8 min read

Microsoft Entra ID

Using Managed Identities in Logic App HTTP triggers

Robbe Van den Daele
Aug 3, 2023 • 8 min read

Photo by Kelly Sikkema / Unsplash

Introduction

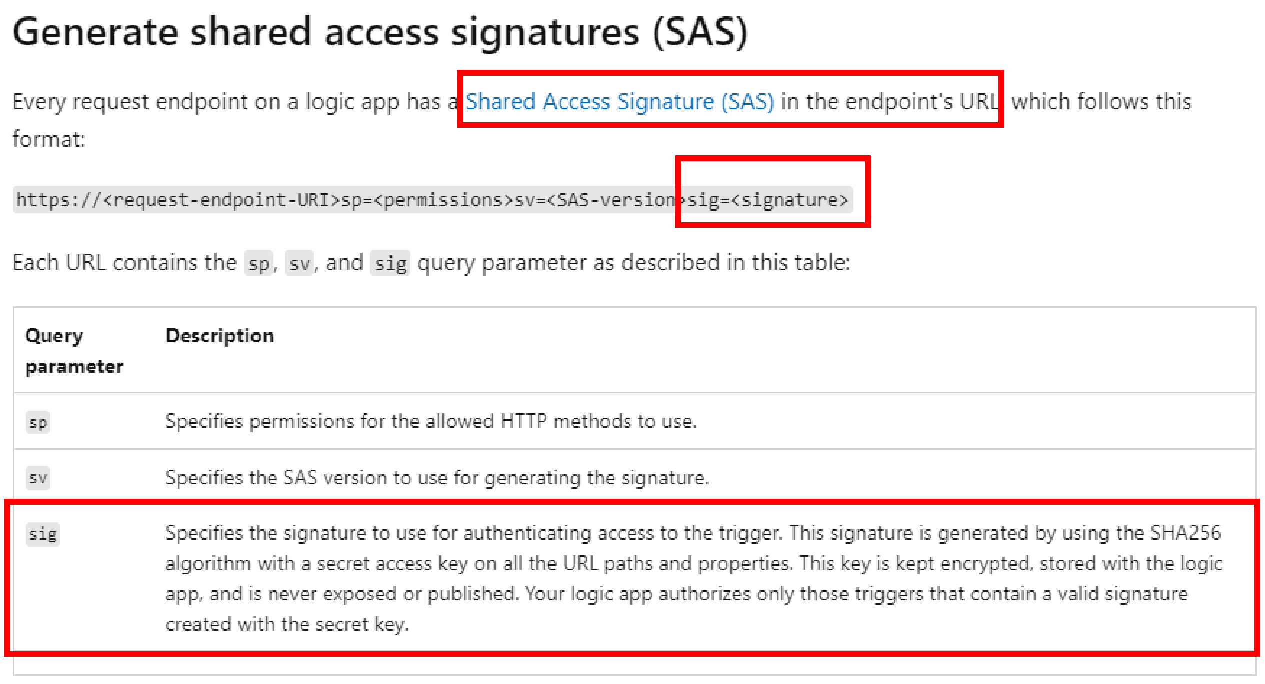
During the past few months, I have been assessing the security configurations of applications created in Azure. During these assessments, I found that people like to use Logic Apps and HTTP triggers to create simple APIs or integration flows. Often these HTTP triggers are being called by other Azure resources, which is why they use Azure Managed Identities for authentication to the Logic App HTTP endpoints. At least, they think that they are using managed identities…

Common HTTP trigger misconception

An Azure Logic App using an HTTP trigger supports two methods for authenticating incoming calls, these are:

- SAS Tokens
- Azure AD OAuth

By default, an HTTP trigger will generate an URL that you can use to access the trigger. What most people forget is that **this URL contains a SAS token by default**. Since the default authentication method of the HTTP trigger is SAS authentication, a lot of people are authenticating to Logic Apps HTTP triggers via SAS tokens without knowing they are.



Generate shared access signatures (SAS)

