**Lab: Access resource secrets securely across services**

**Lab scenario**

**Objectives**

After you complete this lab, you will be able to:

* Create an Azure Key Vault and store secrets in the Key Vault.
* Create a server-assigned managed identity for an Azure App Service instance.
* Create an Azure Key Vault access policy for an Azure Active Directory identity or application.

**Lab setup**

* **Estimated Time**: 45 minutes

**Instructions**

**Exercise 1: Create Azure resources**

**Task 1: Open the Azure portal**

1. Sign in to the **Azure Portal** ([https://portal.azure.com](https://portal.azure.com/)).
2. If this is your first time signing in to the Azure portal, you will see a dialog box offering a tour of the portal. Click **Get Started** to skip the tour.

**Task 2: Create an Azure Storage account**

1. Create a new **storage account** with the following details:
   * **New resource group**: SecureFunction
   * **Name**: securestor[your name in lowercase]
   * **Location**: East US
   * **Performance**: Standard
   * **Account kind**: StorageV2 (general purpose v2)
   * **Replication**: Locally redundant storage (LRS)
   * **Access tier**: Hot

**Note**: Wait for Azure to finish creating the storage account before you move forward with the lab. You will receive a notification when the account is created.

1. Open the **Access Keys** blade of your newly created **storage account** instance.
2. Record the value in the **Connection string** field. You will use this value later in this lab.

**Task 3: Create an Azure Key Vault**

1. Create a new **Key Vault** with the following details:
   * **Existing resource group**: SecureFunction
   * **Name**: securevault[your name in lowercase]
   * **Region**: East US
   * **Pricing tier**: Standard

**Note**: Wait for Azure to finish creating the Key Vault before you move forward with the lab. You will receive a notification when the vault is created.

**Task 4: Create an Azure Functions app**

1. Create a new **function app** with the following details:
   * **Existing resource group**: SecureFunction
   * **App name**: securefunc[your name in lowercase]
   * **Publish**: Code
   * **Runtime Stack**: .NET Core
   * **Region**: East US
   * **Storage account**: securestor[your name in lowercase here]
   * **Operating System**: Windows
   * **Plan**: Consumption
   * **Enable Application Insights**: No

**Note**: Wait for Azure to finish creating the function app before you move forward with the lab. You will receive a notification when the app is created.

**Review**

In this exercise, you created all the resources that you will use for this lab.

**Exercise 2: Configure secrets and identities**

**Task 1: Configure a system-assigned managed service identity**

1. Access the **securefunc\*** function app that you created earlier in this lab.
2. Navigate to the **Identity** settings located in the **Platform features** tab.
3. Enable the **system-assigned** managed identity and save your changes.

**Task 2: Create a Key Vault secret**

1. Access the **securevault\*** Key Vault that you created earlier in this lab.
2. Navigate to the **Secrets** link located in the **Settings** section.
3. Create a new **secret** with the following settings:
   * **Name**: storagecredentials
   * **Value**: <Storage Connection String>
   * **Enabled**: Yes

**Note**: Use the storage account **connection string** that you recorded earlier in this lab for the **value** of this secret.

1. Click through the secret to view the metadata for its latest version.
2. Record the value of the **Secret Identifier** field because you will use this later in the lab.

**Task 3: Configure a Key Vault access policy**

1. Access the **securevault\*** Key Vault that you created earlier in this lab.
2. Navigate to the **Access Policies** link located in the **Settings** section.
3. Create a new **access policy** with the following settings:
   * **Principal**: securefunc[your name in lowercase]
   * **Key permissions**: none
   * **Secret permissions**: GET
   * **Certificate permissions**: none
   * **Authorized application**: none
4. **Save** your changes to the list of **Access Policies**.

**Review**

In this exercise, you created a server-assigned managed service identity for your function app and then gave that identity the appropriate permissions to get the value of a secret in your Key Vault. Finally, you created a secret that you will use within your function app.

**Exercise 3: Write function app code**

**Task 1: Create a Key Vault-derived application setting**

1. Access the **securefunc\*** *Function App* that you created earlier in this lab.
2. Navigate to the **Configuration** settings located in the **Platform features** tab.
3. Create a new **application setting** by using the following details:
   * **Name**: StorageConnectionString
   * **Value**: @Microsoft.KeyVault(SecretUri=<Secret Identifier>)
   * **deployment slot setting**: Not selected

