

# Table of Contents

## RPS Patching v4.0.0

### Introduction & Overview

#### Introduction

### Design

#### Access Control for Patch Management and Sync

#### RPS Patch Management Workflow

#### Sync Service Information

#### RPS Patching Script Framework

#### RPS Patch Manifest Definition

#### New Configurations for CDN

#### RPS Package Provider

### How To

#### How to Patch Using RPS

#### How to Create a Patchable Target Type

#### How to Use Maintenance Windows

#### How to Create an RPS Patch

#### How to Disable or Enable an RPS Patch Using PowerShell

#### How to Create a Patch Stream

#### How to Transfer Non-Patching Content Delivery with RPS

#### How to Load a Patch Stream

#### How to Add a Patch Using PowerShell

#### How to Remove a Patch From a Patch Stream

#### How to Remove a Patch Stream

#### Sideloaded RPS Patches

#### How to Approve and Reject Patch Streams

#### Viewing Patch Stream Deployment Telemetry

#### How to View All Patches

#### How to Enable and Disable CDN Communication

### Additional Resources

#### How to Create and Use Patch Chains

### External Patching

#### Patch Telemetry UI

#### Sidewinder

# RPS Patching v4.0.0

*Last updated on June 9, 2021.*

*Last Reviewed and Approved on PENDING REVIEW*

## Welcome to the RPS v4.0.0 Patching Documentation Landing Page

Rapid Provisioning System (**RPS**) is a flexible and powerful automation tool for managing software installation, updates, and configuration. This repository has been created specifically for RPS developers and other RPS administration patching roles.

## What is RPS Patching Documentation?

RPS Patching documentation provides details about the RPS patching system, including how to properly operate and maintain RPS; and how to load, deploy, and check the status of patches and ISOs in RPS.

### **IMPORTANT**

Documentation bundled with RPS v4.0.0 is accurate as of **9/20/2021**.

Updated documentation can be found at: <https://reactr.azurewebsites.us>

To access documentation on how to create, edit, delete, and download patches and ISOs in REACTR, please visit [RPS Patching in REACTR](#), located at the website above.

# Access Control for Patch Management and Sync

*Last updated on August 3, 2021.*

*Last Reviewed and Approved on PENDING REVIEW*

## Introduction

This reference document describes the service account(s) and Windows services required for the Patch Management capability in the Rapid Provisioning System (RPS).

## Intended Audience

IT professionals and administrators who routinely build software deployment packages to update servers and workstations are primary users of RPS.

## Overview

1. RPS servers and clients, called targets, use a combination of certificates, Windows services, and service accounts.
2. A service account is a user account that is mapped into the logon of the Windows service on RPS servers.
3. Some services, such as DSC (Desired State Configuration) only need to run in the local system context, using the standard Windows local system account.

### View Windows Services

As an RPS Administrator,

1. logon to an RPS Server.
2. open Computer Management and Services.
3. Or use PowerShell.

### Sync Service

- Service Name: SyncService
- Logon (service account): WebApiServiceAccount.
- The Sync Service is a custom RPS windows service.
- Startup Type: Automatic
- The account lets the service communicate across RPS parent and child servers.
- The service/account uses certificates for authentication which allows cross-AD-domain authentication.
- The service collects and investigates which files it needs to download for BITS.

### Distributed File System Replication Service

- Service Name: DFS Replication (DFSR)
- Logon (service account): WebApiServiceAccount.
- The Distributed File System Replication (DFSR) is a Windows server feature and Windows service.
- Startup Type: Automatic
- The service/account uses certificates for authentication which allows cross-AD-domain authentication.
- DFSR is a native Windows multi-master replication engine running on RPS servers to keep file folders synchronized.
- DFSR was chosen for RPS as a more efficient and bandwidth-saving way to replicate RPS deployment files.

### Background Intelligent Transfer Service

- Service Name: BITS
- Logon: Local System
- Startup Type: Automatic (Delayed Start)
- BITS is a common Windows service running on all Windows clients and servers, such as for common Windows updates.

- BITS is used also for RPS client-server communications.

#### Desired State Configuration

- Service Name: DSC
- Service Account Name: Local System.
- The Desired State Configuration (DSC) or "Windows PowerShell DSC" is a foundational capability of RPS.
- DSC is a management platform where RPS users can configure, deploy, and manage RPS packages.

# RPS Patch Management Workflow

Last updated on April 21, 2021.

Last Reviewed and Approved on PENDING REVIEW

The RPS patch management workflow is described in 19 steps, in the following figure:

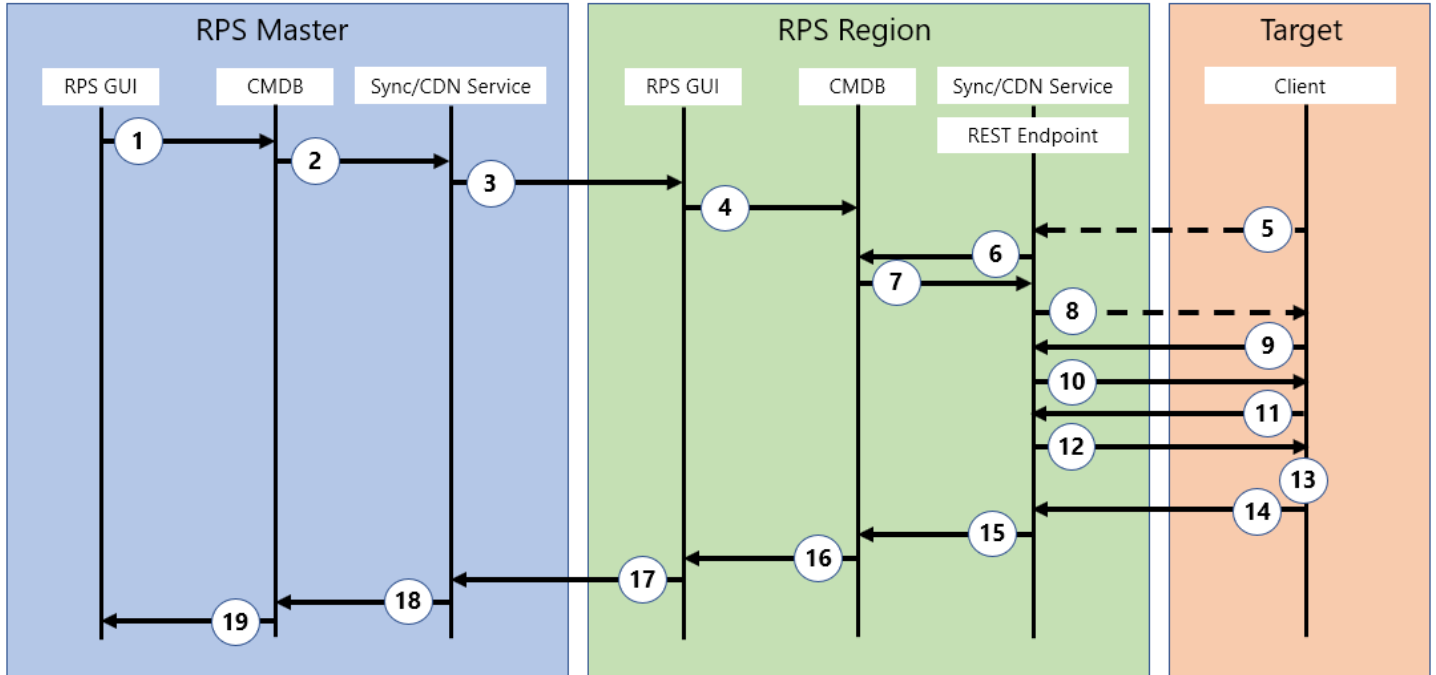


Figure 1: RPS Patch Management Workflow.

## Patch Workflow

The workflow begins at the parent node, from users, who build package streams and packages using the RPS website or PowerShell.

1. Patch and folder-to-target resource assignments are made by approving a Patch Stream, stored as resource assignments in the RPS Configuration Management Database (CMDB).
2. RPS Sync service: This windows service (Sync service) detects an updated resource assignment in the CMDB.
3. The Sync/CDN Services work to transmit resource assignments to RPS child nodes.
4. The Child Node RPS server windows Sync Service updates the local CMDB with new resource assignments.
5. Clients (that run RPS services including the DSC) query a child node's CMDB and register its Target Item ID in the database.
6. Child node REST endpoints query the CMDB for all resource assignments of type Package for the specified Target ID.
7. CMDB: Returns requested Resource Assignment objects to the REST endpoint.
8. Child nodes return a list of patch name/version, ensure states, and deploy attempt counters to the RPS Client.
9. Client then sends a request for specific metadata to the REST endpoint.
10. REST endpoints provide metadata to clients and clients perform a state check.
11. Clients request missing patches needed to reach the desired state and also requests maintenance windows pertinent to the client.

12. Child node REST endpoints provide patches and maintenance windows to the client.
13. If target is within the Maintenance Window, then the patch is deployed to the Target.
14. Clients send state reports to the REST endpoint.
15. REST endpoints update resource assignment's states (InDesiredState, DeployedOn, PackagesInMaintenanceWindow, DeployedStatus) based on received report data.
16. Child Node Sync services detect the updates occurring in the CMDB.
17. Updates are detected by the parent node Sync service.
18. Parent node CMDB is updated.
19. GUI telemetry data is updated for each package and target.

## RPS Components

RPS is similar to systems management suites in that it has a parent-child infrastructure of RPS servers, databases, and Windows services. In this way, target clients that need software packages communicate with locally or geographically distributed RPS child servers, who in turn communicate with RPS parent node servers. RPS users who build packages interact with the Parent Node website user interface (or PowerShell) and child nodes are updated automatically. Clients register with RPS, scan for needed updates, and RPS provides needed packages to each target client for install, via maintenance window.

**RPS Servers:** Parent node RPS servers and child node RPS servers that communicate with each other primarily with the RPS Sync and CDN (Content Delivery Network) services. Child node RPS servers have a REST API endpoint listening for RPS clients that need to communicate with the child CMDB.

**CMDB** Configuration Management Database: A parent/child set of databases that run on RPS parent and child servers, a sort of "brain" for RPS.

**Child Node:** Child RPS servers communicate with RPS parent servers for packages and resource assignments, as well as with clients via REST API endpoint.

**Client:** Target endpoint Windows computers or other devices that need RPS packages.

**DSC** Desired State Configuration: a service running on RPS clients (targets) that can scan, query, and remediate to the desired configuration, via RPS users from the parent node.

**Resource Assignment:** The method that RPS uses to assign the packages to RPS-registered clients (targets).

**Target ID:** A unique identifier of the RPS registered computer client or endpoint.

**Sync Service:** A windows service running on RPS servers that communicates with other sync services and the CMDB.

## Sync Service Settings

*Last updated on December 21, 2020.*

*Last Reviewed and Approved on PENDING REVIEW*

=====

These settings live within the Sync Service's configuration (app.config) file. Any changes to these settings would require a restart of the Sync Service in order to take effect. It is assumed that the reader has a knowledge of how time requirements as well as the specific functionality desired for a specific instance of RPS.

## General

The configuration settings inside the Sync Service application configuration file are located in the <appSettings> section of the document. This section will define each setting purpose and the valid values for the setting.

### ContextCommandTimeout

*Type:* Integer

*Default:* 3600

*Range:* Any integer greater than or equal to 1. Recommended are value is 3600

Timeout, in seconds, for database calls from the Sync Service. Changing this value would alter the maximum amount of time a database call is allowed to execute. This prevents locking the database for abnormal amounts of time.

### HttpClientTimeout

*Type:* Integer

*Default:* 3600

*Range:* Any integer greater than or equal to 1. Recommended are value is 3600

Timeout, in seconds, for HTTP(s) calls from the Sync Service. Changing this value would alter the amount of time a connection is allowed to be alive for a single HTTP(s) call. This would affect both the Sync operations as well as the requests to the Fileserver when synchronizing CDN packages

### CreateClient

*Type:* Boolean

*Default:* True

Indicates whether or not to create a client to used to synchronize data changes with to and from the current node's parent. The type of data synchronized when this setting is set to true includes target data, resources and properties, patch stream telemetry, etc. If this setting is set to false, all synchronization both to and from the parent will be disabled.

## Logging

*The following settings are for setting specific levels of Sync Service application logging.*

### SyncLogs

*Type:* Boolean

*Default:* True

*Remarks:* If the CreateClient setting is off, these logs will not synchronize because data synchronization is turned off.

Indicates whether or not to Synchronize logs with the parent Sync Service. Setting the value to true will cause the logs on the client to synchronize with its parent. Setting the value to false will disable this synchronization.

## LogLevel

*Type:* Integer

*Default:* 4

Indicates the level of logging the sync service will use when synchronizing the logs to the parent node. When a log is made in RPS, it is assigned a log level. When the synchronization of logs occurs, it will send all logs with a level equal to and higher than the value set in this setting to the parent.

1. Verbose – Setting this as the value will synchronize everything logged by the Sync service
2. Debug – Contains information useful for debugging.
3. Information – Contains general information.
4. Warning – Contains warnings for the user to take note of. General indicates that a potential error may occur due to some action taken, but its not guaranteed that it will result in an error.
5. Error – Contains information about an error that the Sync Service encountered.
6. Fatal – Contains information about a catastrophic error that usually results in the partial or total loss of the Sync Service.

## API Server

This section covers settings that are specific to the rest service endpoints for the Sync Service.

### CreateServer

*Type:* Boolean

*Default:* True

Indicates whether or not to start a server instance to host the Sync Api endpoints. This server is needs to be enabled on both nodes in order to allow communications between them for both Syncing of data and CDN items. Setting this value to true will start the REST API server internally by the Sync Service. By this value to false, calls into the SyncService via remote sources would fail.

## CDN Settings

This section contains settings for the content delivery network.

### CreateCDN

*Type:* Boolean

*Default:* True

Indicates whether or not to enable the CDN indexer functionality of the Sync Service. By this option this option to true, you will turn on the CDN and any subsequent synchronization of the CDN with a parent CDN. Setting this value to false will not start the CDN and its synchronizations, thus stopping any new BITS transfers from occurring. This setting has no ties to enabling the file system or api server.

### IndexerInterval

*Type:* Integer

*Default:* 3

*Range:* Any integer greater than or equal to 1. Recommended are value is 3



The interval, in minutes, that the CDN will synchronize. A larger value will increase the time between synchronizations. CDN synchronization will look at what files are currently on the system vs what files should be there. If the system is set to BITS, it will then request any missing files from its parent and begin the download.

## CDN File Server

This section covers settings that are specific to the Content Delivery Network (CDN) File Server. The CDN File Server is the mechanism that transmits files to parent and child nodes.

### CreateStaticFiles

*Type:* Boolean

*Default:* True

Indicates whether or not to start the FileServer in the Sync Service. This is required if the current node is expected to host CDN packages and distribute them downstream. In order to enable this service both the CreateServer setting must be true AND this setting must be true. That is because the file server endpoint is hosted in the API Server. Assuming the CreateServer setting is enabled, setting this value to true will allow for the downstream nodes to request packages from the current node. By setting this value to false, any requests for CDN packages by a child node will fail, stating the endpoint is not there (404 error).

### FileServerOptions.RequestPath

*Type:* String

*Default:* /files

*Remarks:* The value MUST start with a "/" and must be a valid url

The part of the url that allows access to the CDN File Server where CDN files will be available from Sync Service. For example if the base was *www.contoso.com*, and the request path was */files*, the endpoint to access the file server *www.contoso.com/files*.

### FileServerOptions.EnableDirectoryBrowsing

*Type:* Boolean

*Default:* True

Indicates whether or not to allow directory browsing of the file server. Setting this option to true will allow an individual to hit the root file server endpoint and get the directory view for all static files (CDN files) on the node. Setting the value to false disables this ability.

### FileServerOptions.EnableDefaultFiles

*Type:* Boolean

*Default:* False

Indicates whether or not to enable default files on the file server. When this option is set to true, the server will attempt to try to server out an index.html file when visiting the file server root. Setting this value to false will stop the attempt to serve out the index.html file.

## Sync Testing

### General

The goal of this testing is to verify correct data replication between nodes in various scenarios. See the below section for an overview of the sync scenarios and concepts which are tested.

# Testing Scenarios

This section contains the scenarios for the Sync Service that are continually tested through the DevOps pipeline.

## Synchronizing Upstream and Downstream Changes

Validates that downstream and upstream changes are created at the appropriate times (not creating upstream changesets when **only** downstream changes occurred, and vice versa). However, sync is a bi-directional process, and both Nodes will evaluate what changes need to be merged when a sync occurs. These tests verify the proper creation and use of changesets between two Nodes.

## Sync Versioning

Each Node keeps track of its *SyncReceivedVersion* and *SyncSentVersion*. These two Node properties are used to determine when, and with who, a Node last synced. Additionally, Nodes also track a list of recent committed versions. This list can be used to obtain the previous sync version of a subscriber from a distributor (the distributor being the Node where the changes were made, and the subscriber the Node in need of the changes). The unit tests verify that this list of versions is maintained properly throughout synchronization.

## Sync Scope

These tests validate the correct adherence to *SyncScopes* with respect to the properties of entities such as *TargetItems*, *ResourceItems*, etc. The five different *SyncScopes* are as follows:

- **Public** -> Will Sync
- **Private** -> Will NOT Sync
- **Internal** -> Syncs to internal Nodes only
- **InternalDownStream** -> Syncs to internal children only
- **InternalUpstream** -> Syncs to internal parents only

## Synchronizing Target Data

Validates that *TargetItems* and *TargetGroups* only sync in one of two circumstances - always sync upstream, and only sync downstream when the *NodeId* matches that of the target data.

## Synchronizing Task Assignments

Validates that Tasking data is correctly synced both upstream and downstream. *TaskMaps* will sync both ways as long as the corresponding *TargetItem* syncs. This is the same for *TaskFilters*, *TaskAssignments*, and dependencies. Any changes made to Tasking data, such as updates and deletes, are also synced bi-directionally between Nodes. *TaskItems* with protected properties are also synced, and the protected properties should return an *IsProtected* flag.

## Synchronizing Create, Read, Update, and Delete

Validates the synchronization of CRUD operations across multiple tiers (Master -> Region -> Site 1 & 2).

## Synchronizing Resource Data

Validates the correct synchronization of resource data both upstream and downstream. Resource data, such as *ResourceItems*, always sync upstream. However, *ResourceItems* will only sync downstream if they are marked as Global, or assigned to a target on a child node. The following cases are tested:

- Synchronization of *ResourceItems*, *ResourceGroups*, and *ResourceAssignments* both upstream and downstream when applicable.
- The correct handling of empty *ResourceGroups* (which should not sync).
- Synchronization of new *ResourceItem* properties, as well as deleted properties.

## Conflict Behavior

The following Sync conflicts and corresponding resolutions are tested:

- Duplicate TargetItem name and type should rename source TargetItem
- Duplicate TargetGroup name and type should rename source TargetGroup
- Duplicate ResourceItem name and type should rename source ResourceItem
- Duplicate ResourceGroup name and type should rename source ResourceGroup
- Duplicate TargetItem property name should rename source property item
- Duplicate TargetGroup property name should rename source property item
- Duplicate ResourceItem property name should rename source property item
- Duplicate ResourceGroup property name should rename source property item
- Duplicate TaskItem should rename source item

*The entity is renamed by appending a timestamp to the current name.*

### Synchronizing Logging

Validates the correct synchronization of logs upstream depending on log level (inclusion and exclusion). Logs do not sync downstream.

### Sync Dependency Order

Verifies proper synchronization of dependent CRUD operations (i.e inserting a TargetItem on a Node, and subsequently deleting it). Additionally, these tests also validate that certain operations such as *Delete* are tracked based on the Sync version.

# RPS Patching Script Framework

Last updated on August 26, 2021.

**Document Status:** Document Feature Complete as of August 26, 2021; PENDING EXTERNAL REVIEW.

This article introduces the Rapid Provisioning System (RPS) PowerShell script framework. This provides additional deployment options over the standard deployment features available in the RPS application.

## NOTE

Users may see "patch" and "package" used interchangeably in the code and log outputs during this process.

## Intended Audience

This document is intended for RPS patching roles, Lead Systems Integrators (LSI), Field Service Representatives (FSR), IT staff, and Developers. To use the framework, RPS users will need strong knowledge of PowerShell.

## Use Cases

In many cases, standard RPS patching is not possible, for example:

- Installing patches on appliances where PowerShell cannot directly run.
  - Example: a firewall appliance.
- Installing patches that also need custom pre-steps and post steps.
  - Example: Configure custom registry values.
- Installing patches for a type of patch that does not have an installer.
  - Example: Copying files to a location.
- Installing a patch package on a target system that does not have the Local Configuration Manager (LCM) enabled.
  - Example: Install a patch on a system without the LCM by using script framework and setting property on that target to say `RunPackagesOnTms = $true`.

Now this article can be used to build PowerShell scripts using the required PowerShell functions, introduced below.

## Script Framework Requirements

RPS users with RPS accounts need to:

1. Log into a functioning RPS server.
2. Ensure their user account has the RPS patching role.
3. Launch Windows PowerShell ISE, for example.
4. Build the script needed for the use case.
5. Include the required functions defined in the table below.
6. Run the script.

## PowerShell Functions

Three required RPS-specific PowerShell functions must exist when building the script, as shown in the following table:

FUNCTION NAME	DESCRIPTION	RETURN TYPE
<b>Test- PackageResource</b>	This function tests if the patch is in the desired state.	Must return a boolean. True for in-desired state and false for not-in-desired state.
<b>Set- PackageResource</b>	This function installs or uninstalls.	None
<b>Get- ParameterMapping</b>	This function helps with complex mappings using RPS-Mapped parameters.	Function will return a Hashtable for each custom parameter it needs to map.

Next, the function Get-ParameterMapping is introduced.

### Function: Get-ParameterMapping

```
function Get-ParameterMapping
{
    return @{
        DscEncryptionCertificate = @{
            EntityClass = 'ResourceItem'
            EntityType = 'Certificate'
            Role = 'DscEncryption'
            IsAssigned = $true
        }
    }
}
```

The PowerShell module can include other supporting functions and scripts as required.

### Required Parameters

The **Test-PackageResource** and **Set-PackageResource** functions only have one required parameter, called Ensure. The Ensure parameter states if the patch should be 'Present' or 'Absent'. The two methods can have any other parameters that are required for the custom script to run using RPS-Mapped Parameters.

#### NOTE

For more information on RPS-Mapped Parameters, see the RPS article [How to Configure RPS-Mapped Parameters](#).

### Example Script Framework

The next example now includes the two other functions introduced above to patch VMWare ESXi.

- Test-PackageResource
- Set-PackageResource

```
$null = Import-Module -Name 'VMware.PowerCLI'
$null = Set-PowerCLIConfiguration -InvalidCertificateAction Ignore -Confirm:$false -ErrorAction SilentlyContinue -DefaultVIServerMode Multiple -ParticipateInCEIP $false -Scope Session

function Test-PackageResource
{
    [CmdletBinding()]
    Param
    (
        [Parameter(Mandatory = $true)]
        [ValidateScript({[IPAddress]::Parse($_)}])
        [string]
        $IPAddress,
```

```

[Parameter(Mandatory = $true)]
[string]
$ComputerName,

[Parameter(Mandatory = $true)]
[PSCredential]
$LocalAdmin,

[Parameter(Mandatory = $true)]
[string]
$Ensure
)

```

Write-Verbose "Connecting to ESXi at \$IPAddress"

```
$server = Connect-ViServer -Server $IPAddress -Credential $LocalAdmin -WarningAction SilentlyContinue -ErrorAction Stop
```

```
$virtualSwitches = Get-VirtualSwitch -Name 'PatchedVirtualSwitch' -ErrorAction SilentlyContinue
```

```

if($virtualSwitches)
{
    if ($ensure -eq 'present')
    {
        return $true
    }
    else
    {
        return $false
    }
}

```

```

if ($ensure -eq 'Absent')
{
    return $true
}
return $false

```

Write-Verbose "Connected to ESXi at \$IPAddress"

```
}
```

```
function Set-PackageResource
```

```

{
    [CmdletBinding()]
    Param
    (
        [Parameter(Mandatory = $true)]
        [ValidateScript({[ipaddress]::Parse($_)})]
        [string]
        $IPAddress,

        [Parameter(Mandatory = $true)]
        [string]
        $ComputerName,

        [Parameter(Mandatory = $true)]
        [PSCredential]
        $LocalAdmin,

        [Parameter(Mandatory = $true)]
        [string]
        $Ensure
    )

```

```

$server = Connect-ViServer -Server $IPAddress -Credential $LocalAdmin -WarningAction SilentlyContinue -ErrorAction Stop
if ($ensure -eq 'Present')
{

```

```

    Write-Verbose "Connected to ESXi at $IPAddress"
    $server = Connect-ViServer -Server $IPAddress -Credential $LocalAdmin -WarningAction SilentlyContinue -ErrorAction Stop
    $virtualSwitches = Get-VirtualSwitch -Name 'PatchedVirtualSwitch' -ErrorAction SilentlyContinue
    if($virtualSwitches)
    {
        if ($ensure -eq 'present')
        {
            return $true
        }
        else
        {
            return $false
        }
    }
    if ($ensure -eq 'Absent')
    {
        return $true
    }
    return $false
}

```

```

New-VirtualSwitch -Server $server -Name 'PatchedVirtualSwitch' -ErrorAction SilentlyContinue
}
else
{
    $vs = Get-VirtualSwitch -Server $server -Name 'PatchedVirtualSwitch'
    $vs | Remove-VirtualSwitch -Server $server -ErrorAction SilentlyContinue -Confirm:$false
}
Write-Verbose "Connecting to ESXi at $IPAddress"
}

# Sample parameter mapping. We are not using any complex parameters in this script.
function Get-ParameterMapping
{
    @{
        DscEncryptionCertificate = @{
            EntityClass = 'ResourceItem'
            EntityType = 'Certificate'
            Role = 'DscEncryption'
            IsAssigned = $true
        }
    }
}

```

## Windows Installer Patch (.msp) Example - Deprecated in 4.0

### ⚠ IMPORTANT

This example is deprecated from the RPS 4.0 C:\ContentStore\Packaging folder, but will still exist in older RPS 3.1 servers. For RPS 4.0 servers, build .msp style patches using the RPS application.

### How It Works

The example below will install .msp patches on any RPS Windows target, configured with the patch stream Desired State Configuration (DSC) feed resource.

- The packaged script and .msp will be used to ensure the state of the target.
- The PowerShell script will need to be constructed in a way to be able to test the current state and if necessary, set the configured state of the patch.
- The number of different parameters needed to install the patch determines how complex the packaged script will be.
- For most .msp installs, the only required parameter will be the 'Ensure' property. This parameter will control whether the patch is installed or uninstalled during set operations and used to test the current state for test operations.

### Windows Installer Requirements

For scripted patch streams and patches to execute, clients and targets must be declared, with the RPS feed resource configured through Desired State Configuration (DSC).

### Patch File Components

In order to use this framework to install a .msp style patch, the file layout should include each of the following components:

#### 📌 NOTE

Examples of the Package.RPS (manifest) file and the ExampleMspInstall.ps1 PowerShell script can be found below in this section.

ExampleMspInstall\_1.0.0.zip

1. Package.RPS
2. ExampleMspInstall.ps1
3. ExampleProgram.msp

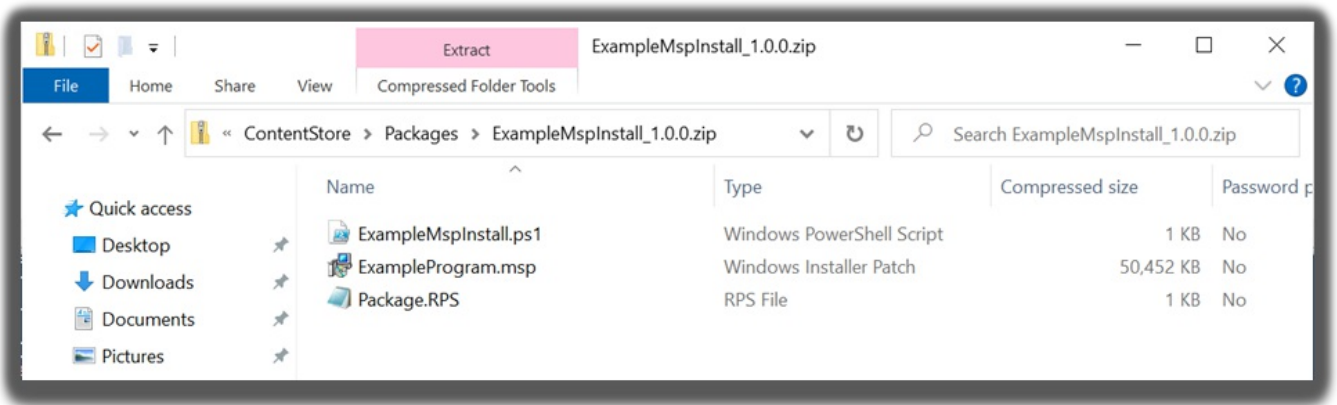


Figure 1: File Explorer view of the contents of an RPS .msp file layout.

### 1. Package.RPS

Use Notepad to create a patch manifest file below with a file name of Package.RPS. The manifest file is used by the script to install the patch. Later in the manifest, at the `ExecutableName` tag, provide a name of .msp to be installed. Both the manifest file and the script below must be zipped.

#### **NOTE**

The `InstallerFileName` should match the PowerShell script file name (ExampleMspInstall.ps1 in our example) and the `Product Type` should be set to "ScriptFramework".

The following is an example of a Package.RPS file that can be modified as required. Notice the use of the RPS cmdlet "Get-RpsAuditEntry" from the RPS-API module.



```
<?xml version="1.0" encoding="utf-8"?>
<PackageManifest xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<PackageName>ExampleMspInstall</PackageName>
<PackageVersion>1.0.0</PackageVersion>
<Description>This is a test patch</Description>
<OsVersion>*</OsVersion>
<Architecture>x64</Architecture>
<OsType>Windows</OsType>
<MsCatalogProductName>ScriptFrameWorkTest1</MsCatalogProductName>
<MsCatalogTitle />
<MsCatalogId />
<Products />
<MsCatalogUpdateId />
<PackageClassification>General</PackageClassification>
<MsCatalogSupcededByKbls />
<MsCatalogLinkUrls />
<UninstallArguments>/s</UninstallArguments>
<InstallArguments />
<SupressReboot>>true</SupressReboot>
<ProductName>ExampleMspInstall</ProductName>
<ProductType>ScriptFramework</ProductType>
<ProductVersion>1.0.0</ProductVersion>
<ProductId>{null}</ProductId>
<InstallerFileName>ExampleMspInstall.ps1</InstallerFileName>
<ExecutableName>ExampleMspInstall.msp</ExecutableName>
</PackageManifest>
```

## 2. ExampleMspInstall.ps1

Use this example PowerShell script to install the .msp style patch using the manifest file defined above.

For the first two variables:

- Ensure that the file names and values match those defined in the manifest file above.
- These values will be used by the script to ensure either the program is installed or uninstalled from the local system.

Also take note of the two functions embedded in the script:

1. Function Set-PackageResource
2. Function Test-PackageResource
3. The Get-ParameterMapping function is for future use and should always return an empty hashtable.

The following script is file-named ExampleMspInstall.ps1 that can be modified for actual use.

```
# Ensure the following two variables match the manifest file.
$Path = "$PSScriptRoot\Package.RPS"
$XPath = "/PackageManifest"
$script:xml = Select-Xml -Path $Path -XPath $XPath

function Set-PackageResource
{
    [CmdletBinding()]
    param
    (
        [Parameter(Mandatory = $true)]
        [ValidateSet('Present','Absent')]
        [System.String]
        $Ensure
    )

    try
    {
        if ($Ensure -eq 'Present')
        ,
    }
}
```

```

    Write-Verbose $('Installing MSP {0}' -f $script:xml.Node.ExecutableName)
    $arguments = '/p "{0}" /quiet /norestart' -f "$PSScriptRoot\$($script:xml.Node.ExecutableName)"
    $result = Invoke-ManagedProcess -Program "$env:winDir\system32\msiexec.exe" -Arguments $arguments
}
else
{
    Write-Verbose $('Uninstalling MSP {0}' -f $script:xml.Node.ExecutableName)
    $packageSettings = Get-Package -Name $script:xml.Node.ProductName -RequiredVersion $script:xml.Node.ProductVersion -
ProviderName msi,programs
    $packageSettings = $packageSettings | Select-Object -First 1
    $result = Invoke-ManagedProcess -Program $packageSettings.Metadata.Item('UninstallString') -Arguments '/s'
}
}
}
catch
{
    Write-Warning "$_"
}
}

function Test-PackageResource
{
    [CmdletBinding()]
    param
    (
        [Parameter(Mandatory = $true)]
        [ValidateSet('Present','Absent')]
        [System.String]
        $Ensure
    )

    try
    {
        Write-Verbose $('Testing state of MSP {0}' -f $script:xml.Node.ExecutableName)
        $params = @{
            Name = $script:xml.Node.ProductName
            RequiredVersion = $script:xml.Node.ProductVersion
            ProviderName = 'msi','Programs'
            ErrorAction = 'SilentlyContinue'
        }
        $packageResult = Get-Package @params

        if ($packageResult -and $Ensure -eq 'Present')
        {
            return $true
        }
        else
        {
            return $false
        }
    }
    catch
    {
        return $false
    }
}

function Get-ParameterMapping
{
    return @{}
}

```

## Installing Multiple Windows Hotfix (MSU) Files: Example

## How It Works

The example below will install multiple .msu (hotfix) files on RPS Windows target(s).

- The PowerShell script.ps1 and .msu (hotfix) files will be used to ensure the state of each target.
- The PowerShell script.ps1 file will need to be constructed in a way to be able to test the current state and, if necessary, set the configured state of the patch.

The example consists of two pieces:

- An example Package.RPS manifest file that describes the Script Framework script that will kick off the patching process.
- An example script.ps1 PowerShell script that provides an example Script Framework script that installs multiple .msu (hotfix) files.

### **NOTE**

Both of the above example files, the Package.RPS manifest file and the script.ps1 PowerShell script, will need to be modified to user requirements of the specific patching scenario.

- The Package.RPS manifest should be modified so that it provides correct information about the bundle of .msu (hotfix) files that will be contained in the patch.
- The script.ps1 PowerShell script should be modified by changing the value of the \$patches variable at the top of the script to contain correct information about the .msu (hotfix) files that will be deployed. This includes correct information such as: ProductId (the KB number), Filepath (path to the msu within the patch ZIP - if it is at the root of the patch ZIP then the filename of the msu will suffice), and Ensure (Present or Absent, which designates whether the msu should be installed or removed).

After the Package.RPS manifest file and the script.ps1 have been modified and verified as correct, create a ZIP archive that contains:

- The Package.RPS manifest file
- The .msu (hotfix) files
- the script.ps1 PowerShell file

The ZIP file should follow the naming convention based on the values in the Package.RPS manifest file:

`<ProductName><ProductVersion>.zip`. For example: "MyWindowsUpdates1.0.0.zip".

## Windows Installer Requirements

For scripted patch streams and patches to execute, clients and targets must be declared, with the RPS feed resource configured through Desired State Configuration (DSC).

## Patch Manifest - Package.RPS

```

<?xml version="1.0" encoding="utf-8"?>
<PackageManifest version="1.0">
  <PackageName>MultipleMSU</PackageName>
  <PackageVersion>1.0.0</PackageVersion>
  <Description>This example patch will install multiple MSU files.</Description>
  <OsVersion>*</OsVersion>
  <Architecture>*</Architecture>
  <OsType>Windows</OsType>
  <PackageClassification>General</PackageClassification>
  <UninstallArguments />
  <InstallArguments />
  <SuppressReboot>>true</SuppressReboot>
  <ProductName />
  <ProductType>ScriptFramework</ProductType>
  <ProductVersion />
  <ProductId />
  <InstallerFileName>multiple_msu.ps1</InstallerFileName>
</PackageManifest>

```

### Script Framework Script - script.ps1

```

$patches = @(
  @{
    ProductId = "KB4601050" #Product ID (KB number) of the MSU/Hotfix
    Filepath = Join-Path -Path $PSScriptRoot -ChildPath "KB4601050.msu" #fully qualified path to the MSU/Hotfix file
    Ensure = "Present" #Present or Absent
  },
  @{
    ProductId = "KB4601051" #Product ID (KB number) of the MSU/Hotfix
    Filepath = Join-Path -Path $PSScriptRoot -ChildPath "KB4601051.msu" #fully qualified path to the MSU/Hotfix file
    Ensure = "Present" #Present or Absent
  },
  @{
    ProductId = "KB4601052" #Product ID (KB number) of the MSU/Hotfix
    Filepath = Join-Path -Path $PSScriptRoot -ChildPath "KB4601052.msu" #fully qualified path to the MSU/Hotfix file
    Ensure = "Present" #Present or Absent
  }
)

function Test-PatchResource
{
  [CmdletBinding()]
  param()

  Write-Verbose -Message 'Started testing patches'

  $result = $true
  foreach($patch in $patches)
  {
    Write-Verbose -Message "Testing state for patch $($patch.ProductId)"

    $patchId = $patch.ProductId.ToLower().Replace("kb", "")
    $hotfix = Get-Hotfix -Id $patchId -ErrorAction SilentlyContinue

    if ($patch.Ensure -eq 'Present' -and $null -eq $hotfix)
    {
      Write-Verbose -Message "Patch $($patch.ProductId) is not in desired state. Patch is not installed"
      $result = $false
    }
    elseif ($patch.Ensure -eq 'Absent' -and $null -ne $hotfix)
    {
      Write-Verbose -Message "Patch $($patch.ProductId) is not in desired state. Patch is installed"
      $result = $false
    }
  }
  else

```

```

    }
    Write-Verbose -Message "$($patch.ProductId) is in desired state."
  }
}

Write-Verbose -Message 'Finished testing patches'

return $result
}

function Set-PatchResource
{
  [CmdletBinding()]
  Param
  (
  )

  Write-Verbose 'Started installing patches'

  foreach($patch in $patches)
  {
    $hotfix = Get-Hotfix -Id $patch.ProductId -ErrorAction SilentlyContinue

    if ($patch.Ensure -eq 'Present' -and $null -eq $hotfix)
    {
      Write-Verbose -Message "Installing $($patch.ProductId)"

      $arguments = "{0} /quiet /norestart" -f $patch.Filepath
      $result = Invoke-ManagedProcess -Program "$env:winDir\system32\wusa.exe" -Arguments $arguments

      if ($result -eq 0)
      {
        Write-Verbose -Message "Finished installing $($patch.ProductId)"
      }
      else
      {
        Write-Verbose -Message "Error installing $($patch.ProductId)"
      }
    }
    elseif ($patch.Ensure -eq 'Absent' -and $null -ne $hotfix)
    {
      Write-Verbose -Message "Uninstalling $($patch.ProductId)"

      $updateId = $($patch.ProductId) -ireplace [regex]::Escape('KB'), ""
      $arguments = '/uninstall /KB:{0} /quiet /norestart' -f $updateId
      $result = Invoke-ManagedProcess -Program "$env:winDir\system32\wusa.exe" -Arguments $arguments

      if ($result -eq 0)
      {
        Write-Verbose -Message "Finished uninstalling $($patch.ProductId)"
      }
      else
      {
        Write-Verbose -Message "Error uninstalling $($patch.ProductId)"
      }
    }
    else
    {
      Write-Verbose -Message "$($patch.ProductId) is in desired state."
    }
  }

  Write-Verbose 'Finished installing patches'
}

```

```

function Invoke-ManagedProcess
{
    [Cmdletbinding()]
    [OutputType([System.UInt32])]
    param
    (
        [Parameter(Mandatory = $true)]
        [System.String]
        $Program,

        [Parameter()]
        [System.String]
        $Arguments = "",

        [Parameter()]
        [System.UInt16]
        $IdleTimeout = 60
    )

    $processInfo = New-Object System.Diagnostics.ProcessStartInfo
    $processInfo.FileName = $Program
    $processInfo.RedirectStandardError = $true
    $processInfo.RedirectStandardOutput = $true
    $processInfo.UseShellExecute = $false
    $processInfo.Arguments = $Arguments
    $managedProcess = New-Object System.Diagnostics.Process
    $managedProcess.StartInfo = $processInfo
    $managedProcess.Start() | Out-Null

    while (Get-Process -Id $managedProcess.ID -ErrorAction SilentlyContinue)
    {
        Test-TaskTimeOut -Process $managedProcess -IdleTimeout $IdleTimeout
    }

    return $managedProcess.ExitCode
}

function Test-TaskTimeOut
{
    [CmdletBinding()]
    param
    (
        [Parameter(Mandatory = $true)]
        [PSObject]
        $Process,

        [Parameter()]
        [System.UInt16]
        $IdleTimeOut = 60
    )

    if ($null -eq $memUsageStack)
    {
        $script:memUsageStack = New-Object -TypeName System.Collections.Stack
    }

    if ($IdleTimeout -gt 0)
    {
        $lastMemUsageCount = Get-ProcessTreeMemoryUsage -ProcessId $Process.ID
        $memUsageStack.Push($lastMemUsageCount)
        if ($lastMemUsageCount -eq 0 -or ($null -ne ($memUsageStack.ToArray() | Where-Object -FilterScript { $_ -ne $lastMemUsageCount
    })))
    {
        if (-not (Get-Process -Id $Process.ID -ErrorAction SilentlyContinue))
        {
            break
        }
    }
}

```

```

        break
    }

    $memUsageStack.Clear()
}
if ($memUsageStack.Count -gt $IdleTimeOut)
{
    Stop-Process -Id $Process.ID
}
}

Start-Sleep -Second 1
}

function Get-ProcessTreeMemoryUsage
{
    [CmdletBinding()]
    [OutputType([System.UInt64])]
    param
    (
        [Parameter(Mandatory = $true)]
        [System.UInt32]
        $ProcessId
    )

    $ReservedMemory = 0

    $childProcessObject = Get-CimInstance Win32_Process -Filter "ParentProcessID=$ProcessId" -Verbose:$false
    if ($childProcessObject)
    {
        foreach ($processObject in $childProcessObject)
        {
            if ($null -ne $processObject.ProcessID)
            {
                $currentProcess = Get-Process -ID $processObject.ProcessID -ErrorAction SilentlyContinue
                $ReservedMemory += $currentProcess.PrivateMemorySize + $currentProcess.WorkingSet
                $ReservedMemory += (Get-ProcessTreeMemoryUsage -ProcessId $processObject.ProcessID)
            }
        }
    }
    else
    {
        $currentProcess = Get-Process -ID $ProcessId -ErrorAction SilentlyContinue

        if ($currentProcess)
        {
            $ReservedMemory += $currentProcess.PrivateMemorySize + $currentProcess.WorkingSet
        }
        else
        {
            $ReservedMemory = 0
        }
    }

    return $ReservedMemory
}

```

# RPS Patch Manifest Definition

Last updated on August 11, 2021.