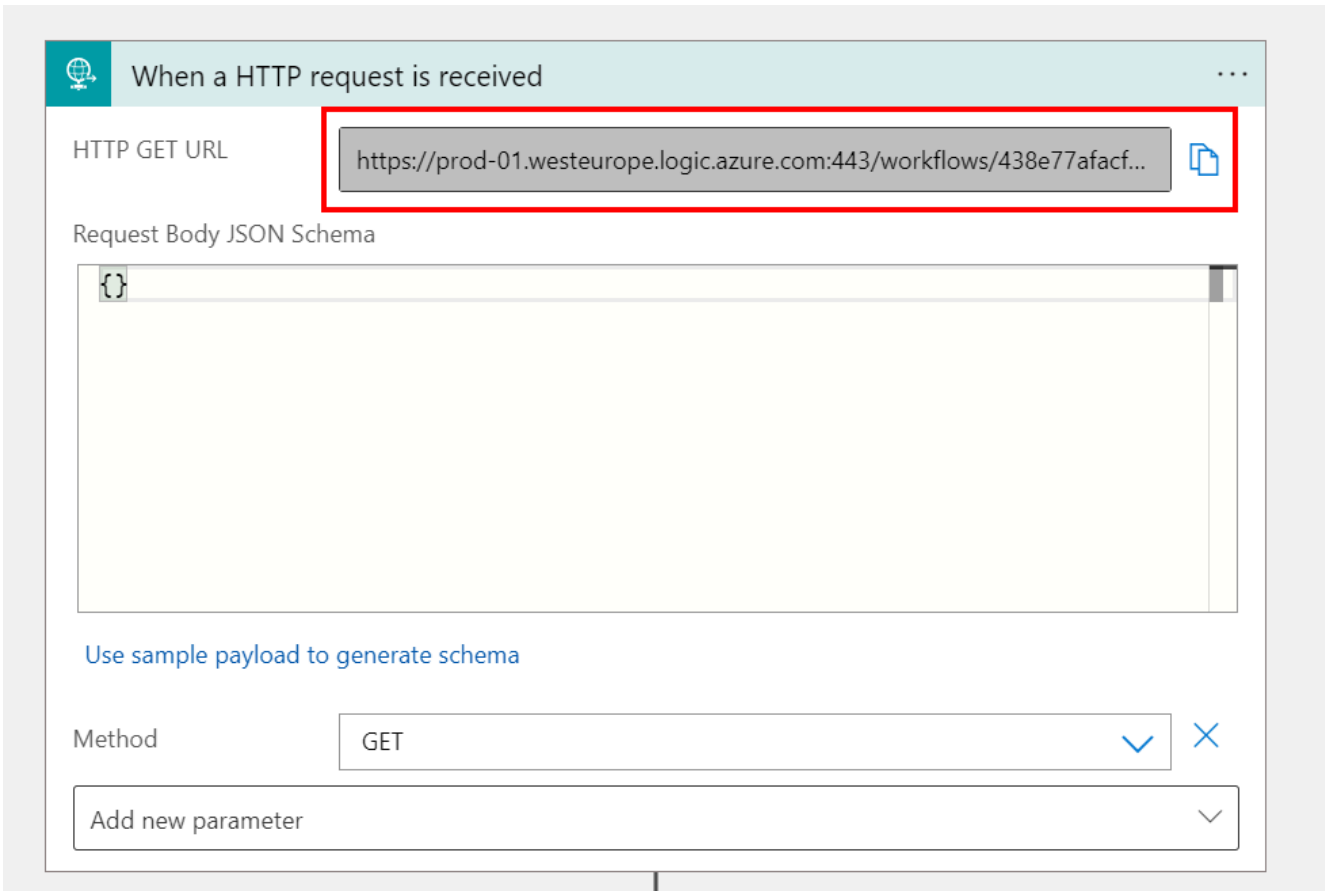
Every request endpoint on a logic app has a Shared Access Signature (SAS) in the endpoint's URL which follows this format:

`https://<request-endpoint-URI>sp=<permissions>sv=<SAS-version>sig=<signature>`

Each URL contains the `sp`, `sv`, and `sig` query parameter as described in this table:

Query parameter	Description
`sp`	Specifies permissions for the allowed HTTP methods to use.
`sv`	Specifies the SAS version to use for generating the signature.
`sig`	Specifies the signature to use for authenticating access to the trigger. This signature is generated by using the SHA256 algorithm with a secret access key on all the URL paths and properties. This key is kept encrypted, stored with the logic app, and is never exposed or published. Your logic app authorizes only those triggers that contain a valid signature created with the secret key.