**Note**: You will need to build a reference to your **Secret Identifier** by using the above syntax. For example, if your Secret Identifier is [**https://securevaultstudent.vault.azure.net/secrets/storagecredentials/17b41386df3e4191b92f089f5efb4cbf**](https://securevaultstudent.vault.azure.net/secrets/storagecredentials/17b41386df3e4191b92f089f5efb4cbf), then your value would be **@Microsoft.KeyVault(SecretUri=**[**https://securevaultstudent.vault.azure.net/secrets/storagecredentials/17b41386df3e4191b92f089f5efb4cbf**](https://securevaultstudent.vault.azure.net/secrets/storagecredentials/17b41386df3e4191b92f089f5efb4cbf)**)**

1. Save your changes to the **Application settings**.

**Task 2: Create a HTTP-triggered function**

1. Access the **securefunc\*** function app that you created earlier in this lab.
2. Create a new **function** by using the following settings:
   * **Development Environment**: In-portal
   * **Template**: HTTP trigger
   * **Name**: FileParser
   * **Authorization level**: Anonymous
3. In the function editor, replace the example function script with the following placeholder C# code:

using System.Net;

using Microsoft.AspNetCore.Mvc;

public static async Task<IActionResult> Run(HttpRequest req)

{

return new OkObjectResult("Test Successful");

}

1. Click **Save and run** to perform a test execution of the function. The output from the execution should be **Test Successful**.

**Task 3: Test the Key Vault-derived application setting**

1. Delete all the existing code within the **Run** method.
2. Get the value of the **StorageConnectionString** application setting by using the **Environment.GetEnvironmentVariable** method:

string connectionString = Environment.GetEnvironmentVariable("StorageConnectionString");

1. Return the value of the **connectionString** variable by using the **OkObjectResult** class constructor:

return new OkObjectResult(connectionString);

1. Click **Save and run** to perform a test execution of the function. The output from the execution should be your **storage account** connection string stored in **Azure Key Vault**.

**Review**

In this exercise, you securely used a service identity to read the value of a secret stored in **Azure Key Vault** and return that value as the result of an **Azure Function**.

### Exercise 4: Access Storage Account blobs

#### Task 1: Upload a sample storage blob

1. On the navigation menu located on the left side of the portal, select the **Resource groups** link.
2. In the **Resource groups** blade, locate and select the **SecureFunction** resource group that you created earlier in this lab.
3. In the **SecureFunction** blade, select the **securestor\*** storage account that you created earlier in this lab.
4. In the **Storage account** blade, select the **Containers** link located in the **Blob service** section on the left side of the blade.
5. In the **Containers** section, select **+ Container**.
6. In the **New container** pop-up, perform the following actions:
   1. In the **Name** text box, enter **drop**.
   2. In the **Public access level** drop-down list, select **Blob (anonymous read access for blobs only)**.
   3. Select **OK**.
7. Back in the **Containers** section, select the newly created **drop** container.
8. In the **Container** blade, select **Upload**.
9. In the **Upload blob** pop-up, perform the following actions:
   1. In the **Files** section, select the **Folder** icon.
   2. In the File Explorer dialog box, go to **LabFiles**, select the **records.json** file, and then select **Open**.
   3. Ensure that **Overwrite if files already exist** is selected.
   4. Select **Upload**.
10. Wait for the blob to be uploaded before you continue with this lab.
11. Back in the **Container** blade, select the **records.json** blob from the list of blobs.
12. In the **Blob** blade, view the blob metadata.
13. Copy the **URL** for the blob.
14. On the taskbar, right-select the **Microsoft Edge** icon and then select **New window**.
15. In the new browser window, navigate to the **URL** that you copied for the blob.
16. You should now see the **JSON** contents of the blob. Close the browser window showing the **JSON** contents.
17. Return to the browser window with the **Azure portal.**
18. Close the **Blob** blade.
19. Back in the **Container** blade, select **Change access level policy** located at the top of the blade.
20. In the **Change access level** pop-up that appears, perform the following actions:
    1. In the **Public access level** drop-down list, select **Private (no anonymous access)**.
    2. Select **OK**.
21. On the taskbar, right-select the **Microsoft Edge** icon and then select **New window**.
22. In the new browser window, navigate to the **URL** that you copied for the blob.
23. You should now see an error message indicating that the resource was not found.

**Note**: If you do not see the error message, your browser might have cached the file. Use **Ctrl+F5** to refresh the page until you see the error message.

#### Task 2: Pull the Storage Account SDK from NuGet

1. On the navigation menu located on the left side of the portal, select the **Resource groups** link.
2. In the **Resource groups** blade, locate and select the **SecureFunction** resource group that you created earlier in this lab.
3. In the **SecureFunction** blade, select the **securefunc\*** function app that you created earlier in this lab.
4. In the **Function App** blade, locate and select the existing **FileParser** function to open the editor for the function.