**Document Status:** Document Feature Complete as of August 11, 2021; PENDING EXTERNAL REVIEW.

This document describes the patch manifest of the Rapid Provisioning System (RPS).

## NOTE

The word "package" is used instead of "patch" within the manifest XML to maintain backwards compatibility with RPS v3.1.

Users may see "patch" and "package" used interchangeably in the code and log outputs during this process.

## What is the Patch Manifest File

When creating the ZIP file that houses all the data needed to install a patch, there also needs to be a patch manifest file created with the patch. It is used by RPS to apply the patches and to determine what targets are to receive the patches.

The manifest file lives at the root level of the ZIP archive and is named *Package.RPS*. It contains information such as the Operating System (OS) Type, OS Architecture, patch name, etc.

## IMPORTANT

Patch names **must** begin with a letter to be valid. They cannot start with a number or special character.

## Structure of the Patch Manifest File

The internal structure of the patch manifest file is JSON with XML, containing a predetermined set of elements. The values contained in these elements will be the details applied to the patch and used to transfer, apply, and manage the patches on an RPS enabled system.

## IMPORTANT

Any XML element that has a `String` field must have the first letter of "String" capitalized for this field to be valid. Example:

```
<String>Windows 10</String>
```



```
<?xml version="1.0" encoding="utf-8"?>
<PackageManifest version="1.0">
  <Architecture></Architecture>
  <Conditions>
    <PackageAssignmentCondition>
      <Property />
      <Operator />
      <Value />
    </PackageAssignmentCondition>
  </Conditions>
  <DependsOn />
  <Description></Description>
  <InstallArguments />
  <InstallerFileName></InstallerFileName>
  <Miscellaneous />
  <MsCatalogId />
  <MsCatalogLinkUrls />
  <MsCatalogProductName></MsCatalogProductName>
  <MsCatalogSupersededByKblids />
  <MsCatalogTitle />
  <MsCatalogUpdateId />
  <OsType></OsType>
  <OsVersion></OsVersion>
  <PackageClassification></PackageClassification>
  <PackageName></PackageName>
  <PackageVersion></PackageVersion>
  <ProductId></ProductId>
  <ProductName></ProductName>
  <ProductType></ProductType>
  <ProductVersion></ProductVersion>
  <Products />
  <Supersedes>
    <String>PackageName/PackageVersion</String>
  </Supersedes>
  <SuppressReboot></SuppressReboot>
  <UninstallArguments></UninstallArguments>
</PackageManifest>
```

## Patch Manifest XML Attributes

*Version* Attribute

**Type:** String

**IsRequired:** Yes

The version of the patch manager that will process this manifest file.

Currently, the following version(s) exist:

- 1.0

```
<PatchManifest version="1.0">
```

## Patch Manifest XML Elements

*Architecture* Element

**Type:** String

**IsRequired:** Yes

The architecture of the operating system to whom the patch is applicable. A wildcard (\*) can be passed in to target **any** OS version. Acceptable values are:

- x86
- x64
- \*

#### **NOTE**

The patch will only be assigned to targets that have an `Architecture` value that passes the match test on this element, in addition to the `OsType` and `OsVersion` values.

### Conditions Element

**Type:** Collection of `PackageAssignmentCondition`

**IsRequired:** No

A collection of custom conditions for filtering patch assignments to a target.

```
<Conditions>
  <PackageAssignmentCondition>
    <Property>IsApp</Property>
    <Operator>Eq</Operator>
    <Value>True</Value>
  </PackageAssignmentCondition>
</Conditions>
```

#### **IMPORTANT**

The fields inside the `PackageAssignmentCondition` must be in the following order to be valid:

```
<Property></Property>
<Operator></Operator>
<Value></Value>
```

### Known Issue

Patch manifest `Conditions` element `Value` field does not support multiple values separated by the pipe delimiter |

- **Error Details:** The following PackageManifest code snippet is an example using pipe delimiter | in `Conditions`, which will fail:

```
<InstallerFileName>opera.msi</InstallerFileName>
<Conditions>
  <PackageAssignmentCondition>
    <Property>Name</Property>
    <Operator>Eq</Operator>
    <Value>ad.unit.domain|nosc.local.rps</Value>
  </PackageAssignmentCondition>
</Conditions>
```

**The resulting behavior:** Only the first value listed in the `Value` field will receive an assignment; all other values after the pipe delimiter | are ignored.

```

PS C:\Windows\system32> Get-RpsResourceAssignment -ResourceItemType 'Package' -TargetItem (Get-RpsTargetItem -Name 'ad.unit.domain' -Type 'virtualmachine') | ft ResourceItem
ResourceItem
-----
Package - Opera/70.0.0
Package - Firefox/70.0.0
Package - AdobeReaderDC/19.12.20034
Package - x64-Windows8.1-KB4486105-x64/4486105.0.0
Package - windows8.1-kb4519990-x64/2019.10.8

PS C:\Windows\system32> Get-RpsResourceAssignment -ResourceItemType 'Package' -TargetItem (Get-RpsTargetItem -Name 'nosc.rps.local' -Type 'virtualmachine') | ft ResourceItem
ResourceItem
-----
Package - Firefox/70.0.0
Package - x64-Windows8.1-KB4486105-x64/4486105.0.0
Package - windows8.1-kb4519990-x64/2019.10.8
Package - AdobeReaderDC/19.12.20034

```

Figure 1: Pipe delimiter error example.

In the above example, ad.unit.domain is assigned the *opera* patch, because it was listed before the pipe delimiter |. nosc.local.rps is not assigned the *opera* patch, because it was listed after the pipe delimiter |.

- **Current Workaround for pipe delimiter |** : Utilize the *Match* Operator field `<Operator>Match</Operator>`, with each value in the *Value* field wrapped in parentheses ( ) and with a trailing question mark ?. Example:

```

<Conditions>
  <PackageAssignmentCondition>
    <Property>ComputerName</Property>
    <Operator>Match</Operator>
    <Value>(NFA)?(WNM)?(WNMA)?</Value>
  </PackageAssignmentCondition>
</Conditions>

```

In the above example, a target with a *Property* of *ComputerName* will be assigned if its *Value* contains *NFA*, *WNM*, **and/or** *WNMA*. This implementation only requires a **partial** value match.

For an **exact** value match, the full string in the *Value* field must be enclosed with a caret ^ and a dollar sign \$ . Example:

```

<Conditions>
  <PackageAssignmentCondition>
    <Property>ComputerName</Property>
    <Operator>Match</Operator>
    <Value>^(NFA)?(WNM)?(WNMA)?$</Value>
  </PackageAssignmentCondition>
</Conditions>

```

In the above example, a target with a *Property* of *ComputerName* will be assigned if its *Value* contains *NFA*, *WNM*, **and** *WNMA*.

### DependsOn Element

**Type:** Collection of Strings

**IsRequired:** No

A collection of products a patch depends on. The value is in the format of *ProductName/ProductVersion*.

```

<DependsOn>
  <String>Patch1/1.0.0</String>
  <String>Patch2/1.5.2</String>
</DependsOn>

```

### Description Element

**Type:** String

**IsRequired:** No

A description of the patch and what it is installing.

#### *InstallArguments* Element

**Type:** String

**IsRequired:** No

Arguments passed in when executing the [InstallerFileName](#) in order to install the patch.

#### *InstallerFileName* Element

**Type:** String

**IsRequired:** Yes

The name of the file inside of the patch ZIP archive that should be executed to begin the install/uninstall process. This file must live at the root of the patch.

#### *Miscellaneous* Element

**Type:** Collection of Strings

**IsRequired:** No

Allows extra information to be stored in a patch manifest. Any child element(s) can be added to the `Miscellaneous` element.

```
<Miscellaneous>  
  <CreatedByUser>myUserName</CreatedByUser>  
  <CreatedOnDate>4/21/2020 12:00:00 PM</CreatedOnDate>  
</Miscellaneous>
```

#### *MsCatalogId* Element

**Type:** String

**IsRequired:** No

The catalog ID of the patch. Used/Populated by updates that come from the Microsoft Catalog.

#### *MsCatalogLinkUrls* Element

**Type:** Collection of Strings

**IsRequired:** No

The catalog link URLs for the patch. Used/Populated by updates that come from the Microsoft Catalog.

```
<MsCatalogLinkUrls>  
  <String>https://www.catalog.update.microsoft.com/ScopedViewInline.aspx?updateid=db57861b-e22b-4107-8c78-1ae8d63310d2</String>  
  <String>https://www.catalog.update.microsoft.com/ScopedViewInline.aspx?updateid=567d7a0d-7f11-4c75-ba80-e7dd1b88fbe3</String>  
  <String>https://www.catalog.update.microsoft.com/ScopedViewInline.aspx?updateid=5f46c0b9-c57c-484e-b6e5-80dded34bfa3</String>  
</MsCatalogLinkUrls>
```

#### *MsCatalogProductName* Element

**Type:** String

**IsRequired:** No

The catalog product name of the patch. Used/Populated by updates that come from the Microsoft Catalog.

#### *MsCatalogSupersededByKblids* Element

**Type:** Collection of Strings

**IsRequired:** No

The catalog IDs of KBs that supersede the patch. Used/Populated by updates that come from the Microsoft Catalog.

```
<MsCatalogSupercededByKblds>  
<String>4532693</String>  
<String>4532695</String>  
<String>4528760</String>  
</MsCatalogSupercededByKblds>
```

*MsCatalogTitle* Element

**Type:** String

**IsRequired:** No

The catalog title of the patch. Used/Populated by updates that come from the Microsoft Catalog.

*MsCatalogUpdateId* Element

**Type:** String

**IsRequired:** No

The catalog update ID of the patch. Used/Populated by updates that come from the Microsoft Catalog.

*OsType* Element

**Type:** String

**IsRequired:** Yes

The type of the operating system. The supported values are:

- Windows
- Linux

#### **NOTE**

The patch will only be assigned to targets that have an `OsType` value that passes the match test on this element, in addition to the `Architecture` and `OsVersion` values.

*OsVersion* Element

**Type:** String

**IsRequired:** Yes

The version of the operating system to whom the patch is applicable. A wildcard (\*) can be passed in to target **any** OS version. Partial wildcard matching is also allowed, which means it will match the values before and after the wildcard.

For example:

- `10.*` would match 10.0.0.2, 10.2, etc., because they all start with "10."
- `*.1` would match 10.0.1, 10.1, etc because they all end with ".1".

#### **NOTE**

The patch will only be assigned to targets who have an `OsVersion` value that passes the match test on this element, in addition to the `Architecture` and `OsType` values.

### *PackageClassification* Element

**Type:** String

**IsRequired:** Yes

The classification of the patch. Accepted values are:

- Critical
- Definition
- General
- New
- Script
- Security

### *PackageName* Element

**Type:** String

**IsRequired:** Yes

The name of the patch to be used by RPS. This name is used in combination with the [PackageVersion element](#) to create a unique name.

### *PackageVersion* Element

**Type:** String

**IsRequired:** Yes

The version of the patch to be used by RPS. This name is used in combination with the [PackageName element](#) to create a unique name. The value stored in here typically follows Semantic Versioning (e.g., 1.0.0).

### *ProductId* Element

**Type:** String

**IsRequired:** Yes, if [ProductType](#) is *WindowsCabinet*, *WindowsExe*, *WindowsHotfix*, *WindowsMsi*, or *WindowsMsp*.

The ID of the product. Used when registering the product with the OS, if needed.

### *ProductName* Element

**Type:** String

**IsRequired:** Yes, if [ProductType](#) is *WindowsExe*, *WindowsMsi*, or *WindowsMsp*.

The name of the product. Used when registering the product with the OS, if needed.

### *ProductType* Element

**Type:** String

**IsRequired:** Yes

The type of product. Determines the type of patch being installed.

Acceptable values are:

- LinuxRpm
- ScriptFramework
- WindowsCabinet
- WindowsExe
- WindowsHotfix
- WindowsMsi
- WindowsMsp

#### *ProductVersion* Element

**Type:** String

**IsRequired:** Yes, if [ProductType](#) is *WindowsExe*, *WindowsMsi*, or *WindowsMsp*.

The Version of the product. Used when registering the patch with the OS, if needed.

#### *Products* Element

**Type:** Collection of Strings

**IsRequired:** No

A collection of products a patch affects/supports. For example, a Windows update might patch Windows 10 and Visual Studio.

```
<Products>
  <String>Windows 10</String>
  <String>Visual Studio</String>
</Products>
```

#### *Supersedes* Element

**Type:** Collection of Strings

**IsRequired:** No

A collection of products a patch supersedes. The value is in the format of *ProductName/ProductVersion*.

```
<Supersedes>
  <String>Patch1/1.0.0</String>
  <String>Patch2/1.5.2</String>
</Supersedes>
```

#### *SuppressReboot* Element

**Type:** Boolean (True/False)

**IsRequired:** Yes

Determines whether or not the machine should reboot after installing/uninstalling the patch.

If *True*, the machine should **not** reboot. If *False*, the machine should reboot.

#### *UninstallArguments* Element

**Type:** String

**IsRequired:** No

Arguments passed in when executing the [InstallerFileName](#) in order to uninstall the patch.

## PackageManifest Examples

### 3<sup>rd</sup> Party Application Patch Manifest

Example of a manifest used to install Firefox to a Windows machine:

```
<?xml version="1.0" encoding="utf-8"?>
<PackageManifest version="1.0">
  <Architecture>x64</Architecture>
  <Description>This will install Firefox v70 on all Windows x64 bit machines</Description>
  <InstallArguments />
  <InstallerFileName>firefox.msi</InstallerFileName>
  <MsCatalogId />
  <MsCatalogLinkUrls />
  <MsCatalogProductName>Firefox70</MsCatalogProductName>
  <MsCatalogSupersededByKblds />
  <MsCatalogTitle />
  <MsCatalogUpdateId />
  <OsType>Windows</OsType>
  <OsVersion>*</OsVersion>
  <PackageClassification>General</PackageClassification>
  <PackageName>Firefox</PackageName>
  <PackageVersion>1.0.0</PackageVersion>
  <ProductId>{74994757-3b19-4c54-afe4-ae84e398a3f7}</ProductId>
  <ProductName>Mozilla Firefox 70.0 (x64 en-US)</ProductName>
  <ProductType>WindowsMsi</ProductType>
  <ProductVersion>70.0</ProductVersion>
  <Products />
  <Supersedes>
    <String>Firefox/0.9.0</String>
  </Supersedes>
  <SuppressReboot>true</SuppressReboot>
  <UninstallArguments>/s</UninstallArguments>
</PackageManifest>
```

### Linux Software Patch Manifest

Example of a manifest used to install Socat to a Linux machine:



```
<?xml version="1.0" encoding="utf-8"?>
<PackageManifest version="1.0">
  <Architecture>x64</Architecture>
  <Description>This contains the install for Socat</Description>
  <InstallArguments />
  <InstallerFileName>socat-1.7.3.2-2.el7.x86_64.rpm</InstallerFileName>
  <OsType>Linux</OsType>
  <OsVersion>*</OsVersion>
  <PackageClassification>General</PackageClassification>
  <PackageName>socat</PackageName>
  <PackageVersion>1.7.3.2</PackageVersion>
  <ProductId>socat</ProductId>
  <ProductName>socat</ProductName>
  <ProductType>LinuxRpm</ProductType>
  <ProductVersion>1.7.3.2</ProductVersion>
  <Products />
  <UninstallArguments>/s</UninstallArguments>
  <SuppressReboot>>true</SuppressReboot>
</PackageManifest>
```

## Appliance Patch Manifest

Example of a manifest used to update an ESX environment:

```
<?xml version="1.0" encoding="utf-8"?>
<PackageManifest version="1.0">
  <Architecture>x64</Architecture>
  <Description>This is a test patch</Description>
  <InstallArguments />
  <InstallerFileName>windows8.1-kb4519990-x64.msu</InstallerFileName>
  <MsCatalogId />
  <MsCatalogLinkUrls />
  <MsCatalogSupercededByKblids />
  <MsCatalogProductName>kb4519990</MsCatalogProductName>
  <MsCatalogTitle />
  <MsCatalogUpdateId />
  <OsType>Windows</OsType>
  <OsVersion>*</OsVersion>
  <PackageClassification>General</PackageClassification>
  <PackageName>windows8.1-kb4519990-x64</PackageName>
  <PackageVersion>2019.10.8</PackageVersion>
  <ProductId>kb4519990</ProductId>
  <ProductName>windows8.1-kb4519990-x64</ProductName>
  <ProductType>WindowsHotfix</ProductType>
  <ProductVersion>2019.10.8</ProductVersion>
  <Products />
  <SuppressReboot>>false</SuppressReboot>
  <UninstallArguments>/s</UninstallArguments>
</PackageManifest>
```

## Get-RpsPatchManifest

The cmdlet `Get-RpsPatchManifest` will return the XML schema for the patch manifest file. The cmdlet takes one optional parameter:

`Version`.

- `Version`: Optional. Used to specify a patch manifest version to validate against. If this parameter is not provided, this cmdlet will return the most current version of the manifest schema.

### **NOTE**

Currently the only valid value for `Version` is `1.0`.

```
$currentSchema = Get-RpsPatchManifest
```

## Test-RpsPatchManifest

The cmdlet `Test-RpsPatchManifest` will take the path to a patch manifest file and validate the file XML format. The cmdlet takes two parameters: `PatchFilePath` and, optionally, `SchemaVersion`.

- `PatchFilePath`: Required. Provides the path to the patch manifest file to be validated.
- `SchemaVersion`: Optional. Used to specify a patch manifest version to validate against. If this parameter is not provided, the latest version of the manifest schema will be used to perform the validation.

### NOTE

Currently the only valid value for `SchemaVersion` is `1.0`.

If the manifest is valid, `Test-RpsPatchManifest` will return true. If the manifest is invalid, `Test-RpsPatchManifest` will return false, with a list of validation exceptions.

```
$result = Test-RpsPatchManifest -PatchFilePath C:\patch\manifest.xml
```

# New Configurations for CDN

Last updated on September 1, 2021.

**Document Status:** Document Feature Complete as of September 1, 2021; PENDING EXTERNAL REVIEW.

## Overview

The following document describes the configuration changes from RPS v3.1 to v4.0.0 and provides a PowerShell script which can be executed by the LSI when implementing a new RPS deployment.

## Script

```
# We no longer need DSC resource Rps_xDFSR
$contentStoreBasePath = <Content Store Base Path>
Remove-Item (Join-Path $contentStoreBasePath -ChildPath "DSC\Modules\RPS_xDFSR") -Recurse

# The 'WebApiServiceAccount' needs to belong to the 'DFSRAadministrators' group
Add-ADGroupMember -Identity DFSRAadministrators -Members WebApiServiceAccount

# New Active Directory configurations for 'DFSRAadministrators' group
$domain = Get-RpsResourceItem -Type ADDomain -Name <domain name>
$accessEntryName = "CN=Computers,$($domain.Path)"

$accessEntry = New-RpsResourceItem -Type $Rps.ResourceTypes.ADAccessEntry -Name $accessEntryName -Properties @{
    DistinguishedName = $accessEntryName
}

$accessControlList = New-RpsResourceItem -Type $Rps.ResourceTypes.ADAccessControlList -Name 'CN=Computers-
AdAccessControlList1' -Parent $accessEntry -Properties @{
    Principal = 'DFSRAadministrators'
}

$null = New-RpsResourceItem -Type $Rps.ResourceTypes.ADAccessRule -Name 'CN=Computers-ADAccessRule3' -Parent
$accessControlList -Properties @{
    AccessControlType = 'Allow'
    ActiveDirectoryRights = 'GenericAll'
    InheritanceType = 'Descendants'
    InheritedObjectType = 'ms-DFSR-LocalSettings'
    ObjectType = ""
    Ensure = 'Present'
}

$null = New-RpsResourceItem -Type $Rps.ResourceTypes.ADAccessRule -Name 'CN=Computers-ADAccessRule4' -Parent
$accessControlList -Properties @{
    AccessControlType = 'Allow'
    ActiveDirectoryRights = 'CreateChild,DeleteChild'
    InheritanceType = 'Descendants'
    InheritedObjectType = 'Computer'
    ObjectType = 'ms-DFSR-LocalSettings'
    Ensure = 'Present'
}
```

### NOTE

The DFSRAdmin user account is no longer required and can be removed from the CMDB.

# RPS Package Provider

Last updated on April 30, 2021.

Last Reviewed and Approved on PENDING REVIEW

The RPS Package Provider is a PowerShell module that provides RPS with the ability to Connect and find, download, test, and install patches.

How the Package Provider tests or installs a patch depends on the patch's product type. For more information on the product types see: [RPS Patch Product Types](#)

## Table of Contents

- [WindowsHotfix](#)
- [WindowsMsi](#)
- [WindowsMSP](#)
- [WindowsCabinet](#)
- [WindowsExe](#)
- [LinuxRpm](#)
- [ScriptFramework](#)

## WindowsHotfix

To **test** if a WindowsHotfix patch is installed the following command is used by the Package Provider:

```
Get-Hotfix -Id <ProductId> -ErrorAction SilentlyContinue
```

- ProductId - The product id from the patch manifest, which is the unique identifier of the product.

To **install** a WindowsHotfix patch the following command is used by the Package Provider:

```
wusa.exe "<Path to installer executable>" /quiet /norestart
```

- Path to installer executable - this is the path to the actual file that kicks off the install process.
  - For example: C:\packages\MyPatch1.0.0\installer.exe

## WindowsMsi

To **test** if a WindowsMsi patch is installed the following command is used by the Package Provider:

```
$params = @{  
    Name          = "<ProductName>"  
    RequiredVersion = "<ProductVersion>"  
    ProviderName  = 'programs', 'msi'  
    ErrorAction   = 'SilentlyContinue'  
}  
Get-Package @params
```

- ProductName - The product name from the patch manifest that identifies the name of the installed product.
- ProductVersion - The product version from the patch manifest that identifies the version of the installed product.

To **install** a WindowsMsi patch the following command is used by the Package Provider:

```
msiexec.exe /i "<Path to installer msi>" /quiet /norestart
```

- Path to installer msi - this is the path to the actual file that kicks off the install process
  - For example: C:\packages\MyPatch1.0.0\installer.msi

## WindowsMsp

To **test** if a WindowsMsp patch is installed the following command is used by the Package Provider:

```
$params = @{
    Name      = "<ProductName>"
    RequiredVersion = "<ProductVersion>"
    ProviderName = 'programs', 'msi'
    ErrorAction = 'SilentlyContinue'
}
Get-Package @params
```

- ProductName - The product name from the patch manifest that identifies the name of the installed product.
- ProductVersion - The product version from the patch manifest that identifies the version of the installed product.

To **install** a WindowsMsp patch the following command is used by the Package Provider:

```
msiexec.exe /p "<Path to installer msp>" /quiet /norestart
```

- Path to installer msp - this is the path to the actual file that kicks off the install process
  - For example: C:\packages\MyPatch1.0.0\patchInstaller.msp

## WindowsCabinet

### NOTE

The RPS Package Provider is only compatible with Windows KB (Knowledge Base) updates that have the CAB file format. Other CAB files may not work using this method. You can test if a CAB file is compatible by using the Dism command explained below to see if its compatible with the RPS Package Provider. If they are not compatible then they may be able to be installed using the script framework and a custom PowerShell script that can install the file.

To **test** if a WindowsCabinet patch is installed the following command is used by the Package Provider:

```
Get-Hotfix -Id <ProductId> -ErrorAction SilentlyContinue
```

- ProductId - The product id from the patch manifest, which is the unique identifier of the product.

To **install** a WindowsCabinet patch the following command is used by the Package Provider:

```
Dism\Add-WindowsPackage -PackagePath <Path to cab archive> -Online -NoRestart
```

- Path to cab archive - this is the path to the actual file for the cab archive
  - for example: C:\packages\MyPatch1.0.0\cabArchive.cab

## WindowsExe

To **test** if a WindowsExe patch is installed use following script which is used internally by the Package Provider:

```

$Name = "<ProductName>"
$IdentifyingNumber = "<ProductId>"
$Version = "<ProductVersion>"

$uninstallRegistryKey = 'HKLM:\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall'
$uninstallRegistryKeyWow64 = 'HKLM:\SOFTWARE\Wow6432Node\Microsoft\Windows\CurrentVersion\Uninstall'

$productEntry = $null
$productEntryPath = $null

if ([String]::IsNullOrEmpty($Version) -or ([String]::IsNullOrEmpty($IdentifyingNumber) -and [String]::IsNullOrEmpty($Name)))
{
    return $productEntry
}

if (-not [String]::IsNullOrEmpty($IdentifyingNumber))
{
    $productEntryKeyLocation = Join-Path -Path $uninstallRegistryKey -ChildPath $IdentifyingNumber
    $productEntryPath = Get-Item -Path $productEntryKeyLocation -ErrorAction 'SilentlyContinue'

    if ($null -eq $productEntryPath)
    {
        $productEntryKeyLocation = Join-Path -Path $uninstallRegistryKeyWow64 -ChildPath $IdentifyingNumber
        $productEntryPath = Get-Item -Path $productEntryKeyLocation -ErrorAction 'SilentlyContinue'
    }

    if ($productEntryPath -and $Version -eq (Get-LocalizedRegistryKeyValue -RegistryKey $productEntryPath -ValueName 'DisplayVersion'))
    {
        Write-Host "True"
    }
}
else
{
    foreach ($registryKeyEntry in (Get-ChildItem -Path @( $uninstallRegistryKey, $uninstallRegistryKeyWow64) -ErrorAction 'Ignore' ))
    {
        if ($Name -eq (Get-LocalizedRegistryKeyValue -RegistryKey $registryKeyEntry -ValueName 'DisplayName') -and
            $Version -eq (Get-LocalizedRegistryKeyValue -RegistryKey $registryKeyEntry -ValueName 'DisplayVersion'))
        {
            Write-Host "True"
        }
    }
}

Write-Host "False"

```

- ProductName - The product name from the patch manifest that identifies the name of the installed product.
- ProductVersion - The product version from the patch manifest that identifies the version of the installed product.
- ProductId - The product id from the patch manifest, which is the unique identifier of the product.

To **install** a WindowsExe patch the following command is used by the Package Provider:

```
<Path to executable> <InstallArguments>
```

- Path to executable - this is the path to the actual file for the exe
  - For example: C:\packages\MyPatch1.0.0\installer.exe /quiet

## LinuxRpm

To **test** if a LinuxRpm patch is installed the following command is used by the Package Provider:

```
rpm -qa <ProductName> | grep <ProductVersion>
```

- ProductName - The product name from the patch manifest that identifies the name of the installed product.
- ProductVersion - The product version from the patch manifest that identifies the version of the installed product.

To **install** a LinuxRpm patch the following command is used by the Package Provider:

```
yum -y --nogpgcheck localinstall <Path to RPM>
```

- Path to RPM - this is the path to the actual file for the RPM
  - For example: /root/packages/MyPatch1.0.0/installer.rpm

## ScriptFramework

To **test** if a ScriptFramework patch is installed you can run the "Test" method of your ScriptFramework script directly.

To **install** a ScriptFramework patch you can execute the "Set" method of your ScriptFramework script directly.

For more information and examples on this see: [RPS Patching Script Framework](#)

# How to Patch Using RPS

Last updated on August 2, 2021.

Last Reviewed and Approved on PENDING REVIEW

This document describes the patching process of the Rapid Provisioning System (RPS).

## Intended Audience

This article is intended for use by RPS Administrators and patching roles.

## Prerequisites

For the Content Delivery Network (CDN) to transfer content between nodes, the CDN will need to know which protocol to use between nodes: Bits or DFSR. To configure this setting, two new properties will need to be created on each local RPS server node:

- ParentCdnProtocol
  - The value will be either 'Dfsr' or 'Bits'. This is the connection RPS will use when communicating with its parent.
- ChildCdnProtocol
  - The value will be either 'Dfsr' or 'Bits'. This is the connection RPS will use when communicating with all its children. You cannot configure each child differently.

### NOTE

The ChildCdnProtocol value **does not** need to match the ParentCdnProtocol value.

## How to Patch Using RPS

The RPS Patching feature lets users deploy patches and new software through the RPS system. This section will provide the information on supported patch types, supported operating systems, and instructions on using RPS to patch.

### Supported Patch Types

The following are a list of valid patch types that can be applied using RPS:

- MSU (Windows)
- EXE (Windows)
- MSI (Windows)
- CAB (Windows)
- RPM (Linux)
- Script Framework (PowerShell)

To learn more, see: [RPS Patch Product Types](#).

### Supported Operating Systems

RPS supports the following operating systems and versions:

- Windows 10



- Windows Server 2012 (or newer)
- CentOS Linux 5 (or newer)
- Red Hat Linux 5 (or newer)
- Appliances that use PowerShell

## How to Patch Windows or Linux

### **NOTE**

For a patch stream to deploy, the target(s) must be in a valid maintenance window.

1. (Optional) Create Maintenance Window(s) for the Targets that you want to patch. See [How to Use Maintenance Windows](#).
2. Create the patch file. See [How to Create an RPS Patch](#).
3. Place all the patch files in a single directory. There should be a single directory for each patch stream.
4. Copy the patch files to the node. You can use an external media device to do this, such as a thumb drive, to copy the files to the node.
5. Create the patch stream for the patch files. See [How to Create a Patch Stream](#).
6. Approve the patch stream. See [How to Approve and Reject Patch Streams](#).

## How to Patch an Appliance

1. Create a target type that is 'patchable' and properties for the appliance. See [How to Create a Patchable Target Type](#).
2. Create the patch zip file. See [How to Create an RPS Patch](#).
3. Place the patch file(s) in a single directory. there should be a single directory for each patch stream.
4. Copy the patch file(s) to the node. You can use an external media device to do this, such as a thumb drive, to copy the files to the node.
5. Create the patch stream for the patch file(s). See [How to Create a Patch Stream](#).
6. Approve the patch stream. See [How to Approve and Reject Patch Streams](#).

# How to Create a Patchable Target Type

Last updated on July 23, 2021.

Last Reviewed and Approved on PENDING REVIEW

## Overview

Systems requiring patches must be marked as patchable targets in order to receive patch assignments. This is achieved by using the `Set-RpsTypeProperty` with the `-CanPatch` switch parameter when the target items are created.

## How CanPatch is Used

The `-CanPatch` switch parameter signifies that a target is patchable and can receive patch assignments. Patches in RPS use specific conditions to automatically be assigned patchable targets when a patch stream is approved. The patches and targets that receive RPS patch assignments must have the same conditions set.

*Conditions* are properties on patches and targets that have a *Name*, *Value*, and *Operator*. By default, all patches have these conditions added to them:

1. **OsType** - This value is pulled from the patch's manifest. This will match with any target that has an *OsType* that is the same as the value on the patch.
2. **OsVersion** - This value is pulled from the patch's manifest. This will match with any target that has an *OsVersion* that is the same as the value on the patch.
3. **Architecture** - This value is pulled from the patch's manifest. This will match with any target that has an *Architecture* that is the same as the value on the patch.
4. **CanPatch** - This value is based on the target item's *Target Type*. This will match with any target whose *Target Type* has a `-CanPatch` parameter set to *true*. This means that only "patchable" targets will get patch assignments.

## How to Create a Patchable Target Type

To create a target type and properties for a patchable target, use the `Set-RpsTargetType` cmdlet in an Administrative PowerShell window, then set the `-CanPatch` switch parameter to mark it as being "patchable", as shown in the following example:

```
$hostType = Set-RpsTargetType -Name 'EsxHost' -IsRoot -CanPatch
```

A patchable *Target Type* can then be used to create *Target Items* that are patchable using the `Set-RpsTypeProperty` cmdlet.

### NOTE

Target types and target items that are not created with the `-CanPatch` attribute are **not** patchable.

The following example lists the **required** properties to create a patchable target type:

```
$null = Set-RpsTypeProperty -Parent $hostType -Name 'ComputerName' -PropertyType Text -IsRequired
$null = Set-RpsTypeProperty -Parent $hostType -Name 'IPAddress' -PropertyType Text -IsRequired
$null = Set-RpsTypeProperty -Parent $hostType -Name 'RunPackagesOnTms' -PropertyType Boolean -DefaultValue 'True'
$null = Set-RpsTypeProperty -Parent $hostType -Name 'OsType' -PropertyType Text - DefaultValue 'ESX'
$null = Set-RpsTypeProperty -Parent $hostType -Name 'Architecture' -PropertyType Text - DefaultValue 'x86'
$null = Set-RpsTypeProperty -Parent $hostType -Name 'OsVersion' -PropertyType Text - DefaultValue '10.12.10'
```

The following table describes the required properties:

PROPERTY NAME	PROPERTY TYPE	DESCRIPTION
ComputerName	text	Name of the target
IPAddress	text	IP address of the target
RunPackagesOnTms	boolean	Indicates whether to deploy Patches from TMS instead of from the target itself (TMS acts as a proxy)
OsType	text	OS Type (e.g. Linux, Windows, ESX)
Architecture	text	Architecture of the target
OsVersion	text	OS version

The following example lists the **optional** properties which can be added when creating a patchable target type:

```
$null = Set-RpsTypeProperty -Parent $virtualMachine -Name 'VMId' -PropertyType Text
$null = Set-RpsTypeProperty -Parent $virtualMachine -Name 'MemoryMB' -PropertyType Number -IsRequired -DefaultValue 2048
$null = Set-RpsTypeProperty -Parent $virtualMachine -Name 'ProcessorCount' -PropertyType Number -IsRequired -DefaultValue 1
$null = Set-RpsTypeProperty -Parent $virtualMachine -Name 'State' -PropertyType Text -DefaultValue 'Running' # Running | Paused | Off
$null = Set-RpsTypeProperty -Parent $virtualMachine -Name 'Path' -PropertyType Path
$null = Set-RpsTypeProperty -Parent $virtualMachine -Name 'Generation' -PropertyType Number -DefaultValue '2' # 1 | 2
$null = Set-RpsTypeProperty -Parent $virtualMachine -Name 'SecureBoot' -PropertyType Boolean -DefaultValue $true
$null = Set-RpsTypeProperty -Parent $virtualMachine -Name 'MinimumMemoryMB' -PropertyType Number
$null = Set-RpsTypeProperty -Parent $virtualMachine -Name 'MaximumMemoryMB' -PropertyType Number
$null = Set-RpsTypeProperty -Parent $virtualMachine -Name 'IsCDN' -PropertyType Boolean
$null = Set-RpsTypeProperty -Parent $virtualMachine -Name 'IsTms' -PropertyType Boolean
$null = Set-RpsTypeProperty -Parent $virtualMachine -Name 'IsDB' -PropertyType Boolean
$null = Set-RpsTypeProperty -Parent $virtualMachine -Name 'IsDC' -PropertyType Boolean
$null = Set-RpsTypeProperty -Parent $virtualMachine -Name 'IsAppliance' -PropertyType Boolean -DefaultValue $false
$null = Set-RpsTypeProperty -Parent $virtualMachine -Name 'ProvisionIpAddress' -PropertyType Text
$null = Set-RpsTypeProperty -Parent $virtualMachine -Name 'Status' -PropertyType Text
$null = Set-RpsTypeProperty -Parent $virtualMachine -Name 'AutomaticCheckPointsEnabled' -PropertyType Boolean
$null = Set-RpsTypeProperty -Parent $virtualMachine -Name 'EnableGuestService' -PropertyType Boolean
$null = Set-RpsTypeProperty -Parent $virtualMachine -Name 'SecureBoot' -PropertyType Boolean
```

# How to Use Maintenance Windows

Last updated on July 07, 2021.

Last Reviewed and Approved on PENDING REVIEW

## Intended Audience

This document is intended for RPS patching roles, Lead Systems Integrators (LSI), Field Service Representatives (FSR), IT staff, and Developers.

## What is a Maintenance Window?

Maintenance windows are used in RPS to schedule the exact time needed for system patching or to create a period of no change within the RPS target environment. Maintenance windows are required for scheduling patch streams and are available through the RPS website's Distribution menu. RPS patch streams combine zipped (.zip) RPS patches, an approval process, and a scheduled deployment of patches to RPS targets.

In addition,

- Multiple schedules can be created for each patch stream. This allows the deployment of a patch stream to different targets on different schedules.
- For administrators, the object is a *templated* Resource Item (an object) that gets assigned to one or more Target Items (computers or endpoints) and represents a window of time that a software installation can occur.
- Maintenance windows are particularly useful for defining a period of no change or code freeze.

### IMPORTANT

Without an active maintenance window, no patches will be applied or removed because the Desired State Configuration (DSC) `Set` command will only run inside a maintenance window.

## Creating a Maintenance Window

Maintenance windows can be created through a local RPS node server website, or using [PowerShell](#) as described later in this article.

### NOTE

Users may see "patch" and "package" used interchangeably in the log outputs during this process.

### Distribution Menu

Navigate to the Distribution Menu. Three menu options are presented: Patch Streams, Patches and Certificate Management. Select Patch Streams.

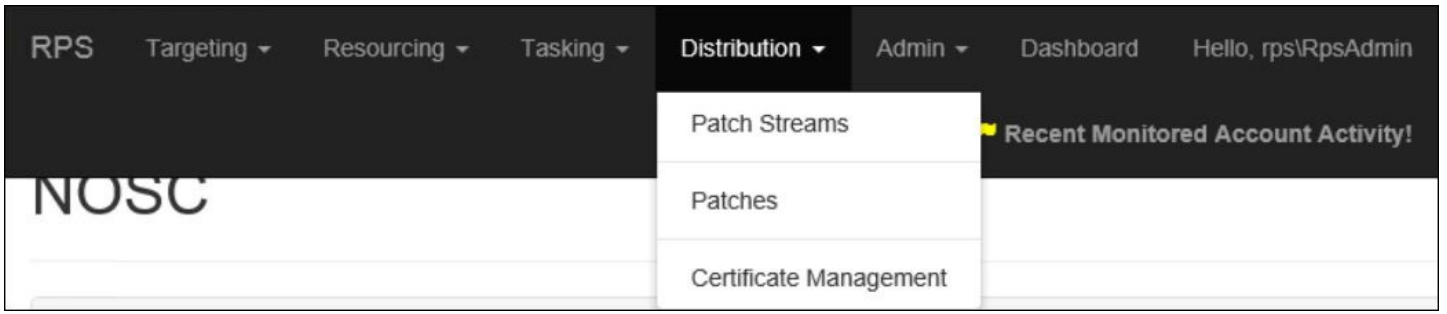


Figure 1: RPS Patching Distribution Menu

### Schedule Tab

From the Patch Stream Maintenance Windows page, select the "Schedule" tab.