Secure access and data - Azure Logic Apps | Microsoft Learn



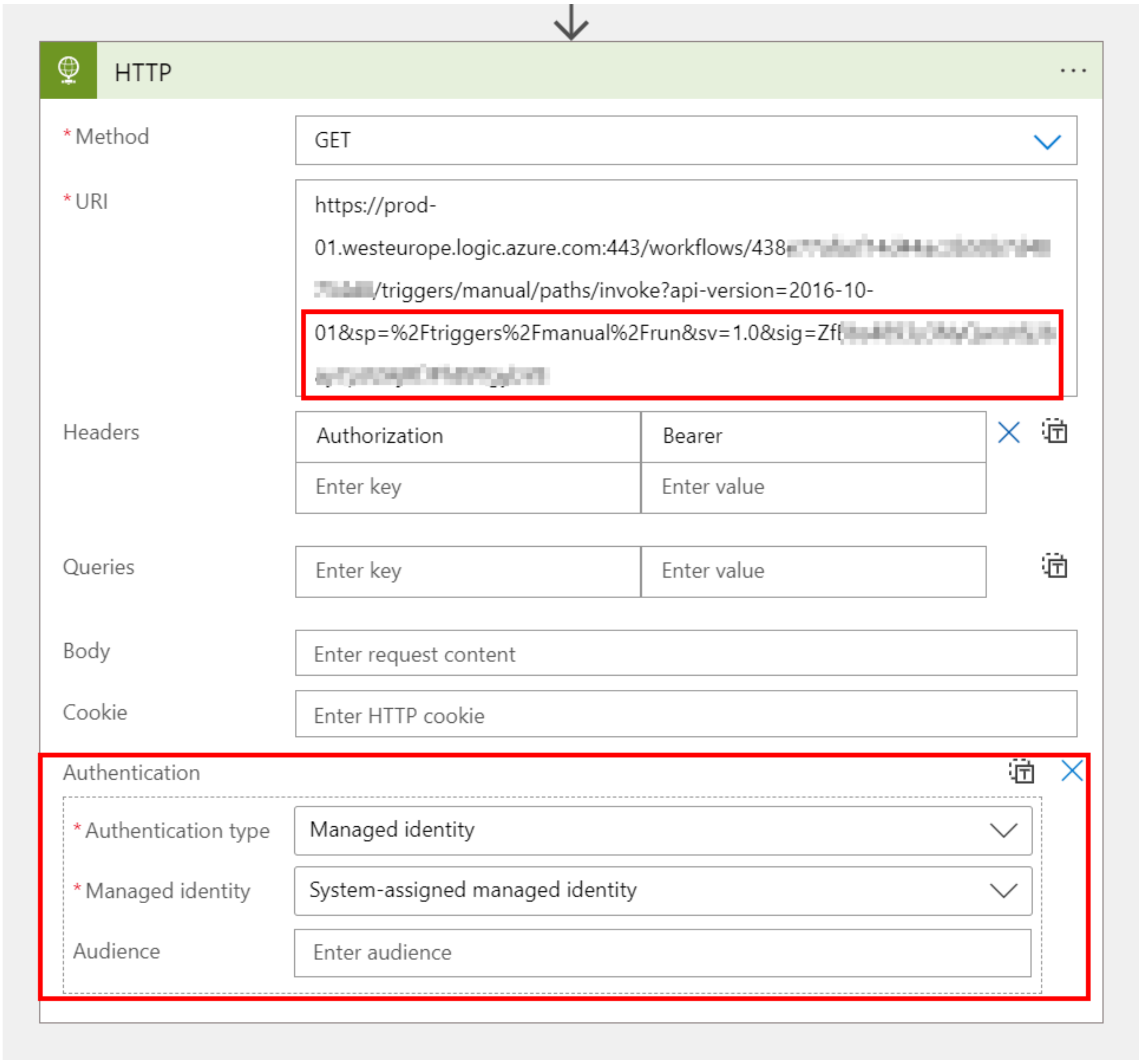
When a HTTP request is received

HTTP GET URL: https://prod-01.westeurope.logic.azure.com:443/workflows/438e77afacf...

Request Body JSON Schema: {}

Method: GET

The misconfiguration I often see in production environments, is when they are triggering an HTTP trigger of a Logic App from another Azure resource, they just set the Authentication method used of the source resource as Managed Identities without changing any other configuration. As an example, I will use another Logic App that will send a HTTP request to a Logic App with a HTTP trigger.


HTTP

* Method	GET
* URI	https://prod-01.westeurope.logic.azure.com:443/workflows/438[redacted]/triggers/manual/paths/invoke?api-version=2016-10-01&sp=%2Ftriggers%2Fmanual%2Frun&sv=1.0&sig=Zf[redacted]
Headers	Authorization: Bearer
Queries	Enter key / Enter value
Body	Enter request content
Cookie	Enter HTTP cookie

Authentication

* Authentication type	Managed identity
* Managed identity	System-assigned managed identity
Audience	Enter audience

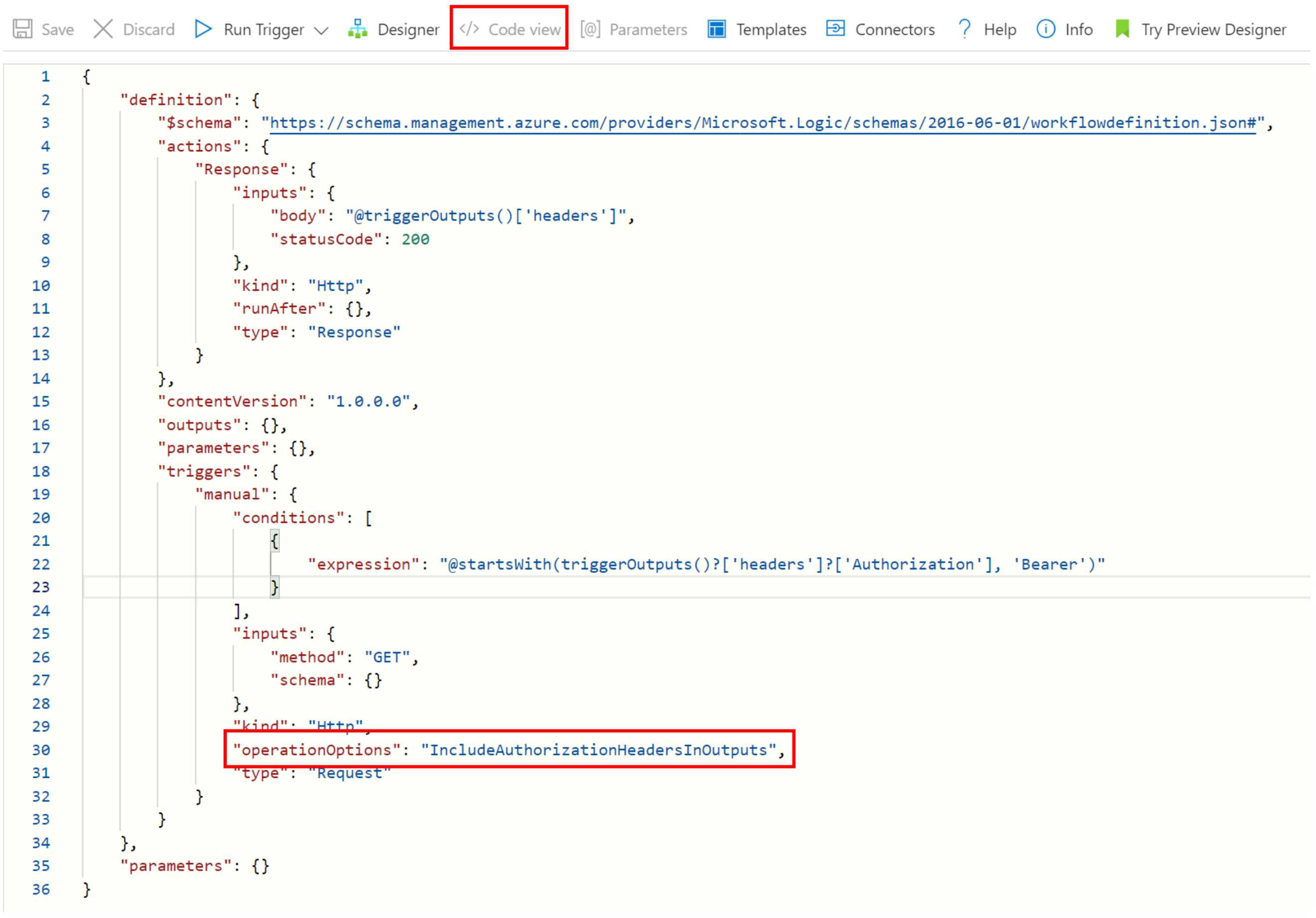
Since the URI still contains the SAS token, and the HTTP trigger in the destination Logic App is configured to use SAS by default, the **managed identity configured in the Authentication field will not be used**. This makes people forget that the URI still is a confidential string, since **anyone with the default URI has the SAS token to authenticate to the Logic App HTTP endpoint**.

