DeviceDecommissionTool - Detailed Function Walkthrough

This document explains every function in the PowerShell script 'DeviceDecommissionTool.ps1' line-by-line. Each function includes the code excerpt followed by an explanation for each line. Keep this guide for developers and advanced helpdesk staff who need to understand the internals.

# How to use this document

Read the high-level overview first, then navigate to any function you need to inspect. For helpdesk, refer to the summary at the end for a short actionable checklist.

# High-Level Overview

The script is a PowerShell WinForms GUI tool that performs the following major tasks:

- Authenticate to Microsoft Graph (supports interactive MFA fallback)  
- Check local prerequisites (modules)  
- Verify Autopilot serial existence and Entra/AD/SCCM device presence  
- Provide modular removal actions with triple confirmation  
- Upload Autopilot HWID CSV (with protections to avoid re-adding existing serials)  
- Detailed logging of actions in the same directory as the script

# Function-by-Function Walkthrough

## Function: Get-ScriptDirectory

Code excerpt:

001: function Get-ScriptDirectory {

-> Function declaration - defines a callable routine.

002: if ($PSScriptRoot) { return $PSScriptRoot }

-> Executes: if ($PSScriptRoot) { return $PSScriptRoot }

003: $inv = $MyInvocation.MyCommand.Definition

-> Assigns a value to a variable.

004: if ($inv) { return (Split-Path -Path $inv -Parent) }

-> Executes: if ($inv) { return (Split-Path -Path $inv -Parent) }

005: return (Get-Location).Path

-> Returns from the function (exits early).

006: }

-> Executes: }

## Function: Log-UI

Code excerpt:

001: function Log-UI {

-> Writes a line to the GUI output textbox for user-visible logging.

002: param([string]$Message, [string]$Status = "INFO")

-> Defines function parameters.

003: $ts = (Get-Date).ToString("yyyy-MM-dd HH:mm:ss")

-> Assigns a value to a variable.

004: $line = "{0} | {1,-6} | {2}" -f $ts, $Status, $Message

-> Assigns a value to a variable.

005: $OutputBox.AppendText("$line`r`n")

-> Executes: $OutputBox.AppendText("$line`r`n")

006: $OutputBox.ScrollToCaret()

-> Executes: $OutputBox.ScrollToCaret()

007: }

-> Executes: }

## Function: Save-ActionLog

Code excerpt:

001: function Save-ActionLog {

-> Saves a full action log to a timestamped file in the script's folder for audit.

002: param(

-> Defines function parameters.

003: [string]$Action,

-> Executes: [string]$Action,

004: [string]$SerialOrDevice,

-> Executes: [string]$SerialOrDevice,

005: [string]$Content

-> Executes: [string]$Content

006: )

-> Executes: )

007: $dir = Get-ScriptDirectory

-> Assigns a value to a variable.

008: if (-not (Test-Path $dir)) { New-Item -Path $dir -ItemType Directory -Force | Out-Null }

-> Executes: if (-not (Test-Path $dir)) { New-Item -Path $dir -ItemType Directory -Force | Out-Null }

009: $stamp = (Get-Date).ToString('yyyyMMdd\_HHmmss')

-> Assigns a value to a variable.

010: $safeName = ($SerialOrDevice -replace '[\\/:\*?"<>| ]','\_')

-> Assigns a value to a variable.

011: $filename = "{0}\_{1}\_{2}.log" -f $stamp, $Action, $safeName

-> Assigns a value to a variable.

012: $full = Join-Path -Path $dir -ChildPath $filename

-> Assigns a value to a variable.

013: $header = "Action: $Action`nTarget: $SerialOrDevice`nTime: $stamp`n`n"

-> Assigns a value to a variable.

014: $Content | Out-File -FilePath $full -Encoding UTF8 -Force

-> Executes: $Content | Out-File -FilePath $full -Encoding UTF8 -Force

015: (Get-Content $full) | Out-File -FilePath $full -Encoding UTF8

-> Executes: (Get-Content $full) | Out-File -FilePath $full -Encoding UTF8

016: Set-Content -Path $full -Value ($header + (Get-Content -Path $full -Raw))

-> Executes: Set-Content -Path $full -Value ($header + (Get-Content -Path $full -Raw))

017: Log-UI "Saved action log to $full" "OK"

-> Writes a line to the GUI output textbox for user-visible logging.

018: }

-> Executes: }

## Function: Confirm-Action

Code excerpt:

001: function Confirm-Action {

-> Function declaration - defines a callable routine.

002: param([string]$Message)

-> Defines function parameters.

003: $result = [System.Windows.Forms.MessageBox]::Show(

-> Displays a dialog box to the user (info/confirm/warning).

004: $Message,

-> Executes: $Message,

005: "Confirm Action",

-> Executes: "Confirm Action",

006: [System.Windows.Forms.MessageBoxButtons]::YesNo,

-> Displays a dialog box to the user (info/confirm/warning).

007: [System.Windows.Forms.MessageBoxIcon]::Warning

-> Displays a dialog box to the user (info/confirm/warning).

008: )

-> Executes: )

009: return $result -eq [System.Windows.Forms.DialogResult]::Yes

-> Returns from the function (exits early).

010: }

-> Executes: }

## Function: Ensure-Connected

Code excerpt:

001: function Ensure-Connected {

-> Function declaration - defines a callable routine.

002: if (-not $global:GraphConnected) {

-> Executes: if (-not $global:GraphConnected) {

003: Log-UI "Not connected to Microsoft Graph. Please Connect/Login first." "FAIL"

-> Writes a line to the GUI output textbox for user-visible logging.

004: return $false

-> Returns from the function (exits early).

005: }

-> Executes: }

006: return $true

-> Returns from the function (exits early).

007: }

-> Executes: }

## Function: Update-ActionButtonsState

Code excerpt:

001: function Update-ActionButtonsState {

-> Function declaration - defines a callable routine.

002: $btnIntune.Enabled = $false

-> Assigns a value to a variable.

003: $btnEntra.Enabled = $false

-> Assigns a value to a variable.

004: $btnAD.Enabled = $false

-> Assigns a value to a variable.

005: $btnSCCM.Enabled = $false

-> Assigns a value to a variable.

006: $btnHWID.Enabled = $false

-> Assigns a value to a variable.