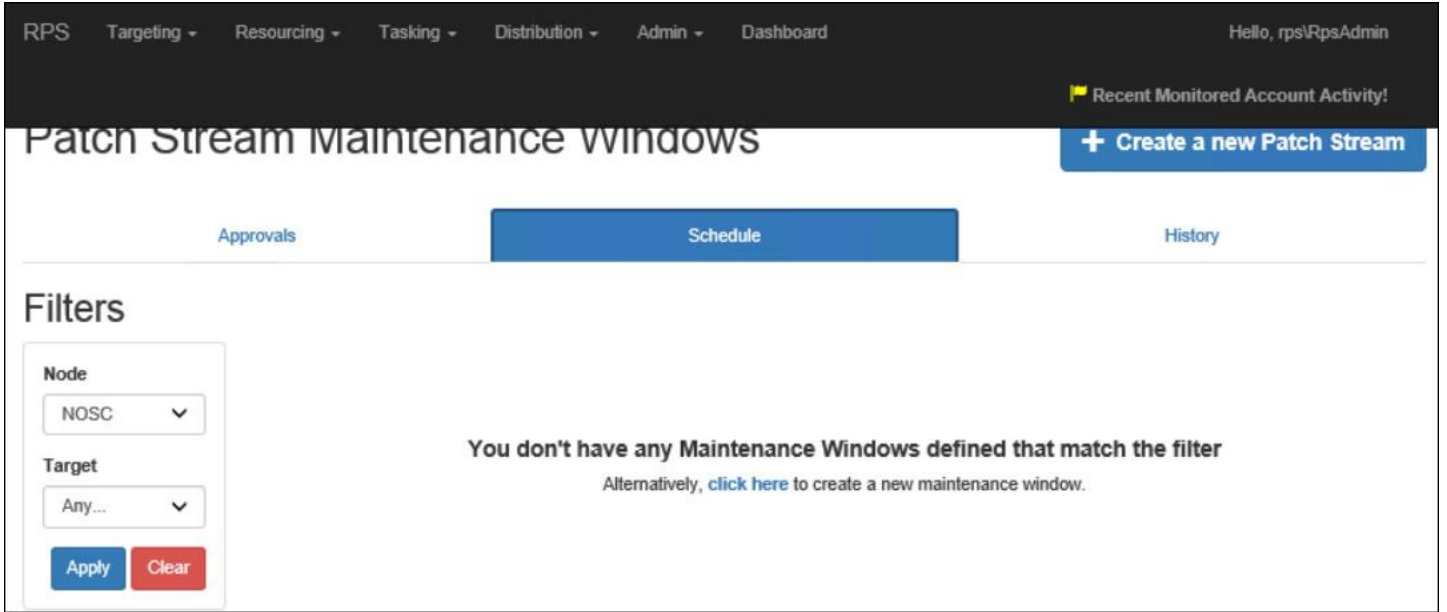


Figure 2: Schedule Tab

Using the **Filters** on the left side of the window, define the Node and Target you would like to schedule a maintenance window for and click **Apply**. This search will return any previously scheduled windows for the Node and/or Target you defined. Use the red **Clear** button to clear the filter.

### Create Maintenance Window

If no maintenance windows are defined matching the filter criteria, use the [click here](#) link to open the "Create Maintenance Window" form and create a new one.

#### **TIP**

Ensure to use a descriptive name for your maintenance window, e.g., Patch Tuesday.

**Create Maintenance Window**

**Name**   
A name to identify this maintenance window recurrence pattern.

**Frequency**   
The frequency the schedule should run.

**Day of the week \***   
The day of the week resources are permitted to run.

**Week of the month**   
The week of the month resources are allowed to run. A value of '0' represents the last week of the month.

**Day of the month**   
The day of the month resources are allowed to run. A value of '0' represents the last day of the month.

**Start date**   
The first day the maintenance window and its recurrence pattern takes effect.

**End date**   
The final day the maintenance window and its recurrence pattern takes effect.

**Start time**    
The time when resources are allowed to run begins.

**Duration**    
The amount of time from the start of the window to allow updates.

**Target items**

- [-] [icon] NOSC-PatchableTargets
  - [-] [icon] AD.unit.domain (VirtualMachine)
  - [-] [icon] NOSC.rps.local (VirtualMachine)
- [-] [icon] TCN-PatchableTargets
  - [-] [icon] TCN.rps.local (VirtualMachine)
  - [-] [icon] Member.unit.domain (VirtualMachine)

Figure 3: Create Maintenance Window form

The following fields are available to define your maintenance window.

**NOTE**

It is best practice to create multiple maintenance windows to satisfy your time, date, and frequency requirements.

PROPERTY	TYPE	DESCRIPTION
Name	string	The friendly name of the maintenance window. { e.g. 'Patch Tuesday' }
Frequency	string	The frequency upon which the window should be open { Daily, Weekly, or Monthly }
Day of the Week	string[]	The day(s) of the week Set-TargetResource will run. { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, or Saturday }
Week of the month	int[]	The week(s) of the month Set-TargetResource could be run. 0 represents the last week of the month. { 0, 1, 2, 3, or 4 }

PROPERTY	TYPE	DESCRIPTION
Day of the month	int[]	The day(s) of the month Set-TargetResource will run. 0 represents the last day of the month. { 0 - 31 }
Start Date	DateTime	The first UTC date Set-TargetResource is allowed to run
End Date	DateTime	The last UTC date Set-TargetResource is allowed to run
Start Time	TimeSpan	The first time the of day Set-TargetResource is allowed to run
Duration	TimeSpan	The amount of time from the start of the window to allow updates.
Target Items	Selectable Tree View	Checkbox the target items that apply, or do not select any.

In addition to identifying the time, date, and frequency of your maintenance window, you will need to identify the target items to which the window applies. This is done by placing checks in the boxes next to items you require assigned to this window. It is possible that you will have maintenance windows with no assigned target items. These assignments can be specified as needed through the UI or via [PowerShell](#) cmdlet discussed below.

## Target Items



Cancel

Submit

Figure 8: Target Items Tree View

### ⚠ IMPORTANT

A value of zero  in the **Day of the month** or **Week of the month** field indicates the last day or week of the month.

### ℹ TIP

Recommended best practice to utilize Universal Time Coordinated (UTC) dates for  and  regardless of target item time zone.

Once you have completed the form, save the maintenance window by clicking the **Submit** button.

## Using PowerShell To Create a Maintenance Window

As seen in the example below, use the  cmdlet to create your maintenance window using PowerShell.

### ℹ NOTE

In the examples below, replace the words in parenthesis with unique names applicable to the maintenance window being created.

```
$startDate = (Get-Date)
$startTime = New-TimeSpan -Hours 0 -Minutes 0
$endDate = $startDate.AddDays(5)
$endTime = New-TimeSpan -Hours 11 -Minutes 59
$frequency = "Daily"

$myWindow = New-RpsMaintenanceWindow -Name 'myWindow' -Frequency $frequency -StartDate $startDate -StartTime $startTime -
EndDate $endDate -EndTime $endTime
```

Checking the result of `$myWindow` will show you what was created, for example: `$myWindow.Properties`:

KEY	VALUE
StartDate	1/30/2020 12:00:00AM
StartTime	00:00
EndTime	23:59
EndDate	2/4/2020 12:00:00AM
Frequency	"Daily"

Use the `Get-RpsMaintenanceWindow`, `Get-RpsTargetItem`, and `New-RpsResourceAssignment` cmdlets to specify the target items for your maintenance window as seen the the following example.

```
$myWindow = Get-RpsMaintenanceWindow -Name 'myWindow'
$myTargetItem = Get-RpsTargetItem -Name 'myTargetItem'
New-RpsResourceAssignment -ResourceItem $myWindow.MaintenanceWindowResourceItem.Id -TargetItem $myTargetItem
```

### Editing Maintenance Windows

To view or edit existing maintenance windows return to the "Schedule" tab located on the Patch Stream Maintenance Windows page. Using the **Filters** on the left side of the window, define the Node and Target you would like to view the scheduled maintenance window for and click **Apply**. This search will return any previously scheduled windows for the Node and/or Target you defined. Use the red **Clear** button to clear the filter.



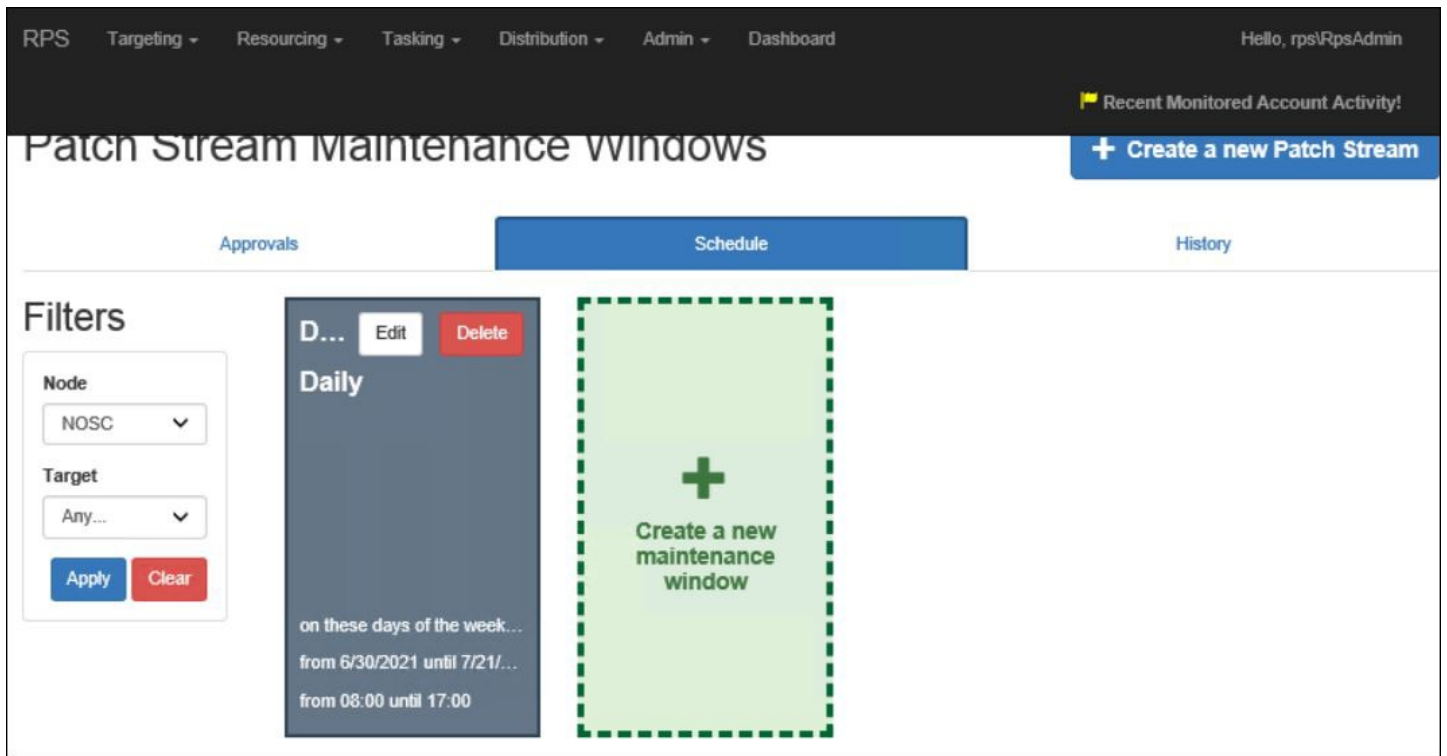


Figure 4: View or Edit Maintenance Windows

To view an existing maintenance window using PowerShell, use the `Get-RpsMaintenanceWindow` cmdlet. The below examples show running this cmdlet against the `-Name` and the `-Id` paramaters.

```
Get-RpsMaintenanceWindow -Name 'myWindow'
```

or

```
Get-RpsMaintenanceWindow -Id <GUID>
```

To make any necessary changes click the **Edit** button on the top-right of the maintenance window tile, shown below:

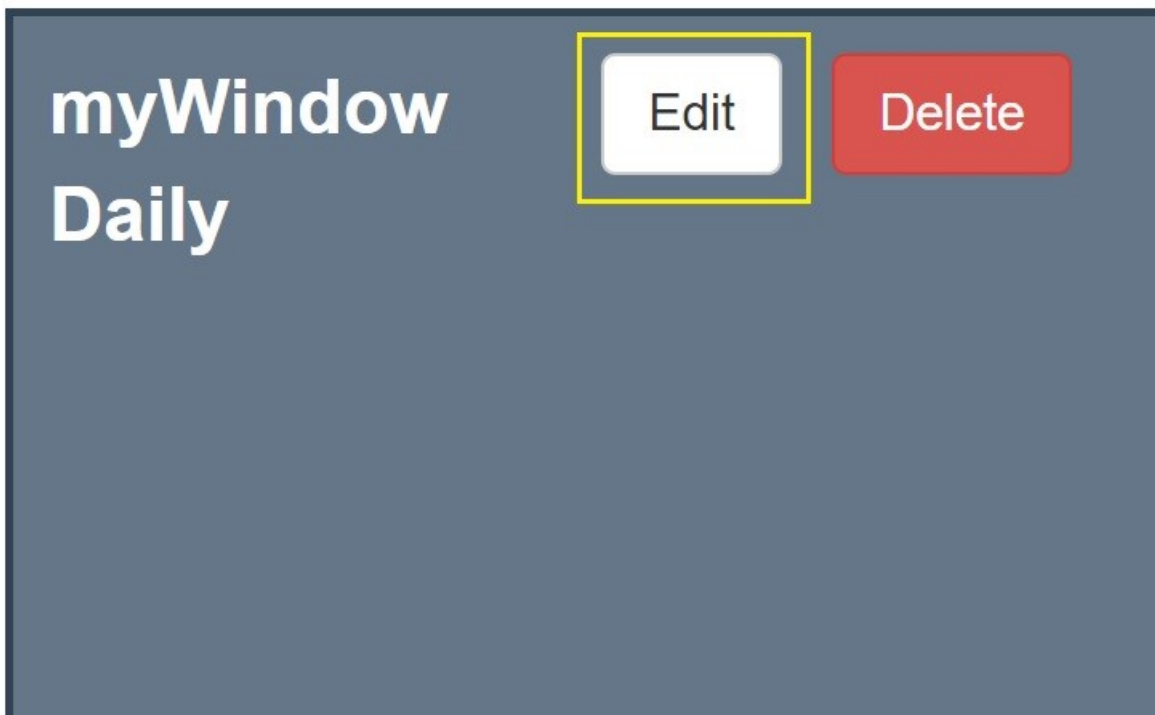


Figure 5: Edit Button

The "Create Maintenance Window" form will open allowing you to make changes. Once completed click **Submit**.

To edit with PowerShell you will use the `Get-RpsMaintenanceWindow` followed by the `Set-RPSMaintenanceWindow` cmdlets as seen in the following examples.

```
$myWindow = Get-RpsMaintenanceWindow -Name 'myWindow'  
Set-RpsMaintenanceWindow -Id $myWindow.Id -DayOfTheWeek 'Sunday'
```

or

```
$myWindow = Get-RpsMaintenanceWindow -Name 'myWindow'  
Set-RpsMaintenanceWindow -Name 'myWindow' -DayOfTheWeek 'Sunday'
```

In both of these examples the `-DayofTheWeek` property was set to Sunday by calling the `-Name` or `-Id` of the 'myWindow' maintenance window.

## Deleting Maintenance Windows

To delete existing maintenance windows return to the "Schedule" tab located on the Patch Stream Maintenance Windows page. Using the **Filters** on the left side of the window, define the Node and Target you would like to view the scheduled maintenance window for and click **Apply**. This search will return any previously scheduled windows for the Node and/or Target you defined.

Select the applicable window you would like to delete and click the red **Delete** button on the top-right of the maintenance window tile, shown below:

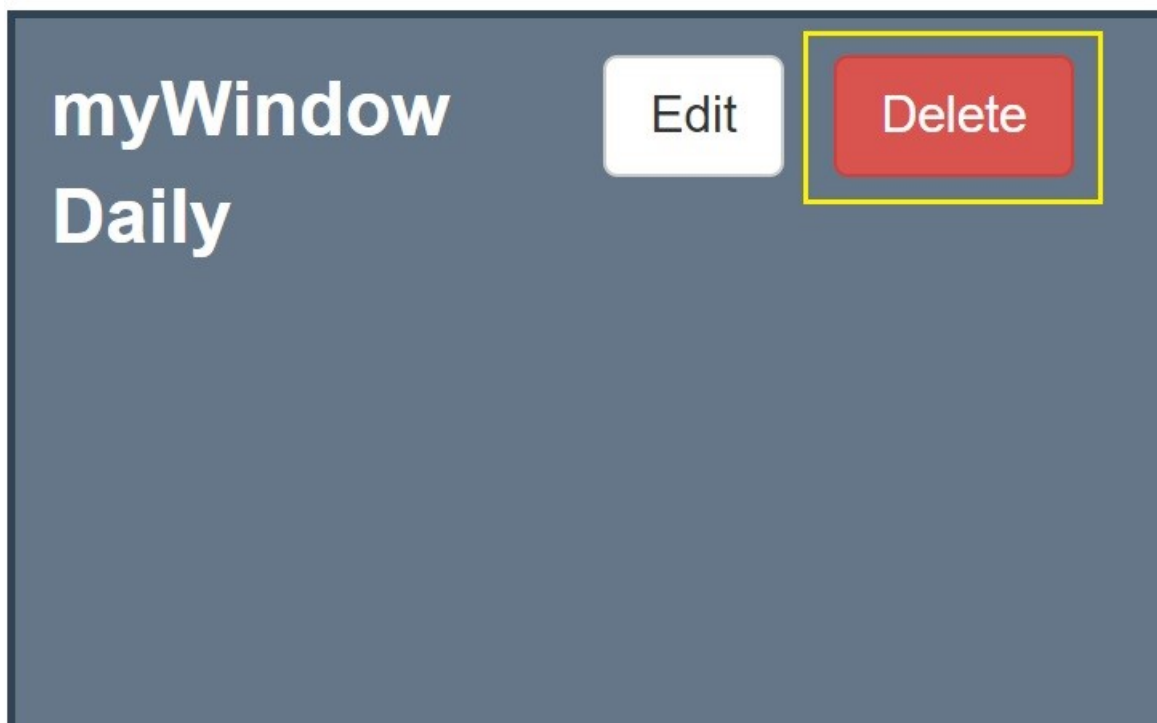


Figure 6: Delete Button

A warning dialog opens displaying information about the maintenance window to be deleted. Click the red **Delete** button to continue or click **Cancel** to close without deleting.

# Warning!

---

**You will not be able to revert deletion of this Maintenance Window**

**Name:** myWindow  
**Frequency:** Daily  
**Effective Date:** from 1/1/0001 until 1/1/0001  
**Effective Time:** from 00:00 until 23:59

---

Cancel

Delete

Figure 7: Deletion Warning Dialog Box

Use the `Delete-RpsMaintenanceWindow` cmdlet to delete a maintenance window using PowerShell, as seen in the example below:

```
Delete-RpsMaintenanceWindow -Id <GUID>
```

# How to Create an RPS Patch

*Last updated on August 26, 2021.*

*Document Status: Document Feature Complete as of August 26, 2021; PENDING EXTERNAL REVIEW.*

## What is a Patch

An RPS Patch is what RPS calls any executable meant to be distributed via RPS to an RPS Target with the intent of managing the software on the target. A patch is a Zip archive that contains the content needed in order to manage the software, such as executables. A patch also requires a patch manifest file, which is an XML formatted text file containing required metadata. That zip archive in its entirety is what would be considered a "Patch" in terms of RPS, not the individual files inside.

## How to Create a Patch Manually

It is recommended to utilize REACTR to create RPS patches. [How to Create, edit, and download patches in REACTR.](#)

To create a patch manually:

1. Create a text file and name it "RPS.Package". Ensure the file format is not .txt.
2. Open RPS.Package with Notepad and copy the empty Patch Manifest example in [Patch Manifest Definition](#).
3. Fill in the required fields for the patch manifest and save the RPS.Package file.
4. Gather all content files needed to run on the target in order to install, upgrade, or uninstall. This must include an executable entry point such as an .exe file.
5. Zip content files and RPS.Package into one archive.
  1. Right-click on the installer or multiple selected files to bring up the context menu
  2. Highlight 'Send To' context menu option
  3. Select 'Compressed (zipped) folder' option
6. Rename the zipped archive file. The ZIP file must be named with the exact ProductName and ProductVersion values from the patch manifest file in the format "ProductNameProductVersion.zip" with no spaces in between the values. For example "Firefox70.0.zip".

### Example: Creating a FireFox Install Package

1. Navigate to the desired directory in File Explorer, right-click, highlight New, and select Text Document. Name the file "Package.RPS". This is the patch manifest file.

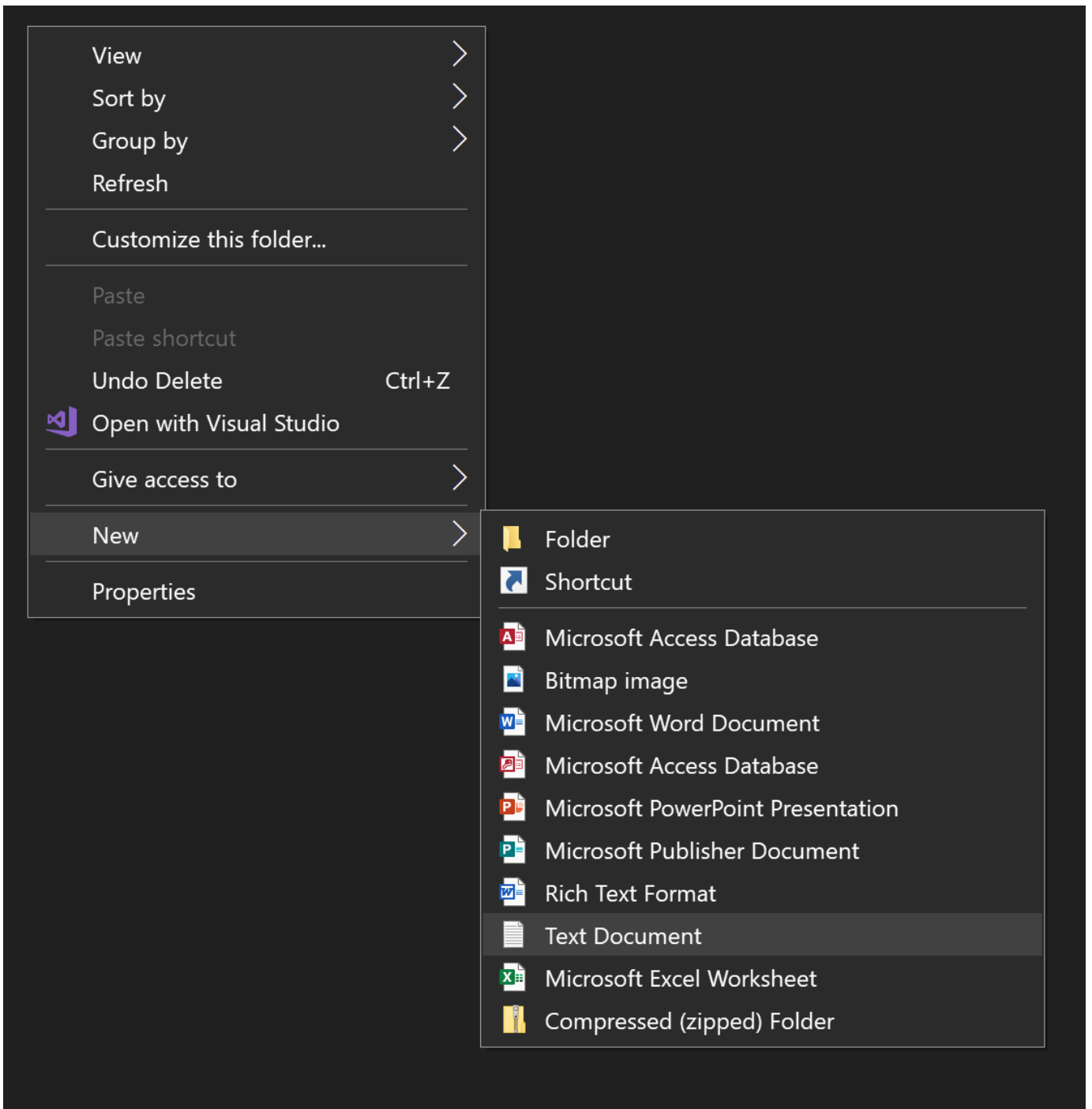


Figure 1 Create the Patch Manifest file

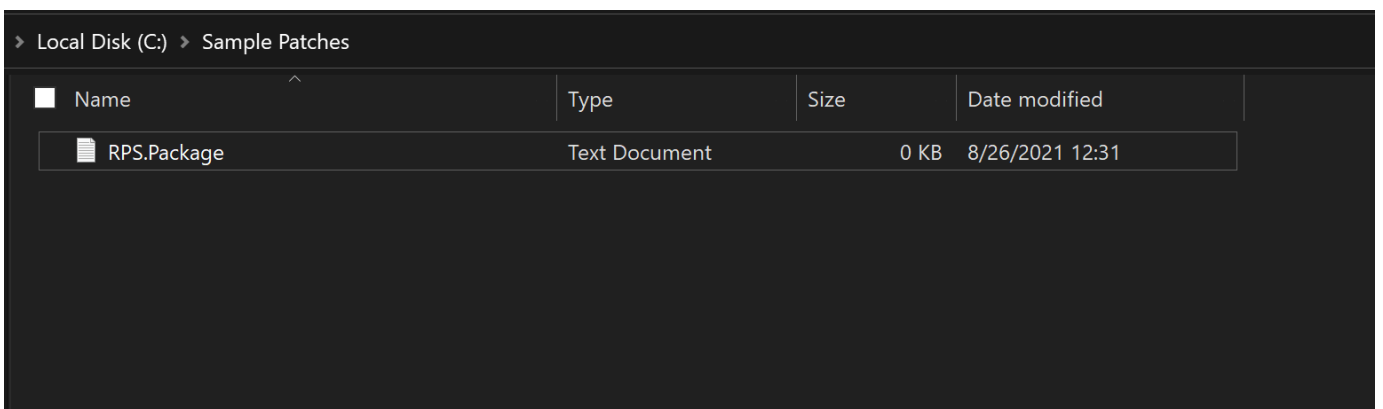


Figure 2 Name the file RPS.Package

2. Edit the PackageManifest file and populate required fields for the Firefox version that is to be installed.

```
<?xml version="1.0" encoding="utf-8"?>
<PackageManifest xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
version="1.0">
<PackageName>Firefox</PackageName>
<PackageVersion>70.0.0</PackageVersion>
<Description>This is a test package</Description>
<OsVersion>*</OsVersion>
<Architecture>x64</Architecture>
<OsType>Windows</OsType>
<MsCatalogProductName>Firefox70</MsCatalogProductName>
<MsCatalogTitle />
<MsCatalogId />
<Products />
<MsCatalogUpdateId />
<PackageClassification>General</PackageClassification>
<MsCatalogSupersededByKblds />
<MsCatalogLinkUrls />
<UninstallArguments>/s</UninstallArguments>
<InstallArguments />
<SuppressReboot>>true</SuppressReboot>
<ProductName>Mozilla Firefox 70.0 (x64 en-US)</ProductName>
<ProductType>WindowsMsi</ProductType>
<ProductVersion>70.0</ProductVersion>
<ProductId>{74994757-3b19-4c54-afe4-ae84e398a3f7}</ProductId>
<InstallerFileName>firefox.msi</InstallerFileName>
</PackageManifest>
```

3. Zip the installer and any files needed for the desired Firefox version
  1. Right-click on the installer or multiple selected files to bring up the context menu
  2. Highlight 'Send To' context menu option
  3. Select 'Compressed (zipped) folder' option

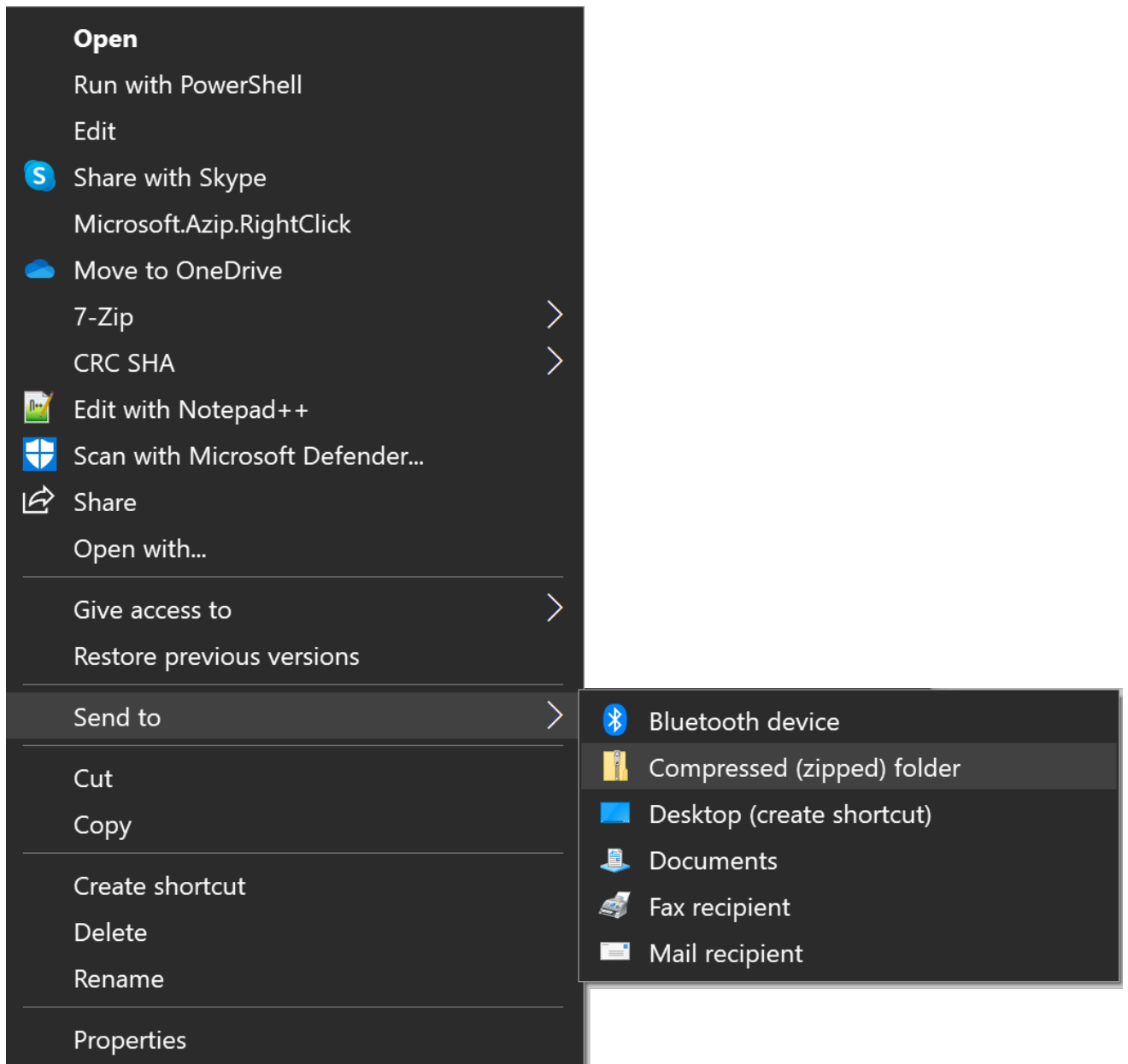


Figure 3 Create a Compressed (zipped) folder

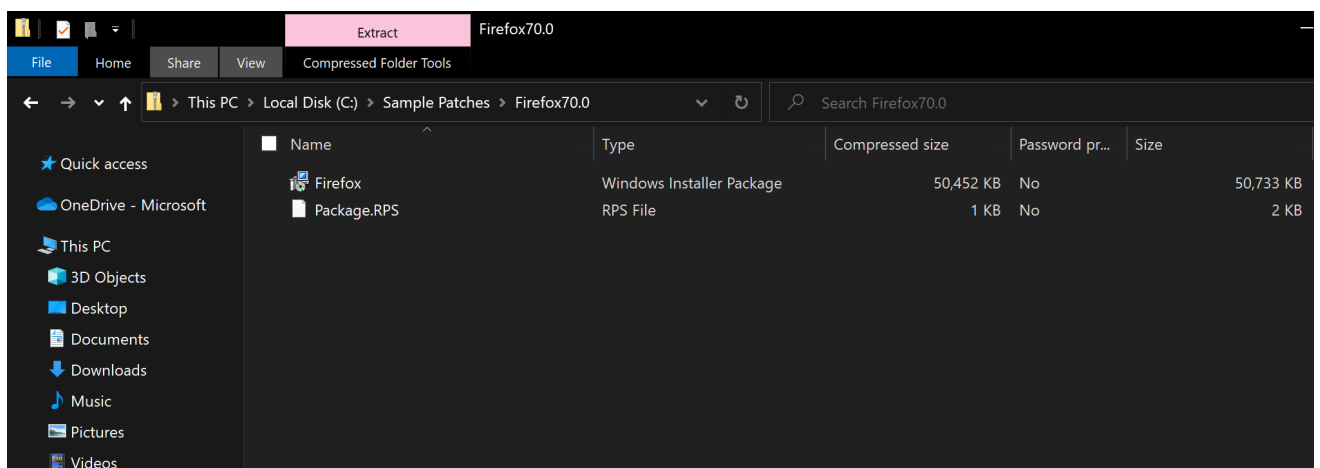


Figure 4 Firefox Patch Contents

- Rename the zip file to "ProductNameProductVersion" based on the patch manifest values. For this example, the filename would be "Firefox70.0.zip".

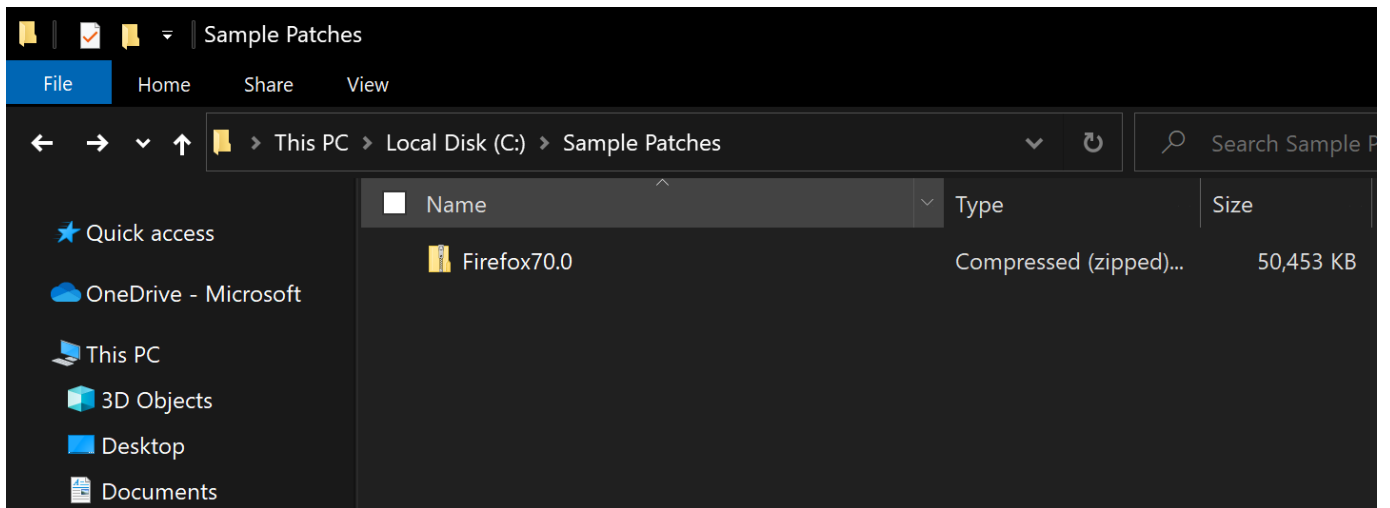


Figure 5 Firefox v70.0 Rps Patch



# How to Disable or Enable an RPS Patch Using PowerShell

Last updated on August 19, 2021.

**Document Status:** Document Feature Complete as of August 19, 2021; PENDING EXTERNAL REVIEW.

This document describes how to disable or enable an RPS patch using PowerShell.

## NOTE

A patch is enabled by default and is not disabled until you direct RPS to disable this patch.

## Prerequisites

1. Log into an RPS Server and launch Windows PowerShell or Windows PowerShell ISE as administrator.

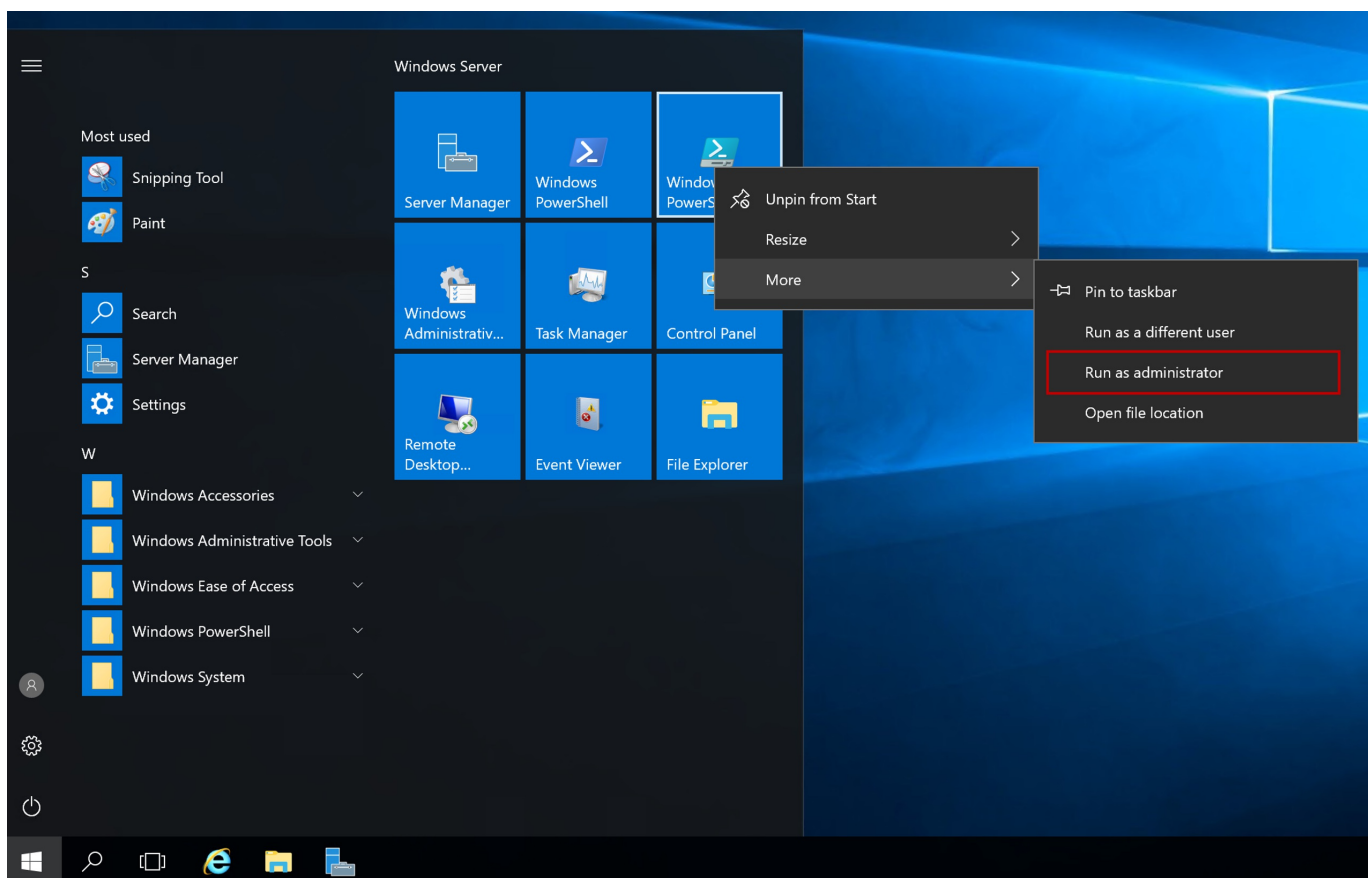


Figure 1: Run PowerShell as administrator.

2. Change your directory to the RPS ContentStore. For example:

```
cd C:\ContentStore
```

3. Import the RPS API module.

```
Import-Module C:\ContentStore\Modules\Rps-Api
```

4. Create a patch. See [How to Create an RPS Patch](#).

## How to Disable a Patch

A patch is enabled by default and is not disabled until you direct RPS to disable this patch. Disabling a patch will not reverse past

deployments, but it will halt future deployments from occurring.

Disable a patch using the PowerShell cmdlet, [Disable-RpsPatch](#).

#### **WARNING**

Disabling a patch will disable all patches that depend on it.

After the disable patch command has been executed, the following will happen:

1. The patch will be disabled (`IsActive` Property set to "False").
2. All of the patches that depend on the disabled patch will also be disabled.