Switching to Managed Identities

To really switch to using Managed Identities, we have to configure a couple of things in the HTTP trigger of the destination Logic App.

Include Authorization Header

First of all, the request trigger or HTTP webhook needs to be configured to include the authorization header (which will represent the Managed Identity used in HTTP requests) in the triggers outputs. This can be done by setting the `IncludeAuthorizationHeadersInOutputs` value to the `operationOptions` field in the HTTP trigger. This can be done by using the code view:



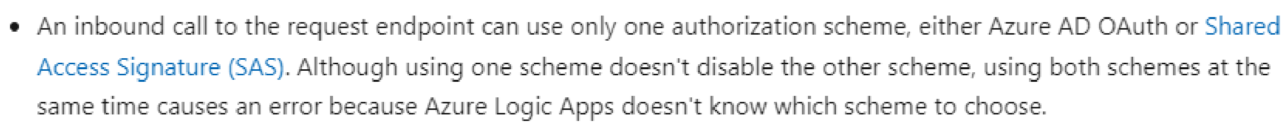
```
{
    "definition": {
        "$schema": "https://schema.management.azure.com/providers/Microsoft.Logic/schemas/2016-06-01/workflowdefinition.json#",
        "actions": {
            "Response": {
                "inputs": {
                    "body": "@triggerOutputs()['headers']",
                    "statusCode": 200
                },
                "kind": "Http",
                "runAfter": {},
                "type": "Response"
            }
        },
        "contentVersion": "1.0.0.0",
        "outputs": {},
        "parameters": {},
        "triggers": {
            "manual": {
                "conditions": [
                    {
                        "expression": "@startsWith(triggerOutputs()?['headers']?['Authorization'], 'Bearer')"
                    }
                ],
                "inputs": {
                    "method": "GET",
                    "schema": {}
                },
                "kind": "Http",
                "operationOptions": "IncludeAuthorizationHeadersInOutputs",
                "type": "Request"
            }
        }
    },
    "parameters": {}
}
```

```
"triggers": {
   "manual": {
      "inputs": {
         "schema": {}
      },
      "kind": "Http",
      "type": "Request",
      "operationOptions": "IncludeAuthorizationHeadersInOutputs"
   }
}
```

By doing this, we made sure that we can validate OAuth tokens when they are being sent by source applications.

Disable SAS Authentication

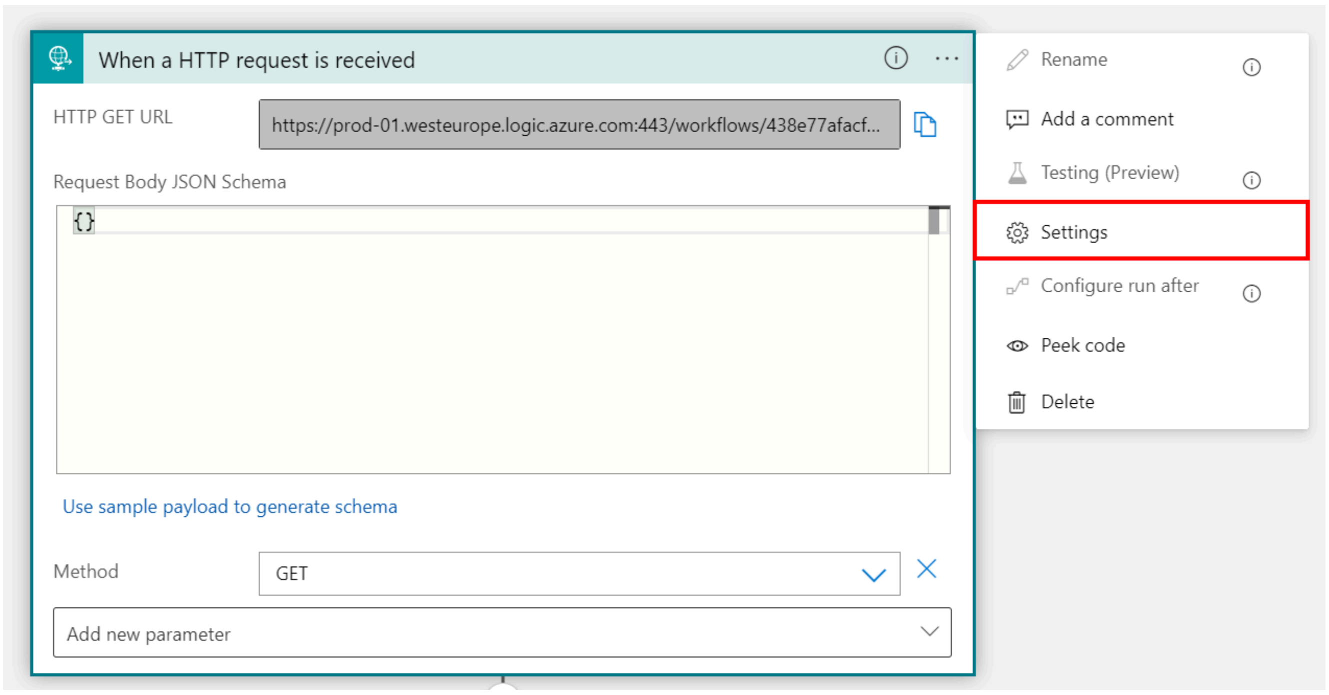
To make sure SAS authentication cannot be used, we have to fiddle a little bit with how these HTTP triggers work. In the Microsoft Docs we can read that enabling OAuth authentication does not disable SAS authentication, but using both will resolve in an error:

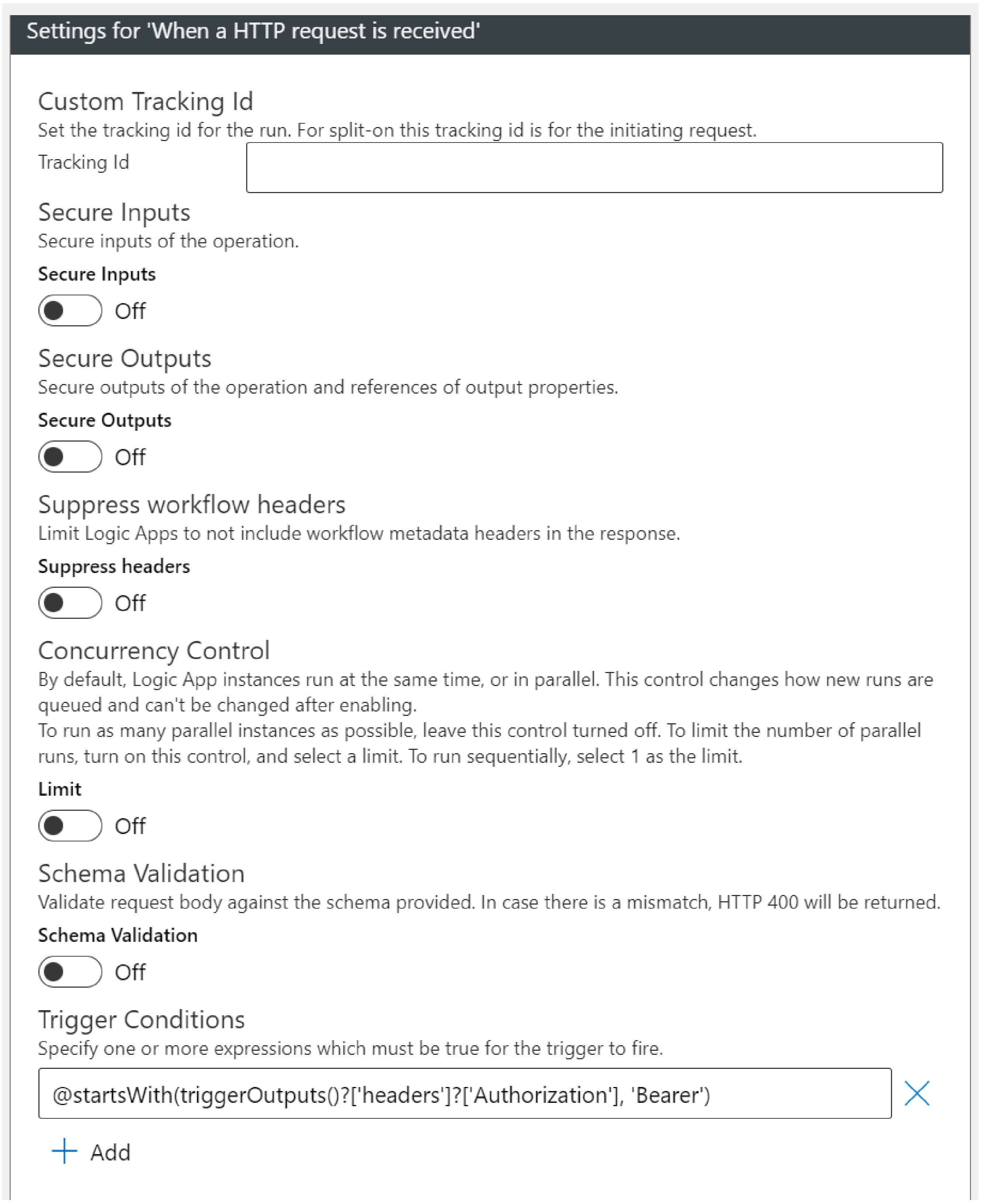


- An inbound call to the request endpoint can use only one authorization scheme, either Azure AD OAuth or Shared Access Signature (SAS). Although using one scheme doesn't disable the other scheme, using both schemes at the same time causes an error because Azure Logic Apps doesn't know which scheme to choose.

