halt on error ($true / $false) |

**📦 Usage Examples**

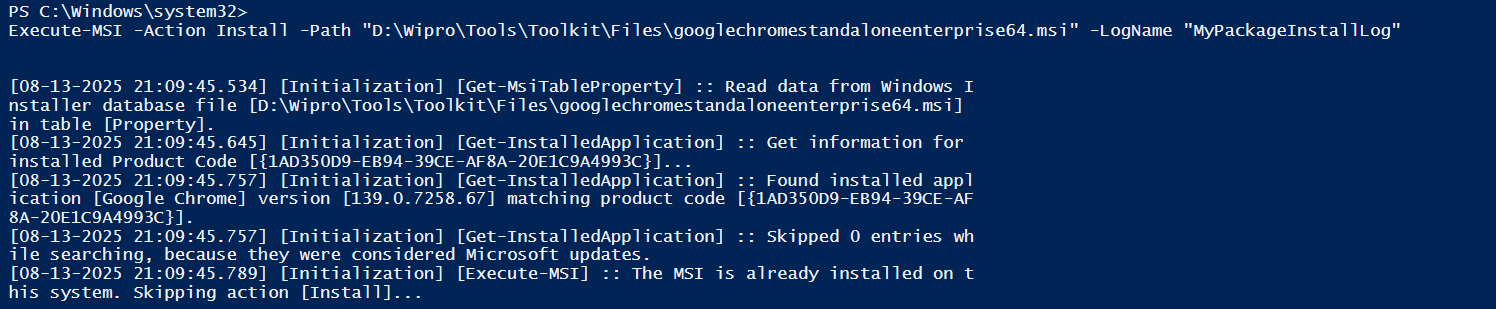
**✅ MSI Installation**

Execute-MSI -Action Install `

-Path "C:\MyApps\MyPackage.msi" `

-LogName "MyPackageInstallLog" `

-LogVerbosity Verbose

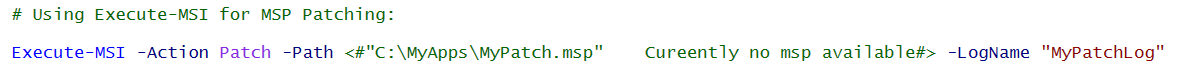


**✅ MSP Patch**

Execute-MSI -Action Patch `

-Path "C:\MyApps\MyPatch.msp" `

-LogName "MyPatchLog"



**✅ MSI Uninstall**

Execute-MSI -Action Uninstall `

-Path "[PRODUCTCODE]" `

-LogName "MyPackageUninstallLog"

A blue screen with red and white text

AI-generated content may be incorrect.

**⚙️ Custom Log Location**

**🔹 Option 1: Set in AppDeployToolkitConfig.xml**

<configToolkitLogDir>C:\CustomLogFolder</configToolkitLogDir>

**🔹 Option 2: Set early in Deploy-Application.ps1**

$configToolkitLogDir = "C:\CustomLogFolder"

**⚠️ Considerations**

* **Default Location**: Logs are saved under the PSADT working directory unless overridden.
* **Log Level**: Use Verbose for full diagnostics; Error for minimal logging.
* **Error Handling**: Combine logging with -ContinueOnError to control script behavior on failure.
* **Network Logging**: For enterprise deployments, consider logging to a network share for centralized access.

**📝 Logging Individual Commands in PSAppDeployToolkit**

**🧠 Purpose**

Track specific actions, command executions, and status updates within your deployment script using Write-Log.

**🔧 Core Logging Function**

**✅ Write-Log**

Logs custom messages to the main PSADT log file.

**🔹 Examples**

Write-Log "Starting the installation of Application X."

Execute-Process -Path "msiexec.exe" -Parameters "/i application.msi /qn"

Write-Log "MSI installation command executed."

If (Test-Path -Path "C:\Program Files\Application X") {

Write-Log "Application X installation successful."

} Else {

Write-Log -Message "Application X installation failed." -Severity "Error"

}

* Use -Severity "Error" or "Warning" for critical messages.
* Helps trace script flow and troubleshoot failures.

**📄 Dedicated Log Files for MSI/MSP**

**✅ Execute-MSI with -LogName**

Creates a separate log file for MSI/MSP operations.

Execute-MSI -Action Install `

-Path "$InstallPath\MyApp.msi" `

-Parameters "/qb" `

-LogName "MyAppInstall.log"

* Saved in default PSADT log directory (e.g., C:\Windows\Logs\Software).
* Useful for isolating installer logs from toolkit logs.

**🧾 PowerShell Transcript Logging**

**✅ Start-Transcript / Stop-Transcript**

Captures all console output to a separate .txt file.

Start-Transcript -Path "C:\Logs\Transcript.txt"

# Run custom PowerShell commands

Write-Output "Running custom registry cleanup..."

Stop-Transcript

* Ideal for debugging non-PSADT commands.
* Use alongside Write-Log for full visibility.

**📌 Best Practices**

* Use Write-Log generously for traceability.
* Log before and after critical operations.
* Include status checks and error messages.
* Use separate MSI logs for installer-level diagnostics.
* Combine transcript logging for deep PowerShell debugging.

**📝 Catching Return Codes in PowerShell**

**🎯 Purpose**

Track success/failure of commands and executables using exit codes and structured error handling.

**🔢 1. $LASTEXITCODE – External Executables**

Start-Process "my\_program.exe" -Wait

$exitCode = $LASTEXITCODE

if ($exitCode -ne 0) {

Write-Host "Error: my\_program.exe failed with exit code $($exitCode)"

}

**📌 Notes**

* Captures exit code from .exe, .bat, or other non-PowerShell commands.
* 0 = success, non-zero = failure or warning.
* Not applicable to native PowerShell cmdlets.

**🛡 2. try...catch – PowerShell Error Handling**

try {

Get-Content "nonexistent\_file.txt"

}

catch {

Write-Host "An error occurred: $($\_.Exception.Message)"

}

**🔹 With $ErrorActionPreference**

$ErrorActionPreference = "Stop"

try {

Get-ChildItem "C:\nonexistent\_folder"

}

catch {

Write-Host "Error: $($\_.Exception.Message)"

}

**📌 Notes**

* Catches terminating errors.
* Use "Stop" to convert non-terminating errors into catchable exceptions.
* Exception object provides detailed error info.

**🚪 3. exit – Custom Script Exit Codes**

if ($someCondition -eq $true) {

exit 1 # Failure

} else {

exit 0 # Success

}

**📌 Notes**

* Sets the script’s exit code for external tools or monitoring systems.
* Can be used in combination with $LASTEXITCODE.

**🧠 Key Takeaways**

|  |  |
| --- | --- |
| **Technique** | **Use Case** |
| $LASTEXITCODE | Track external command results |
| try...catch | Handle PowerShell cmdlet errors |
| exit <code> | Return custom status from script |
| $ErrorActionPreference | Control error behavior globally |