#### Disable-RpsPatch

PowerShell cmdlet that disables an RPS Patch.

```
Disable-RpsPatch -Patch $myPatch -Force
```

Options for the Disable-RpsPatch cmdlet are:

PARAMETER NAME	TYPE	DESCRIPTION	REQUIRED
Patch	Patch	Patch to be disabled.	True
Force	Switch	Force execution of disabling patch without confirmation.	False

## How to Enable a Patch

A patch is enabled by default and is not disabled until you direct RPS to disable this patch. If a patch has been disabled, then the patch can be enabled for future deployments. Re-enabling a patch will not reverse past deployments.

Enable a patch using the PowerShell cmdlet, [Enable-RpsPatch](#).

#### **WARNING**

Enabling a patch will enable all patches that patch depends on.

After the enable patch command has been executed, the following will happen:

1. The patch will be enabled (`IsActive` Property set to "True").
2. All of the patches that the enabled patch depends on will also be enabled.

#### Enable-RpsPatch

PowerShell cmdlet that enables an RPS Patch.

```
Enable-RpsPatch -Patch $myPatch -Force
```

Options for the Enable-RpsPatch cmdlet are:

<b>PARAMETER NAME</b>	<b>TYPE</b>	<b>DESCRIPTION</b>	<b>REQUIRED</b>
Patch	Patch	Patch to be enabled.	True
Force	Switch	Force execution of enabling patch without confirmation.	False

# How to Create a Patch Stream

Last updated on July 9, 2021.

Last Reviewed and Approved on PENDING REVIEW

## Introduction

This document provides step-by-step instructions for loading a patch stream into RPS using the User Interface.

## Prerequisites

- User must have permissions to the *CDN* folder.
- User must be part of the *ContentCreators* Active Directory group.
- Patches must have been previously created. Please visit [How to Create an RPS Patch](#) for detailed instructions.

### NOTE

Users may see "patch" and "package" used interchangeably in the log outputs during this process.

## What is a Patch Stream

A patch stream is what RPS calls a collection of RPS patches that will be distributed via RPS to other RPS targets with the intent of managing the software on the target.

A patch is a .ZIP archive that contains the content to deploy to the desired Targets and an RPS Patch Manifest file. The .ZIP archive in its entirety is what would be considered an "RPS Patch", not the individual files inside.

### TIP

You can find out more about RPS patches in the article "[How to Create an RPS Patch](#)". You can find out more about RPS patch manifests in the article "[RPS Patch Manifest Definition](#)".

## How to Create a Patch Stream Using the RPS GUI

1. Launch the RPS GUI and navigate to any of the patch stream management pages.
2. From the patch streams page select the **Create a new Patch Stream** button.

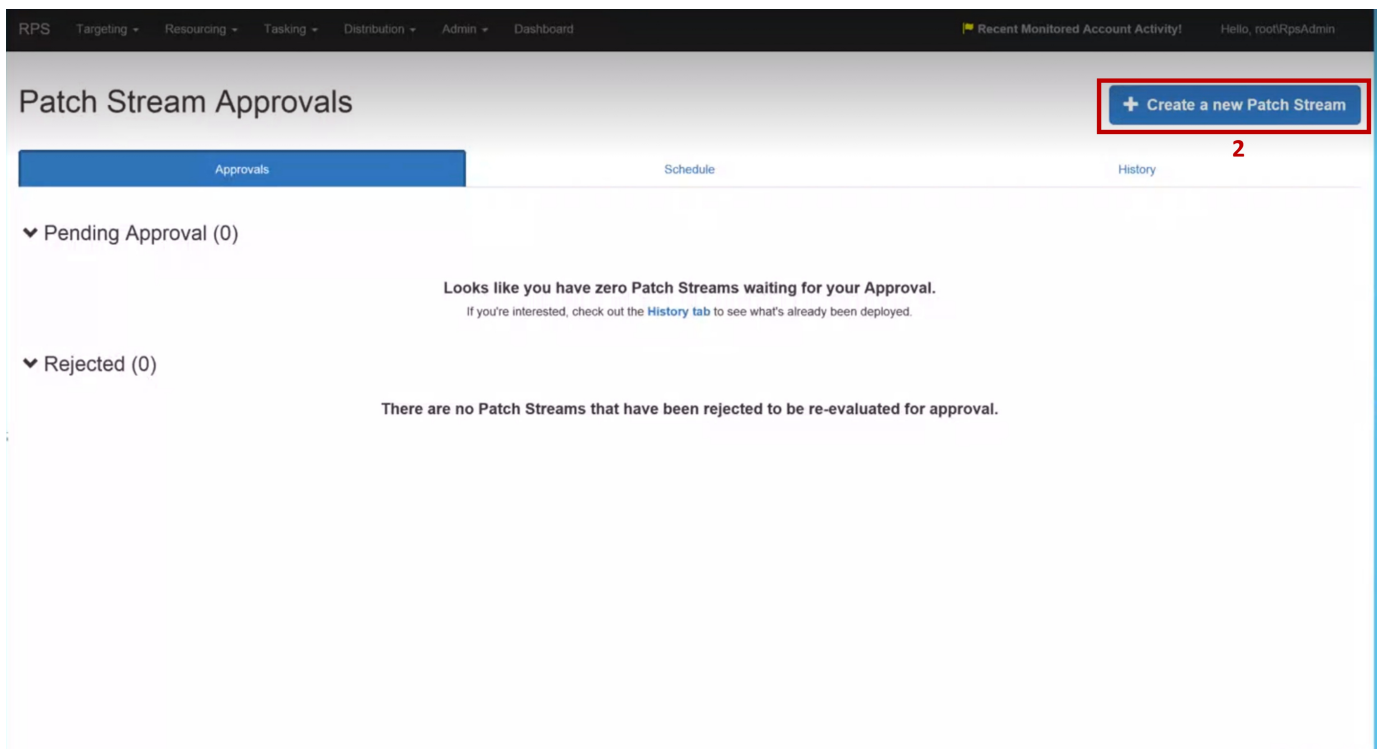


Figure 1: Create a new patch stream button.

3. A new form will appear and you will be prompted to provide a **Unique Name** for your new patch stream.
4. Click **Submit**.

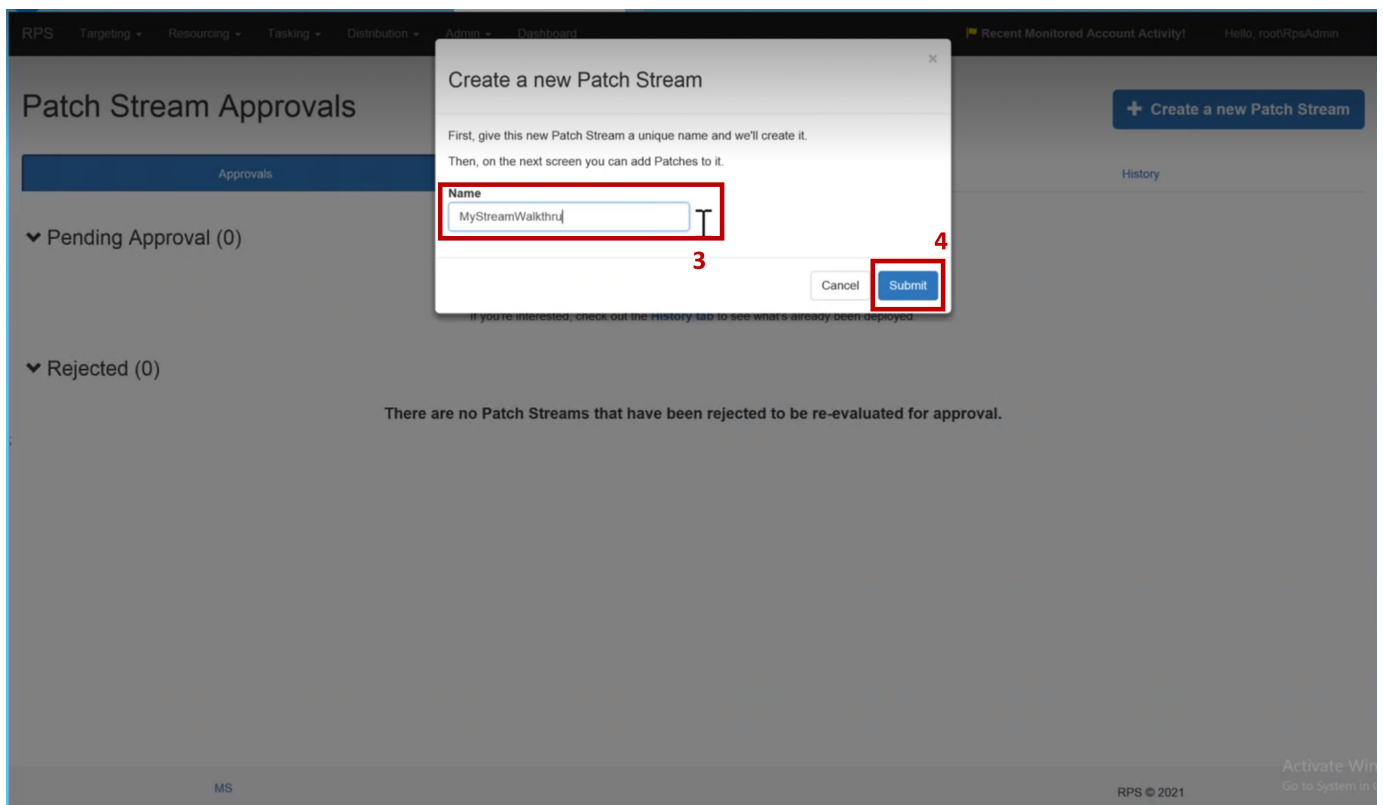


Figure 2: Create a new patch stream Name input.

5. Upon submit, you will be taken to a patch stream Editing page. Select the button labeled **Upload RPS .ZIP Patches to this Stream**.

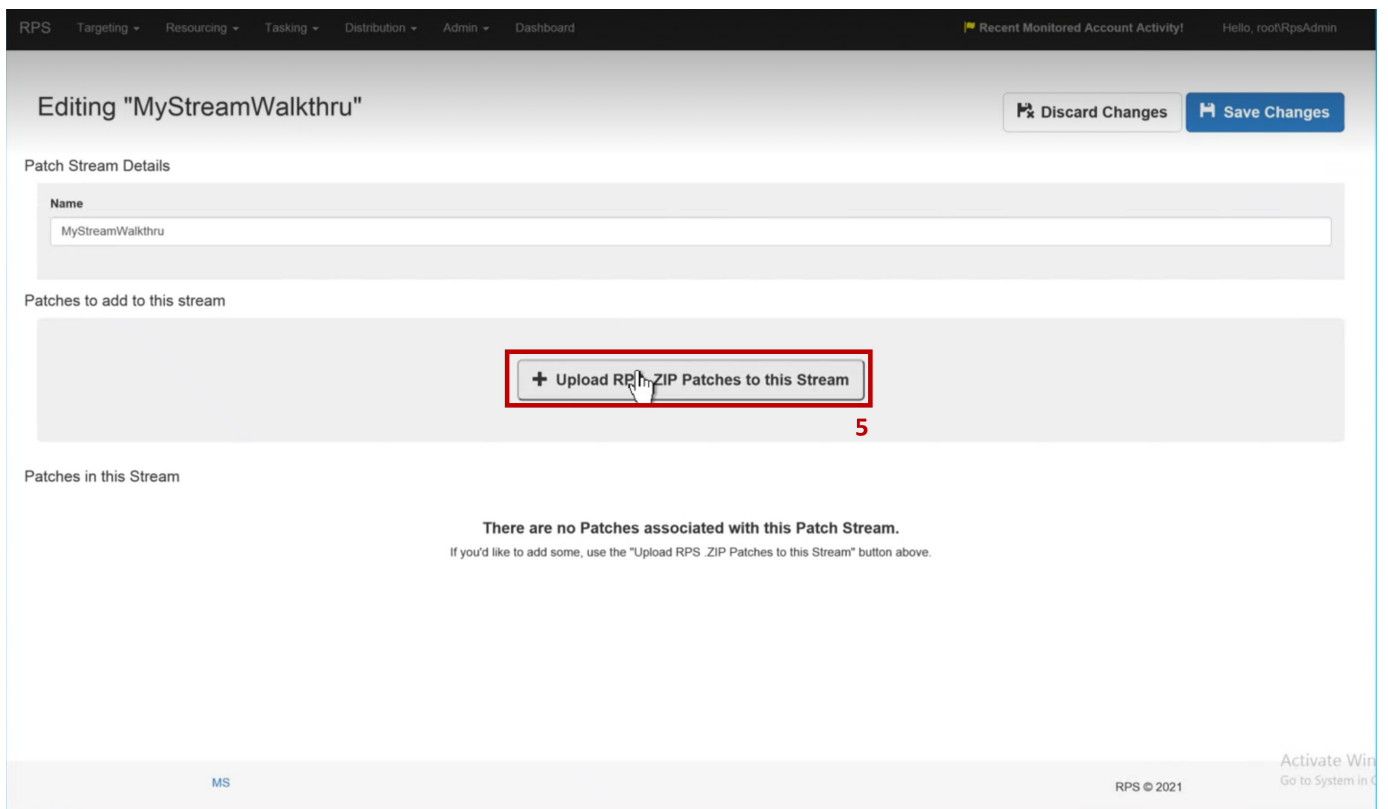


Figure 3: patch stream Edit form.

6. A new upload form will appear. Use the input button to select the **RPS .ZIP patches** you wish to associate with the new patch stream.
7. Select the **Upload** button.

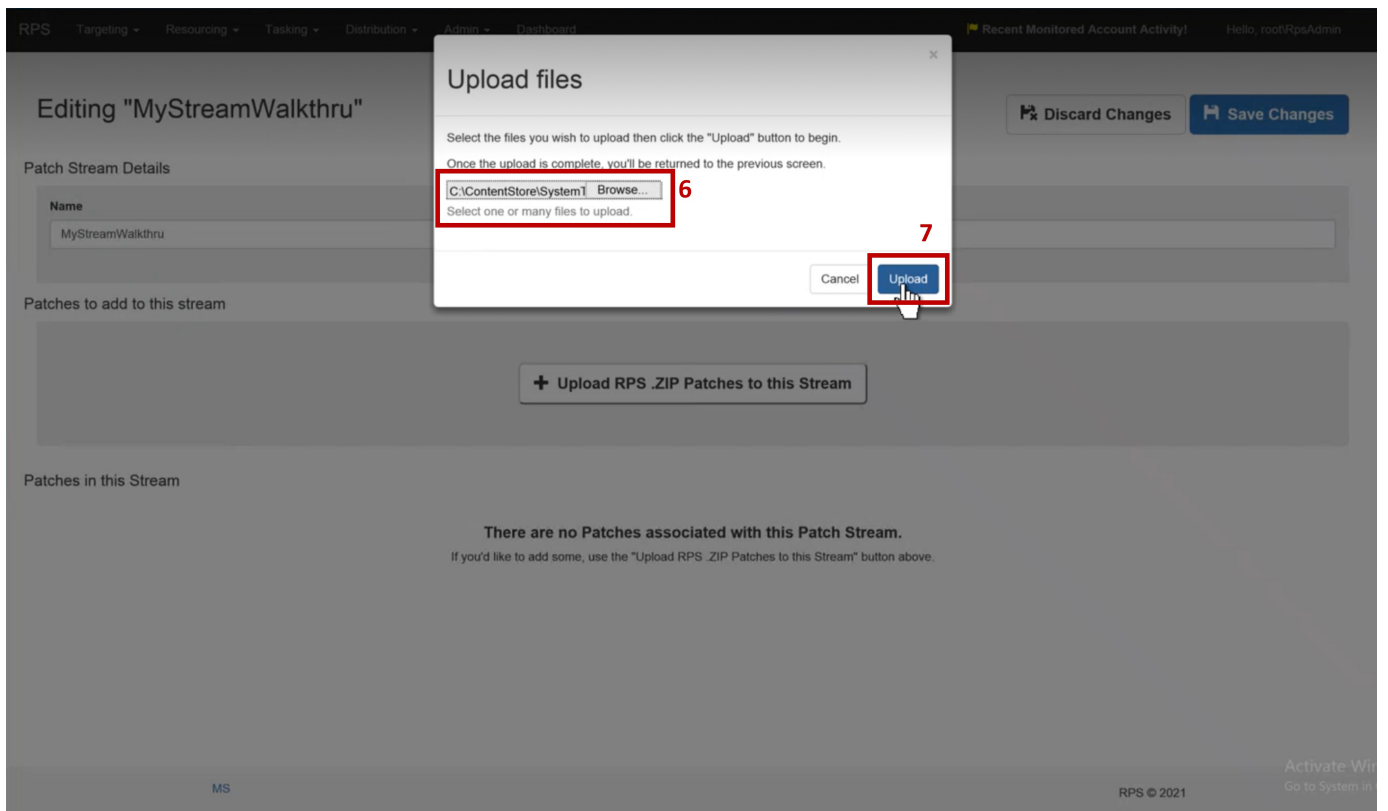


Figure 4: Upload patch files form.

**NOTE**

1. The patch upload location is configurable and can be modified by adjusting the `uploads:Location` property in RPS Web's `web.config` file.
2. The patch upload location "days until clean up" is configurable and can be modified by adjusting the `uploads:DaysUntilCleanup` property in RPS Web's `web.config`.
  1. The default value for `uploads:Location` is: `~/App_Data/Uploads`.
  2. The default value for `uploads:DaysUntilCleanup` is: `30`.

8. Your RPS Patches will begin to upload. Click **OK** once complete.

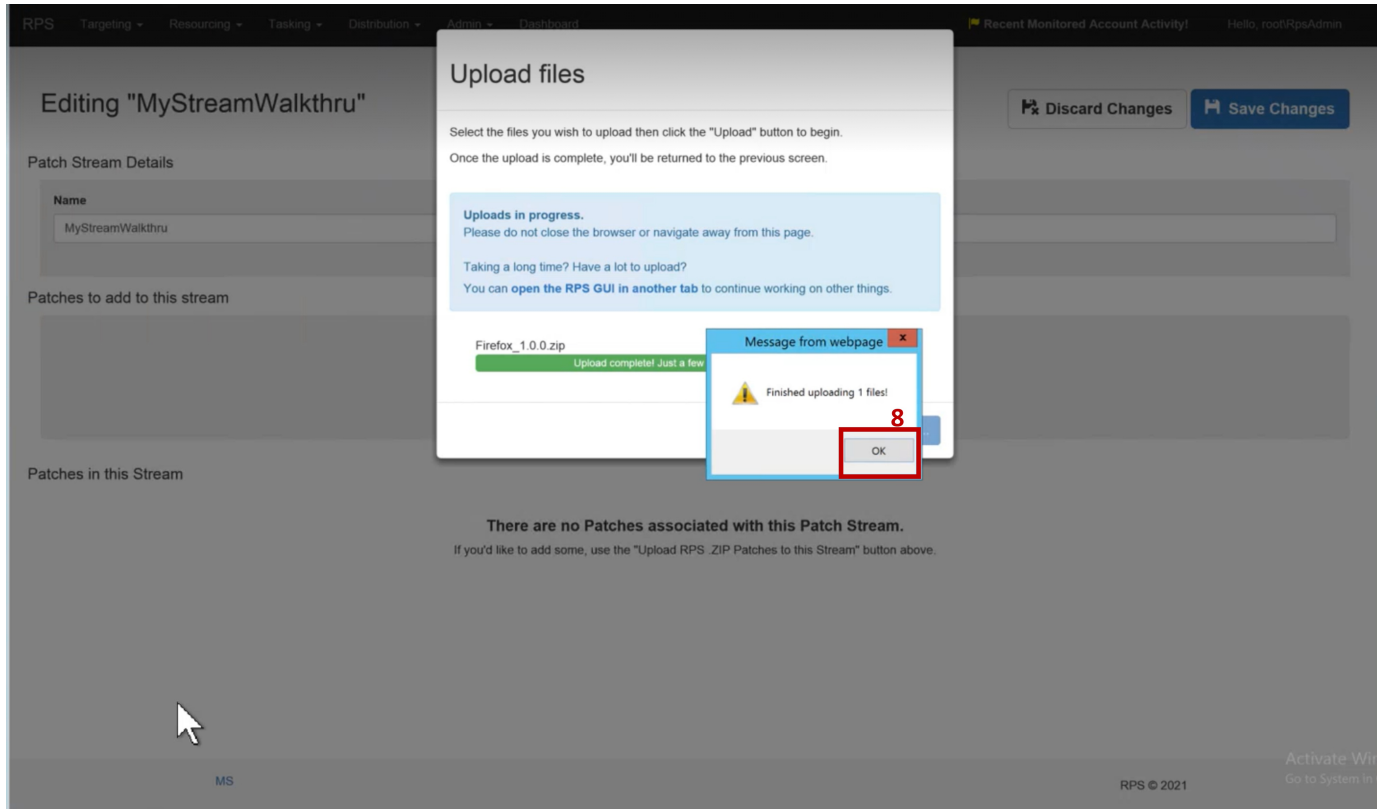


Figure 5: Finished uploading patch file(s).

### **⚠ IMPORTANT**

Do not navigate away from the page while uploads are in progress. You cannot submit partially uploaded patches. If you are uploading extremely large files, you can continue working by clicking the link "open the RPS GUI in another tab".

9. The upload modal will close and the patch stream editing page will refresh, showing the newly uploaded patches.

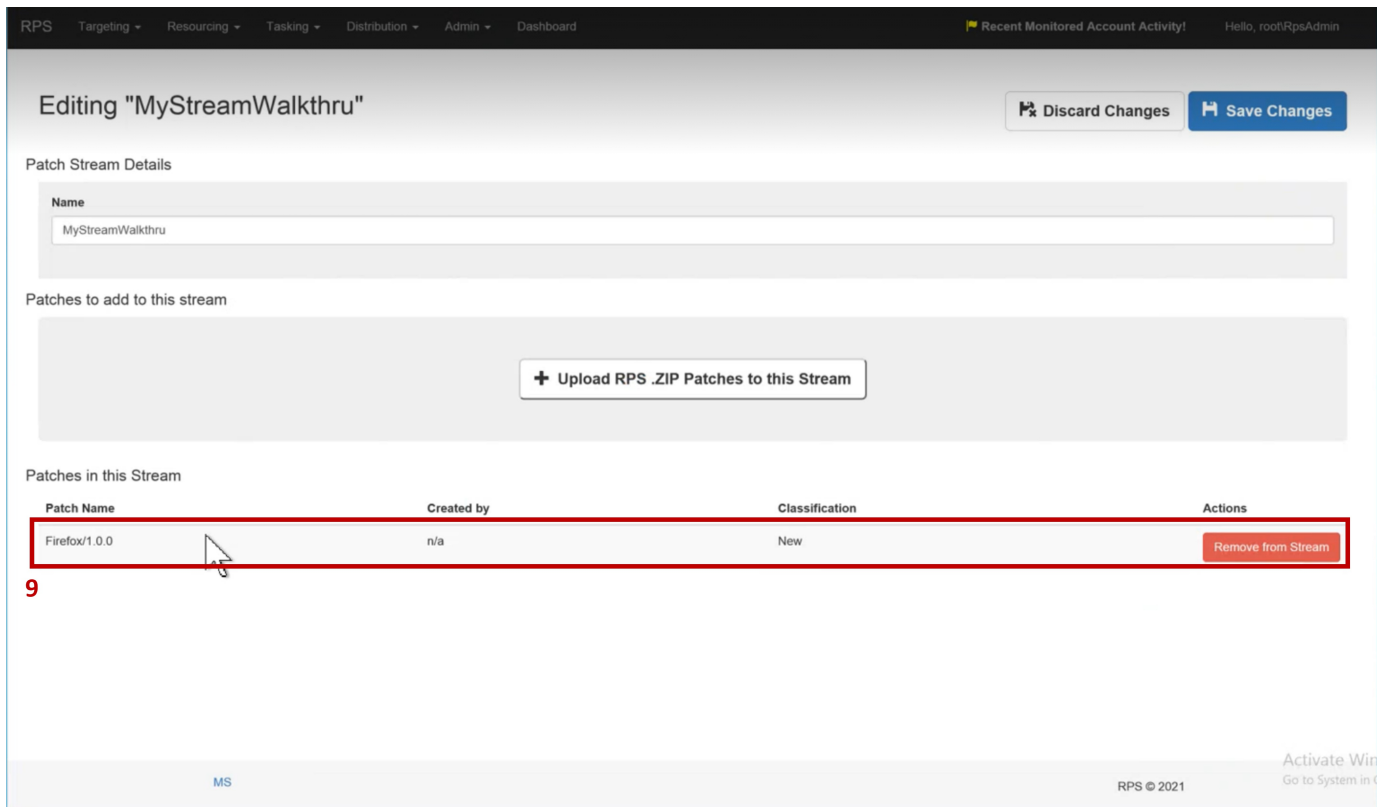


Figure 6: patch stream Edit form with uploaded patches.

10. If the patches provided are invalid, you will see errors specifying what values in the **.Rps file** are non-complitic.

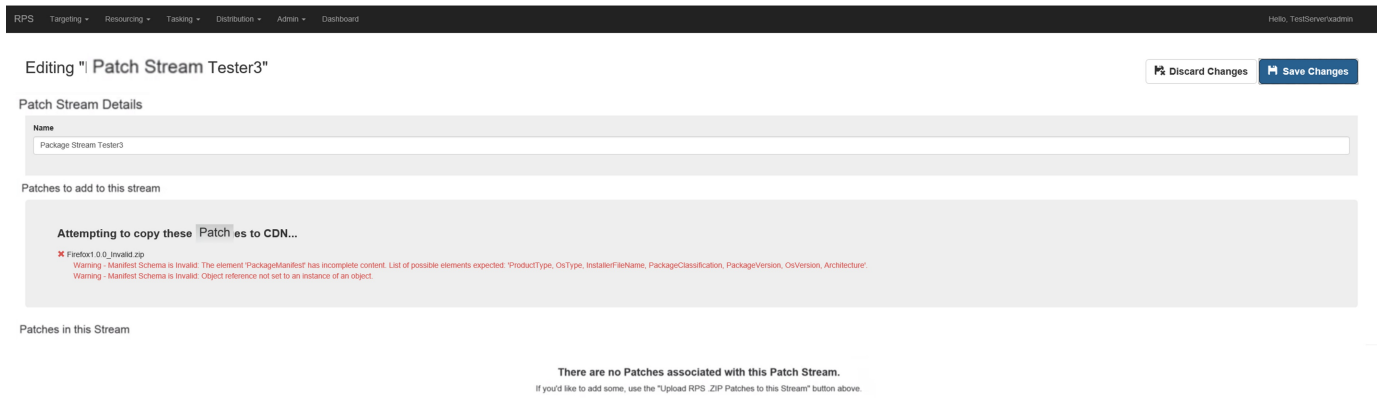


Figure 7: Uploaded patch showing validation exceptions.

NOTE: Uploaded files are stored in a temporary location. These files will be moved to the CDN once a patch stream is created. If a patch stream is created or updated, then the temporary patch uploads will be removed once they are copied to the CDN. If a patch stream is not completed created (e.g. the Save Changes button is not clicked), then the uploaded patches will remain in the temporary upload location for a configurable amount of time (default is 30 days). After the configurable amount of time has passed, the uploaded patches in the temporary upload location will be automatically removed.



# How to Transfer Non-Patching Content Delivery with RPS

Last updated on August 4, 2021.

**Document Status:** Document Feature Complete as of August 4, 2021; PENDING EXTERNAL REVIEW.

## Overview

The Rapid Provisioning System (RPS) can replicate non-patching content, such as log shipping information, to nodes within the local domain and to nodes outside of the local domain. Non-Patching content delivery can be achieved using PowerShell and RPS cmdlets.

## Prerequisites

Users performing non-patching content delivery should have the RPS Patching role or be assigned RPS Admin permissions, and have knowledge and experience using PowerShell and RPS.

## Resource Type Definition

To create transferable non-patching content, the *Resource Type Definition* must be set. This property will enable RPS to include this information when it configures DFSR and/or BITS for replication to the target node. For information on enabling DFSR and BITS, see [How to Enable and Disable CDN Communication](#).

To set the *Resource Type Definition*:

1. Login to RPS and open a PowerShell window as Administrator.
2. Execute the following PowerShell code block:

```
$type = Set-RpsResourceType -Name 'Logs' -IsContentDistribution -CdnDirection Downstream  
$null = Set-RpsTypeProperty -Parent $type -Name 'DisplayName' -PropertyType Text
```

The above command will configure any *Resource Item* (content) of type 'Logs' to be marked as transferable content.

The `-CdnDirection` parameter determines the direction of data transfer and has two possible options:

- *Downstream*: the direction for content transfer will be to child nodes.
- *Upstream*: the direction for content transfer will be to parent nodes.

### NOTE

'Logs' and 'DisplayName' are example values only and should be customized in the script to user requirements.

## Create a Resource Item

Next, the content must be created using the `New-ResourceItem` cmdlet. The following code block will set variables and create a new *ResourceItem*.

```
$contentType = 'Logs'  
$contentName = 'Firewall Logs'  
$newContent = New-RpsResourceItem -Type $contentType -Name $contentName -IsGlobal $true -Properties @{ DisplayName =  
$contentName }
```

### NOTE

Users must specify the `$ContentType` and `$ContentName` variables.

The following code block sets the *FolderID* and the *FolderResourceID* on the newly created content:

```
$folderId = [Rps.Api.Utils.HashingUtils]::Reverse($newContent.Id)

$newContent.FolderResourceId = $folderId
$newContent.Update()
```

The following code block gets the CDN Folder, creates a new directory in the CDN folder, and configures Folder Data in CMDB:

```
$localNode = Get-RpsLocalNode
$cdnPath = $localNode.CdnPath

$null = New-Item -Path $cdnPath -Name $folderId -ItemType Directory

$fileCatalog = [Rps.Api.Utils.FileUtils]::BuildFolderCatalog($(Join-Path -Path $cdnPath -ChildPath $folderId))
[Rps.Api.Utils.FileUtils]::CreateFolderCatalog($fileCatalog, $contentName, $folderId)
```

## Assign the Resource Item (Content) to a Target Node

Now the *ResourceItem* must be assigned to a target in order to be transferred. This is accomplished with the following code block:

```
$regionTarget = Get-RpsTargetItem -Name <app.region.rps> -Type VirtualMachine
$siteTarget = Get-RpsTargetItem -Name <app-s.region.rps> -Type VirtualMachine

$null = $newContent.AssignTo($regionTarget)
$null = $newContent.AssignTo($siteTarget)
```

### **NOTE**

Users must specify the target node in the `-Name` parameter of this code block.

## PowerShell Script Example

The below script can be copied and pasted to perform all of the discussed actions at once.

```

# Variables
$contentType = 'Logs'
$contentName = 'FirewallLogs'

# Create Type Definition for new Content
$type = Set-RpsResourceType -Name 'Logs' -IsContentDistribution -CdnDirection Downstream
$null = Set-RpsTypeProperty -Parent $type -Name 'DisplayName' -PropertyType Text

# Create the resource item with the same type
$newContent = New-RpsResourceItem -Type $contentType -Name $contentName -IsGlobal $true -Properties @{ DisplayName = $contentName }

# Get the folder Id. FolderId is the reverse of the content resource item
$folderId = [Rps.Api.Utils.HashingUtils]::Reverse($newContent.Id)

# Set FolderResourceId on the new content.
$newContent.FolderResourceId = $folderId
$newContent.Update()

# Get CDN Folder
$localNode = Get-RpsLocalNode
$cdnPath = $localNode.CdnPath

# Create folder
$null = New-Item -Path $cdnPath -Name $folderId -ItemType Directory

# Configure Folder data in CMDB
$fileCatalog = [Rps.Api.Utils.FileUtils]::BuildFolderCatalog($(Join-Path -Path $cdnPath -ChildPath $folderId))
[Rps.Api.Utils.FileUtils]::CreateFolderCatalog($fileCatalog, $contentName, $folderId)

# Assign content to targets on each node you want to send the data to
$regionTarget = Get-RpsTargetItem -Name <app.region.rps> -Type VirtualMachine
$siteTarget = Get-RpsTargetItem -Name <app-s.region.rps> -Type VirtualMachine

$null = $newContent.AssignTo($regionTarget)
$null = $newContent.AssignTo($siteTarget)

```

### **NOTE**

The `$contentType` and `$contentName` variables, and the `-CdnDirection` and `-Name` parameter values in the above script are examples only and must be customized by the user for the script to execute properly.

# How to Load a Patch Stream

Last updated on July 7, 2021.

Last Reviewed and Approved on PENDING REVIEW

## Introduction

This document provides step-by-step instructions for loading a patch stream into RPS.

## Prerequisites

- User must have permissions to the *CDN* folder.
- User must be part of the *ContentCreators* Active Directory group.
- Patches must have been previously created. Please visit [How to Create an RPS Patch](#) for detailed instructions.

### NOTE

Users may see "patch" and "package" used interchangeably in the log outputs during this process.

## How to Create a Patch Stream Using PowerShell Cmdlets

### NOTE

If patches have already been copied to the node, proceed to step 3.

1. On an external media device (e.g. USB thumb drive), create a new directory.
2. Zip patch files and and move them to the newly created directory.
3. Log into the node where you will be creating the patch stream.
4. Open a PowerShell window as an Administrator.
5. Change your directory to the RPS ContentStore using the following command:

```
cd C:\ContentStore
```

6. Import the RPS API module using the following command:

```
Import-Module C:\ContentStore\Modules\Rps-Api
```

7. Use the *New-RpsPatchStream* cmdlet to create the patch stream. There are two required parameters and two optional parameters for use with the *New-RPSPatchStream* cmdlet:

The following parameters are **required**:

PARAMETER NAME	TYPE	DESCRIPTION
Name	string	Names the new patch stream (max length = 255). The name can be arbitrary and is valid only to the user.
Path	string	Specifies current location of patches (e.g. on the external media)

The following parameters are optional:

PARAMETER NAME	TYPE	DESCRIPTION
PatchExtensions	List of String	Used to list file extensions of patches that will be searched for in the location specified in the <i>Path</i> parameter. The files must be valid ZIP archives.
Recurse	Switch	Used to search all sub-directories of <i>path</i> parameter for patches.

The following is an example of how these parameters can be used with the *New-RPSPatchStream* cmdlet to create a patch stream:

```
New-RpsPatchStream -Name MyPatchStream1 -Path E:\Patches -PatchExtensions ".zip"
```

In this example, the user has created a patch stream named "MyPatchStream1" containing patches from the `E:\Patches\` directory.

8. The following actions will occur once the *New-RPSPatchStream* command has been successfully executed:
  - a. A patch stream item will be added to the Configuration Management Database (CMDB).
  - b. The patches will be added to the CMDB, one for each patch found in the location specified by the *path* parameter.
  - c. The patches will be copied into the CDN directory, enabling them to be replicated across the CDN.

#### NOTE

The initial state of a patch stream is "Pending" and therefore no assignments between patches and targets will be made.

- d. The patch stream will be approved via PowerShell or the Web UI. For more information on how to approve and reject patch streams, please visit [How to Approve and Reject Patch Streams](#).

# How to Add a Patch Using PowerShell

Last updated on July 30, 2021.

Last Reviewed and Approved on PENDING REVIEW

## Introduction

This document describes the step-by-step PowerShell instructions for:

- Adding a patch without a patch stream.
- Adding a patch to a patch stream.
- Update the patches in a patch stream.

For instructions on how to create a patch using the RPS User Interface, see [How to Create a Patch Stream](#).

### NOTE

Users may see "patch" and "package" used interchangeably in the log outputs during this process.

## Prerequisites

1. Open a PowerShell window as an Administrator.

### NOTE

User must be a Global Admin, Patch Admin, or Patch Creator.

2. Change your directory to the RPS ContentStore. For example:

```
cd C:\ContentStore
```

### NOTE

The ContentStore directory location is user changeable. The current ContentStore directory location can be determined using the following PowerShell snippet:

```
(Get-RpsLocalNode).ContentPath
```

3. Import the RPS API module.

```
Import-Module C:\ContentStore\Rps-API
```

### NOTE

The ContentStore directory location is user changeable. The current ContentStore directory location can be determined with the following PowerShell snippet:

```
(Get-RpsLocalNode).ContentPath
```

## New-RpsPatch

PowerShell cmdlet that is used to create a new RPS Patch.

For more information on patches, see [How to Create an RPS Patch](#). For instructions on how to create a patch stream using PowerShell, see [How to Load a Patch Stream](#).

#### Parameters

PARAMETER NAME	TYPE	DESCRIPTION	REQUIRED
Path	string	Path of the patch.	False
PatchStream	string	Patch Stream object that the patch will be added to.	False

## Adding a Patch

1. Follow the [Prerequisite](#) steps to add the RPS PowerShell API module.
2. Add the patch, without or with a patch stream, using the `New-RpsPatch` cmdlet and parameters mentioned, above.

a. Add a patch **without** a patch stream example:

```
New-RpsPatch -Path C:\Patches\myPatch.zip
```

b. Add a patch **with** a patch stream example:

```
New-RpsPatch -Path C:\Patches\myPatch.zip -PatchStream $myPatchstream1
```

3. After the Patch command has been executed, the following will happen:
  1. The patch item will be added to the CMDB.
  2. The patch file will be copied in to the CDN directory, so it is able to replicate across the CDN.

## Get-RpsPatch

PowerShell cmdlet that gets an RPS Patch that has been added to the CMDB using the `New-RPSPatch` cmdlet.

#### Parameters

PARAMETER NAME	TYPE	DESCRIPTION	REQUIRED
Id	GUID	ID of the patch in CMDB.	False

## Get-RpsPatchStream

PowerShell cmdlet that gets an RPS Patch Stream that has been added using the `New-RpsPatchStream` cmdlet.

For instructions on how to create a patch stream using PowerShell, see [How to Load a Patch Stream](#).

#### Parameters

PARAMETER NAME	TYPE	DESCRIPTION	REQUIRED
Id	GUID	ID of the patch in CMDB.	False

# Update the Patches in a Patch Stream

1. Follow the [Prerequisite](#) steps to add the RPS PowerShell API module.
2. Retrieve the patches you would like to set for the patch stream by using the `Get-RpsPatch` cmdlet.

```
$myFirstPatch = Get-RpsPatch -Id <GUID>  
$mySecondPatch = Get-RpsPatch -Id <GUID>  
$myThirdPatch = Get-RpsPatch -Id <GUID>
```

3. Assign the patches to a collection.

```
$myPatches = $myFirstPatch, $mySecondPatch, $myThirdPatch
```

4. Retrieve the patch stream you'd like to add the patches to by using the `Get-RpsPatchStream` cmdlet.

```
$myPatchstream = Get-RpsPatchStream -Id <GUID>
```

5. Add those patches to the patch stream using the `Add-RpsPatch` cmdlet and passing in your collection of patches to the `-Patch` parameter.

```
Add-RpsPatch -PatchStream $myPatchstream -Patch $myPatches
```

## **NOTE**

To replace all patches in a patch stream use the `-Force` parameter. This is "make it so" behavior. The collection of patches passed to `Add-RpsPatch` will become the only patches in the patch stream if the `-Force` switch parameter is used.

```
Add-RpsPatch -PatchStream $myPatchstream -Patch $myPatches -Force
```



# How to Remove a Patch From a Patch Stream

Last updated on July 13, 2021.

Last Reviewed and Approved on PENDING REVIEW

## Introduction

This document describes the step-by-step process for removing an RPS patch from an RPS patch stream using either PowerShell cmdlet or an RPS server website GUI.

### NOTE

Users may see "patch" and "package" used interchangeably in the log outputs during this process.

## How to Remove a Patch From a Patch Stream Using PowerShell Cmdlets

### NOTE

The following process will not remove any assignments or patches from the Content Delivery Network (CDN). Patches can only be removed from patch streams in a "Pending" state.