Secure access and data - Azure Logic Apps | Microsoft Learn

This means that if we only allow OAuth requests to trigger the HTTP trigger, we can force that all SAS authentication requests fail. To do this, we need to add the `@startsWith(triggerOutputs()?['headers']?['Authorization'], 'Bearer')` line to the trigger conditions in the settings of the HTTP trigger:


When a HTTP request is received
HTTP GET URL
https://prod-01.westeurope.logic.azure.com:443/workflows/438e77afacf...
Request Body JSON Schema
{}
Use sample payload to generate schema
Method
GET
Add new parameter
Rename
Add a comment
Testing (Preview)
Settings
Configure run after
Peek code
Delete


Settings for 'When a HTTP request is received'

Custom Tracking Id
Set the tracking id for the run. For split-on this tracking id is for the initiating request.
Tracking Id

Secure Inputs
Secure inputs of the operation.
Secure Inputs
Off

Secure Outputs
Secure outputs of the operation and references of output properties.
Secure Outputs
Off

Suppress workflow headers
Limit Logic Apps to not include workflow metadata headers in the response.
Suppress headers
Off

Concurrency Control
By default, Logic App instances run at the same time, or in parallel. This control changes how new runs are queued and can't be changed after enabling.
To run as many parallel instances as possible, leave this control turned off. To limit the number of parallel runs, turn on this control, and select a limit. To run sequentially, select 1 as the limit.
Limit
Off

Schema Validation
Validate request body against the schema provided. In case there is a mismatch, HTTP 400 will be returned.
Schema Validation
Off

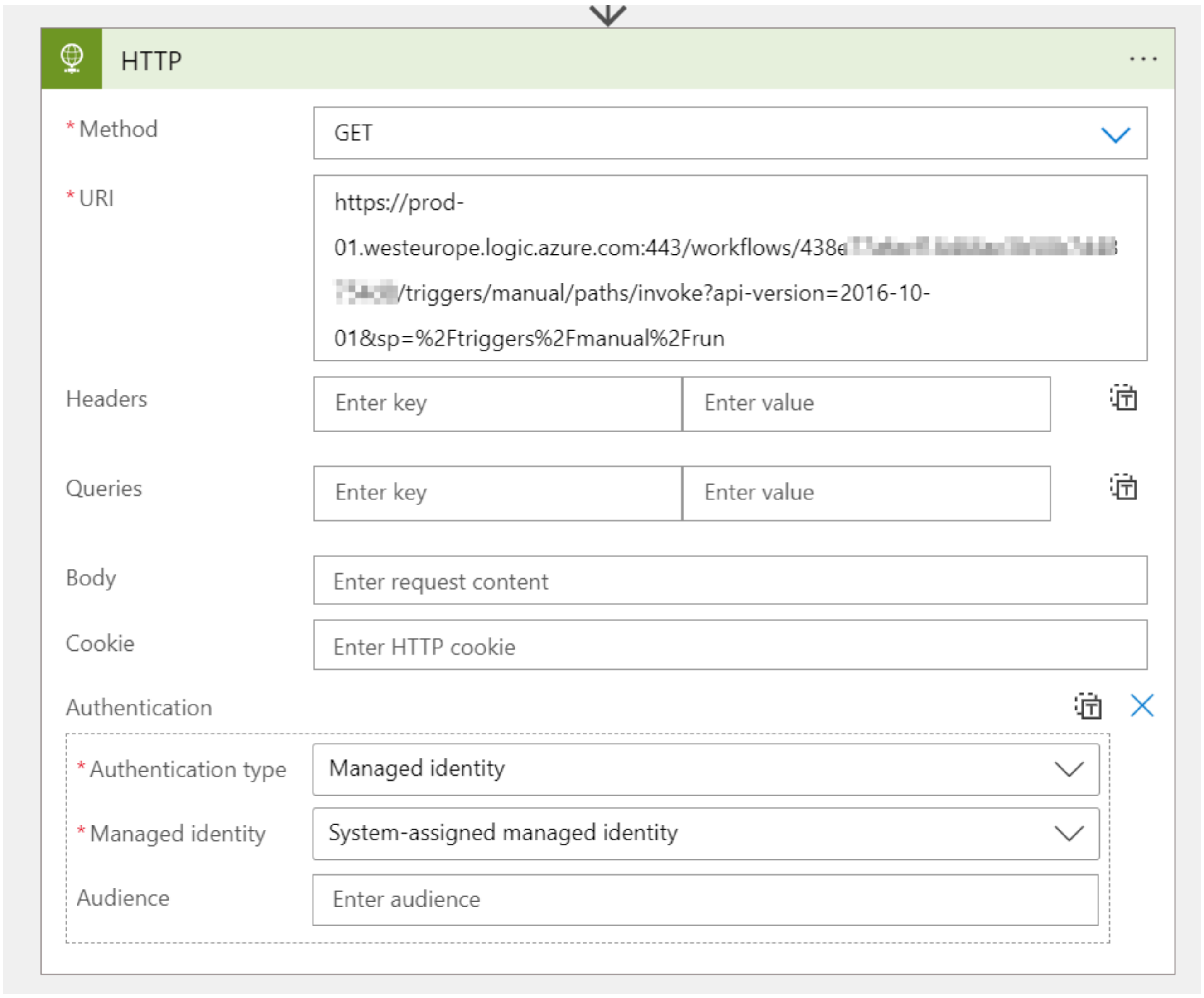
Trigger Conditions
Specify one or more expressions which must be true for the trigger to fire.

```
@startsWith(triggerOutputs()?['headers']?['Authorization'], 'Bearer')
```