007:

008: if ($global:SerialFound -and -not $global:DeviceFound) { $btnIntune.Enabled = $true }

-> Executes: if ($global:SerialFound -and -not $global:DeviceFound) { $btnIntune.Enabled = $true }

009: if ($global:DeviceFound -and -not $global:SerialFound) {

-> Executes: if ($global:DeviceFound -and -not $global:SerialFound) {

010: if ($global:EntraDevice) { $btnEntra.Enabled = $true }

-> Executes: if ($global:EntraDevice) { $btnEntra.Enabled = $true }

011: if ($global:ADFound) { $btnAD.Enabled = $true }

-> Executes: if ($global:ADFound) { $btnAD.Enabled = $true }

012: if ($global:SCCMFound) { $btnSCCM.Enabled = $true }

-> Executes: if ($global:SCCMFound) { $btnSCCM.Enabled = $true }

013: }

-> Executes: }

014: if ($global:SerialFound -and $global:DeviceFound) {

-> Executes: if ($global:SerialFound -and $global:DeviceFound) {

015: $btnIntune.Enabled = $true

-> Assigns a value to a variable.

016: if ($global:EntraDevice) { $btnEntra.Enabled = $true }

-> Executes: if ($global:EntraDevice) { $btnEntra.Enabled = $true }

017: if ($global:ADFound) { $btnAD.Enabled = $true }

-> Executes: if ($global:ADFound) { $btnAD.Enabled = $true }

018: if ($global:SCCMFound) { $btnSCCM.Enabled = $true }

-> Executes: if ($global:SCCMFound) { $btnSCCM.Enabled = $true }

019: }

-> Executes: }

020: # HWID upload always enabled when connected (module may be required)

-> Comment: HWID upload always enabled when connected (module may be required)

021: if ($global:GraphConnected) { $btnHWID.Enabled = $true }

-> Executes: if ($global:GraphConnected) { $btnHWID.Enabled = $true }

022: }

-> Executes: }

## Function: Connect-ServiceAccount

Code excerpt:

001: function Connect-ServiceAccount {

-> Function declaration - defines a callable routine.

002: try {

-> Start of a try block for error handling.

003: if (-not $UserBox.Text -or -not $PassBox.Text) {

-> Executes: if (-not $UserBox.Text -or -not $PassBox.Text) {

004: [System.Windows.Forms.MessageBox]::Show("Please enter Service Account Username and Password before Connect.","Missing credentials",[System.Windows.Forms.MessageBoxButtons]::OK,[System.Windows.Forms.MessageBoxIcon]::Exclamation)

-> Displays a dialog box to the user (info/confirm/warning).

005: return

-> Returns from the function (exits early).

006: }

-> Executes: }

007: $secure = ConvertTo-SecureString -String $PassBox.Text -AsPlainText -Force

-> Assigns a value to a variable.

008: $global:ServiceCredential = New-Object System.Management.Automation.PSCredential ($UserBox.Text, $secure)

-> Assigns a value to a variable.

009: } catch {

-> Catch block to handle exceptions thrown in the try block.

010: Log-UI "Failed to create PSCredential: $\_" "ERROR"

-> Writes a line to the GUI output textbox for user-visible logging.

011: return

-> Returns from the function (exits early).

012: }

-> Executes: }

013:

014: if (-not (Get-Module -ListAvailable -Name Microsoft.Graph)) {

-> Executes: if (-not (Get-Module -ListAvailable -Name Microsoft.Graph)) {

015: $install = [System.Windows.Forms.MessageBox]::Show("Microsoft.Graph module is not installed. Install from PSGallery now?","Install module",[System.Windows.Forms.MessageBoxButtons]::YesNo,[System.Windows.Forms.MessageBoxIcon]::Question)

-> Displays a dialog box to the user (info/confirm/warning).

016: if ($install -eq [System.Windows.Forms.DialogResult]::Yes) {

-> Executes: if ($install -eq [System.Windows.Forms.DialogResult]::Yes) {

017: try {

-> Start of a try block for error handling.

018: Install-PackageProvider -Name NuGet -MinimumVersion 2.8.5.201 -Force -Scope CurrentUser -ErrorAction SilentlyContinue

-> Ensures NuGet provider is available to install PowerShell modules.

019: Install-Module -Name Microsoft.Graph -Scope CurrentUser -Force -AllowClobber -ErrorAction Stop

-> Installs the Microsoft.Graph module from PSGallery (used if not present).

020: Log-UI "Microsoft.Graph installed." "OK"

-> Writes a line to the GUI output textbox for user-visible logging.

021: } catch {

-> Catch block to handle exceptions thrown in the try block.

022: Log-UI "Could not install Microsoft.Graph automatically: $\_" "ERROR"

-> Writes a line to the GUI output textbox for user-visible logging.

023: [System.Windows.Forms.MessageBox]::Show("Automatic install failed. Please install Microsoft.Graph module manually and relaunch the tool.","Install failed",[System.Windows.Forms.MessageBoxButtons]::OK,[System.Windows.Forms.MessageBoxIcon]::Error)

-> Displays a dialog box to the user (info/confirm/warning).

024: return

-> Returns from the function (exits early).

025: }

-> Executes: }

026: } else {

-> Executes: } else {

027: Log-UI "Microsoft.Graph not installed; cannot connect." "FAIL"

-> Writes a line to the GUI output textbox for user-visible logging.

028: return

-> Returns from the function (exits early).

029: }

-> Executes: }

030: }

-> Executes: }

031:

032: Import-Module Microsoft.Graph -ErrorAction SilentlyContinue

-> Imports the Microsoft.Graph PowerShell module required to call Graph API.

033:

034: try {

-> Start of a try block for error handling.

035: Log-UI "Attempting Graph connection using provided credential..." "INFO"

-> Writes a line to the GUI output textbox for user-visible logging.

036: Connect-MgGraph -Credential $global:ServiceCredential -Scopes "Device.ReadWrite.All","DeviceManagementServiceConfig.ReadWrite.All","DeviceManagementManagedDevices.ReadWrite.All" -ErrorAction Stop

-> Connects to Microsoft Graph (attempts credential-based, may fall back to interactive sign-in).

037: $global:GraphConnected = $true

-> Assigns a value to a variable.

038: $lblStatus.Text = "Connected ✅"

-> Assigns a value to a variable.