1. Log into an RPS Server and launch Windows PowerShell or Windows PowerShell ISE as administrator.

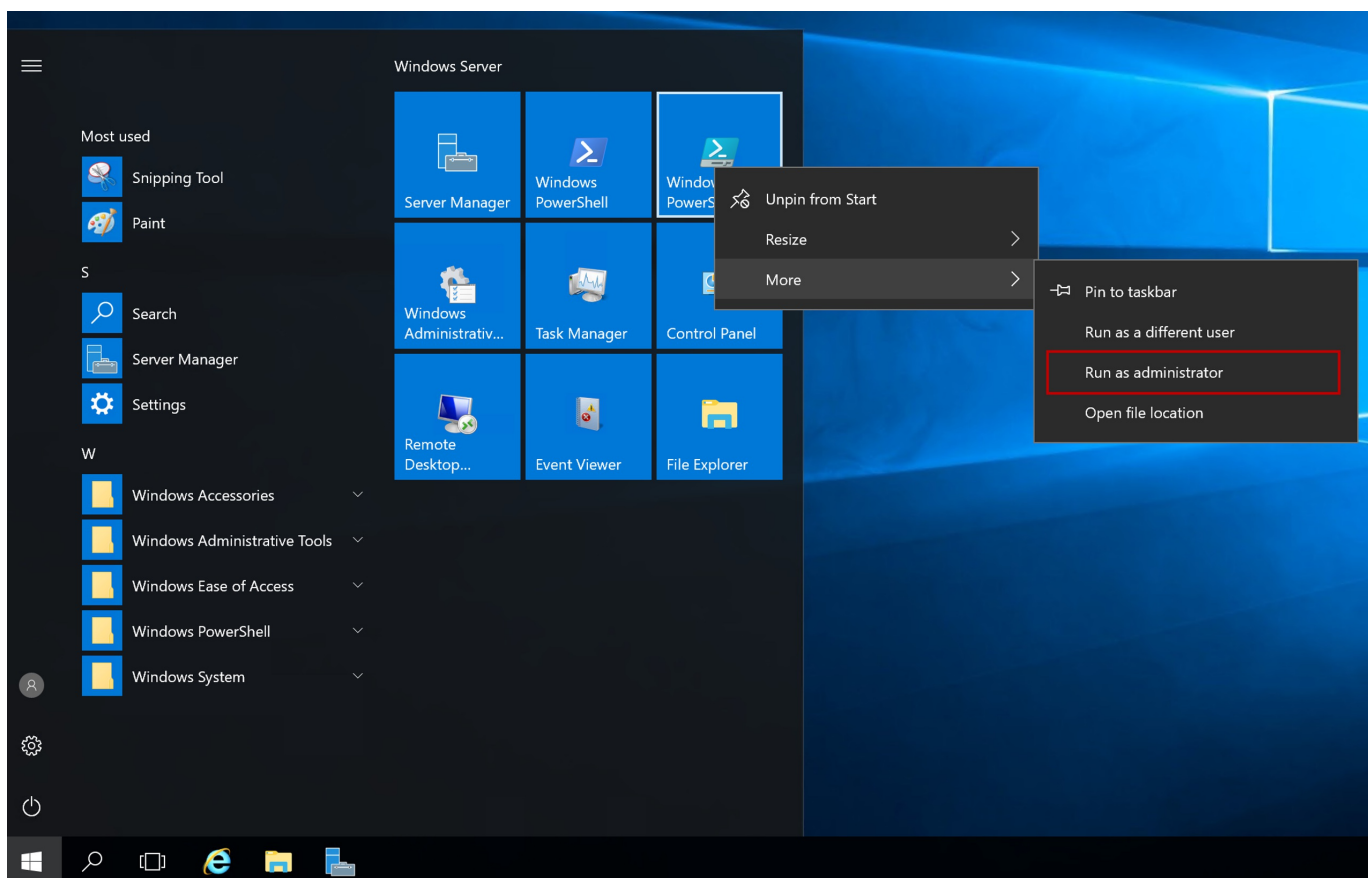


Figure 1: Run PowerShell as administrator.

2. Change directory to the RPS ContentStore using the following command:

```
cd C:\ContentStore
```

3. Import the RPS API module using the following command:

```
Import-Module C:\ContentStore\Modules\Rps-Api
```

4. Remove the patch using the `Remove-RpsPatch` cmdlet as seen in the examples below. The provided examples demonstrate removing a patch by identifying the patch id, name, or patch object and the associated patch stream. The table below describes each parameter.

**NOTE**

`PatchStream` is the only required parameter.

PARAMETER NAME	TYPE	DESCRIPTION	REQUIRED
PatchStream	string	The patch stream you want to remove the patch from.	True
Id	string	Guid of the patch.	False
Name	string	Name of the patch.	False
Patch	string	The patch object.	False
Force	switch	If you would like to bypass the confirmation for execution.	False

Examples:

```
Remove-RpsPatch -Id <GUID> -PatchStream <$stream> -Force
```

```
Remove-RpsPatch -Name <name> -PatchStream <$stream> -Force
```

```
Remove-RpsPatch -Patch <$patch Value> -PatchStream <$stream> -Force
```

## How to Remove a Patch Using the RPS Website GUI

Patch streams can be updated to include the removal of a patch, through a local RPS node server website.

**NOTE**

Patch streams **cannot** be edited once approved. This includes the removal of patches from the patch stream.

1. Log into the RPS website and navigate to the Distribution menu. Select "Patch Streams" from the available options.

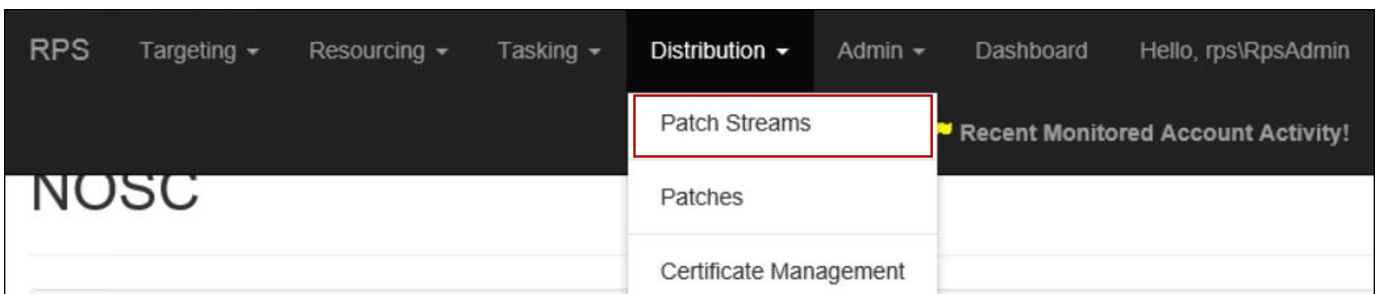


Figure 2: Select "Patch Streams" from the RPS Distribution menu.

2. Select the "Approvals" tab if not currently selected.

3. Click the **Edit** button, as seen in the following screen capture, for the patch stream to be updated.

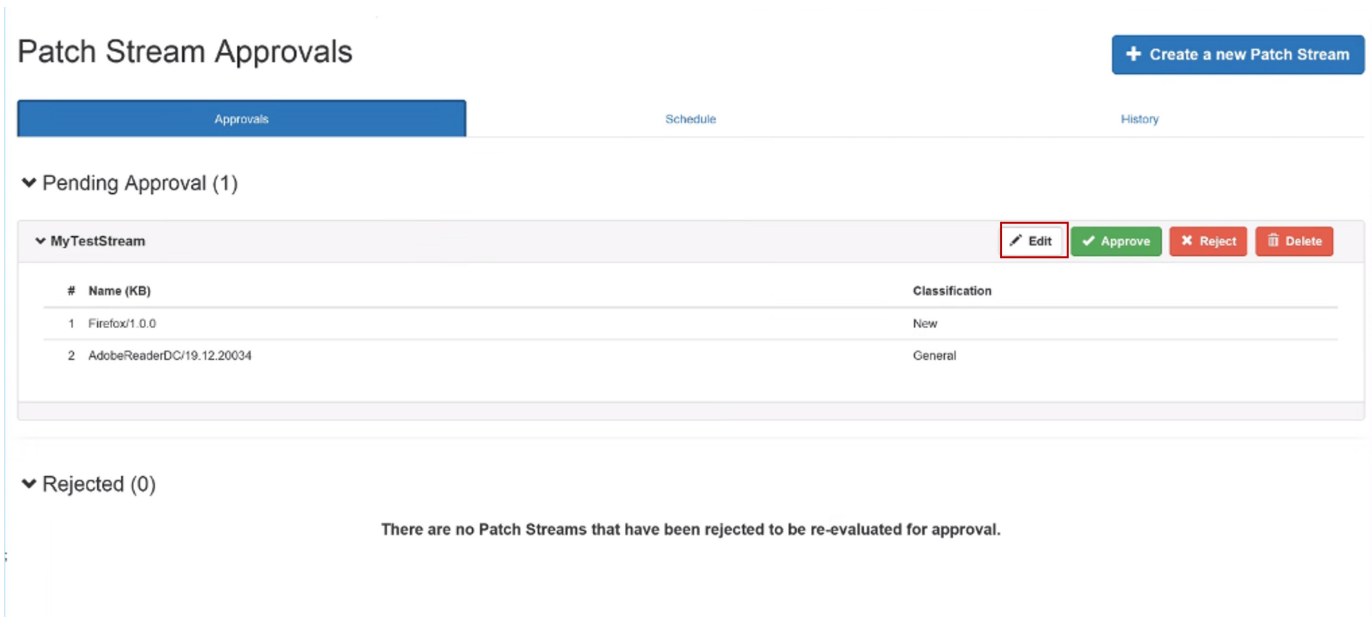


Figure 3: On "Approvals" tab, edit the patch stream MyTestStream.

4. On the right-hand side of the screen click on the red **Remove from Stream** button, as seen in the following screen capture, to remove the patch from the patch stream.

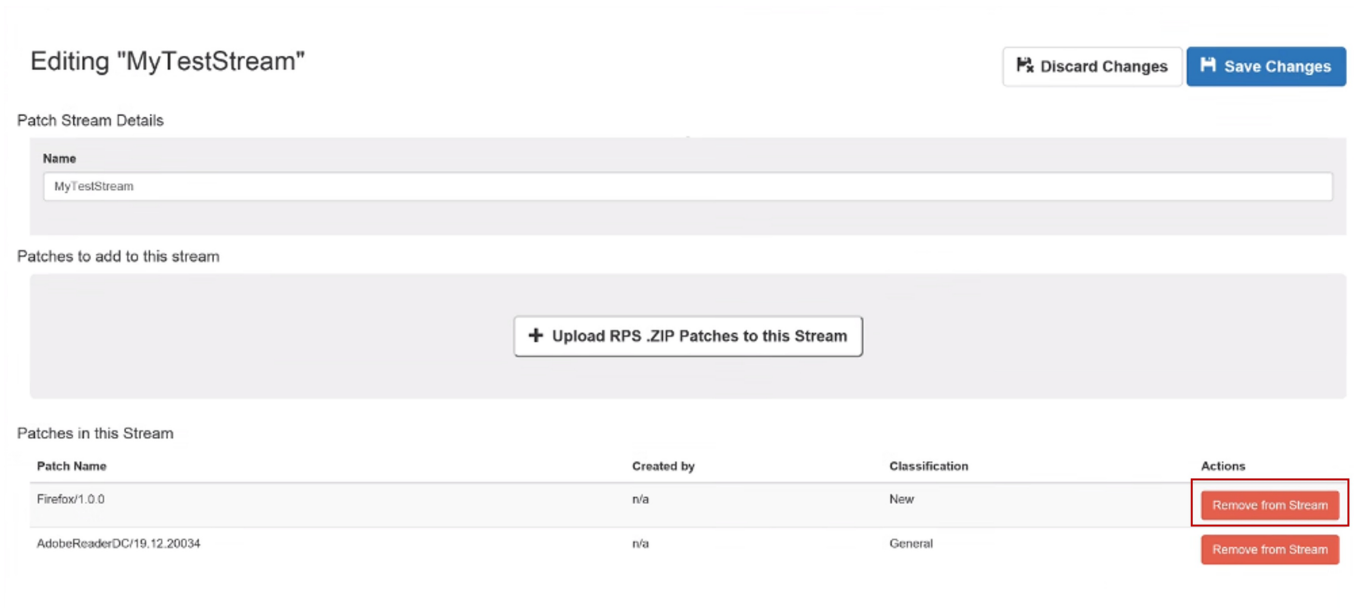


Figure 4: Remove patch from the patch stream MyTestStream.

### **⚠ IMPORTANT**

Users **will not** be prompted to confirm the removal of a patch. Clicking **Remove from Stream** is instantaneous, permanent, and does not require clicking **Save Changes** to commit. The **Discard Changes** button **will not** undo the removal of a patch from a patch stream.

# How to Remove a Patch Stream

Last updated on July 28, 2021.

Last Reviewed and Approved on PENDING REVIEW

## Introduction

This document describes the step by step instructions for removing a Patch Stream.

## How to Remove a Patch Stream in the RPS GUI

1. Launch the RPS GUI and navigate to any of the Patch Stream management pages then select the 'Approvals' Tab.
2. Click on the Delete button of the Patch Stream under 'Pending Approval'

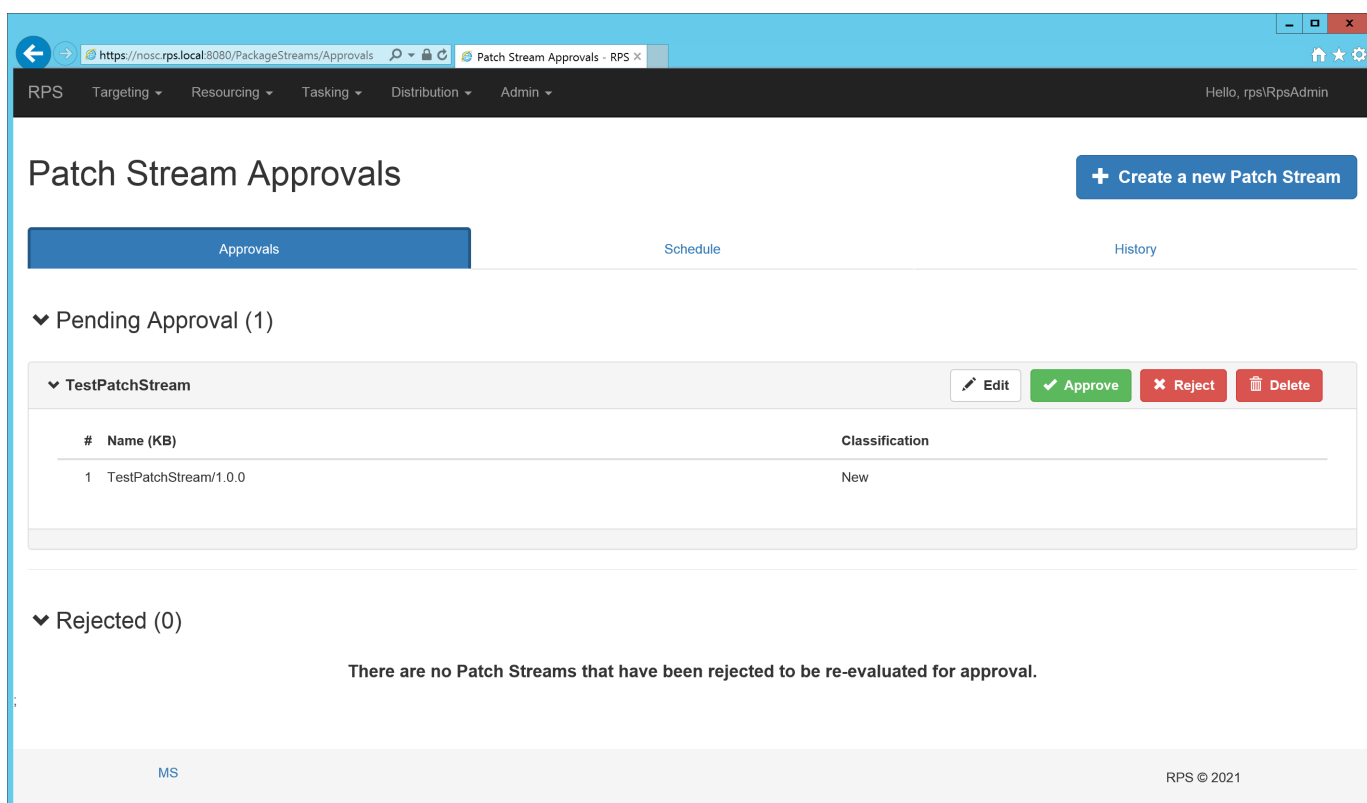


Figure 1: Remove Patch Stream Button.

3. A prompt will appear confirming that you want to delete the patch stream and will list the patches that will be deleted with the stream.

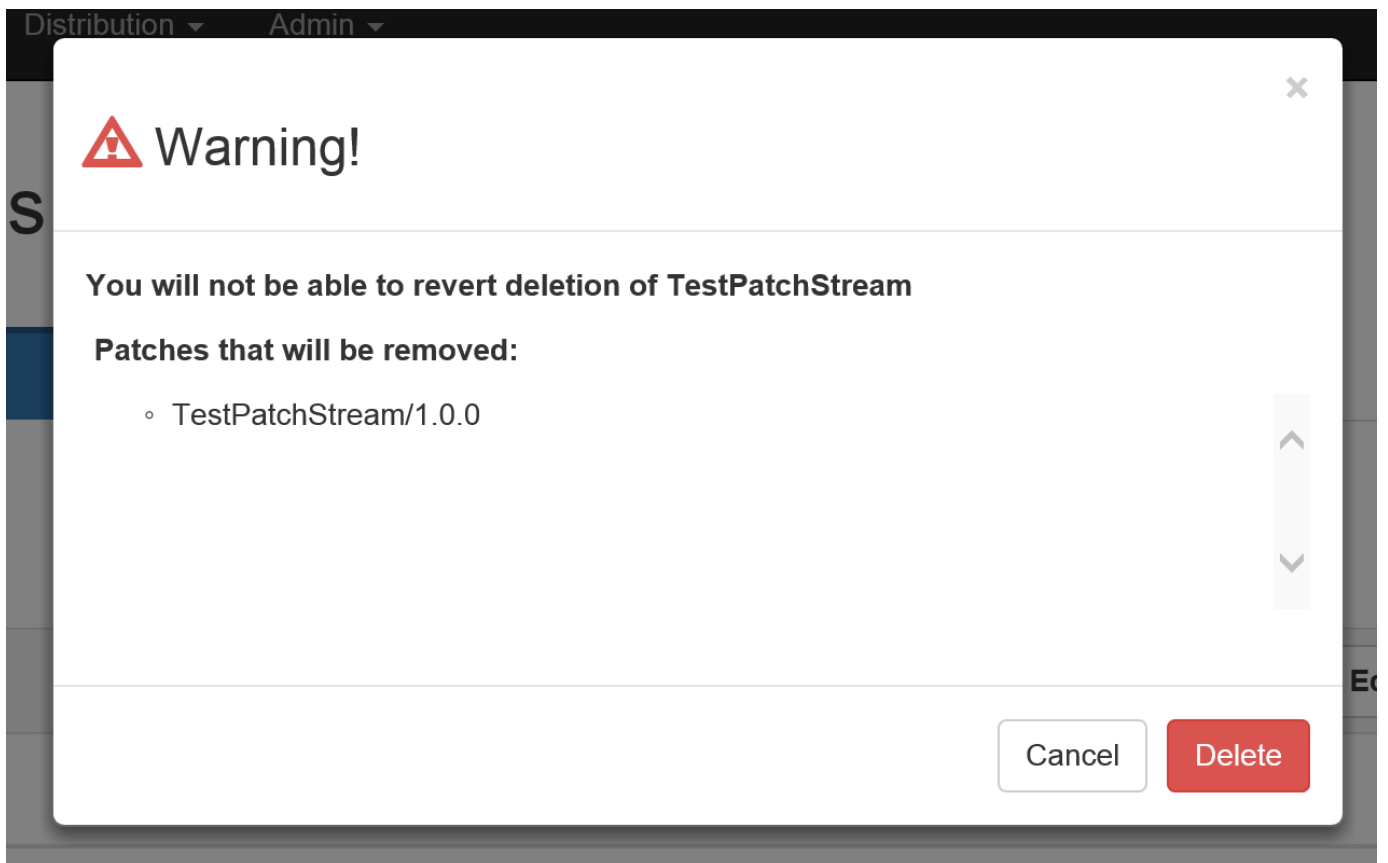


Figure 2: Remove Patch Stream Confirmation.

4. Once you click 'Delete' the patch stream will be deleted along with the patches listed in the prompt.

## How to Remove a Patch Stream with cmdlet

### **i** NOTE

This will not remove the Patches from the Content Delivery Network (CDN) or remove any assignments, you can only remove a Patch Stream in the "Pending" State\*

1. Open a PowerShell window
2. Change your directory to the RPS ContentStore. For example:

```
cd C:\ContentStore
```

### **i** NOTE

The directory for the ContentStore is able to be changed and may be different. User `(Get-RpsLocalNode).ContentPath` PowerShell Command to get the ContentStore directory

3. Import the RPS API module

```
Import-Module C:\ContentStore\Modules\Rps-API
```

### **i** NOTE

The directory for the ContentStore is able to be changed and may be different. User `(Get-RpsLocalNode).ContentPath` PowerShell Command to get the ContentStore directory

#### 4. Remove the Patch Stream using the Remove-RpsPatchStream cmdlet

- Options for the Remove-RpsPatchStream cmdlet

PARAMETER NAME	TYPE	DESCRIPTION
Name	string	Name of the Patch
Patch	string	The Patch object
Force	Switch	Skips the user feedback required to complete the task

- Example:

```
Remove-RpsPatchStream -Name MyPatchStream1
```

#### 5. You will be prompted with what patches will be removed along with the patch stream

```
Windows PowerShell
Copyright (C) 2016 Microsoft Corporation. All rights reserved.

PS C:\Windows\system32> Remove-RpsPatchStream TestPatchStream
You are about to delete the following Patches: TestPatchStream/1.0.0
Are you sure you want to continue
[Y] Yes [N] No [S] Suspend [?] Help (default is "Y"): _
```

- Enter 'Y' or any of the options above to continue

# Sideloaded RPS Patches

Last updated on April 20, 2021.

Last Reviewed and Approved on PENDING REVIEW

This how-to article explains how RPS Administrators and experienced Patch Manager roles can manually stage or position RPS patches to RPS child nodes, in case of Content Delivery Network (CDN) service disruption.

The activity described below is called Sideloaded, which is an acceptable workaround. The method requires remote access to RPS servers and use of PowerShell.

## Intended Audience

RPS Administrators and Patching roles are users of RPS who need to use this article.

## Overview

RPS Patching roles routinely build patch streams and patches for deployment to RPS child nodes.

If the Content Delivery Network (CDN) service communication is disabled, patches may not deploy from RPS parent to RPS child nodes. However, other RPS services will still let the CMDB (Configuration Management Database) update, and will continue to synchronize with the local, child node CMDBs. This is useful so that the local (child) node has a file path to the CDN file location, so that the RPS user can copy the patch files to that location.

In this case the RPS user can manually follow the process below.

### NOTE

CDN notation in this article will mix between CDN, Cdn, and cdn.

## Process

1. Logon to a RPS server.
2. Start an administrative PowerShell session.
3. Perform these four steps:
  1. Get the file hash of the RPS patch.
  2. Get the Folder Resource ID.
  3. Join file paths together.
  4. Copy the RPS patch to the CDN file location.

### Step 1: Get the File Hash of the RPS Patch

```
$filePath = 'C:\Packages\Firefox 70.0.zip'  
  
# Calculate the file hash to get the Resource ID  
$resourceId = [Guid]::new([Rps.Api.Utils.HashingUtils]::ComputeMD5Hash($filePath))
```

### NOTE

When a patch is created, it creates an MD5 Hash ID.

## Step 2: Get the Folder Resource ID

### NOTE

The folder ID is the reverse GUID of the Resource Item.

```
# Get the folder ID. FolderId is the reverse of the content Resource Item.
$folderId = [Rps.Api.Utils.HashingUtils]::Reverse($newContent.Id)
```

## Step 3: Join File Paths Together

```
# Get CDN Folder
$localNode = Get-RpsLocalNode
$cdnPath = $localNode.CdnPath

# Get CDN folder path to store file
$cdnFolderPath = Join-Path -Path $cdnPath -ChildPath $folderId
```

Now that we have the full path to the local CDN, we can copy the RPS patch contents over.

## Step 4: Copy the RPS Patch to the CDN File Location

```
# Copy item
Copy-Item -Path $filePath -Destination $cdnFolderPath -Force
```

## Example

```
$filePath = 'C:\Packages\Firefox 70.0.zip'

# Calculate the file hash to get the Resource ID
$resourceId = [Guid]::new([Rps.Api.Utils.HashingUtils]::ComputeMD5Hash($filePath))

# Get the folder ID. FolderId is the reverse of the content Resource Item.
$folderId = [Rps.Api.Utils.HashingUtils]::Reverse($newContent.Id)

# Get CDN Folder
$localNode = Get-RpsLocalNode
$cdnPath = $localNode.CdnPath

# Get CDN folder path to store file
$cdnFolderPath = Join-Path -Path $cdnPath -ChildPath $folderId

# Copy item
Copy-Item -Path $filePath -Destination $cdnFolderPath -Force
```



# How to Approve and Reject Patch Streams

Last updated on September 02, 2021.

Document Status: Document Feature Complete as of September 02, 2021; PENDING EXTERNAL REVIEW.

## Introduction

This document describes the processes for approving and rejecting RPS patch streams using either PowerShell cmdlet or an RPS server website GUI. RPS patch streams are a collection of one or more patches to be applied to targeted items during a scheduled maintenance window. Click the following links for more information on "[How to Create a Patch Stream](#)" and "[How to Use Maintenance Windows](#)".

When considering the status of patch streams, there are only four valid scenarios allowed within RPS. All other scenarios are considered invalid and are not supported. The processes described within this document are only applicable to the **valid** scenarios found in the following list:

Valid scenarios:

- Pending to Rejected
- Pending to Approved
- Rejected to Approved
- Approved to Approved (Reapprove)

Invalid scenarios:

- Rejected to Pending
- Approved to Pending
- Approved to Rejected

### NOTE

Users may see "patch" and "package" used interchangeably in the log outputs during this process.

## Approve or Reject Patch Streams Using PowerShell Cmdlets

1. Log into an RPS Server and launch Windows PowerShell or Windows PowerShell ISE as administrator.

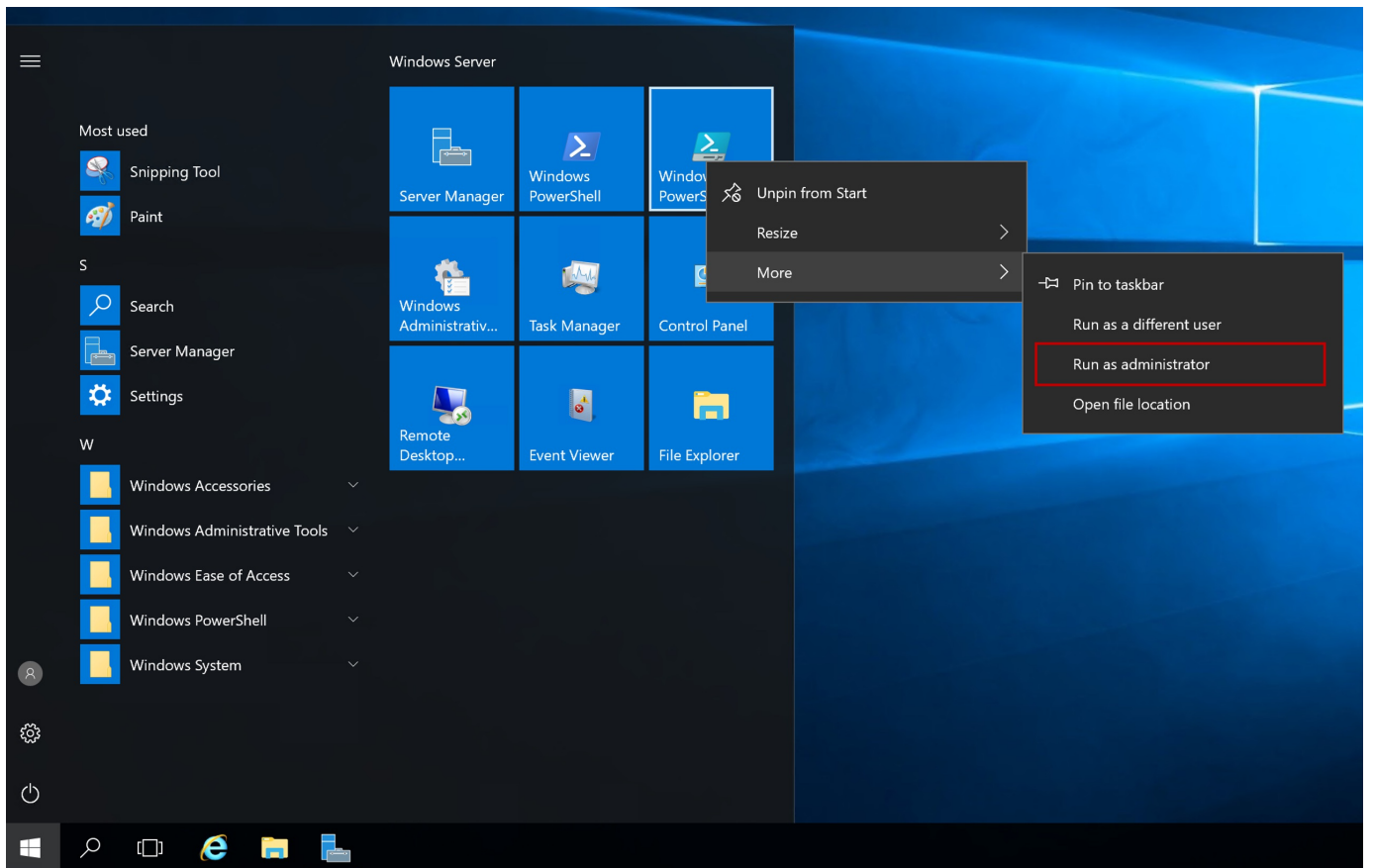


Figure 1: Run PowerShell as administrator.

2. Change directory to the RPS ContentStore using the following command:

```
cd C:\ContentStore
```

3. Import the RPS API module using the following command:

```
Import-Module C:\ContentStore\Modules\Rps-API
```

4. Approve the patch stream using the `Approve-RpsPatchStream` cmdlet as seen in the examples below. Table 1 below describes the available parameters associated with approval and rejection. It is necessary to specify one of these parameters to approve or reject a patch stream.

PARAMETER NAME	TYPE	DESCRIPTION
PatchStream	PackageStream	Patch stream object.
Id	GUID	Guid of the patch stream.
Name	string	Name of the patch stream.

Table 1: `Approve-RpsPatchStream` and `Deny-RpsPatchStream` cmdlet parameters.

Examples of `Approve-RpsPatchStream` cmdlet usage:

```
Approve-RpsPatchStream -Name <name>
```

Or, using the same `Name` parameter as the example above but storing the name as a variable:

```
$myPatchStream = Approve-RpsPatchStream -Name <myPatchStream>
```

The following examples show specifying the patch stream by ID or PatchSteam object:

```
Approve-RpsPatchStream -ID <GUID>
```

```
Approve-RpsPatchStream -PatchStream <myPatchSteam>
```

5. It is possible to reapprove a patch stream to correct for assignment changes between targets and patches. For example, targets from one target group move to a different target group after a patch stream has already been approved. Use the `Approve-RpsPatchStream` cmdlet as described above to reapprove the patch stream.
6. To reject a patch stream use the `Deny-RpsPatchStream` cmdlet as seen in the next examples. Use the parameters listed in Table 1 above to complete the command.

```
Deny-RpsPatchStream -Name <name>
```

```
Deny-RpsPatchStream -ID <GUID>
```

```
Deny-RpsPatchStream -PatchStream <$stream>
```

## Approve or Reject Patch Streams Using the RPS Website GUI

### NOTE

Patch streams **cannot** be edited once approved.

1. Log into the RPS website and navigate to the Distribution menu. Select "Patch Streams" from the available options.

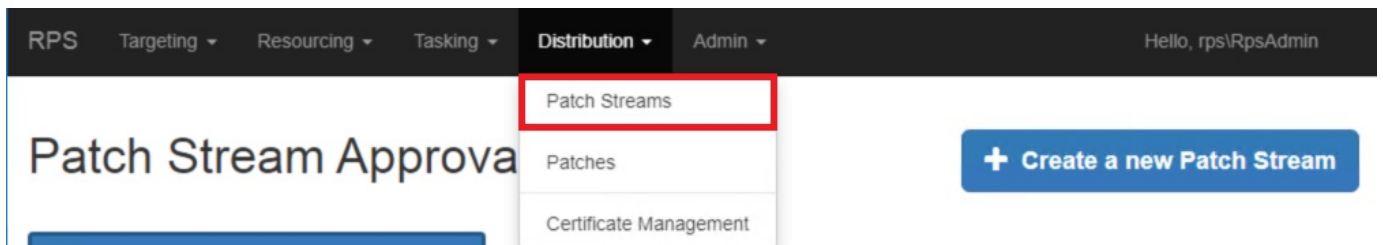


Figure 2: Select "Patch Streams" from the RPS Distribution menu.

2. Select the "Approvals" tab if not currently selected.
3. A patch stream in the "Pending Approval" state can be approved by selecting the green **Disseminate** button on the right-hand side of the "Approvals" tab.

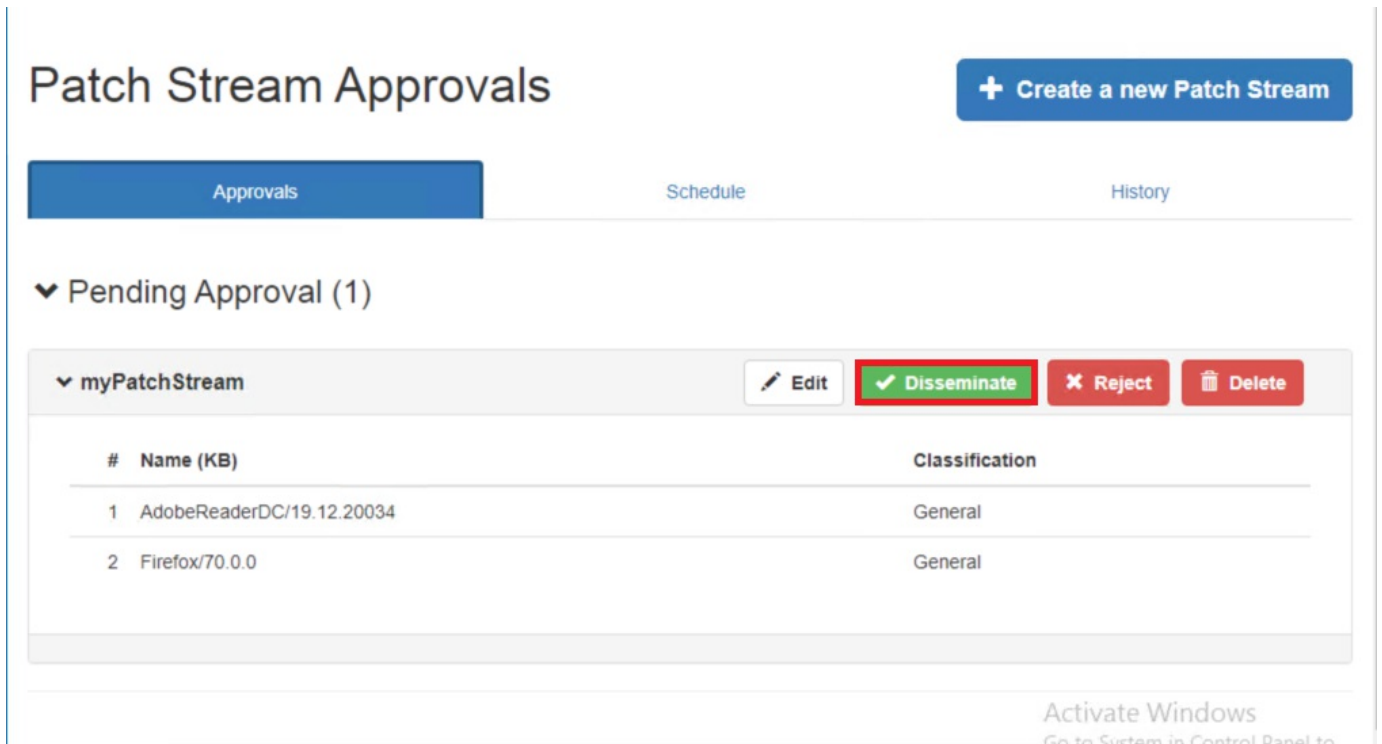


Figure 3: Disseminate the patch stream.

4. Confirm the approval by clicking **Disseminate** on the confirmation window as seen below.

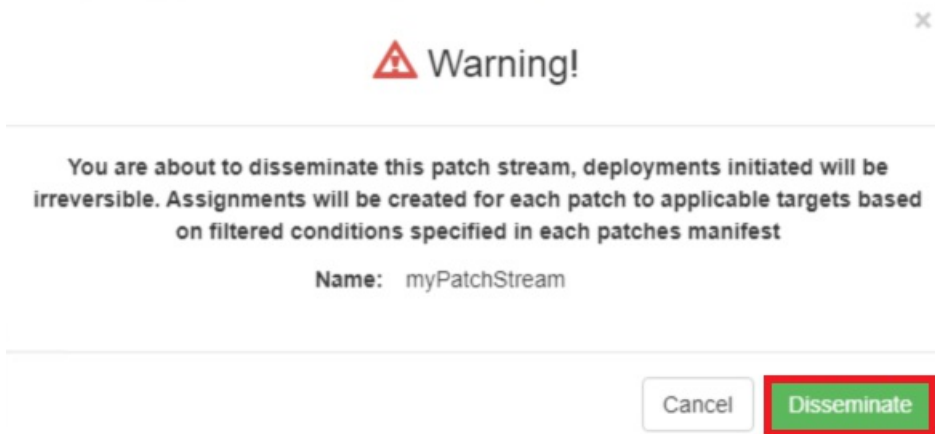


Figure 4: Confirmation Warning Window.

5. Patch streams in the pending approval state can be rejected by selecting the red **Reject** button on the right-hand side of the "Approvals" tab.

# Patch Stream Approvals

[+ Create a new Patch Stream](#)

Approvals      Schedule      History

▼ Pending Approval (1)

▼ myPatchStream      [Edit](#)      [Disseminate](#)      [Reject](#)      [Delete](#)


#	Name (KB)	Classification
1	AdobeReaderDC/19.12.20034	General
2	Firefox/70.0.0	General

Activate Windows  
Go to System in Control Panel to

Figure 5: On "Approvals" tab, reject the patch stream.

6. Confirm the rejection by clicking **Reject** on the confirmation window as seen below.

✕

 **Warning!**

---

**You are about to reject the patch stream, all future deployments will be halted**

**Name:** myPatchStream

---

Cancel
Reject

Figure 6: Confirmation Warning Window.

7. Rejected patch streams will appear at the bottom of the "Approvals" tab under the "Rejected" section. Here individuals with approval permissions can optionally choose to disseminate a previously rejected patch stream by clicking the green **Disseminate** button as seen in the example below.

Approvals Schedule History

▼ Pending Approval (0)

**Looks like you have zero Patch Streams waiting for your Approval.**  
 If you're interested, check out the [History tab](#) to see what's already been deployed.

▼ Rejected (1)

▼ myPatchStream

#	Name (KB)	Classification
1	AdobeReaderDC/19.12.20034	General
2	Firefox/70.0.0	General

Figure 7: Disseminate previously rejected patch stream.

- Confirm the approval by clicking **Disseminate** on the confirmation window as seen below.

⚠ **Warning!** ×

---

**You are about to disseminate this patch stream, deployments initiated will be irreversible. Assignments will be created for each patch to applicable targets based on filtered conditions specified in each patches manifest**

**Name:** myPatchStream

---

Figure 8: Confirmation Warning Window.

- It is possible to disseminate a patch stream to correct for assignment changes between targets and patches. For example, targets from one target group move to a different target group after a patch stream has already been disseminated.

Disseminating patch streams from the RPS website GUI can be accomplished from the "Patch Streams" page "History" tab by selecting the desired patch stream and clicking the green **Disseminate** button as seen in the example below.

# Patch Stream Deployment History

+ Create a new Patch Stream

Approvals

Schedule

History

## Filters

**Node**  
NOSC

**Patch Stream**  
Any...

**Patch**  
Any...

Apply

Clear

## Deployment History - NOSC

▼ NOSC

Child Nodes

TCN

> myPatchStream ⚠ Deployment Incomplete ✔ Disseminate

Approved by rps\RpsAdmin on 2021/09/02 14:27

Figure 9: Disseminate patch stream.

10. Confirm the approval by clicking **Disseminate** on the confirmation window as seen below.

Warning!

You are about to disseminate this patch stream, deployments initiated will be irreversible. Assignments will be created for each patch to applicable targets based on filtered conditions specified in each patches manifest

Name: myPatchStream

Cancel Disseminate

Figure 10: Confirmation Warning Window.

# Viewing Patch Stream Deployment Telemetry

Last updated on March 4, 2021.

Last Reviewed and Approved on PENDING REVIEW

This article helps RPS Patch managers understand how to track the deployment progress of RPS patching, following their approval.

## Intended Audience

RPS patching roles such as Patch Stream Approvers need to use this article to track the status of their deployments.

## Overview

Observe Patch Stream History, by logging into the RPS website. Or, connect to RPS using PowerShell and the RPS PowerShell cmdlets, discussed later in this article.