+ Add

By doing this, we make sure that requests containing SAS tokens with or without OAuth tokens will be rejected. **This means that the new URI you will need to use in the application that is sending requests, is the URI with the `sv` and `sig` parameters omitted**.

```
https://<request-endpoint-URI>sp=<permissions>
```

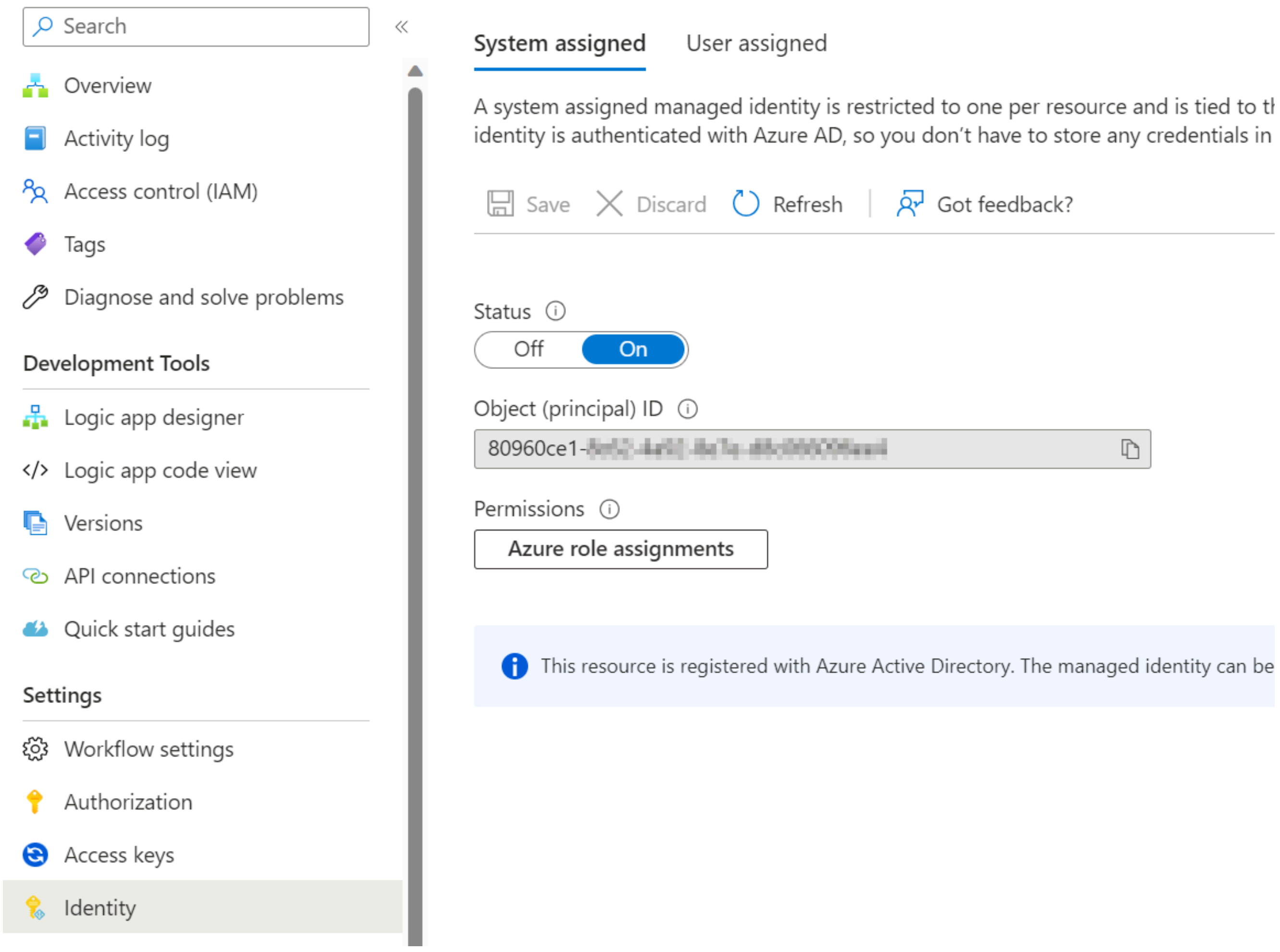


HTTP

* Method	GET
* URI	https://prod-01.westeurope.logic.azure.com:443/workflows/438e[redacted]/triggers/manual/paths/invoke?api-version=2016-10-01&sp=%2Ftriggers%2Fmanual%2Frun
Headers	Enter key / Enter value
Queries	Enter key / Enter value
Body	Enter request content
Cookie	Enter HTTP cookie
Authentication	
* Authentication type	Managed identity
* Managed identity	System-assigned managed identity
Audience	Enter audience

Enable Managed Identity

Before we proceed, we will need to enable a Managed Identity for the Logic App that will be sending requests to the HTTP Endpoint. This can be either a User Assigned Managed Identity or a System Assigned Managed Identity.