039: $lblStatus.ForeColor = [System.Drawing.Color]::Green

-> Assigns a value to a variable.

040: Log-UI "Connected to Microsoft Graph (credential auth)." "OK"

-> Writes a line to the GUI output textbox for user-visible logging.

041: } catch {

-> Catch block to handle exceptions thrown in the try block.

042: Log-UI "Credential-based Graph connection failed (likely MFA/conditional). Falling back to interactive sign-in..." "WARN"

-> Writes a line to the GUI output textbox for user-visible logging.

043: try {

-> Start of a try block for error handling.

044: Connect-MgGraph -Scopes "Device.ReadWrite.All","DeviceManagementServiceConfig.ReadWrite.All","DeviceManagementManagedDevices.ReadWrite.All" -ErrorAction Stop

-> Connects to Microsoft Graph (attempts credential-based, may fall back to interactive sign-in).

045: $global:GraphConnected = $true

-> Assigns a value to a variable.

046: $lblStatus.Text = "Connected ✅"

-> Assigns a value to a variable.

047: $lblStatus.ForeColor = [System.Drawing.Color]::Green

-> Assigns a value to a variable.

048: Log-UI "Connected to Microsoft Graph (interactive auth)." "OK"

-> Writes a line to the GUI output textbox for user-visible logging.

049: } catch {

-> Catch block to handle exceptions thrown in the try block.

050: $global:GraphConnected = $false

-> Assigns a value to a variable.

051: $lblStatus.Text = "Not Connected ❌"

-> Assigns a value to a variable.

052: $lblStatus.ForeColor = [System.Drawing.Color]::Red

-> Assigns a value to a variable.

053: Log-UI "Interactive Graph login failed: $\_" "ERROR"

-> Writes a line to the GUI output textbox for user-visible logging.

054: return

-> Returns from the function (exits early).

055: }

-> Executes: }

056: }

-> Executes: }

057:

058: # detect AD & SCCM availability after connect

-> Comment: detect AD & SCCM availability after connect

059: if (Get-Module -ListAvailable -Name ActiveDirectory) { $global:ADAvailable = $true }

-> Checks whether the ActiveDirectory (RSAT) PowerShell module is available locally.

060: if ($Env:SMS\_ADMIN\_UI\_PATH) { $global:SCCMAvailable = $true }

-> Executes: if ($Env:SMS\_ADMIN\_UI\_PATH) { $global:SCCMAvailable = $true }

061: Update-ActionButtonsState

-> Executes: Update-ActionButtonsState

062: }

-> Executes: }

## Function: Check-Prerequisites

Code excerpt:

001: function Check-Prerequisites {

-> Function declaration - defines a callable routine.

002: Log-UI "Starting prerequisite checks..." "INFO"

-> Writes a line to the GUI output textbox for user-visible logging.

003: $missing = @()

-> Assigns a value to a variable.

004:

005: if (Get-Module -ListAvailable -Name Microsoft.Graph) { Log-UI "Microsoft.Graph module: OK" "OK" } else { Log-UI "Microsoft.Graph module: MISSING" "FAIL"; $missing += "Microsoft.Graph" }

-> Writes a line to the GUI output textbox for user-visible logging.

006:

007: if (Get-Module -ListAvailable -Name ActiveDirectory) { Log-UI "ActiveDirectory module: OK" "OK"; $global:ADAvailable = $true } else { Log-UI "ActiveDirectory module: MISSING (RSAT AD Tools)" "FAIL"; $missing += "ActiveDirectory"; $global:ADAvailable = $false }

-> Checks whether the ActiveDirectory (RSAT) PowerShell module is available locally.

008:

009: $sccmPath = $Env:SMS\_ADMIN\_UI\_PATH

-> Assigns a value to a variable.

010: if ($sccmPath -and (Test-Path ($sccmPath.Substring(0,$sccmPath.Length-5) + '\ConfigurationManager.psd1'))) { Log-UI "SCCM ConfigurationManager module: OK" "OK"; $global:SCCMAvailable = $true } else { Log-UI "SCCM ConfigurationManager module: NOT FOUND (SCCM console missing)" "WARN"; $global:SCCMAvailable = $false; $missing += "ConfigurationManager(SCCMConsole)" }

-> Attempts to import the SCCM ConfigurationManager module (requires SCCM console installed locally).

011:

012: if ($missing.Count -gt 0) {

-> Executes: if ($missing.Count -gt 0) {

013: $msg = "Missing components detected:`n- " + ($missing -join "`n- ") + "`n`nAttempt to install Microsoft.Graph and RSAT-AD (if available) now?"

-> Assigns a value to a variable.

014: $resp = [System.Windows.Forms.MessageBox]::Show($msg,"Missing Components", [System.Windows.Forms.MessageBoxButtons]::YesNo, [System.Windows.Forms.MessageBoxIcon]::Question)

-> Displays a dialog box to the user (info/confirm/warning).

015: if ($resp -eq [System.Windows.Forms.DialogResult]::Yes) {

-> Executes: if ($resp -eq [System.Windows.Forms.DialogResult]::Yes) {

016: if ($missing -contains "Microsoft.Graph") {

-> Executes: if ($missing -contains "Microsoft.Graph") {

017: try { Install-PackageProvider -Name NuGet -MinimumVersion 2.8.5.201 -Force -Scope CurrentUser -ErrorAction SilentlyContinue; Install-Module -Name Microsoft.Graph -Scope CurrentUser -Force -AllowClobber -ErrorAction Stop; Log-UI "Microsoft.Graph installed." "OK" } catch { Log-UI "Failed to install Microsoft.Graph automatically: $\_" "ERROR" }

-> Start of a try block for error handling.

018: }

-> Executes: }

019: if ($missing -contains "ActiveDirectory") {

-> Executes: if ($missing -contains "ActiveDirectory") {

020: try { Log-UI "Attempting to add RSAT ActiveDirectory feature (may require restart & admin)..." "INFO"; Add-WindowsCapability -Online -Name Rsat.ActiveDirectory.DS-LDS.Tools~~~~0.0.1.0 -ErrorAction Stop; Log-UI "RSAT ActiveDirectory installed via WindowsCapability." "OK" } catch { Log-UI "Automatic RSAT add failed or not supported on this OS: $\_" "WARN"; [System.Windows.Forms.MessageBox]::Show("Automatic RSAT install failed or not supported. Please install RSAT Active Directory Tools manually.","Install RSAT", [System.Windows.Forms.MessageBoxButtons]::OK, [System.Windows.Forms.MessageBoxIcon]::Information) }

-> Start of a try block for error handling.