On the RPS website, go to the Distribution menu and select "Patch Streams," shown below.

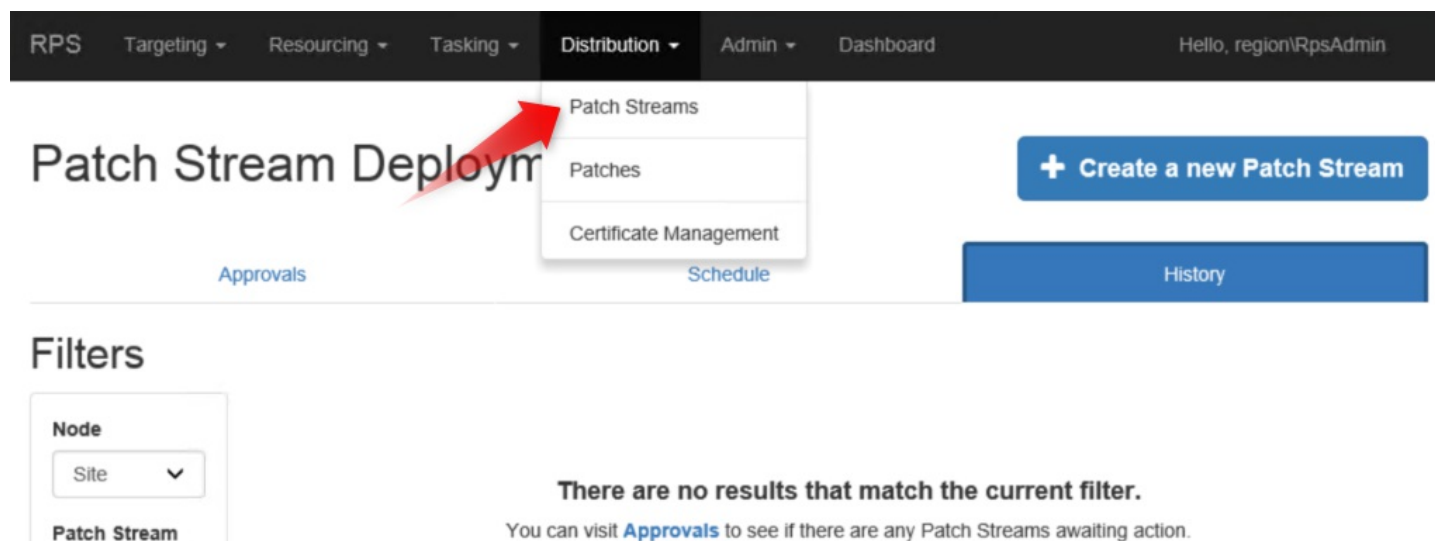


Figure 1: RPS Distribution Menu.

Next, click on History, shown below:



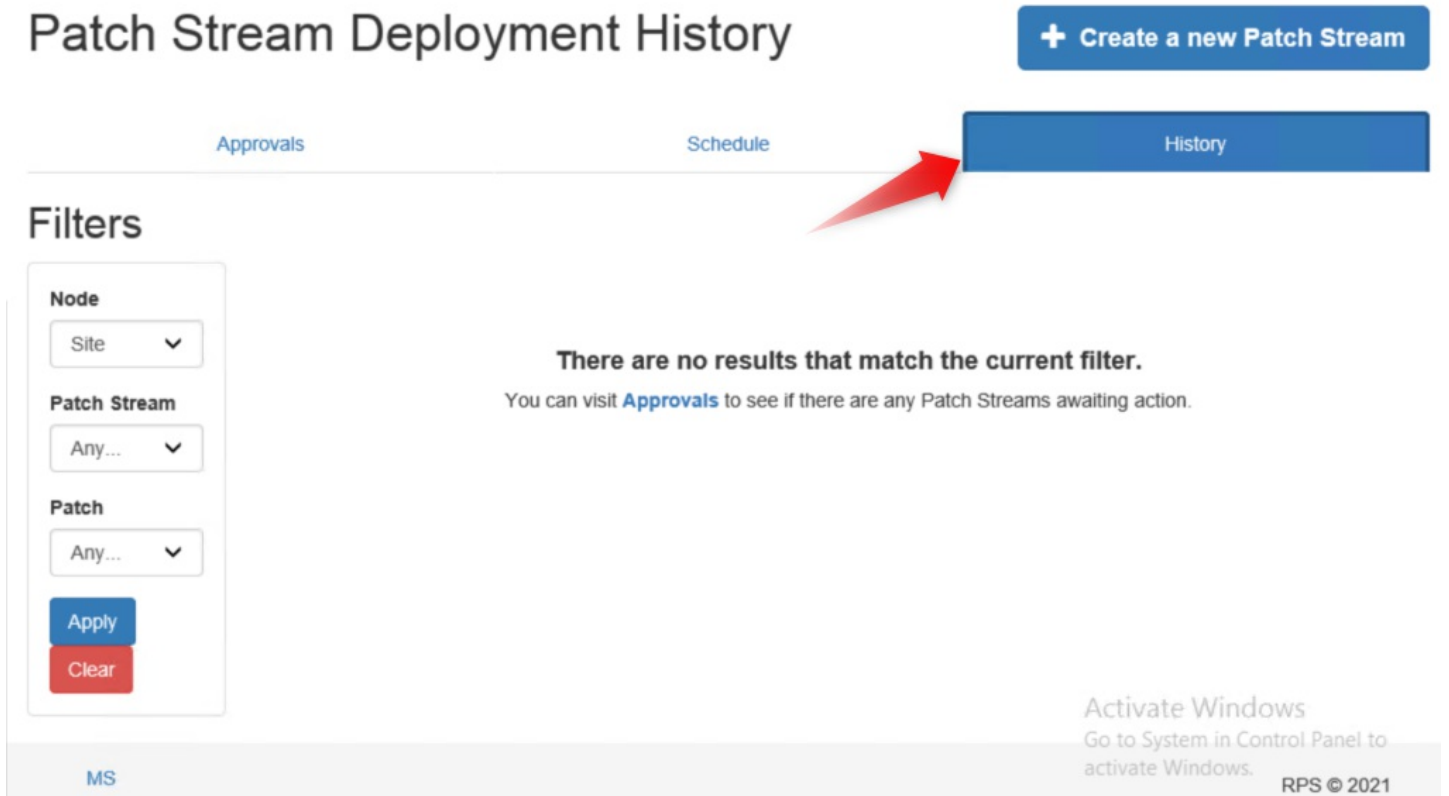


Figure 2: Patch Stream Deployment History.

## For More Information

- Later in this article: "[How RPS Evaluates Deployment Status](#)".
- Link to this article "[RPS Patch Management Workflow](#)" and learn the flow of RPS patches.

## Status Summaries

### Patch Stream Deployment States

An RPS Patch Steam Approver deploys a Patch Stream containing multiple patches. Then, observe the progress of the deployment. RPS will show four states:

- Pending
- Processing
- Successful
- Error

## Patch Assignment Deployment States

For an individual patch assignment, RPS will show several deployment states:

- Pending
- Processing
- Error

- Superseded
- (Successful) IsPresentAndDesirePresent
- (Successful) IsAbsentAndDesireAbsent
- (Failed) IsPresentAndDesireAbsent
- (Failed) IsAbsentAndDesirePresent
- (Failed) IsBelowMinPatchSystemVersion
- (Failed) IsAboveMaxPatchSystemVersion

Later, see "[How RPS Evaluates Deployment Status](#)".

## Viewing Patch Stream Distribution History

From the RPS Website UI

In the RPS Distribution menu, select Patch Streams and the History tab.

1. If there have been no Patch Stream approvals yet, RPS will display a message "*There haven't been any attempts to deploy Patch Streams.*"

### Patch Stream Deployment History

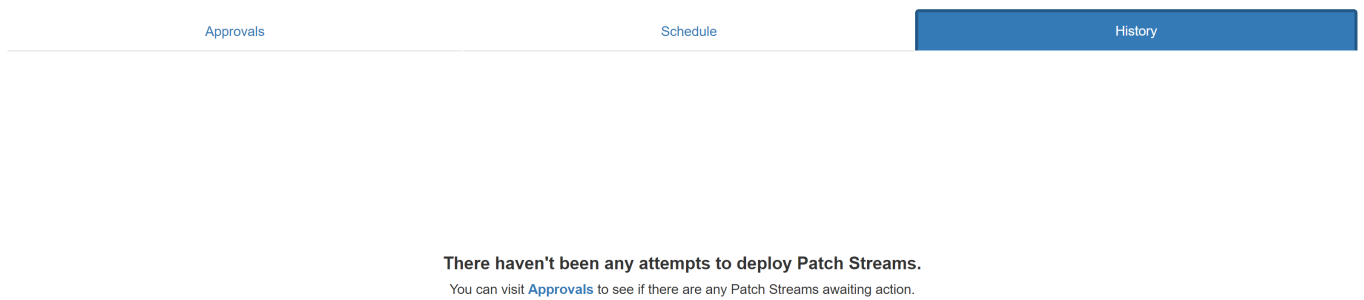
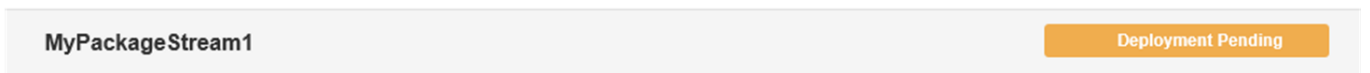


Figure 3: Deployment History

2. If there are Patch Streams that have been previously Approved, the user will see a list of them on the screen, and the overall deployment status for that Patch Stream will be in the top-right of the panel:



(intentionally omitted Patch details and telemetry)

Approved by master\vrpsadmin on 2020/02/06 18:47

Figure 4: Patch Stream Approval

In Figure 4, the Patch Stream Deployment Status is "Deployment Pending."

- If any patch in the package stream has been disabled then you should see a (disabled) title next to the patch name and the patch greyed out.

Figure 7: Disabled Patch

In Figure 7, the Patch - IntelliJ is "Disabled."

## Viewing Patch Stream Telemetry From PowerShell

In PowerShell, users can inspect Patch Stream status using the RPS cmdlet `Get-RpsPatchStream`.

- Retrieve the patch stream by specifying Patch Stream's name in the `-Name` parameter of the `Get-RpsPatchStream` cmdlet:

```
$myPackageStream = Get-RpsPatchStream -Name 'MyPackageStream1'
```

- You can print the Approval Status and approval metadata accessing the `ApprovalStatus`, `ApprovedOn`, and `ApprovedBy` properties:

```
Get-RpsPatchStream -Name 'MyPackageStream1' | Select-Object ApprovalStatus, ApprovedOn, ApprovedBy
```

## Viewing Patch Telemetry

Users in the **Patch Stream Approver** role can determine that the contents of the Patch Stream are not to be deployed to target items by Rejecting the Patch Stream.

### Viewing Patch Stream Status From the UI

Patch Stream Approvers can approve Patch Streams for future deployment.

- Launch the RPS Website.
- Navigate to the Package Page, and select the "History" tab.
- If there have been no Patch Stream approvals yet, RPS will display a message *"There haven't been any attempts to deploy Patch Streams."*

# Patch Stream Deployment History

Approvals

Schedule

History

**There haven't been any attempts to deploy Patch Streams.**  
You can visit [Approvals](#) to see if there are any Patch Streams awaiting action.

Figure 5: Deployment History (empty)

4. If there are Patch Streams that have been previously Approved, RPS will display a list of them with the overall deployment status and show information about 1) the Patch Stream, 2) the Nodes and Targets that have Patches within the Patch Stream, 3) the Patches within the Patch Stream, and 4) the telemetry about the Patch deployments:

Deployment Status	Package Name	Classification	Deployment Date/Time
Pending	Patch Chrome 3	General	
Pending	Patch Compuserve 3	Security	
Pending	Patch Netscape 3	New/Install	

Figure 6: Deployment History

## Viewing Patch Stream Status From PowerShell

In PowerShell, inspect Patch Stream and Patch telemetry RPS cmdlets `Get-RpsPatchStream` and `Get-RpsPatch`.

1. Retrieve the Patch Stream by specifying Patch Stream's name in the:

-Name parameter of the `Get-RpsPatchStream` cmdlet:

```
$myPackageStream = Get-RpsPatchStream -Name 'MyPackageStream1'
```

2. Retrieve the Patches by inspecting that Patch Stream's Packages property:

```
$myPackageStream.Packages | Select-Object Id, PackageResourceItem
```

3. Retrieve the desired Patch by using the `Get-RpsPatch` cmdlet, specifying the `-Id` parameter:

```
$myPackage = Get-RpsPatch -Id '<GUID of Patch>'
```

4. Then, retrieve the status of all of the assigned Target Item deployments by inspecting the Patch Assignments'

`DeployedStatus`, `Ensure`, and `EndPointState` properties:

```
$myPackage.Assignments | Select-Object DeployedStatus, Ensure, EndPointState
```

## How RPS Evaluates Deployment Status

## How Streams, Patches, and Assignments Derive their Status

The Patch Stream Status and Patch Deployment Status are roll-up measures of how a Target Item reports up the successful configuration of an RPS Patch.

A Target Item can be assigned a Patch via a RPS Package Assignment (a templated Resource Assignment).

Within that assignment, RPS tracks the desired state of the Patch (Present (installed) or Absent (uninstalled)).

Whenever the Target Item attempts to reach the desired state, it reports status back through to the Master RPS (Parent Node) Configuration Management Database (CMDB) via the Resource Assignment.

The Target Items will report these statuses on the Patch deployment:

- Pending
- Processing
- Error
- Superseded
- (Successful) IsPresentAndDesirePresent
- (Successful) IsAbsentAndDesireAbsent
- (Failed) IsPresentAndDesireAbsent
- (Failed) IsAbsentAndDesirePresent
- (Failed) IsBelowMinPatchSystemVersion
- (Failed) IsAboveMaxPatchSystemVersion

The Patch within the Stream updates its status as Successful when all the Target Items have successfully reached the desired state (IsPresentAndDesirePresent or IsAbsentAndDesireAbsent) for the assigned Patch.

The Patch Stream updates its status as Successful when all the patches within it have reached the Successful state.

### Patch Assignment and Status Flow Diagram

See the article [RPS Patch Management Workflow](#) diagram for more information on how Patches are assigned.

# How to View All Patches

Last updated on August 3, 2021.

Last Reviewed and Approved on PENDING REVIEW

## From the Graphical User Interface

Using the RPS Graphical User Interface (GUI), users can view patches on the patches page of the RPS GUI

1. Launch the RPS Website and navigate to the **Patches Page** (Distribution > Patches)

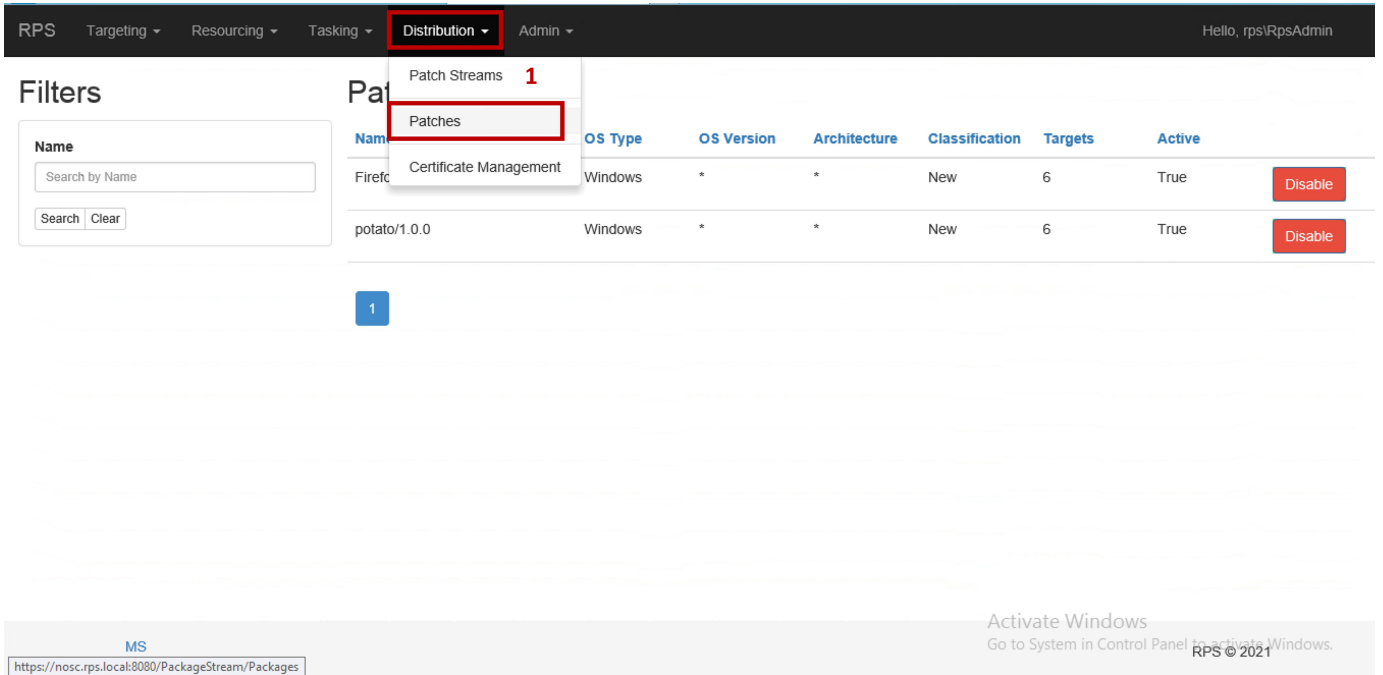


Figure 1: View all patches

2. From this page you can view all patches in the CMDB

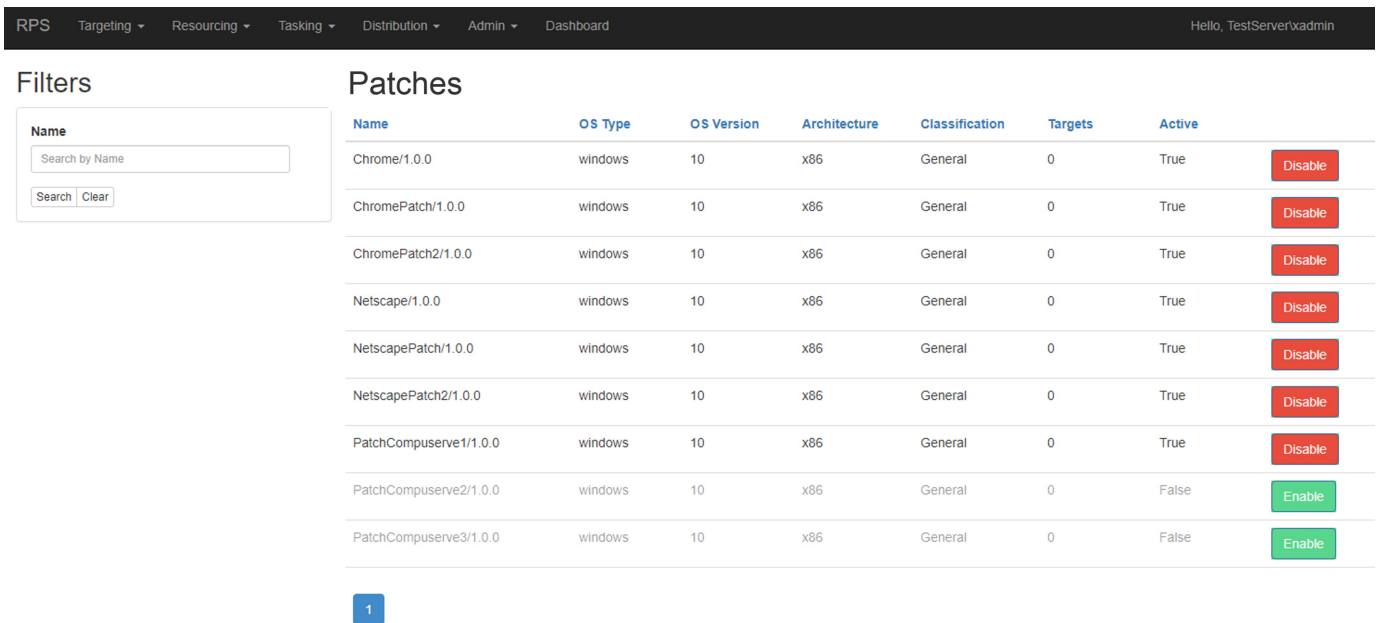
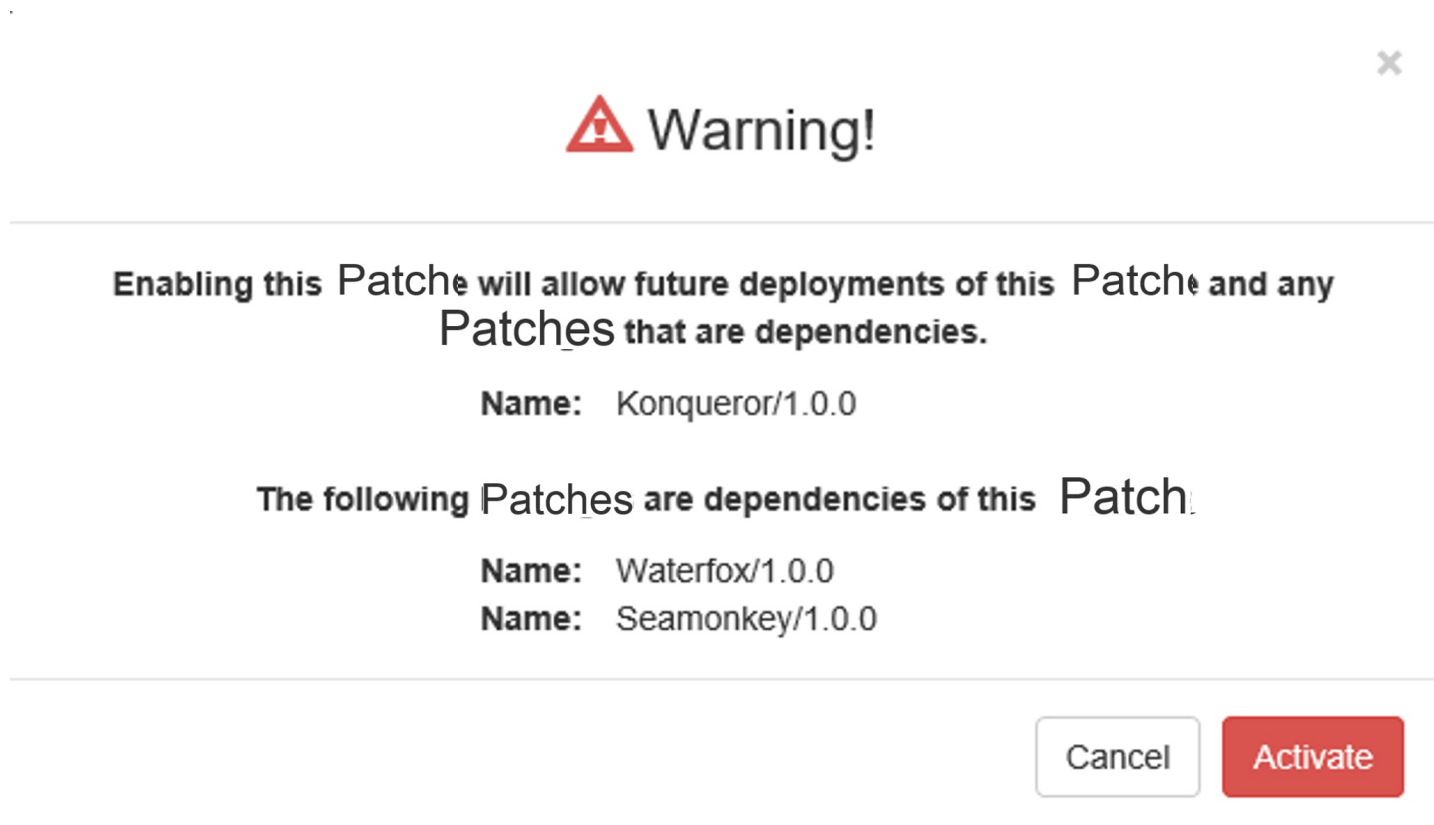


Figure 2: View all patches

## Disable or Enable Patches

## Enable a Patch

To enable a patch, click on the button labeled **Enable** next to the patch you want to enable.



A warning dialog box with a red 'x' in the top right corner. The main heading is 'Warning!' with a red warning triangle icon. Below this, the text reads: 'Enabling this Patch will allow future deployments of this Patch and any Patches that are dependencies.' This is followed by 'Name: Konqueror/1.0.0'. Then, 'The following Patches are dependencies of this Patch:' is shown, followed by 'Name: Waterfox/1.0.0' and 'Name: Seamonkey/1.0.0'. At the bottom right, there are two buttons: 'Cancel' (white with grey border) and 'Activate' (red with white text).

Figure 3: Enable patch

[!Important]

Enabling a **patch** will also enable any dependencies that may be required **for** the **patch**.

## Disable a Patch

To disable a patch, click on the button labeled **Disable** next to the patch you want to disable.

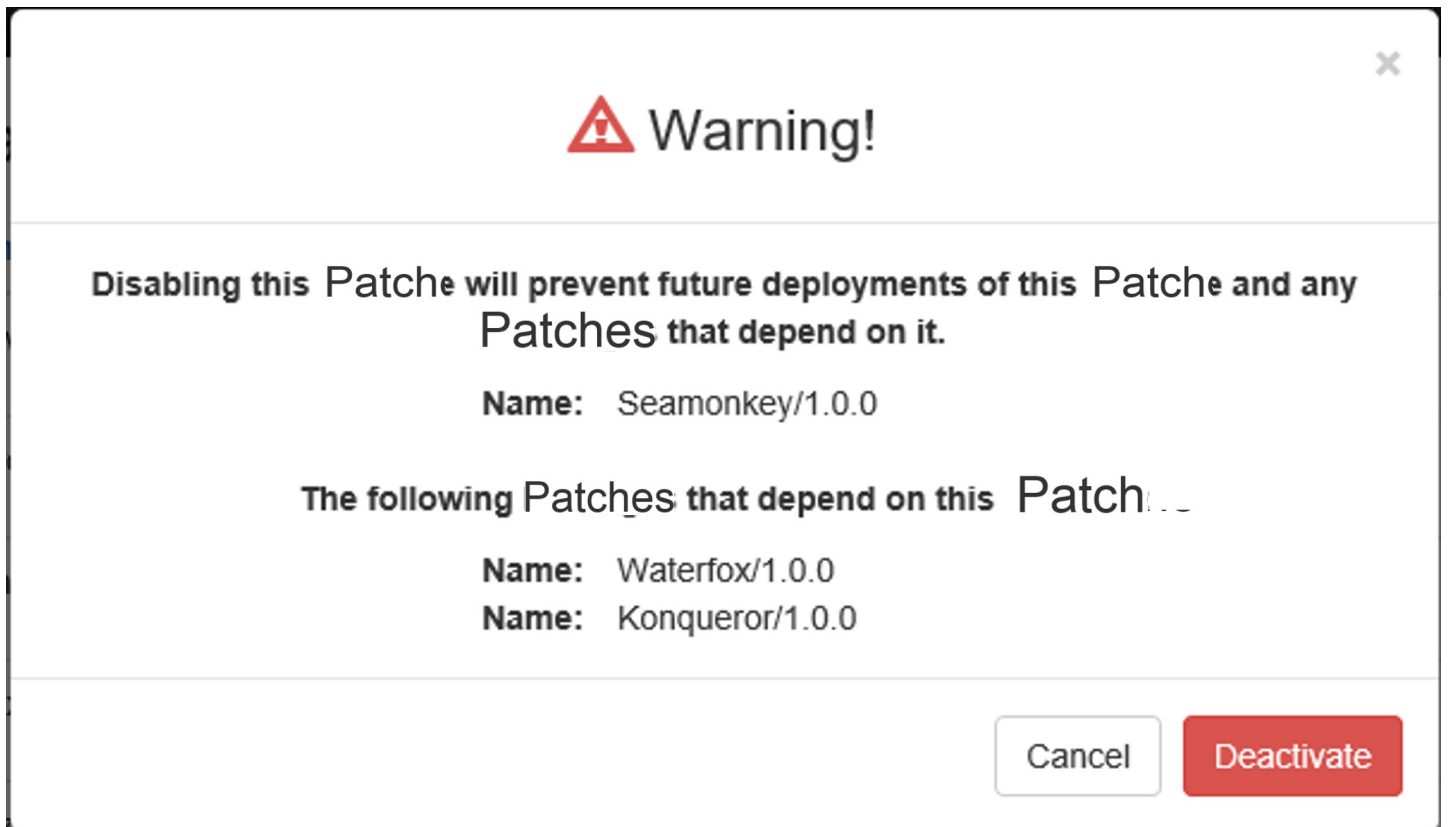


Figure 4: Disable patch

[!Important]

Disabling a **patch** will also disable any dependencies that may be required for the **patch**.

#### From PowerShell

In PowerShell, users can view all patches or specific patches using the RPS cmdlet **Get-RpsPatch**.

#### **⚠ IMPORTANT**

Start by establishing your working session in PowerShell ISE Administrator Mode.

1. Click on the **Search Icon** from the Start Menu.



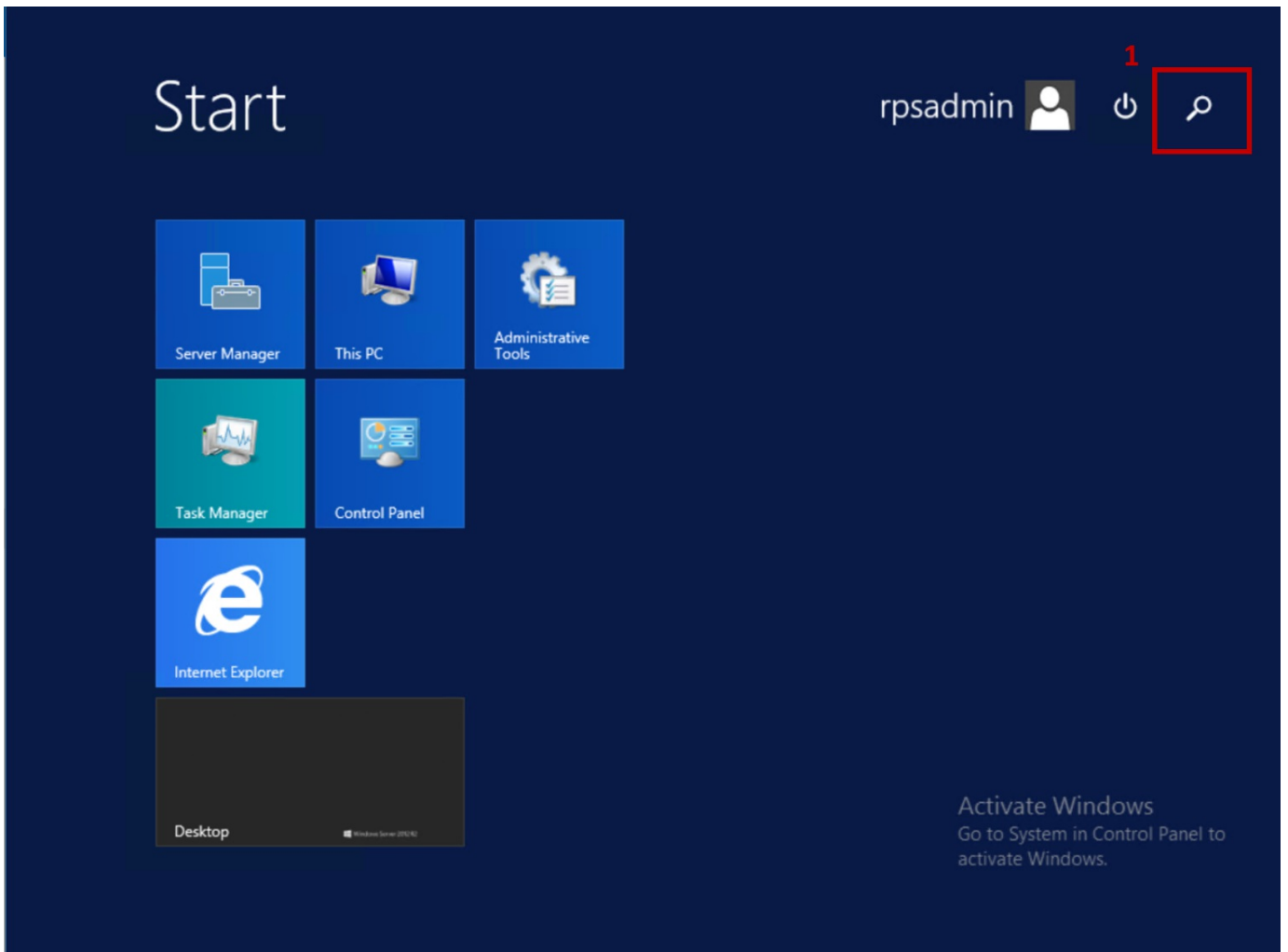


Figure 1: Click on Search Icon.

1. Search for PowerShell ISE by typing *PowerShell ISE* in the Search bar.

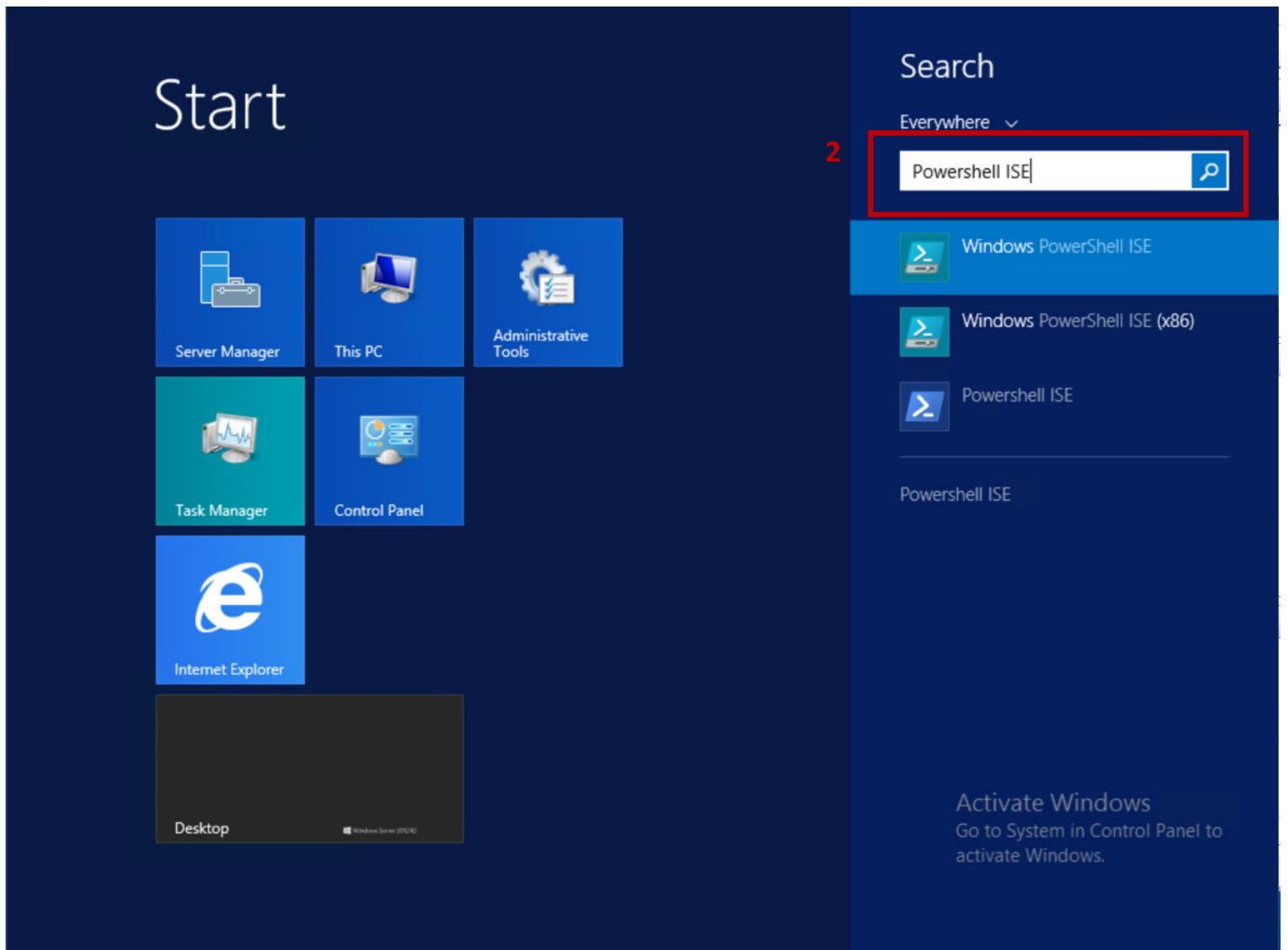


Figure 2: Search for PowerShell ISE.

1. Click on **Run As Administrator**.

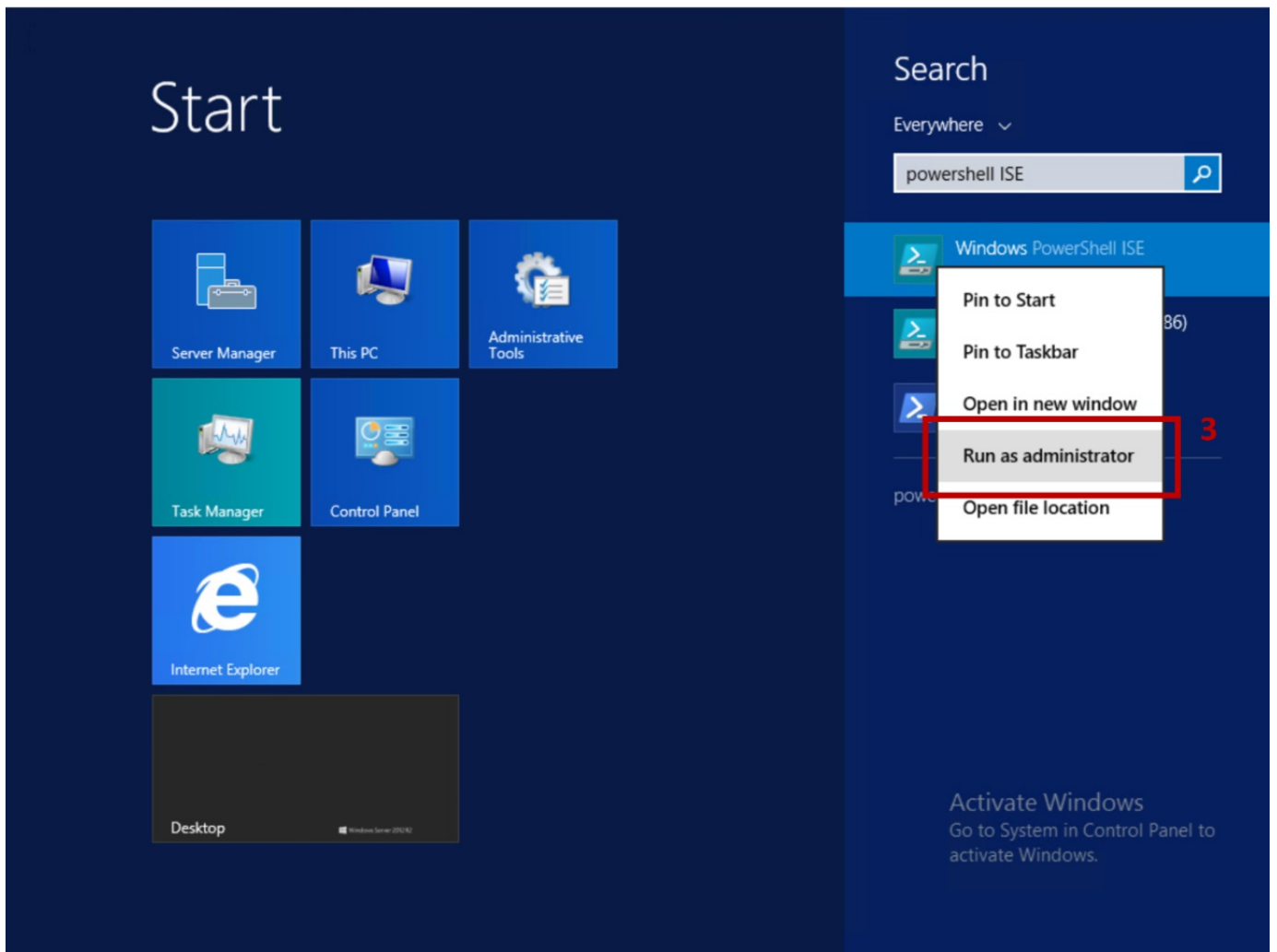


Figure 3: Open PowerShell ISE as Administrator.