**Note**: You might need to expand the **Functions** option in the menu on the left side of the blade.

1. On the right side of the editor, select **View files** to open the tab.
2. In the **View files** tab, select **Add**.
3. In the filename dialog that appears, enter **function.proj** and press Enter (displays an empty code editor).
4. In the file editor, insert this configuration content:

<Project Sdk="Microsoft.NET.Sdk">

<PropertyGroup>

<TargetFramework>netstandard2.0</TargetFramework>

</PropertyGroup>

<ItemGroup>

<PackageReference Include="Azure.Storage.Blobs" Version="12.0.0" />

</ItemGroup>

</Project>

1. In the editor, select **Save** button to persist your changes to the configuration.

**Note**: This **.proj** file contains the NuGet package reference necessary to import the [Azure.Storage.Blobs](https://www.nuget.org/packages/Azure.Storage.Blobs/12.0.0) package.

1. Select the **run.csx** file to return to the editor for the **FileParser** function.
2. Minimize the **View files** tab.

**Note**: You can minimize the tab by selecting the arrow immediately to the right of the tab header.

1. Within the editor, delete the existing code within the **Run** method of the script.
2. At the top of the code file, add the following line of code to create a **using** directive for the **Azure.Storage** namespace:

using Azure.Storage;

1. At the top of the code file, add the following line of code to create a **using** directive for the **Azure.Storage.Blobs** namespace:

using Azure.Storage.Blobs;

1. Add the following line of code to create a **using** directive for the **Azure.Storage.Blobs.Models** namespace:

using Azure.Storage.Blobs.Models;

1. The **Run** method should now look like this:

using System.Net;

using Microsoft.AspNetCore.Mvc;

using Azure.Storage;

using Azure.Storage.Blobs;

using Azure.Storage.Blobs.Models;

public static async Task<IActionResult> Run(HttpRequest req)

{

}

#### Task 3: Write storage account code

1. Add the following line of code within the **Run** method to get the value of the **StorageConnectionString** application setting by using the **Environment.GetEnvironmentVariable** method:

string connectionString = Environment.GetEnvironmentVariable("StorageConnectionString");

1. Add the following line of code to create a new instance of the **BlobServiceClient** class by passing in your connectionString variable to the constructor:

BlobServiceClient serviceClient = new BlobServiceClient(connectionString);

1. Add the following line of code to use the **BlobServiceClient.GetBlobContainerClient** method, while passing in the **drop** container name to create a new instance of the **BlobContainerClient** class that references the container that you created earlier in this lab:

BlobContainerClient containerClient = serviceClient.GetBlobContainerClient("drop");

1. Add the following line of code to use the **BlobContainerClient.GetBlobClient** method, while passing in the **records.json** blob name to create a new instance of the **BlobClient** class that references the blob that you uploaded earlier in this lab:

BlobClient blobClient = containerClient.GetBlobClient("records.json");

1. The **Run** method should now look like this:

using System.Net;

using Microsoft.AspNetCore.Mvc;

using Azure.Storage;

using Azure.Storage.Blobs;

using Azure.Storage.Blobs.Models;

public static async Task<IActionResult> Run(HttpRequest req)

{

string connectionString = Environment.GetEnvironmentVariable("StorageConnectionString");

BlobServiceClient serviceClient = new BlobServiceClient(connectionString);

BlobContainerClient containerClient = serviceClient.GetBlobContainerClient("drop");

BlobClient blobClient = containerClient.GetBlobClient("records.json");

}

#### Task 4: Download a blob

1. Add the following line of code to use the **BlobClient.DownloadAsync** method to download the contents of the referenced blob asynchronously and store the result in a variable named response:

var response = await blobClient.DownloadAsync();

1. Add the following line of code to return the various content stored in the content variable by using the **FileStreamResult** class constructor:

return new FileStreamResult(response?.Value?.Content, response?.Value?.ContentType);

1. The **Run** method should now look like this:

using System.Net;

using Microsoft.AspNetCore.Mvc;

using Azure.Storage;

using Azure.Storage.Blobs;

using Azure.Storage.Blobs.Models;

public static async Task<IActionResult> Run(HttpRequest req)

{

string connectionString = Environment.GetEnvironmentVariable("StorageConnectionString");

BlobServiceClient serviceClient = new BlobServiceClient(connectionString);

BlobContainerClient containerClient = serviceClient.GetBlobContainerClient("drop");

BlobClient blobClient = containerClient.GetBlobClient("records.json");

var response = await blobClient.DownloadAsync();

return new FileStreamResult(response?.Value?.Content, response?.Value?.ContentType);

}

1. Select **Save and run** to save the script and perform a test execution of the function.
2. Observe the **Output** text box in the **Test** pane. You should now see the content of the **$/drop/records.json** blob stored in your **storage account**.

#### Review

In this exercise, you used C# code to access a Storage Account securely and then download the contents of a blob.