021: }

-> Executes: }

022: } else { Log-UI "User chose not to auto-install missing components." "INFO" }

-> Writes a line to the GUI output textbox for user-visible logging.

023: } else { Log-UI "All prerequisites present." "OK" }

-> Writes a line to the GUI output textbox for user-visible logging.

024:

025: $global:PrereqsChecked = $true

-> Assigns a value to a variable.

026: if (Get-Module -ListAvailable -Name ActiveDirectory) { $global:ADAvailable = $true }

-> Checks whether the ActiveDirectory (RSAT) PowerShell module is available locally.

027: if ($Env:SMS\_ADMIN\_UI\_PATH) { $global:SCCMAvailable = $true }

-> Executes: if ($Env:SMS\_ADMIN\_UI\_PATH) { $global:SCCMAvailable = $true }

028: Update-ActionButtonsState

-> Executes: Update-ActionButtonsState

029: }

-> Executes: }

## Function: Check-Serial

Code excerpt:

001: function Check-Serial {

-> Function declaration - defines a callable routine.

002: if (-not $global:PrereqsChecked) { Log-UI "Run 'Check Prerequisites' first." "FAIL"; return }

-> Writes a line to the GUI output textbox for user-visible logging.

003: if (-not $global:GraphConnected) { Log-UI "Connect/Login first." "FAIL"; return }

-> Writes a line to the GUI output textbox for user-visible logging.

004:

005: $serial = $SerialBox.Text.Trim()

-> Assigns a value to a variable.

006: if (-not $serial) { Log-UI "Serial number required." "FAIL"; return }

-> Writes a line to the GUI output textbox for user-visible logging.

007:

008: try {

-> Start of a try block for error handling.

009: Import-Module Microsoft.Graph -ErrorAction SilentlyContinue

-> Imports the Microsoft.Graph PowerShell module required to call Graph API.

010: $found = Get-MgDeviceManagementWindowsAutopilotDeviceIdentity -All | Where-Object { $\_.SerialNumber -eq $serial }

-> Queries Windows Autopilot device identities via Microsoft Graph.

011: if ($found) {

-> Executes: if ($found) {

012: $global:SerialFound = $true

-> Assigns a value to a variable.

013: $global:AutoPilotDevice = $found

-> Assigns a value to a variable.

014: $lblSerialStatus.Text = "Found ✅"

-> Assigns a value to a variable.

015: $lblSerialStatus.ForeColor = [System.Drawing.Color]::Green

-> Assigns a value to a variable.

016: Log-UI "Autopilot record found: Serial=$serial ; Id=$($found.Id)" "OK"

-> Writes a line to the GUI output textbox for user-visible logging.

017: } else {

-> Executes: } else {

018: $global:SerialFound = $false

-> Assigns a value to a variable.

019: $global:AutoPilotDevice = $null

-> Assigns a value to a variable.

020: $lblSerialStatus.Text = "Not Found ❌"

-> Assigns a value to a variable.

021: $lblSerialStatus.ForeColor = [System.Drawing.Color]::Red

-> Assigns a value to a variable.

022: Log-UI "No Autopilot record found for serial: $serial" "FAIL"

-> Writes a line to the GUI output textbox for user-visible logging.

023: }

-> Executes: }

024: } catch {

-> Catch block to handle exceptions thrown in the try block.

025: Log-UI "Error checking Autopilot (Graph): $\_" "ERROR"

-> Writes a line to the GUI output textbox for user-visible logging.

026: }

-> Executes: }

027: Update-ActionButtonsState

-> Executes: Update-ActionButtonsState

028: }

-> Executes: }

## Function: Check-DeviceName

Code excerpt:

001: function Check-DeviceName {

-> Function declaration - defines a callable routine.

002: if (-not $global:PrereqsChecked) { Log-UI "Run 'Check Prerequisites' first." "FAIL"; return }

-> Writes a line to the GUI output textbox for user-visible logging.

003: if (-not $global:GraphConnected) { Log-UI "Connect/Login first." "FAIL"; return }

-> Writes a line to the GUI output textbox for user-visible logging.

004:

005: $dev = $DeviceBox.Text.Trim()

-> Assigns a value to a variable.

006: if (-not $dev) { Log-UI "Device name required." "FAIL"; return }

-> Writes a line to the GUI output textbox for user-visible logging.

007:

008: $global:DeviceFound = $false

-> Assigns a value to a variable.

009: $global:EntraDevice = $null

-> Assigns a value to a variable.

010: $global:ADFound = $false

-> Assigns a value to a variable.

011: $global:ADComputer = $null

-> Assigns a value to a variable.

012: $global:SCCMFound = $false

-> Assigns a value to a variable.

013: $global:SCCMDevice = $null

-> Assigns a value to a variable.

014:

015: try {

-> Start of a try block for error handling.

016: Import-Module Microsoft.Graph -ErrorAction SilentlyContinue

-> Imports the Microsoft.Graph PowerShell module required to call Graph API.

017: $device = Get-MgDevice -All | Where-Object { $\_.DisplayName -eq $dev }

-> Queries Entra ID/Azure AD device objects using Microsoft Graph.

018: if ($device) {

-> Executes: if ($device) {

019: $global:DeviceFound = $true

-> Assigns a value to a variable.

020: $global:EntraDevice = $device

-> Assigns a value to a variable.

021: $lblDeviceStatus.Text = "Entra: Found ✅"

-> Assigns a value to a variable.

022: $lblDeviceStatus.ForeColor = [System.Drawing.Color]::Green

-> Assigns a value to a variable.

023: Log-UI "Entra device found: $dev ; Id=$($device.Id)" "OK"

-> Writes a line to the GUI output textbox for user-visible logging.

024: } else {

-> Executes: } else {

025: $lblDeviceStatus.Text = "Entra: Not Found ❌"

-> Assigns a value to a variable.

026: $lblDeviceStatus.ForeColor = [System.Drawing.Color]::Red

-> Assigns a value to a variable.

027: Log-UI "Entra: No device found with name $dev" "FAIL"

-> Writes a line to the GUI output textbox for user-visible logging.