1. Retrieve a specific patch by specifying the patch's name in the `-Name` parameter of the `Get-RpsPatch` cmdlet:

```
$myPackage = Get-RpsPatch -Name 'MyPackage1'
```

2. Retrieve all Patches in the CMDB by not specifying any parameters for the `Get-RpsPatch` cmdlet:

```
$myPackages = Get-RpsPatch
```

# How to Enable and Disable CDN Communication

Last updated on 4 August, 2021.

**Document Status:** Document Feature Complete as of August 4, 2021; PENDING EXTERNAL REVIEW.

## Overview

In order for content to replicate from one target node to another, the CDN communication protocol must be enabled on the node. There are two different protocols used within RPS: DFSR and BITS. Each is dependent on whether the content is being replicated to a target outside of the local domain or to a target within the local domain. Both DFSR and BITS can be enabled and disabled using PowerShell and the RPS UI.

## Prerequisites

Users enabling or disabling CDN communication should be assigned *CDN Sync* or *RPS Admin* permissions.

## DFSR

DFSR is the Distributed File System Replication windows service, and runs on RPS servers. This is a multi-master replication engine which allows synchronization of folders between servers. DFSR is used when content is being replicated to a target node *within* the local domain. This protocol is faster than BITS and should be used whenever feasible.

### WARNING

DFSR cannot be used to send content to a target outside the local domain.

## BITS

BITS is the Background Intelligent Transfer Service, a windows service which facilitates asynchronous, prioritized, and throttled transfer of files between machines using idle network bandwidth. BITS is used when content is being sent to a target node *outside* of the local domain. While this protocol is also capable of sending content within the local domain, it is much slower than DFSR and therefore should not be enabled for local domain replication.

### TIP

BITS should only be used to replicate content outside of the local domain.

## Enable/Disable CDN Using PowerShell

In PowerShell, the cmdlet `Enable-RpsCdn` is used in conjunction with `$true` to *enable*, or `$false` to *disable*, the CDN communication protocols, as shown in the following examples:

### Enable DFSR

```
Enable-RpsCdn -Dfsr $true
```

### Disable DFSR

```
Enable-RpsCdn -Dfsr $false
```

### Enable BITS

Enable-RpsCdn -Bits \$true

#### Disable BITS

Enable-RpsCdn -Bits \$false

#### Enable DFSR and BITS

Enable-RpsCdn -Dfsr \$true -Bits \$true

#### Disable DFSR and BITS

Enable-RpsCdn -Dfsr \$false -Bits \$false

#### Enable DFSR and Disable BITS

Enable-RpsCdn -Dfsr \$true -Bits \$false

#### Disable DFSR and Enable BITS

Enable-RpsCdn -Dfsr \$false -Bits \$true

## Enable/Disable CDN Using the RPS UI

DFSR and BITS may also be enabled or disabled using the RPS UI. To accomplish this, follow the 4-step process below:

1. Access the RPS UI website using the following URL convention: <https://<server hosting Rps UI>:8080>
2. If the browser does not immediately open to the local node, users can access it by selecting "Targeting" from the menu ribbon and clicking **Nodes**, as shown in the following figure:

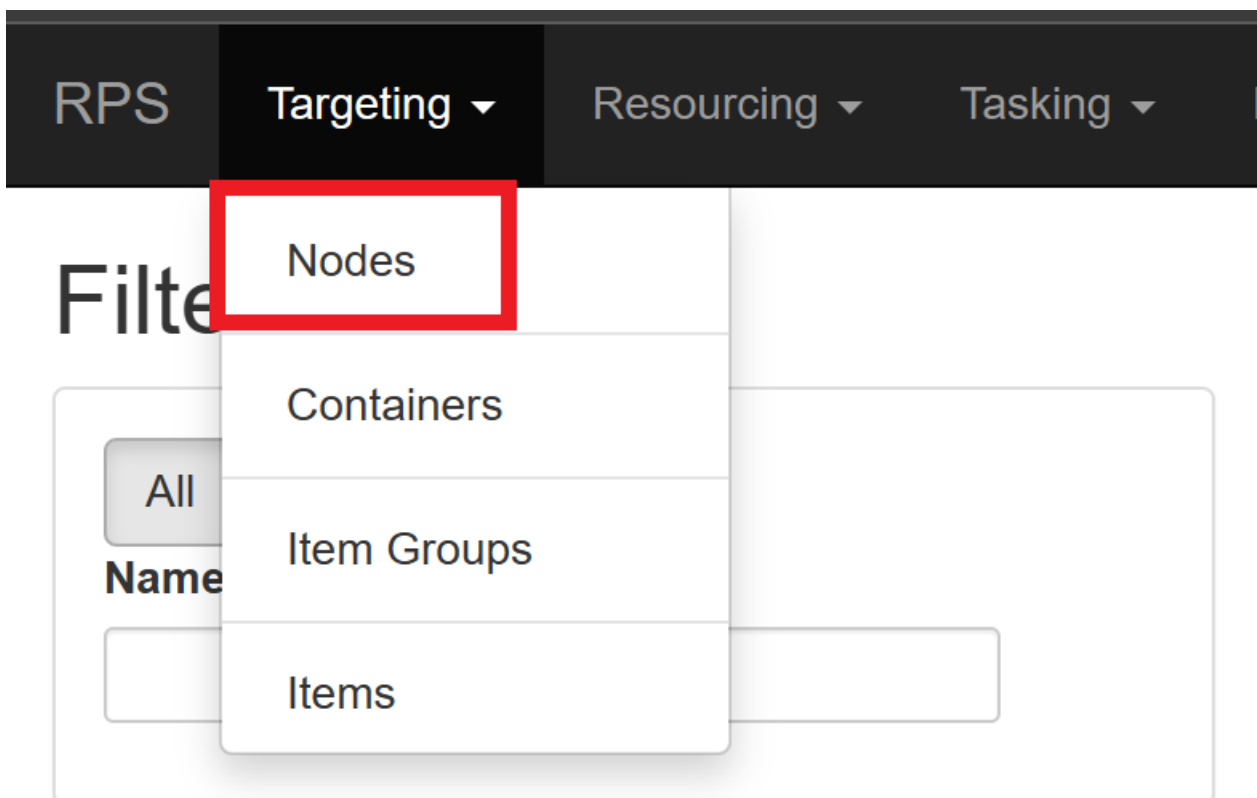


Figure 1: Navigate to the local node.

From this screen, users can then select the desired node by clicking the name of the node from the list. Shown as "Default"

in the following figure:

Name ↑	Host Name	# Of Children	IPAddress	Endpoint Url	Thumbprint	Local Node	Active	
Default	Default	0	10.0.0.1			✓	✓	<a href="#">+ Add Node</a> <a href="#">Edit</a> <a href="#">Remove</a>

Figure 2: Select the desired node.

- Under the "Data Replication" section, click **Edit** and a "Data Replication Settings" box will appear.

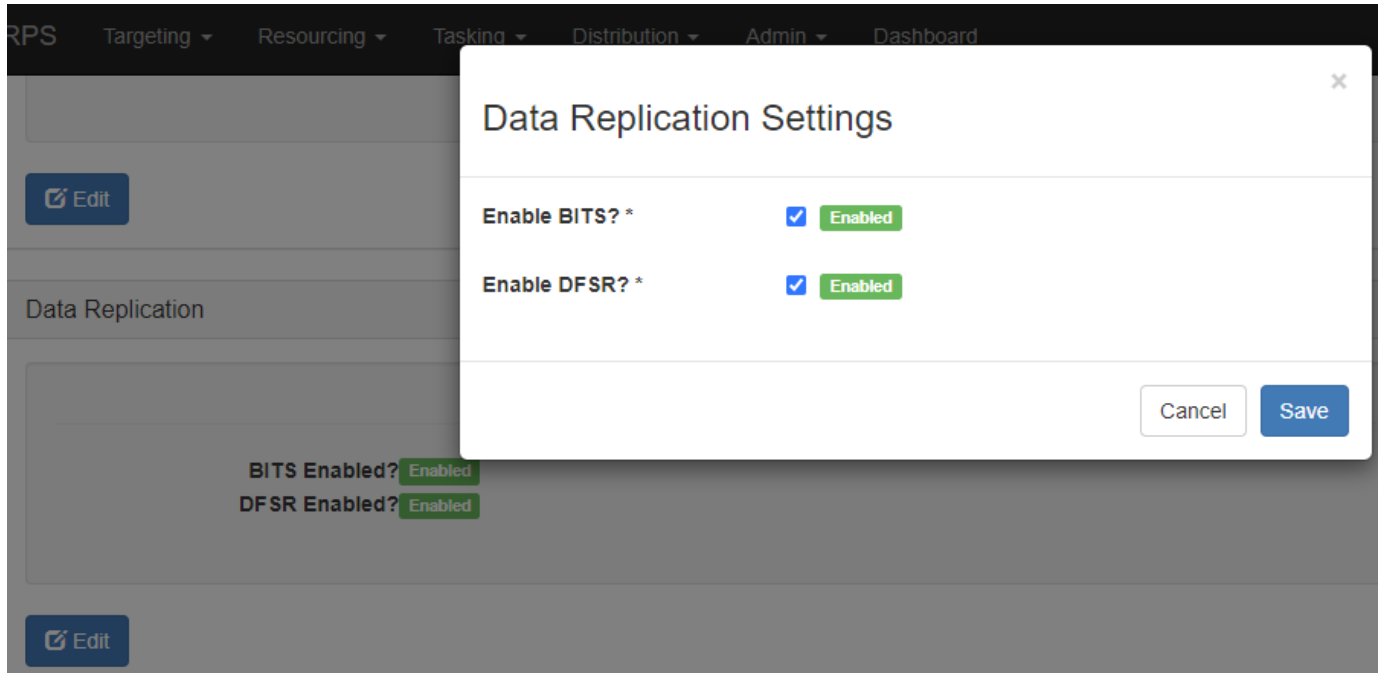


Figure 3: Data Replication Settings box.

- Check** to *Enable* or **UnCheck** to *Disable* the appropriate CDN Communication protocol, then click **Save** to apply your settings, as shown in the following figure:

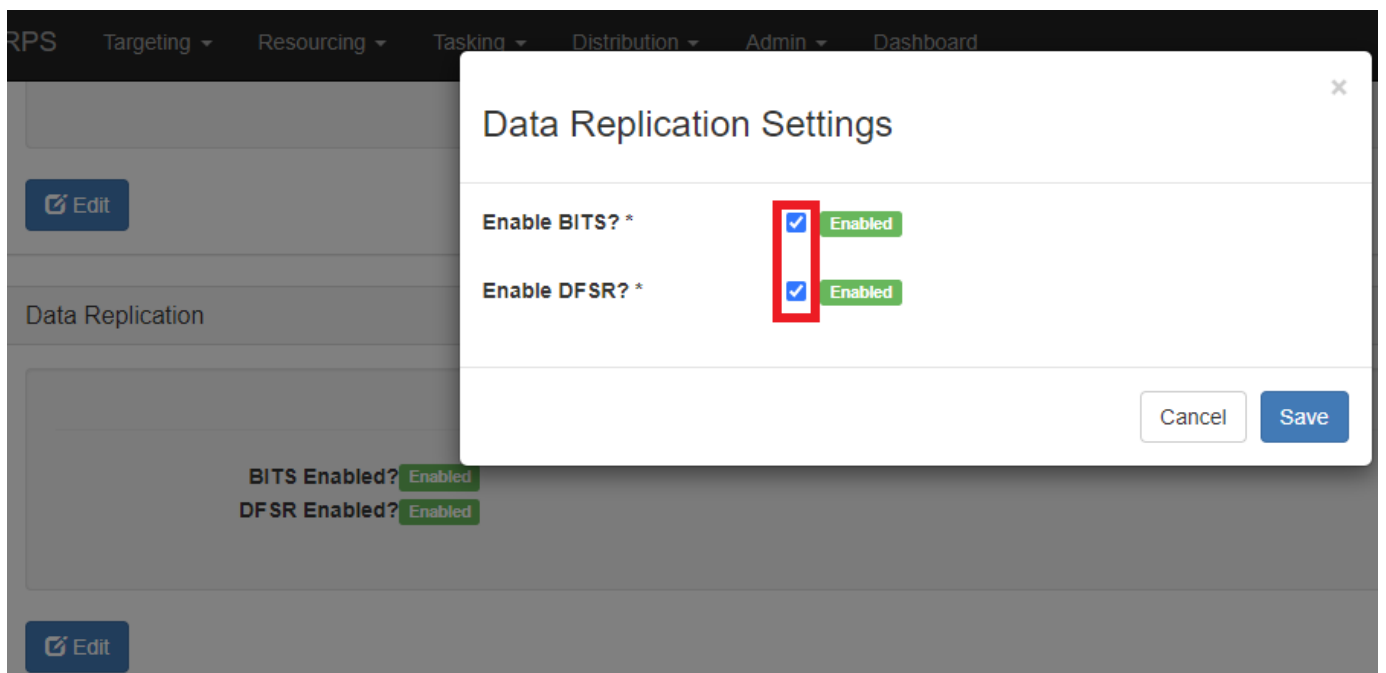


Figure 4: Enable/Disable CDN from RPS UI.

# How to Create and Use Patch Chains

Last updated on July 30, 2021.

Last Reviewed and Approved on PENDING REVIEW

## Overview

Certain patches require one or more legacy patches to be installed before the current patch will run or install successfully. When this happens, a patch stream is created containing all the required versions of the patch. This is called a patch chain.

## How is a Patch Chain Different from a Patch Stream?

Patch chains and patch streams are both created using the *New-RPSPatchStream* cmdlet. However, a patch chain is a patch stream which contains multiple versions of one specific patch type. In contrast, a patch stream may contain many different application types. When a patch chain is loaded, the application versions will be sorted and installed in sequential order. This is similar to a Dependency, except that the patch names must be identical in a patch chain, and the version numbers determine the order of patching. For more information on Dependencies, see the *DependsOn* Element section in the [RPS Patch Manifest Definition](#) article.

## How a Patch Chain Works

Patch chains must have identical `Patch Name` values, (which are case sensitive), and include the `Patch Version` in the name as well. The `Patch Version` determines the order that the patches will be installed. When the LCM (Local Configuration Manager) executes on a target, it will attempt to install the next highest version of a patch in a chain. If none of the patches in the chain are installed, then it will start installation with the lowest version, then work its way up the version chain until the highest version is installed.

There are two events that can cause a chain to stop processing all patches:

- An error occurs during installation of the previous patch in the chain.
- The patch is outside the current Maintenance Window. For more information on Maintenance Windows, see [How to Use Maintenance Windows](#).

### WARNING

Patches with a version less than the currently installed patch in the chain will no longer be tested by DSC to verify installation status. To resolve this, the highest installable patch in the chain must be uninstalled and the target system must re-install the entire chain by starting over with the lowest version.

## Patch Version Constraints

Version numbers consist of two to four segments: major, minor, build, and revision, as discussed in the table below:

SEGMENT	TYPE	SEGMENT	DESCRIPTION
<i>Major</i>	<b>Required</b>	First	Major rewrite of product between major numbers. Backwards compatibility cannot be assumed.
<i>Minor</i>	<b>Required</b>	Second	Indicates significant enhancement with the intention of backwards compatibility where all major numbers are the same.
<i>Build</i>	<b>Optional</b>	Third	Indicates when processor, platform, or compiler changes.



SEGMENT	TYPE	SEGMENT	DESCRIPTION
Revision	<b>Optional</b>	Fourth	Usually indicates a minor fix or patch for security hole.

Example: `.NET Framework 4.7.0`

- Where the `4` is the *major* and indicates that this may not be backwards compatible with .NET versions where *majors* are lower than 4. For example: .NET Framework 4.7.0 may not be backwards compatible with .NET Framework 3.5.0.
- Where the `7` is the *minor* and indicates there has been an enhancement added to version 4, typically backwards compatible with lower *minor* numbers where the *major* number is the same. For example: .NET Framework 4.7 would be backwards compatible with .NET Frameworks 4.5, but **not** .NET Framework 3.5.
- Where the `0` is the *build* and indicates that this is the first release of version 4.7.0. Higher *build* numbers indicate minor changes and are backwards compatible with .NET Framework applications where the *major* and *minor* numbers are the same.

## To Create a Patch Chain

To ensure a patch chain is created properly, the following conditions apply:

1. Each of the patches in the chain must have the exact same name when they are created in REACTR, with version numbers following the conventions stated above. An example follows:

### Correct:

```
.NET Framework 4.7.0
.NET Framework 4.8.0
.NET Framework 5.0.0
```

Notice that the name *.NET Framework* is the same in all three versions of the application. Variations of the application name will not be executed properly. An example follows:

### Incorrect:

```
MS .NET Framework
.NET Frmwork 3.5
.NET_Framework4_7
```

### **i** NOTE

Patch names are case sensitive and must be exact to be processed sequentially.

2. When patches are copied to the RPS node to create the patch chain, all patches must reside in the same folder.
3. Create the patch chain using the *New-RPSPatchStream* cmdlet as shown in the following example:

```
New-RPSPatchStream -Name MyPatchChainExample -Path C:\MyPatchChain
```

For more information on how to use the *New-RPSPatchStream* PowerShell cmdlet, see [How To Load a Patch Stream](#).

4. Approve the Patch Chain. See [How to Approve and Reject Patch Streams](#) for detailed instructions on accomplishing this step.

Once the patches sync through the CDN (Content Delivery Network) and are loaded onto the RPS node, the DSC (Desired State Configuration) will run to make sure the patch chain installs the applications in the right order.

# Patch Telemetry UI

Last updated on September 16, 2021.

**Document Status:** Document Developer Quality Complete.

## ⚠ IMPORTANT

Patch Telemetry UI will only work with a modern browser (Edge v93.0+ and Firefox v92.0+).

Documentation bundled with Patch Telemetry UI is accurate as of **9/16/2021**.

Updated documentation can be found at: <https://reactr.azurewebsites.us>

## Introduction

The purpose of the Patch Telemetry UI is to review the patching status for the nodes under the root node.

### 📘 NOTE

Patch Telemetry UI is only available at the root node. The telemetry files will only exist for RPS v4.0 nodes.

## How to Use the Patch Telemetry Desktop Application

This is a desktop application that will be installed on the root node from the Azure image.

### Set the Patch Telemetry Folder Location

Open the appsettings.json configuration file and update the value for **CDNFilePath**. This is the path to the patch telemetry data files stored by the packaging service. This data is synced between the root and child nodes using RPS.CDN.

### 📘 NOTE

By default, the path is `C:/CDN/PatchTelemetry`.

## Application

Once the desktop application is opened, you can select the available telemetry nodes in the left-hand side panel.

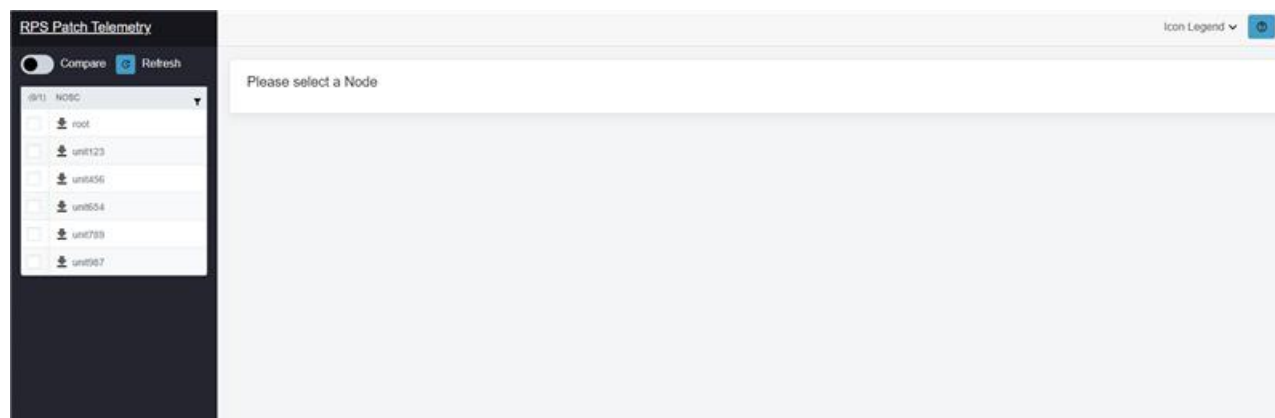



Figure 1: RPS Patch Telemetry UI landing page.

The list of available nodes can be filtered. To view the filter options, click the advanced filter icon  at the top of the list of nodes in the left-hand side panel.

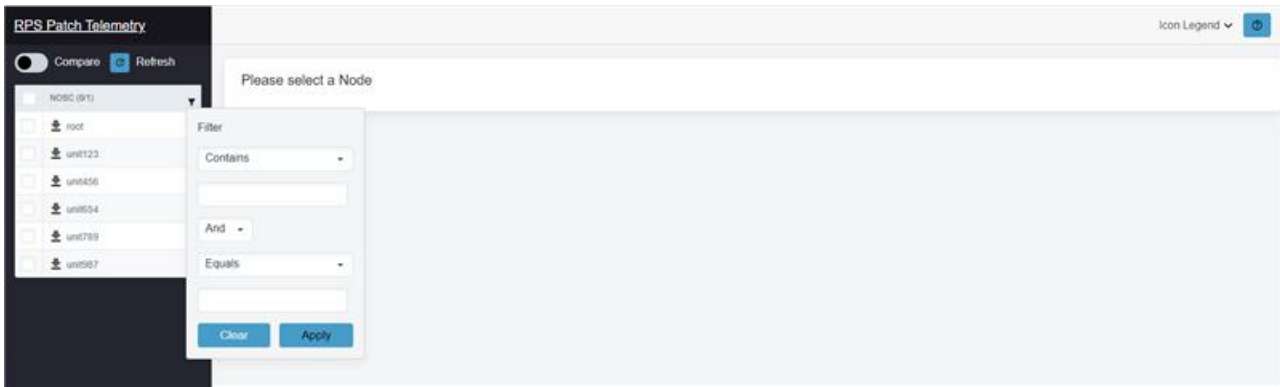


Figure 2: RPS Patch Telemetry UI advanced filter.

Once a node is selected, the available telemetry data will be displayed in the Node Details panel on the right. The details panel groups the data by patch stream.

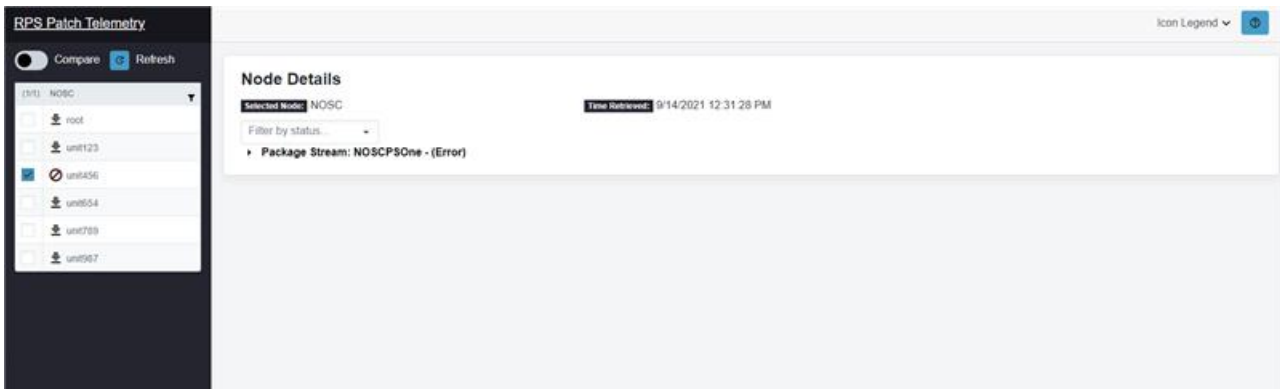


Figure 3: Select a node from the list to view the related telemetry data.

Using the **Filter by status...** dropdown in the Node Details panel, you can filter the data by the following patch stream statuses:

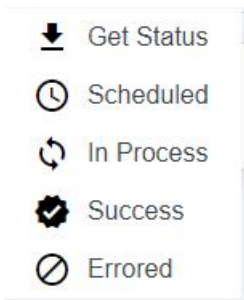


Figure 4: RPS Patch Telemetry UI Icon Legend.

### Comparing Multiple Nodes

At the top of the left-hand side panel, there is a **Compare** toggle that enables "compare mode". By toggling this switch, you can compare two nodes.

#### **NOTE**

Only two nodes can be compared with the **Compare** toggle.

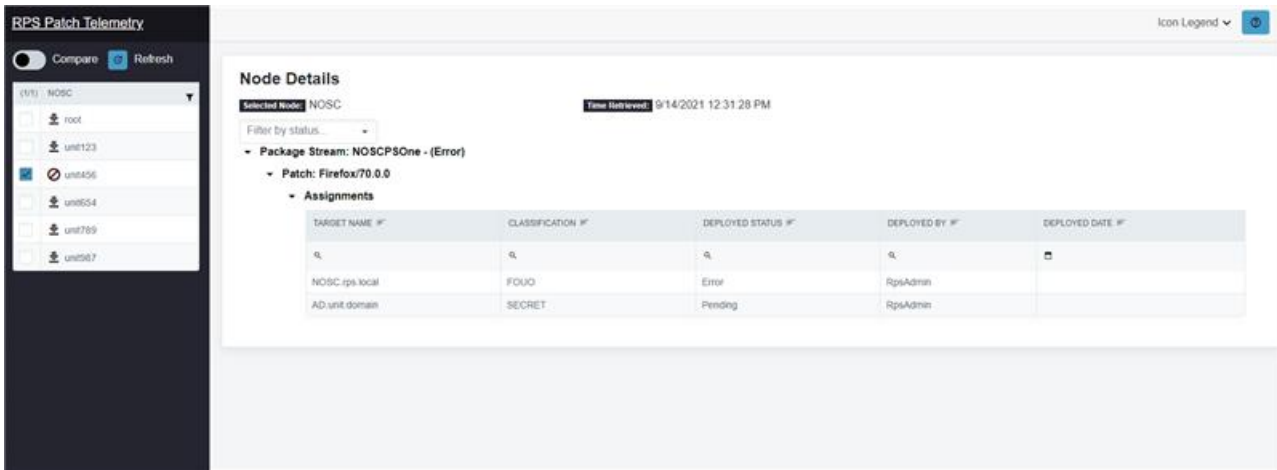


Figure 5: Node Details panel with expanded data (Compare mode off).

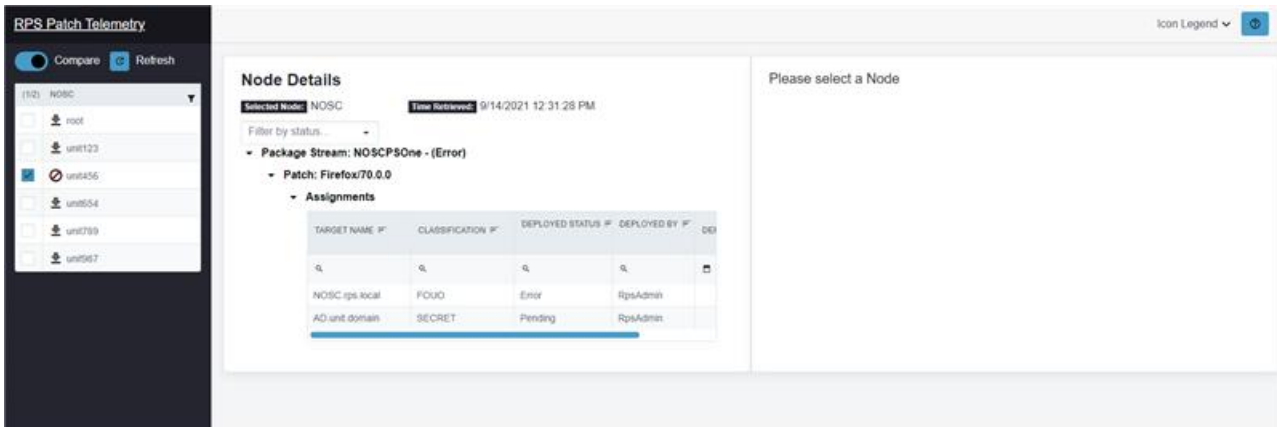


Figure 6: Node Details panel with expanded data (Compare mode on).

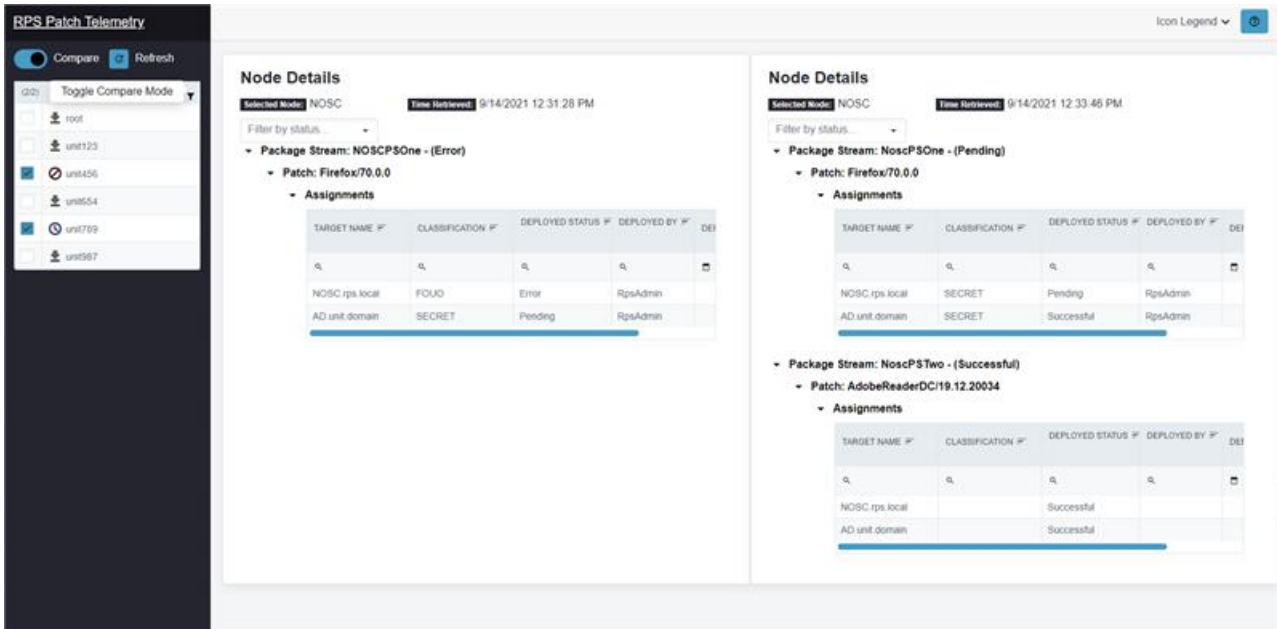


Figure 7: Node Details panel with compare node selected (Compare mode on).

## Troubleshooting

Unable to Reach the Patch Telemetry UI Home Page

**Issue:** Attempting to reach the Patch Telemetry UI home page in a browser does not show a web page.

**Remedy Steps:**

- Verify PatchTelemSvc is running.
- Make sure browser version is correct (Edge/Firefox).
- Verify correct home page for deployed Patch Telemetry UI.

### Empty Node Panel

**Issue:** The list of available nodes on the left-hand side panel is empty on startup.

### **Remedy steps:**

- Make sure the browser version is correct (Edge/Firefox).
- Verify the appsettings config for CDNFilePath is correct.
- Verify valid files exist in the CDN Patch Telemetry folder.

# Sidewinder

Last updated on August 25, 2021.

**Document Status:** Document Developer Quality Complete.

## ⚠ IMPORTANT

Documentation bundled with Sidewinder is accurate as of **9/20/2021**.

Updated documentation can be found at: <https://reactr.azurewebsites.us>

## Introduction

Sidewinder is an application that enables sideloading of approved patch streams from one RPS instance to another.

## Prerequisites

### General

- Permissions to execute an application.
- Non-administrative permissions to the current user's My Documents.
- SMB ports and protocols to RPS instance (source and target).
- HTTP ports and protocols to RPS instance (source and target, if target is 4.0).

### Export

- Access to RPS 4.0 CDN with either logged in user or alternate account with username and password.
- Access to RPS 4.0 API (ports and protocols only; no RPS RBAC needed).
- Access to file system to create archive (.zip file).

### Import

- Access to RPS 4.0 or 3.1 CDN with either logged in user or alternate account with username and password.
- Access to RPS 4.0 API (when importing to RPS 4.0).
- Source patch stream archive created previously by Sidewinder from another instance of RPS.

## Settings

A valid configuration must be specified in settings for all of the tabs in Sidewinder to be accessible. The **Save button** must be selected for updated configurations to take effect. Settings will be persisted to disk for that particular user.

### RPS Version

This setting only impacts importing, as Sidewinder only supports exporting from RPS 4.0.

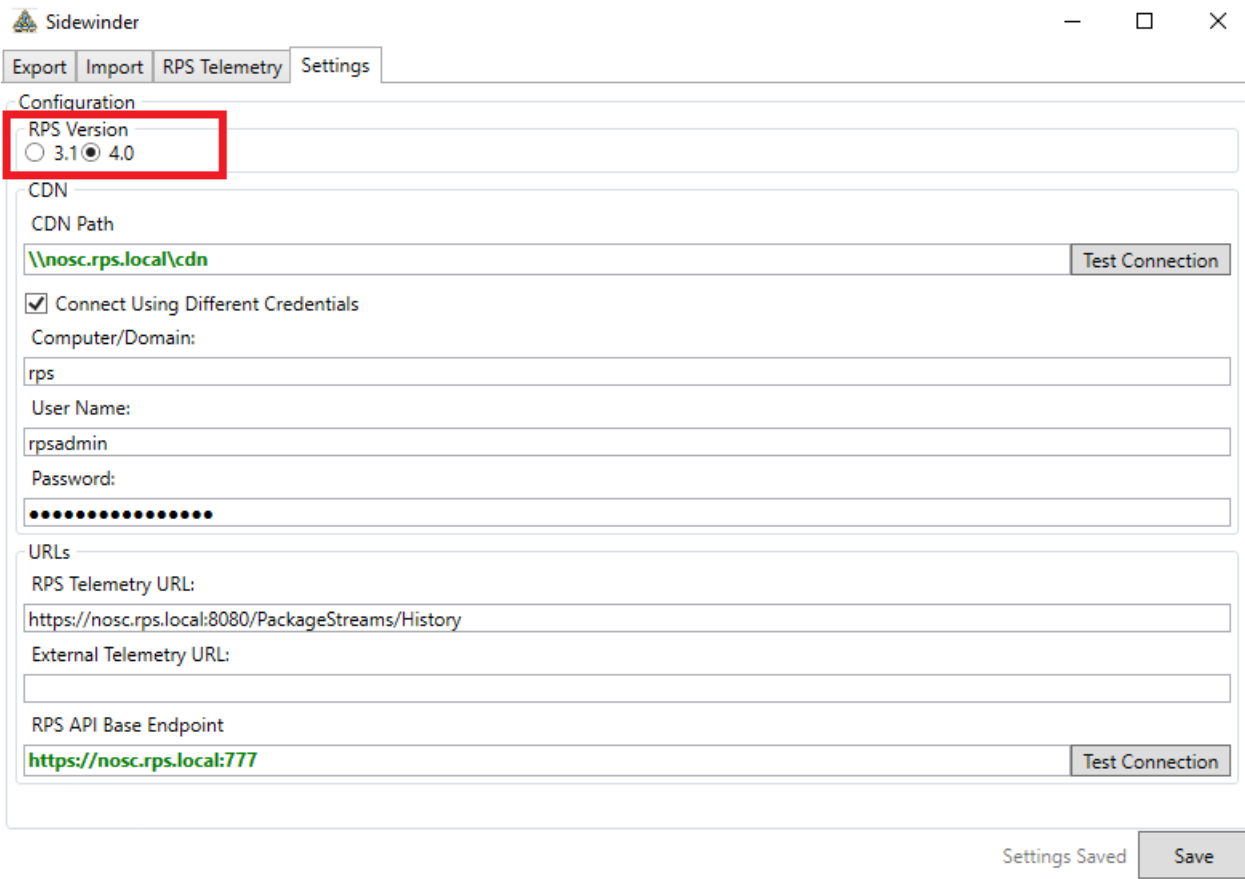


Figure 1: Settings - RPS Version Selection

### CDN / Credentials

SETTING	DETAILS
CDN Path	UNC path to the CDN folder. This can be network or local path. Click the <b>Test Connection</b> button to test the CDN location in combination with the credentials specified (if any). Defaults to <code>C:\CDN</code> .
Connect Using Different Credentials	If checked, the user must specify a domain, user name, and password to connect to the CDN. By default, this setting is not selected (unchecked).
Computer/Domain	Computer, domain, or workgroup to provide when "Connect Using Different Credentials" is selected (checked). Defaults to the current computer or domain (if domain joined).
User Name	The user name to provide when "Connect Using Different Credentials" is selected (checked). Defaults to current logged in user.
Password	The user name's password to provide when "Connect Using Different Credentials" is selected (checked). This data is encrypted in memory and is destroyed when the application closes. For security reasons, the password will need to be entered each time the application is opened.

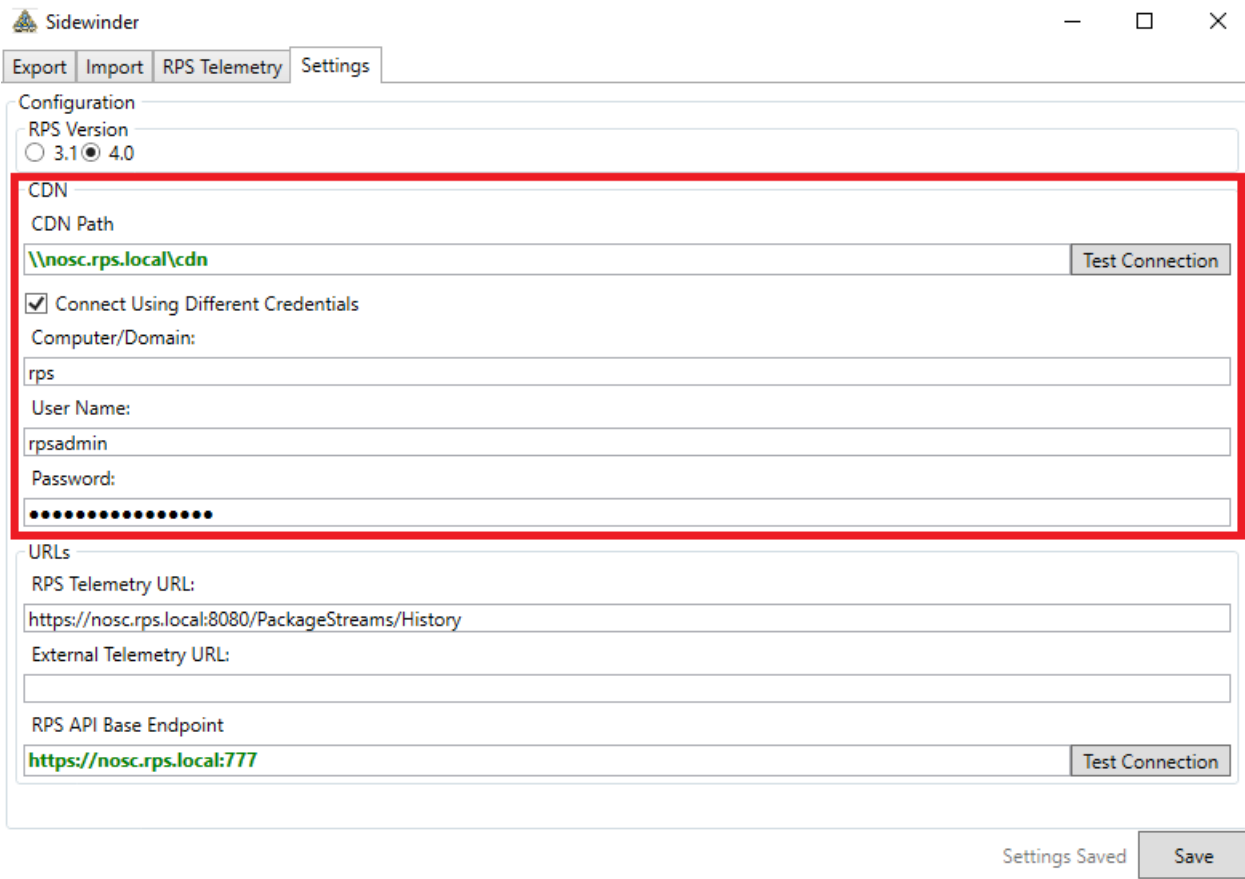


Figure 2: Settings - CDN/Credential Section

#### URLs

SETTING	DETAILS
RPS Telemetry URL	Specifies the URL to use for the RPS Telemetry page under the RPS Telemetry tab.
External Telemetry URL	Specifies the URL to use for the External Telemetry page under the RPS Telemetry tab.
RPS API Base Endpoint	This is the base endpoint for the RPS API of the RPS instance to connect to. The base URLs are used during export (version 4.0 only) and import to either: a. Generate patch stream files on export, or b. Ingest the new patch stream after import has been completed.