028: }

-> Executes: }

029: } catch {

-> Catch block to handle exceptions thrown in the try block.

030: Log-UI "Error checking Entra device: $\_" "ERROR"

-> Writes a line to the GUI output textbox for user-visible logging.

031: }

-> Executes: }

032:

033: if ($global:ADAvailable) {

-> Executes: if ($global:ADAvailable) {

034: try {

-> Start of a try block for error handling.

035: $cred = $global:ServiceCredential

-> Assigns a value to a variable.

036: $comp = Get-ADComputer -Identity $dev -Credential $cred -ErrorAction SilentlyContinue

-> Queries Active Directory for a computer object by name.

037: if ($comp) {

-> Executes: if ($comp) {

038: $global:ADFound = $true

-> Assigns a value to a variable.

039: $global:ADComputer = $comp

-> Assigns a value to a variable.

040: $lblADStatus.Text = "AD: Found ✅"

-> Assigns a value to a variable.

041: $lblADStatus.ForeColor = [System.Drawing.Color]::Green

-> Assigns a value to a variable.

042: Log-UI "AD Computer found: $dev ; DN=$($comp.DistinguishedName)" "OK"

-> Writes a line to the GUI output textbox for user-visible logging.

043: } else {

-> Executes: } else {

044: $lblADStatus.Text = "AD: Not Found ❌"

-> Assigns a value to a variable.

045: $lblADStatus.ForeColor = [System.Drawing.Color]::Red

-> Assigns a value to a variable.

046: Log-UI "AD: No computer found with name $dev" "FAIL"

-> Writes a line to the GUI output textbox for user-visible logging.

047: }

-> Executes: }

048: } catch {

-> Catch block to handle exceptions thrown in the try block.

049: Log-UI "Error checking AD computer: $\_" "ERROR"

-> Writes a line to the GUI output textbox for user-visible logging.

050: }

-> Executes: }

051: } else {

-> Executes: } else {

052: $lblADStatus.Text = "AD: Module N/A"

-> Assigns a value to a variable.

053: $lblADStatus.ForeColor = [System.Drawing.Color]::Orange

-> Assigns a value to a variable.

054: }

-> Executes: }

055:

056: if ($global:SCCMAvailable) {

-> Executes: if ($global:SCCMAvailable) {

057: try {

-> Start of a try block for error handling.

058: if (-not (Get-Module ConfigurationManager)) {

-> Executes: if (-not (Get-Module ConfigurationManager)) {

059: Import-Module ($Env:SMS\_ADMIN\_UI\_PATH.Substring(0,$Env:SMS\_ADMIN\_UI\_PATH.Length-5) + '\ConfigurationManager.psd1') -ErrorAction SilentlyContinue

-> Attempts to import the SCCM ConfigurationManager module (requires SCCM console installed locally).

060: }

-> Executes: }

061: $d = Get-CMDevice -Name $dev -ErrorAction SilentlyContinue

-> Queries SCCM for a device record by name.

062: if ($d) {

-> Executes: if ($d) {

063: $global:SCCMFound = $true

-> Assigns a value to a variable.

064: $global:SCCMDevice = $d

-> Assigns a value to a variable.

065: $lblSCCMStatus.Text = "SCCM: Found ✅"

-> Assigns a value to a variable.

066: $lblSCCMStatus.ForeColor = [System.Drawing.Color]::Green

-> Assigns a value to a variable.

067: Log-UI "SCCM device found: $dev ; ResourceID=$($d.ResourceID)" "OK"

-> Writes a line to the GUI output textbox for user-visible logging.

068: } else {

-> Executes: } else {

069: $lblSCCMStatus.Text = "SCCM: Not Found ❌"

-> Assigns a value to a variable.

070: $lblSCCMStatus.ForeColor = [System.Drawing.Color]::Red

-> Assigns a value to a variable.

071: Log-UI "SCCM: No device found with name $dev" "FAIL"

-> Writes a line to the GUI output textbox for user-visible logging.

072: }

-> Executes: }

073: } catch {

-> Catch block to handle exceptions thrown in the try block.

074: Log-UI "Error checking SCCM: $\_" "ERROR"

-> Writes a line to the GUI output textbox for user-visible logging.

075: }

-> Executes: }

076: } else {

-> Executes: } else {

077: $lblSCCMStatus.Text = "SCCM: N/A"

-> Assigns a value to a variable.

078: $lblSCCMStatus.ForeColor = [System.Drawing.Color]::Orange

-> Assigns a value to a variable.

079: }

-> Executes: }

080:

081: Update-ActionButtonsState

-> Executes: Update-ActionButtonsState

082: }

-> Executes: }

## Function: Action-RemoveIntune

Code excerpt:

001: function Action-RemoveIntune {

-> Function declaration - defines a callable routine.

002: if (-not (Ensure-Connected)) { return }

-> Executes: if (-not (Ensure-Connected)) { return }

003: if (-not $global:SerialFound -or -not $global:AutoPilotDevice) { Log-UI "No Autopilot/Serial found for removal." "FAIL"; return }

-> Writes a line to the GUI output textbox for user-visible logging.

004:

005: $serial = $global:AutoPilotDevice.SerialNumber

-> Assigns a value to a variable.

006: $id = $global:AutoPilotDevice.Id

-> Assigns a value to a variable.

007: $msg = "Autopilot record found:`r`nSerial: $serial`r`nID: $id`r`n`r`nDo you confirm deletion of this Autopilot enrollment?"

-> Assigns a value to a variable.

008: if (-not (Confirm-Action $msg)) { Log-UI "User cancelled Autopilot deletion." "CANCEL"; return }

-> Writes a line to the GUI output textbox for user-visible logging.

009:

010: try {

-> Start of a try block for error handling.

011: Remove-MgDeviceManagementWindowsAutopilotDeviceIdentity -WindowsAutopilotDeviceIdentityId $id -Confirm:$false -ErrorAction Stop

-> Removes an Autopilot device identity from Intune/Autopilot via Graph.

012: $out = "Autopilot enrollment removed for serial $serial (Id=$id)."

-> Assigns a value to a variable.

013: Log-UI $out "OK"

-> Writes a line to the GUI output textbox for user-visible logging.

014: Save-ActionLog -Action "RemoveAutopilot" -SerialOrDevice $serial -Content $out

-> Saves a full action log to a timestamped file in the script's folder for audit.

015: } catch {

-> Catch block to handle exceptions thrown in the try block.

016: Log-UI "Failed to remove Autopilot device: $\_" "ERROR"

-> Writes a line to the GUI output textbox for user-visible logging.

017: Save-ActionLog -Action "RemoveAutopilot\_FAILED" -SerialOrDevice $serial -Content $\_.ToString()

-> Saves a full action log to a timestamped file in the script's folder for audit.

018: }

-> Executes: }

019: }

-> Executes: }

## Function: Action-RemoveEntra

Code excerpt:

001: function Action-RemoveEntra {

-> Function declaration - defines a callable routine.

002: if (-not (Ensure-Connected)) { return }

-> Executes: if (-not (Ensure-Connected)) { return }

003: if (-not $global:EntraDevice) { Log-UI "No Entra device found." "FAIL"; return }

-> Writes a line to the GUI output textbox for user-visible logging.

004:

005: $name = $global:EntraDevice.DisplayName

-> Assigns a value to a variable.

006: $id = $global:EntraDevice.Id

-> Assigns a value to a variable.

007: $msg = "Entra device found:`r`nName: $name`r`nID: $id`r`n`r`nDo you confirm deletion of this Entra device?"

-> Assigns a value to a variable.

008: if (-not (Confirm-Action $msg)) { Log-UI "User cancelled Entra deletion." "CANCEL"; return }

-> Writes a line to the GUI output textbox for user-visible logging.

009:

010: try {

-> Start of a try block for error handling.

011: Remove-MgDevice -DeviceId $id -Confirm:$false -ErrorAction Stop

-> Deletes a device object from Entra ID using Microsoft Graph.

012: $out = "Entra device removed: $name (Id=$id)."

-> Assigns a value to a variable.

013: Log-UI $out "OK"

-> Writes a line to the GUI output textbox for user-visible logging.

014: Save-ActionLog -Action "RemoveEntra" -SerialOrDevice $name -Content $out

-> Saves a full action log to a timestamped file in the script's folder for audit.

015: } catch {

-> Catch block to handle exceptions thrown in the try block.

016: Log-UI "Failed to remove Entra device: $\_" "ERROR"

-> Writes a line to the GUI output textbox for user-visible logging.

017: Save-ActionLog -Action "RemoveEntra\_FAILED" -SerialOrDevice $name -Content $\_.ToString()

-> Saves a full action log to a timestamped file in the script's folder for audit.

018: }

-> Executes: }

019: }

-> Executes: }

## Function: Action-ADDisableMove

Code excerpt:

001: function Action-ADDisableMove {

-> Function declaration - defines a callable routine.

002: if (-not $global:ADFound -or -not $global:ADComputer) { Log-UI "No AD computer found to modify." "FAIL"; return }

-> Writes a line to the GUI output textbox for user-visible logging.

003: $dev = $global:ADComputer.Name

-> Assigns a value to a variable.

004: $dn = $global:ADComputer.DistinguishedName

-> Assigns a value to a variable.

005: $targetOU = $OUBox.Text.Trim()

-> Assigns a value to a variable.

006: if (-not $targetOU) { Log-UI "Target OU is required." "FAIL"; return }

-> Writes a line to the GUI output textbox for user-visible logging.

007:

008: $msg = "AD Computer found:`r`nName: $dev`r`nDN: $dn`r`nTarget OU: $targetOU`r`n`r`nProceed to Disable account and move to OU?"

-> Assigns a value to a variable.

009: if (-not (Confirm-Action $msg)) { Log-UI "User cancelled AD disable/move." "CANCEL"; return }

-> Writes a line to the GUI output textbox for user-visible logging.

010:

011: try {

-> Start of a try block for error handling.

012: $cred = $global:ServiceCredential

-> Assigns a value to a variable.

013: Disable-ADAccount -Identity $global:ADComputer -Credential $cred -ErrorAction Stop

-> Disables the AD account for the computer (prevents logon).

014: Move-ADObject -Identity $global:ADComputer.DistinguishedName -TargetPath $targetOU -Credential $cred -ErrorAction Stop

-> Moves the AD object to a target OU (used for quarantine or decommissioning).

015: $out = "Disabled and moved AD computer $dev to $targetOU."

-> Assigns a value to a variable.

016: Log-UI $out "OK"

-> Writes a line to the GUI output textbox for user-visible logging.

017: Save-ActionLog -Action "ADDisableMove" -SerialOrDevice $dev -Content $out

-> Saves a full action log to a timestamped file in the script's folder for audit.

018: } catch {

-> Catch block to handle exceptions thrown in the try block.

019: Log-UI "Failed AD disable/move: $\_" "ERROR"

-> Writes a line to the GUI output textbox for user-visible logging.

020: Save-ActionLog -Action "ADDisableMove\_FAILED" -SerialOrDevice $dev -Content $\_.ToString()

-> Saves a full action log to a timestamped file in the script's folder for audit.

021: }

-> Executes: }

022: }

-> Executes: }

## Function: Action-RemoveSCCM

Code excerpt:

001: function Action-RemoveSCCM {

-> Function declaration - defines a callable routine.

002: if (-not $global:SCCMFound -or -not $global:SCCMDevice) { Log-UI "No SCCM device found to remove or SCCM console missing." "FAIL"; return }

-> Writes a line to the GUI output textbox for user-visible logging.

003: $dev = $global:SCCMDevice.Name

-> Assigns a value to a variable.

004: $id = $global:SCCMDevice.ResourceID

-> Assigns a value to a variable.

005: $msg = "SCCM Device found:`r`nName: $dev`r`nResourceID: $id`r`n`r`nProceed to remove from SCCM?"

-> Assigns a value to a variable.

006: if (-not (Confirm-Action $msg)) { Log-UI "User cancelled SCCM removal." "CANCEL"; return }

-> Writes a line to the GUI output textbox for user-visible logging.

007:

008: try {

-> Start of a try block for error handling.

009: if (-not (Get-Module ConfigurationManager)) {

-> Executes: if (-not (Get-Module ConfigurationManager)) {

010: Import-Module ($Env:SMS\_ADMIN\_UI\_PATH.Substring(0,$Env:SMS\_ADMIN\_UI\_PATH.Length-5) + '\ConfigurationManager.psd1') -ErrorAction SilentlyContinue

-> Attempts to import the SCCM ConfigurationManager module (requires SCCM console installed locally).