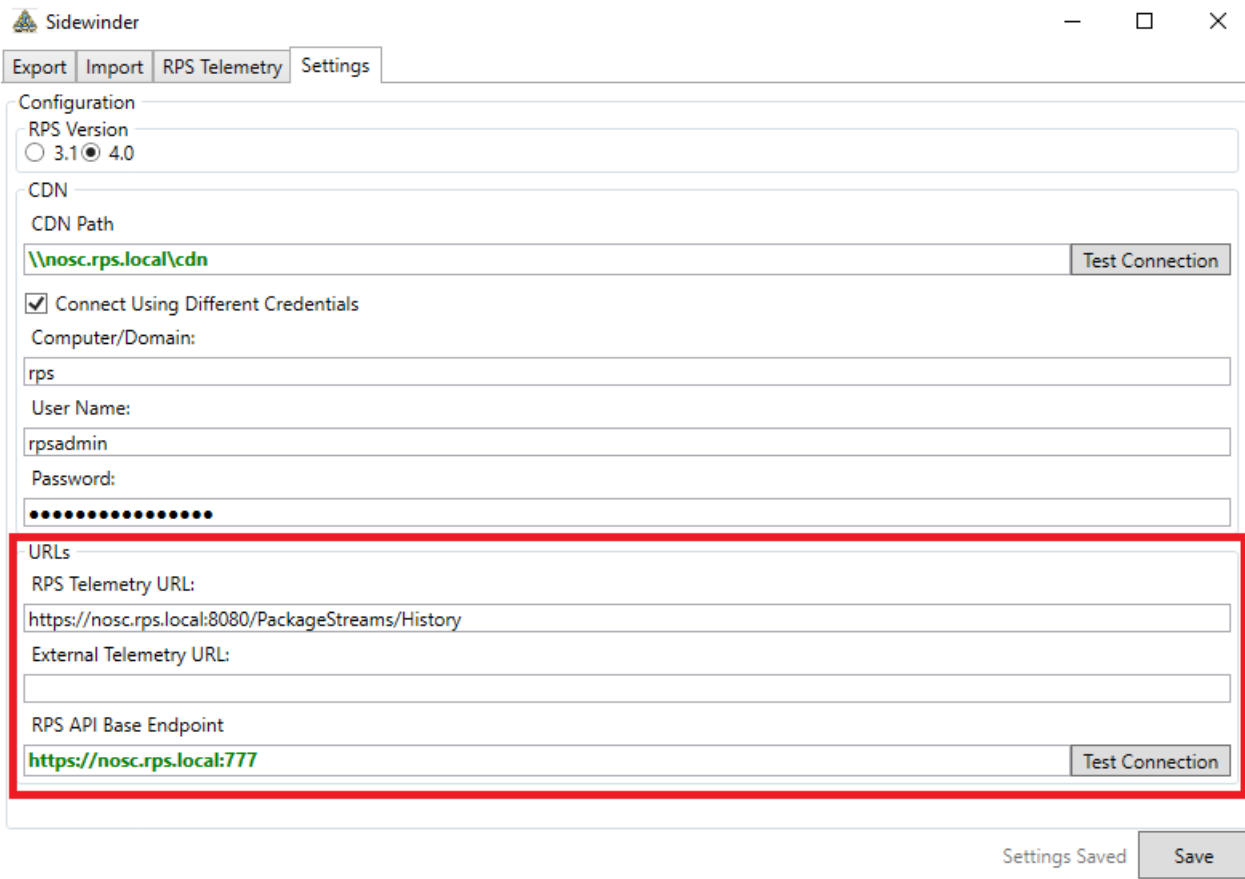


Figure 3: Settings - URLs Section

### Saving

It is important to save the settings configuration once editing is complete. A confirmation message will appear when the settings have been successfully saved.

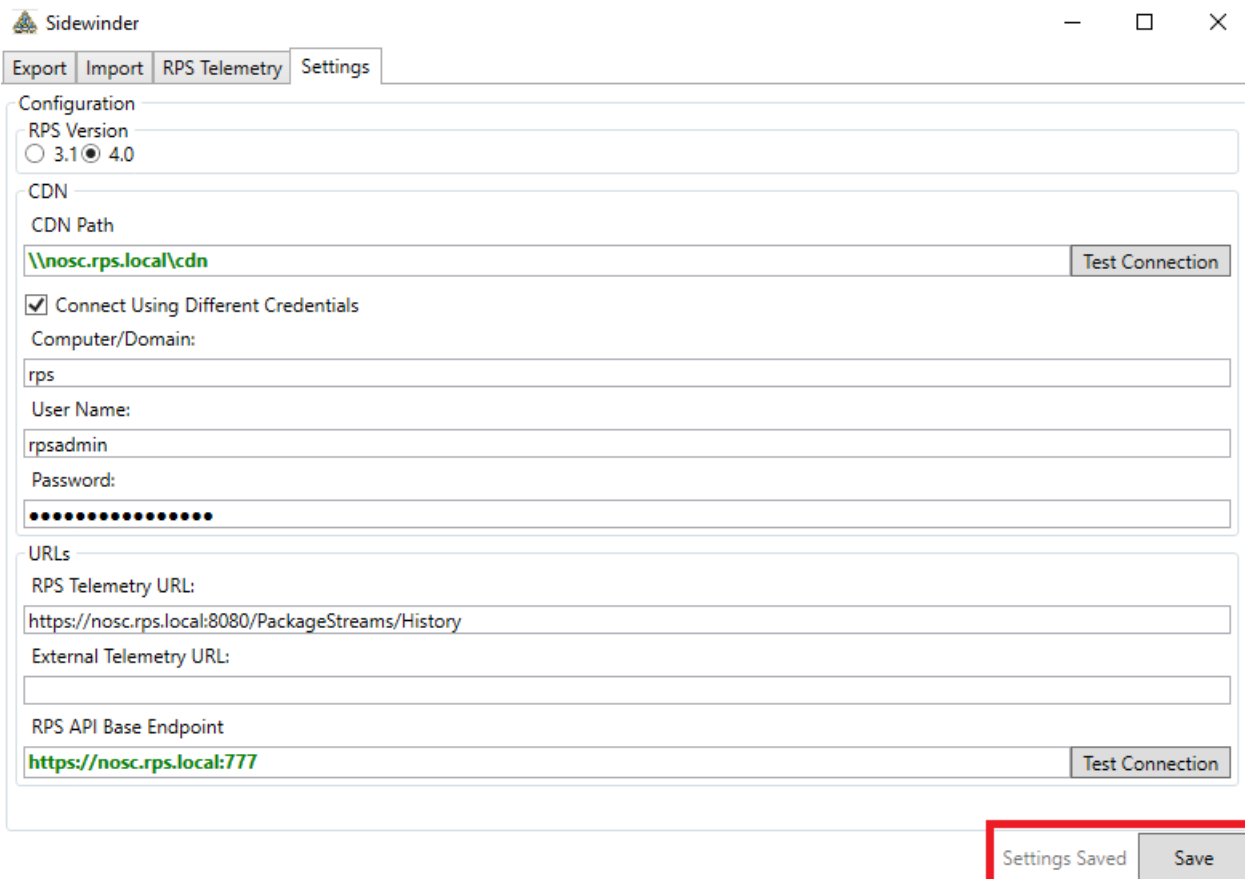


Figure 4: Settings - Save Message

## How to Export an RPS v4.0 Patch Stream

Follow these steps to export a patch stream from an RPS 4.0 instance.

1. On the **Export** tab, click **Find Streams**. This will load the patch streams available for export.

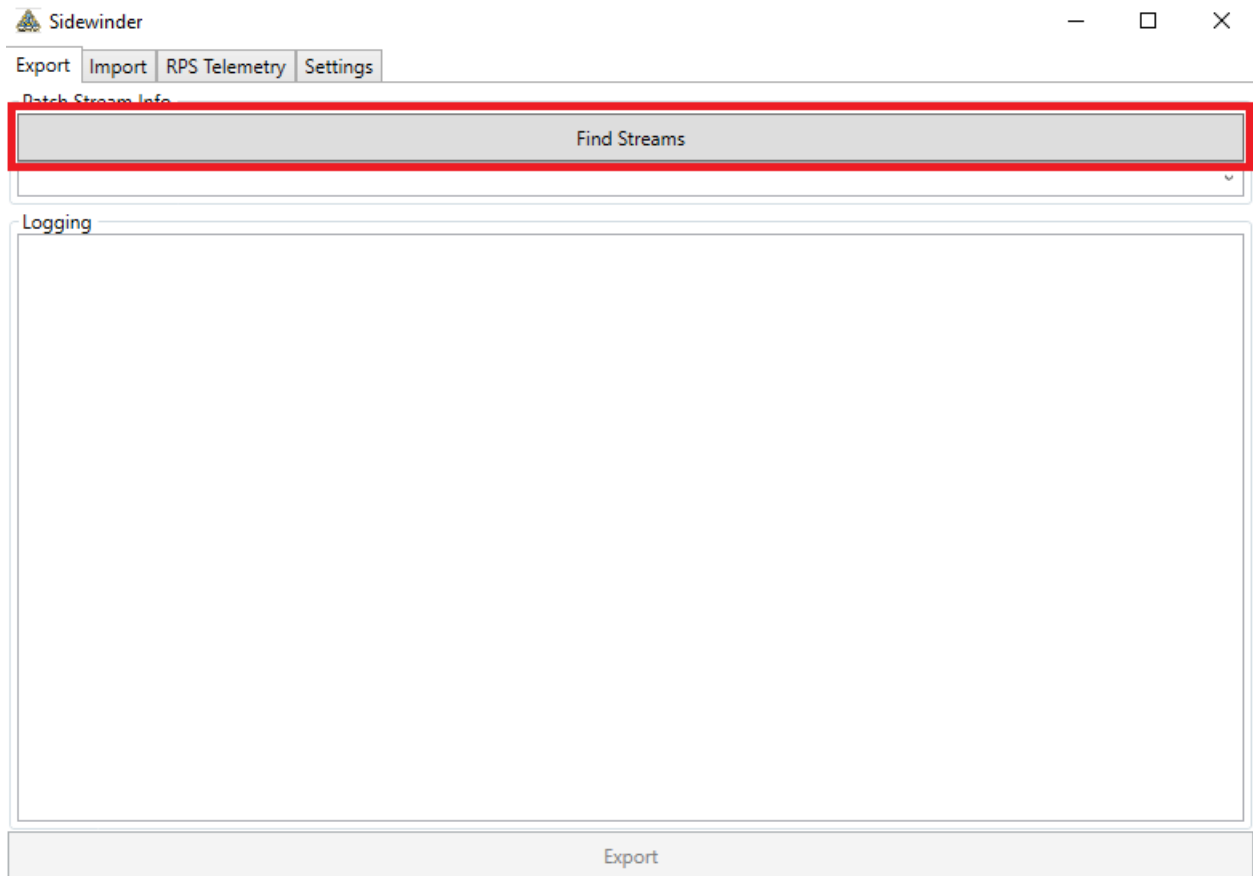


Figure 5: Export - Find Streams

2. Select the stream to export from the dropdown.

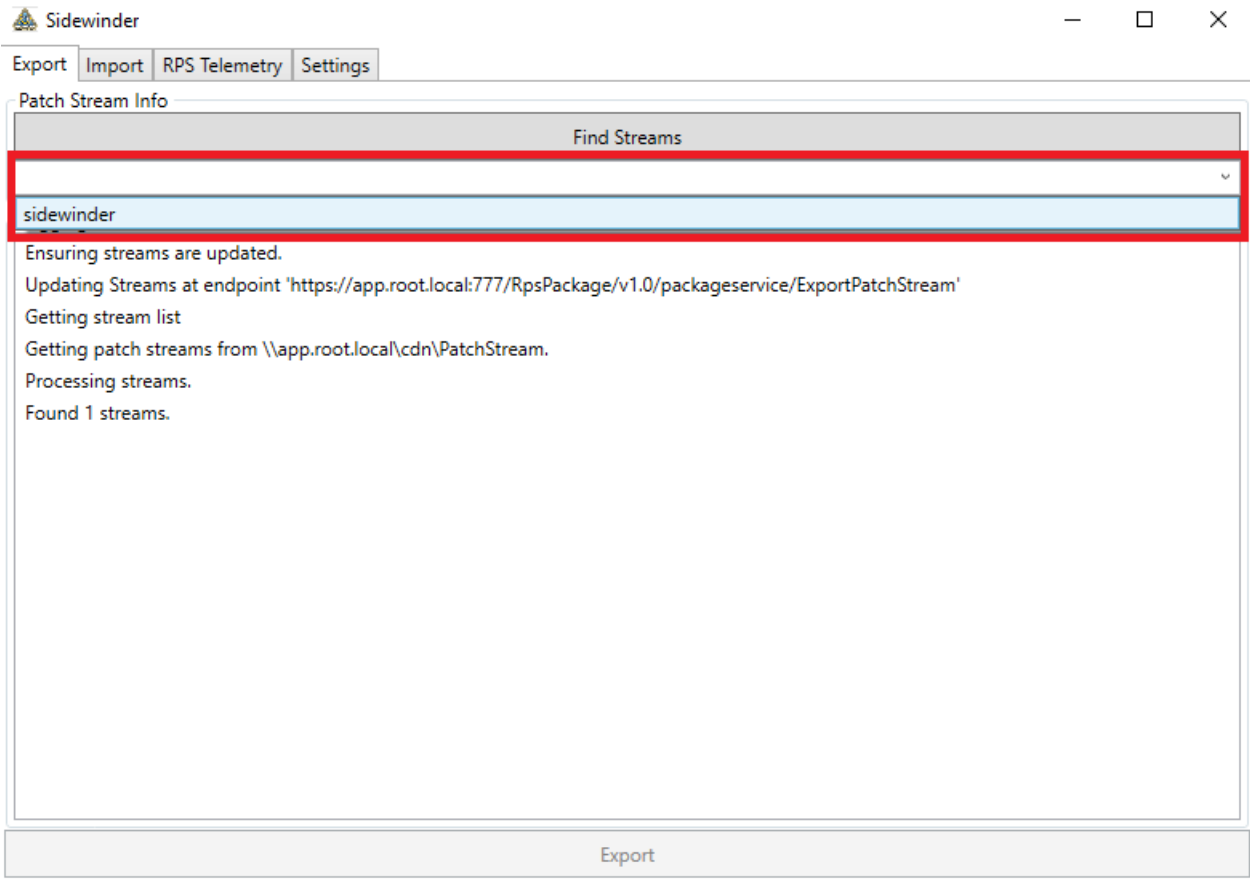


Figure 6: Export - Select Stream

3. Click **Export**.

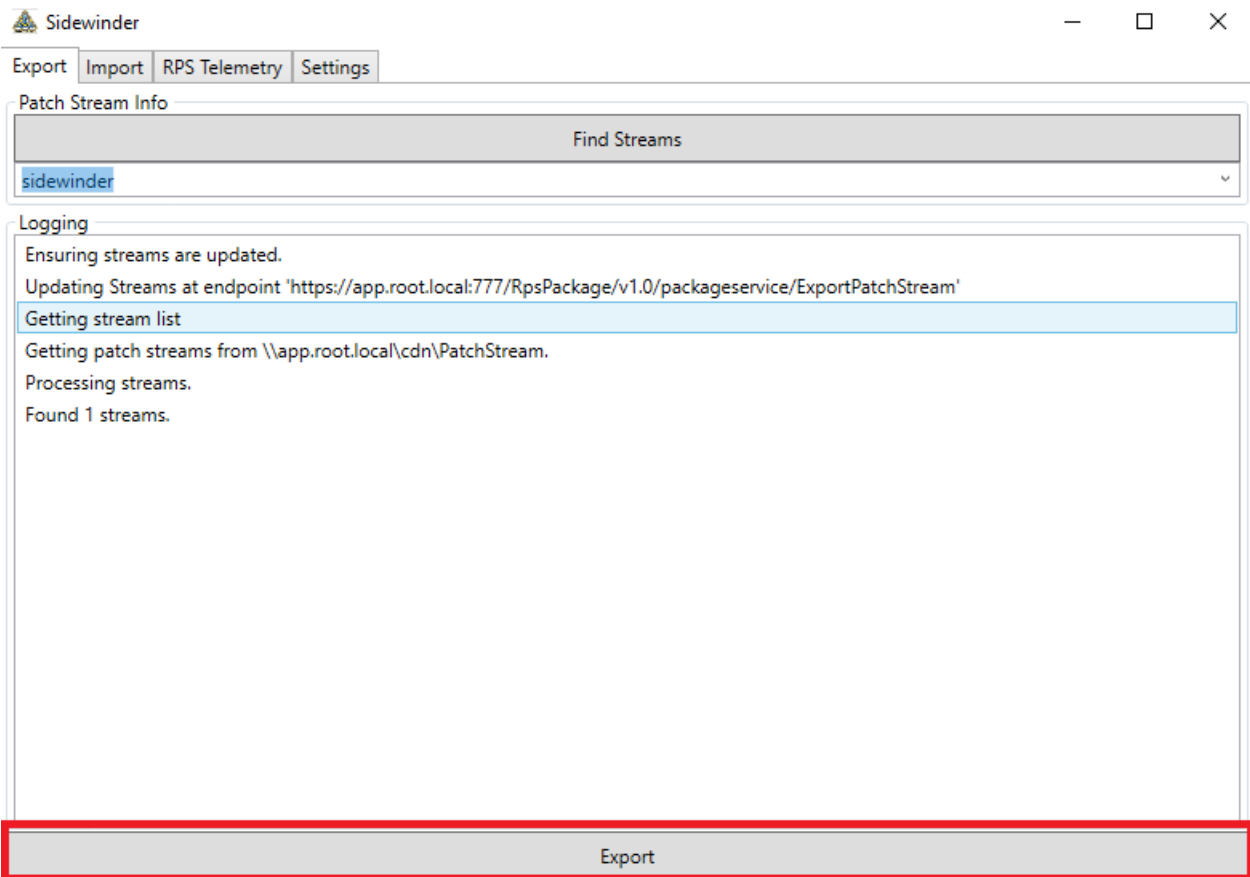


Figure 7: Export - Click Export

4. Choose the file location to save to and click **Save**.

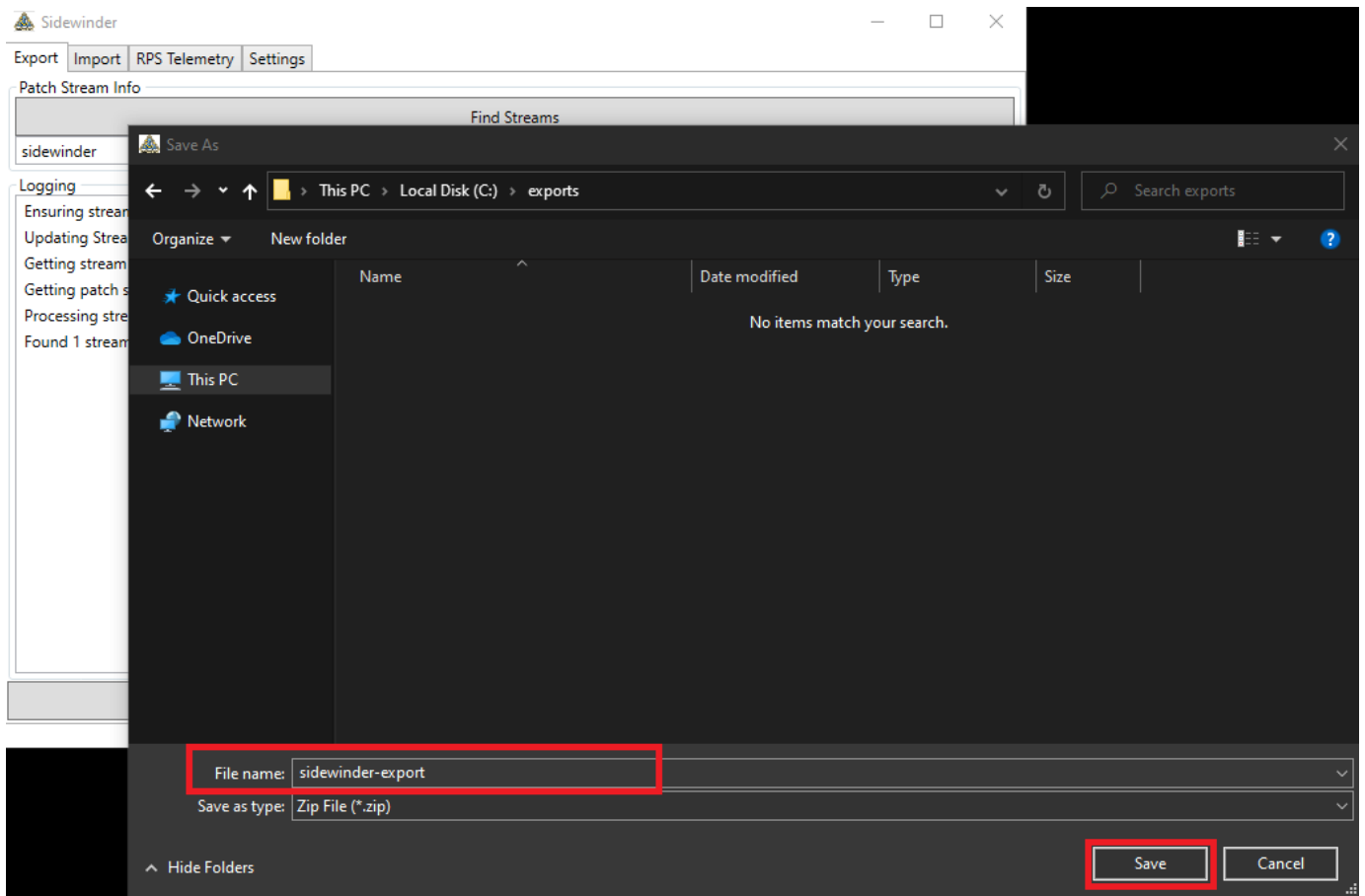
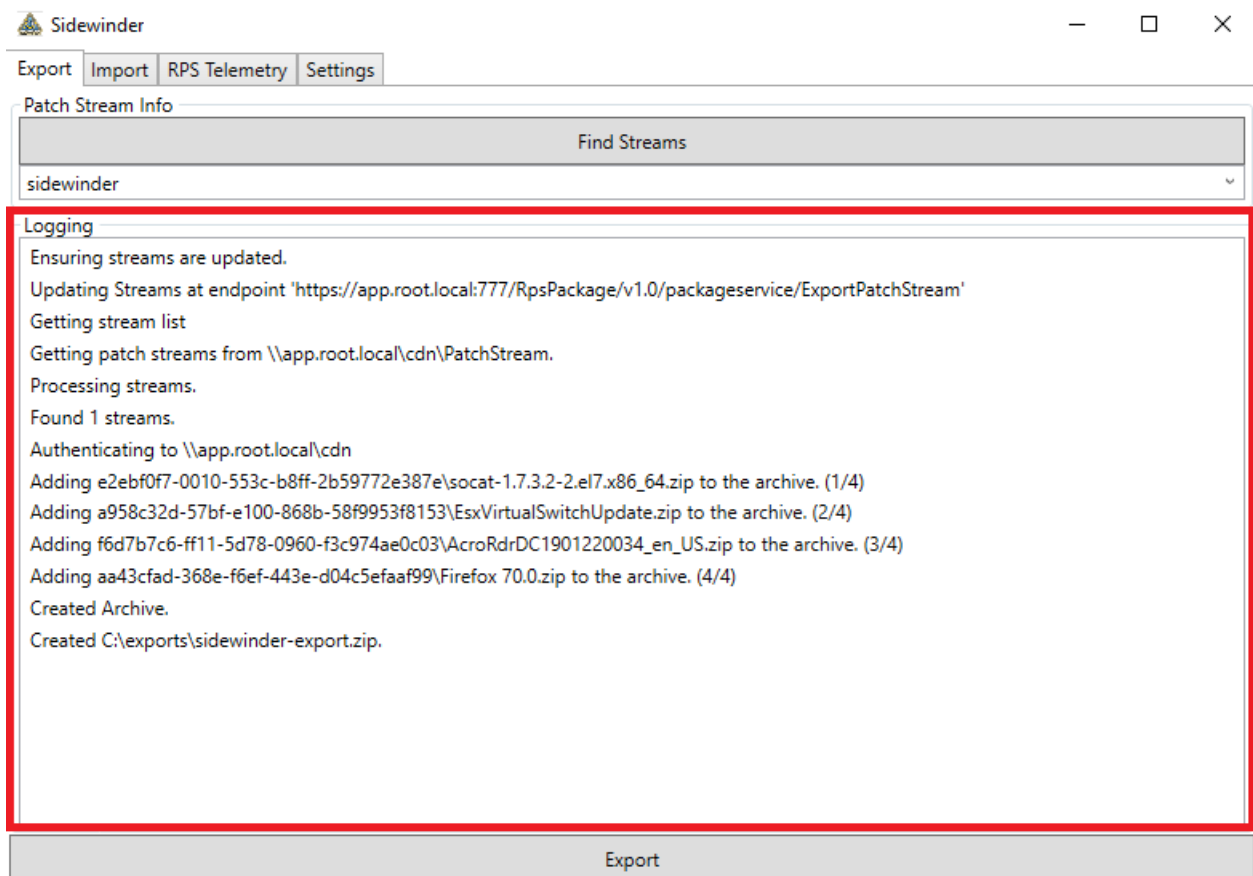


Figure 8: Export - Export File Picker

5. Check the Logging section for processing status and any error messages.



## How to Import a Patch Stream

Follow these steps to import to an RPS Instance.

### NOTE

The steps for importing to RPS 4.0 and RPS 3.1 are the same. Be sure to pick the correct version of RPS from the **Settings** tab.

1. On the **Import** tab, click **Browse...** and navigate to the archive file.

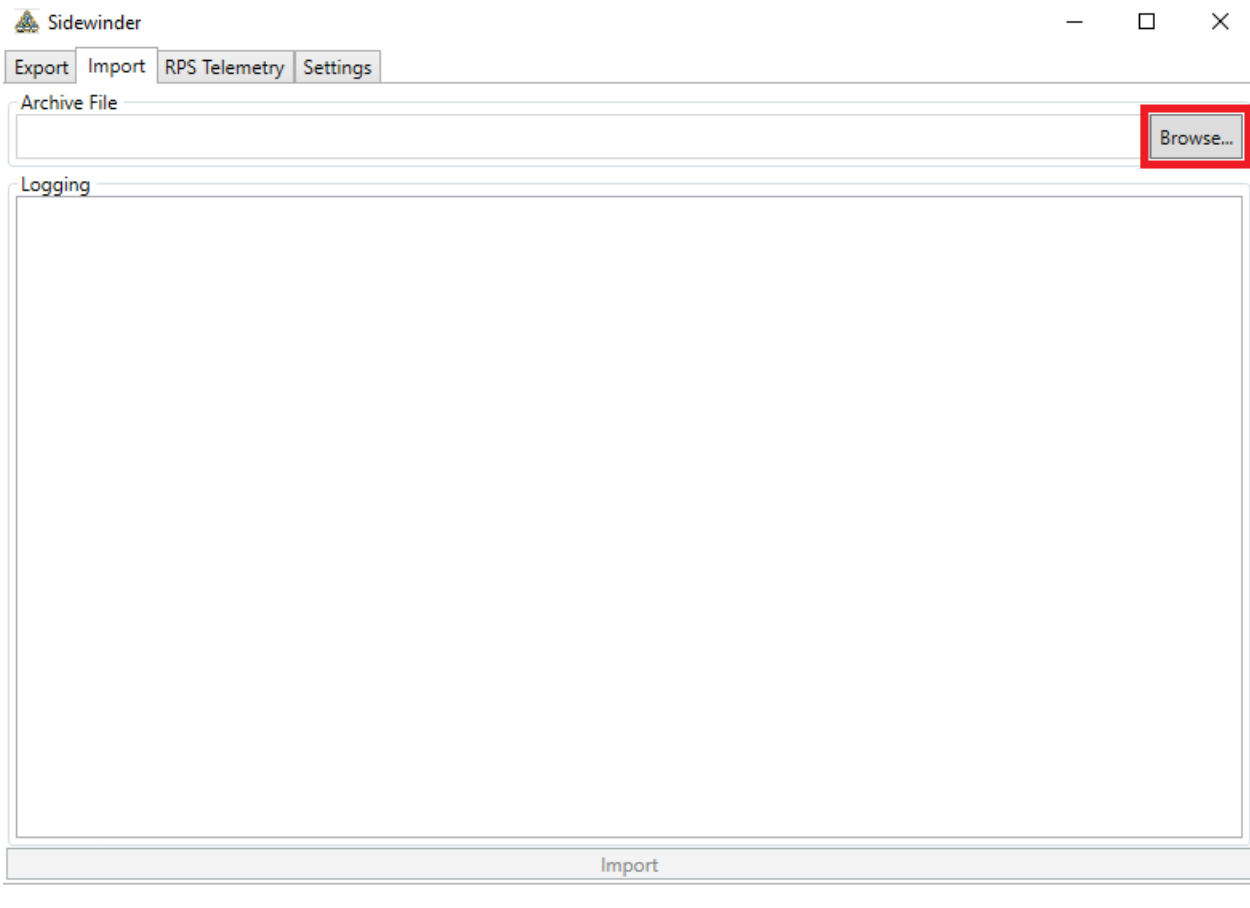


Figure 10: Import - Browse for Archive File

2. Select the exported patch stream zip file, and then click **Open**.

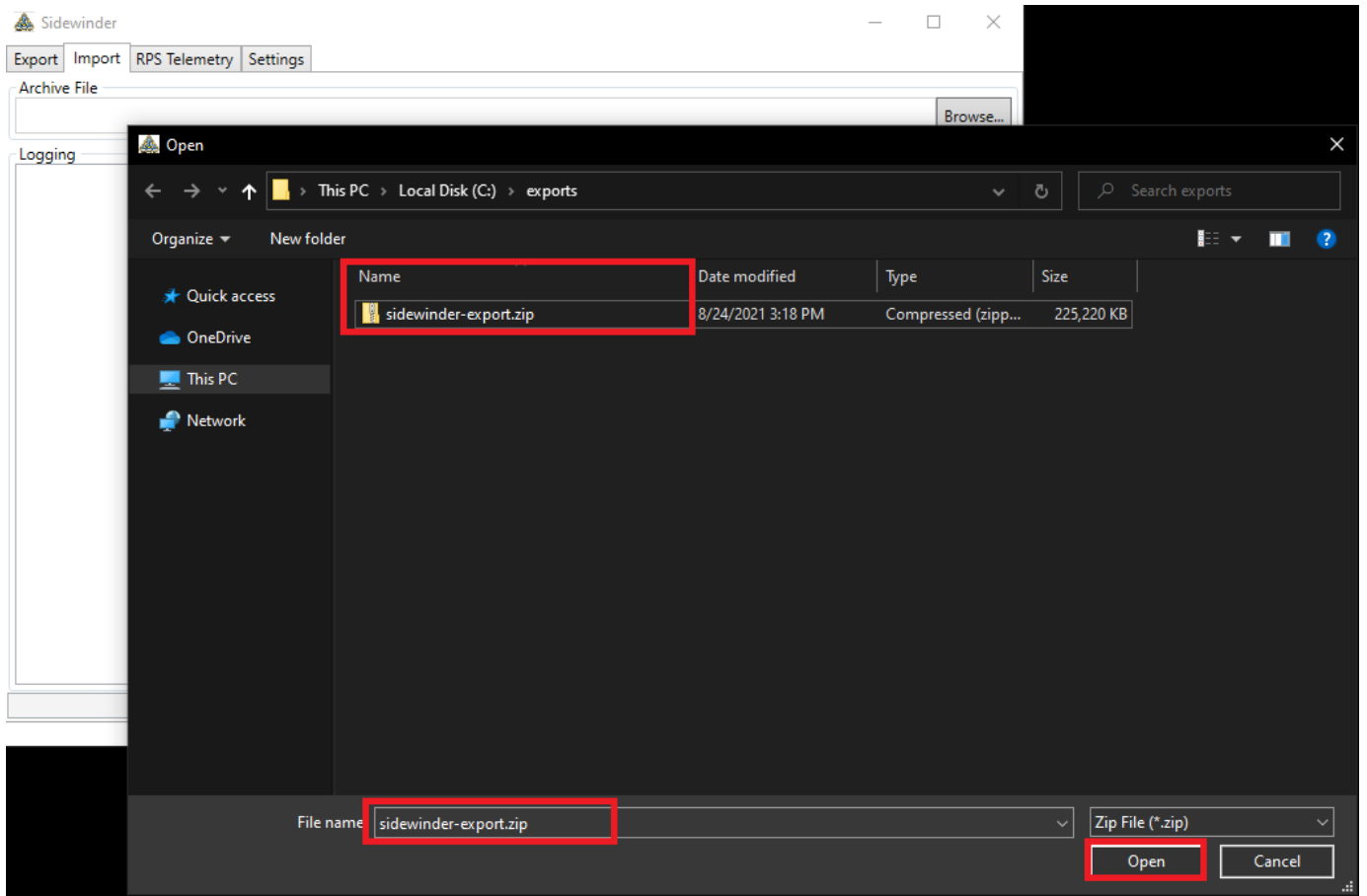


Figure 11: Import - Select Patch Stream Zip File

3. Click **Import**.

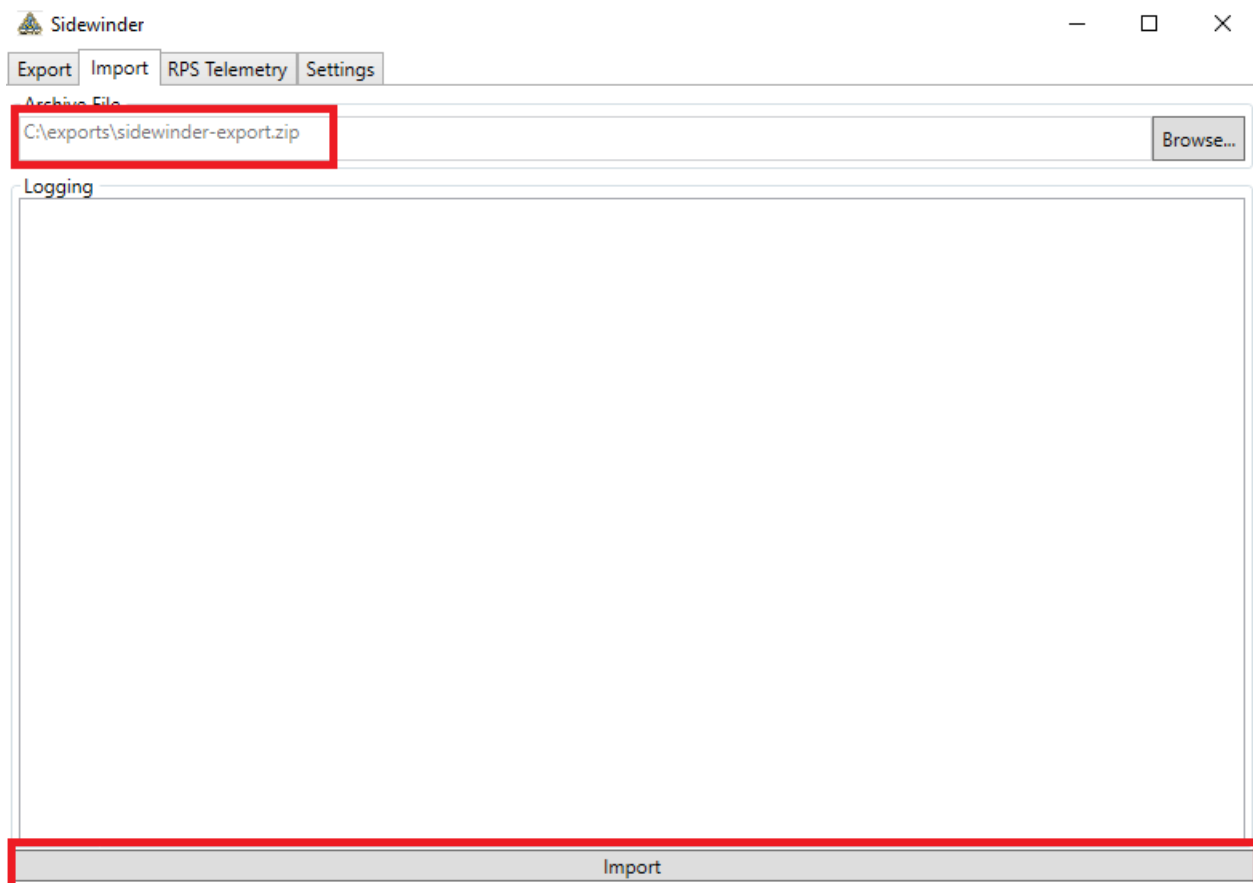


Figure 12: Import - Click Import

**⚠ IMPORTANT**

When importing to RPS 3.1, you may see a PowerShell window open. Do not close this window during processing.

4. Check the Logging section for processing status and any error messages.

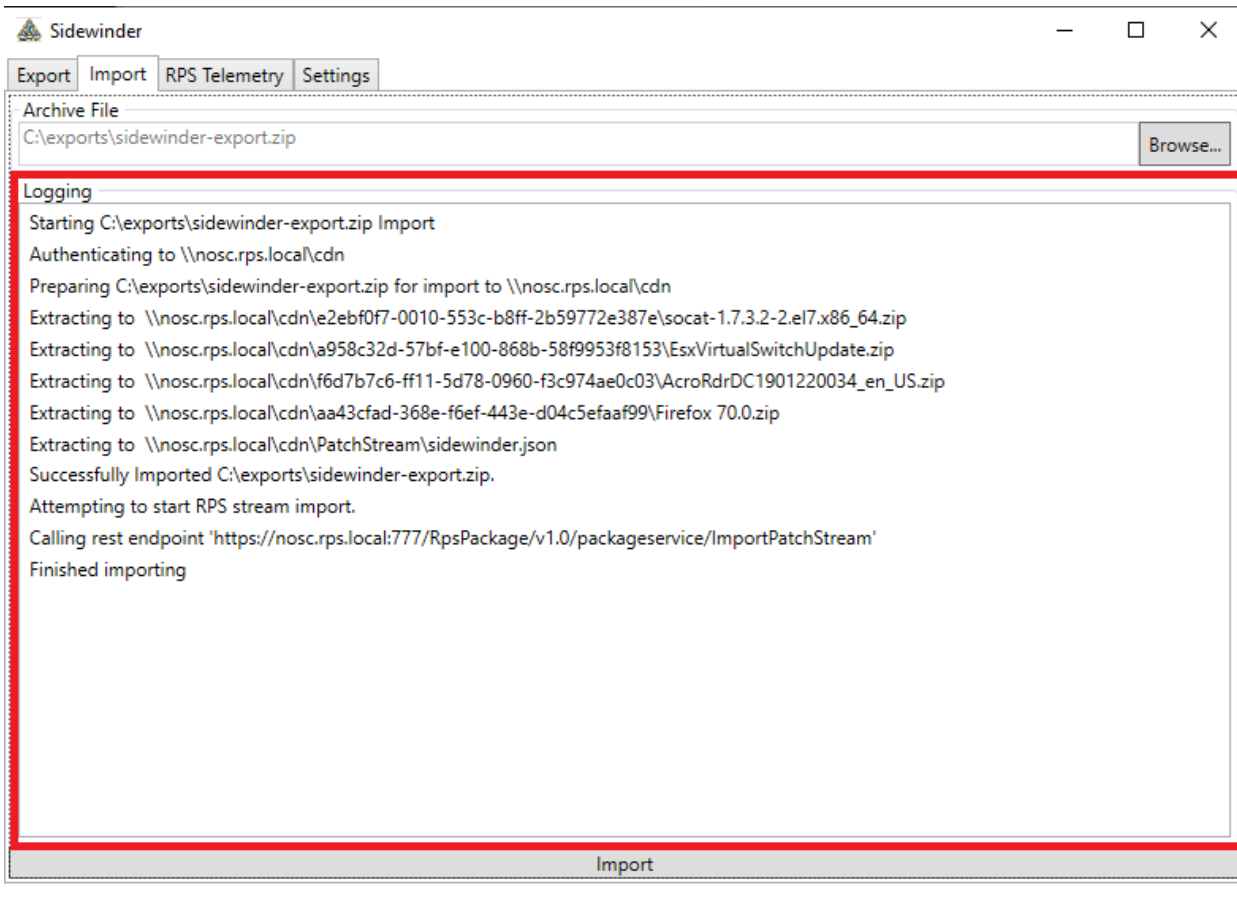


Figure 13: Import - Monitor Import Logs