011: }

-> Executes: }

012: Remove-CMDevice -DeviceName $dev -Force -ErrorAction Stop

-> Removes a device from SCCM (ConfigMgr) collection/database.

013: $out = "Removed device $dev from SCCM."

-> Assigns a value to a variable.

014: Log-UI $out "OK"

-> Writes a line to the GUI output textbox for user-visible logging.

015: Save-ActionLog -Action "RemoveSCCM" -SerialOrDevice $dev -Content $out

-> Saves a full action log to a timestamped file in the script's folder for audit.

016: } catch {

-> Catch block to handle exceptions thrown in the try block.

017: Log-UI "Failed to remove SCCM device: $\_" "ERROR"

-> Writes a line to the GUI output textbox for user-visible logging.

018: Save-ActionLog -Action "RemoveSCCM\_FAILED" -SerialOrDevice $dev -Content $\_.ToString()

-> Saves a full action log to a timestamped file in the script's folder for audit.

019: }

-> Executes: }

020: }

-> Executes: }

## Function: Upload-HWIDCSV

Code excerpt:

001: function Upload-HWIDCSV {

-> Function declaration - defines a callable routine.

002: if (-not (Ensure-Connected)) { return }

-> Executes: if (-not (Ensure-Connected)) { return }

003:

004: $ofd = New-Object System.Windows.Forms.OpenFileDialog

-> Opens a file selection dialog to let the user pick a CSV file.

005: $ofd.Filter = "CSV files (\*.csv)|\*.csv|All files (\*.\*)|\*.\*"

-> Assigns a value to a variable.

006: $ofd.Multiselect = $false

-> Assigns a value to a variable.

007: $ok = $ofd.ShowDialog()

-> Assigns a value to a variable.

008: if ($ok -ne [System.Windows.Forms.DialogResult]::OK) { Log-UI "HWID CSV upload cancelled by user." "CANCEL"; return }

-> Writes a line to the GUI output textbox for user-visible logging.

009:

010: $path = $ofd.FileName

-> Assigns a value to a variable.

011: Log-UI "Selected HWID CSV: $path" "INFO"

-> Writes a line to the GUI output textbox for user-visible logging.

012:

013: try {

-> Start of a try block for error handling.

014: $rows = Import-Csv -Path $path -ErrorAction Stop

-> Reads CSV rows into PowerShell objects for processing.

015: if ($rows.Count -eq 0) { Log-UI "CSV contains no rows." "FAIL"; return }

-> Writes a line to the GUI output textbox for user-visible logging.

016:

017: # Confirm action

-> Comment: Confirm action

018: $cnt = $rows.Count

-> Assigns a value to a variable.

019: $msg = "CSV contains $cnt rows. Proceed to upload to Autopilot?"

-> Assigns a value to a variable.

020: if (-not (Confirm-Action $msg)) { Log-UI "User cancelled HWID upload." "CANCEL"; return }

-> Writes a line to the GUI output textbox for user-visible logging.

021:

022: $success = 0; $failed = 0; $details = @()

-> Assigns a value to a variable.

023: foreach ($r in $rows) {

-> Iterates over a collection of objects.

024: # Expecting at minimum: "SerialNumber" and "HardwareHash" fields as columns; other columns optional (Manufacturer, Model, OrderIdentifier)

-> Comment: Expecting at minimum: "SerialNumber" and "HardwareHash" fields as columns; other columns optional (Manufacturer, Model, OrderIdentifier)

025: $serial = $r.SerialNumber -or $r.DeviceSerialNumber -or $r."Device Serial Number"

-> Assigns a value to a variable.

026: $hw = $r.HardwareHash -or $r.HardwareIdentifier -or $r."Hardware Hash"

-> Assigns a value to a variable.

027: $manufacturer = $r.Manufacturer -or $r.Mfg -or $null

-> Assigns a value to a variable.

028: $model = $r.Model -or $null

-> Assigns a value to a variable.

029:

030: if (-not $serial -or -not $hw) {

-> Executes: if (-not $serial -or -not $hw) {

031: $failed++;

-> Executes: $failed++;

032: $details += "Row missing required SerialNumber or HardwareHash: $($r | Out-String)"

-> Assigns a value to a variable.

033: continue

-> Executes: continue

034: }

-> Executes: }

035:

036: # Use Microsoft.Graph to create Autopilot identity

-> Comment: Use Microsoft.Graph to create Autopilot identity

037: try {

-> Start of a try block for error handling.

038: # New-MgDeviceManagementWindowsAutopilotDeviceIdentity is used when available

-> Comment: New-MgDeviceManagementWindowsAutopilotDeviceIdentity is used when available

039: New-MgDeviceManagementWindowsAutopilotDeviceIdentity -BodyParameter @{serialNumber = $serial; hardwareIdentifier = $hw; manufacturer = $manufacturer; model = $model} -ErrorAction Stop

-> Creates/Registers a new Autopilot device identity via Microsoft Graph.

040: $success++

-> Executes: $success++

041: } catch {

-> Catch block to handle exceptions thrown in the try block.

042: $failed++

-> Executes: $failed++

043: $details += "Failed to upload serial $serial : $\_"

-> Assigns a value to a variable.

044: }

-> Executes: }

045: }

-> Executes: }

046:

047: $summary = "HWID upload completed. Success: $success ; Failed: $failed"

-> Assigns a value to a variable.

048: Log-UI $summary "OK"

-> Writes a line to the GUI output textbox for user-visible logging.

049: Save-ActionLog -Action "UploadHWID" -SerialOrDevice (Split-Path -Leaf $path) -Content ($summary + "`n`n" + ($details -join "`n"))

-> Saves a full action log to a timestamped file in the script's folder for audit.

050:

051: } catch {

-> Catch block to handle exceptions thrown in the try block.

052: Log-UI "Failed to process/upload HWID CSV: $\_" "ERROR"

-> Writes a line to the GUI output textbox for user-visible logging.

053: }

-> Executes: }

054: }

-> Executes: }

# GUI Layout & Event Flow

Device Decommission & Re-Enroll Tool - User Guide

This document provides detailed information about the Device Decommission Tool, a PowerShell-based GUI application designed to automate device cleanup and re-enrollment across Microsoft Intune, Entra ID (Azure AD), Active Directory, and SCCM. The tool also supports uploading new Autopilot HWID CSV files, with enhanced safety checks.

**1. Purpose of the Tool**

The tool streamlines repetitive decommission tasks by consolidating them into a single GUI. It prevents mistakes, ensures consistent cleanup, and provides safe re-enrollment of devices into Autopilot.

**2. Requirements**

- Windows 10/11 with PowerShell 5.1 or later  
- Required Modules: Microsoft.Graph, ActiveDirectory, ConfigurationManager (if SCCM), WindowsAutopilotIntune  
- Service Account with delegated rights (Intune, Entra ID, AD, SCCM if applicable)  
- MFA support enabled  
- Daily password updates (if password rotation policy is in place)

**3. Workflow & Sequence**

1. Login with service account credentials (MFA supported).  
2. Module Checker validates prerequisites; SCCM is optional.  
3. Serial Checker verifies device existence in Intune Autopilot.  
4. Device Name Checker verifies device in Entra ID/AD.  
5. Dynamic Button Rules:  
 - Serial exists only → Enrollment removal enabled.  
 - Device name exists only → Entra/AD/SCCM removal enabled.  
 - Both exist → all buttons enabled.  
 - SCCM missing → SCCM button disabled.  
6. HWID Upload Button:  
 - Disabled until Serial Checker run.  
 - If Serial exists in Autopilot → button stays disabled.  
 - If Serial not found → button enabled.  
 - After removal + re-check → button becomes enabled.  
7. Triple Confirmation:  
 - Confirm upload intent.  
 - Confirm CSV file is correct.  
 - Final proceed/cancel dialog.  
8. Each action produces logs stored with timestamp.  
9. Exit reminder notifies user where logs are stored.

**4. Safety Features**

- Serial/Device check before enabling buttons.  
- Triple confirmation for all destructive or enrollment actions.  
- HWID upload is blocked if device already exists in Autopilot.  
- Logs stored locally for audit.

**5. Limitations**

- Requires interactive login; service principals not supported.  
- SCCM only works if console + module installed locally.  
- Only one device or CSV upload processed per run.  
- Bulk automation may hit Graph API throttling.

**6. Logging**

Logs saved as:  
YYYYMMDD\_HHMMSS\_Action\_SerialOrDeviceName.log  
  
Each log entry includes:  
- Action executed  
- Account used  
- Target device(s)  
- Result (success, failure, error)

**7. Example Use Cases**

- Decommission laptops for departing employees.  
- Clean stale device objects.  
- Upload HWID for re-enrolling wiped devices.  
- Safe guided workflow for Helpdesk teams.

**8. Troubleshooting**

- Login issues → verify credentials and MFA.  
- Missing modules → run module checker.  
- HWID upload disabled → device already exists, must be removed first.  
- SCCM disabled → console not installed.  
- CSV upload errors → validate format and permissions.

**9. Future Enhancements**

- Bulk device operations  
- ServiceNow integration  
- Automated reporting  
- Scheduled cleanup jobs  
  
This document provides a comprehensive, line-by-line walkthrough of the Device Decommission Tool PowerShell script. It explains each function in detail, including why specific modules, commands, and permissions are required. This expanded version includes Microsoft Graph API permission requirements for each action.

**Authentication and Service Account**

The tool uses delegated rights with a service account that supports MFA. Authentication is established using Connect-MgGraph with the required scopes.  
Example line:  
 Connect-MgGraph -Scopes 'DeviceManagementManagedDevices.ReadWrite.All', 'Device.ReadWrite.All', 'Directory.AccessAsUser.All'  
  
Explanation:  
- Connects to Microsoft Graph API.  
- Requires delegated rights and MFA.  
- Fails if scopes are missing or account lacks role assignments.

**Graph API Permissions Matrix**

|  |  |  |  |
| --- | --- | --- | --- |
| Function | Graph Endpoint | Permission Scope | Reason |
| Remove from Intune Enrollment | DELETE /deviceManagement/managedDevices/{id} | DeviceManagementManagedDevices.ReadWrite.All | Required to delete managed devices from Intune enrollment. |
| Remove from Entra ID | DELETE /devices/{id} | Directory.AccessAsUser.All or Directory.ReadWrite.All | Required to delete a device object in Entra ID. |
| Upload Autopilot HWID | POST /deviceManagement/windowsAutopilotDeviceIdentities/import | DeviceManagementServiceConfig.ReadWrite.All | Required to upload hardware hashes (CSV) for Windows Autopilot enrollment. |
| Check Intune Devices | GET /deviceManagement/managedDevices | DeviceManagementManagedDevices.Read.All | Required to query and confirm Intune device serial number or name. |
| Check Entra Devices | GET /devices | Directory.Read.All | Required to confirm device objects by name or ID in Entra ID. |

**Active Directory Operations**

Example function:  
 Move-ADObject -Identity $Device.DistinguishedName -TargetPath 'OU=Decommissioned,DC=contoso,DC=com'  
  
Explanation:  
- This command moves a computer object into a specific OU.  
- Requires AD module installed.  
- Service account must have delegated rights to move and disable devices in Active Directory.

**SCCM Operations**

SCCM device deletion uses the Configuration Manager PowerShell module. If the module or console is missing, the tool prompts the user and disables SCCM actions.  
  
Example function:  
 Remove-CMDevice -Name $DeviceName  
  
Required Rights:  
- Full Administrator or custom RBAC with delete device rights in SCCM.

**HWID Upload Safeguards**

The HWID upload button remains disabled unless the serial number check confirms that the device does not already exist in Intune.  
  
Triple-Check Protection:  
1. User is prompted that the CSV will be uploaded.  
2. A second confirmation popup verifies the serial number and CSV path.  
3. A final confirmation asks the user to type 'YES' before execution.  
  
This prevents accidental enrollment of duplicate devices.

**Helpdesk Quick Summary**

1. Login with service account (MFA required). Ensure status shows 'Connected'.  
2. Run 'Check Modules'. If missing, allow install.  
3. Run 'Check Serial' or 'Check Device Name'.  
4. Use only buttons that are enabled based on checks.  
5. Each action will prompt for triple confirmation before proceeding.  
6. Logs are saved in the same folder as the script with the format:  
 Date+Time+Action+SerialOrDeviceName.log  
7. If SCCM console is not installed, SCCM button will be disabled.