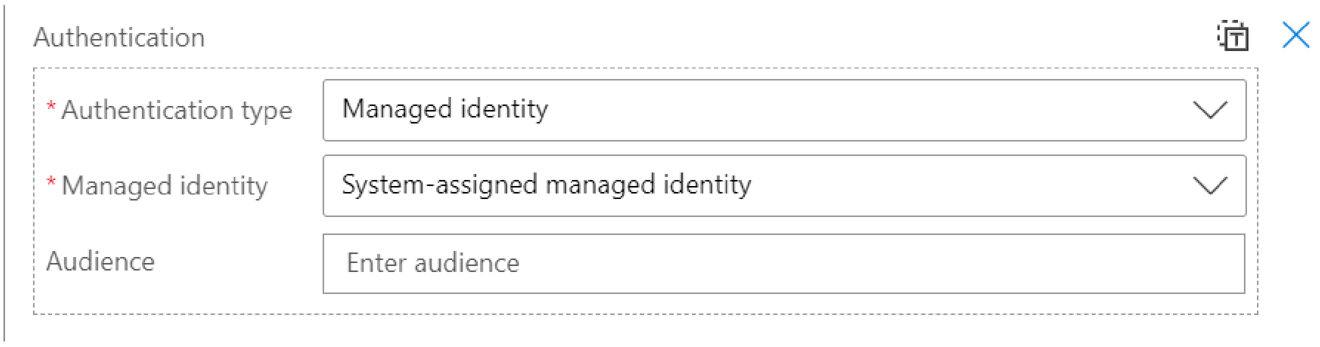
A system assigned managed identity is restricted to one per resource and is tied to tl
identity is authenticated with Azure AD, so you don't have to store any credentials in

Status: Off [✓ On]

Object (principal) ID: 80960ce1-[redacted]

This resource is registered with Azure Active Directory. The managed identity can be

Once this is done, the Managed Identity needs to be configured in the Logic App that will be sending requests:

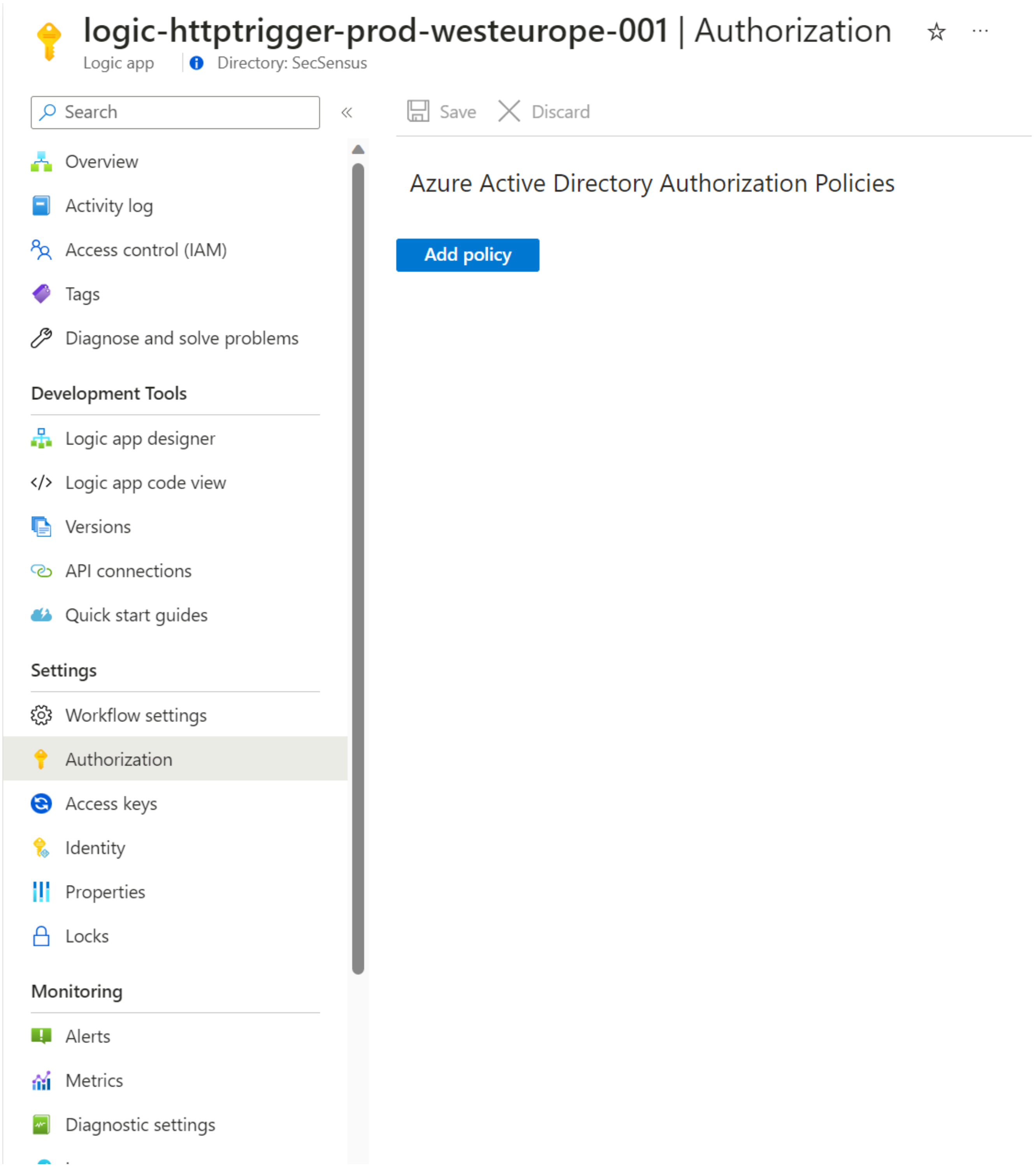


* Authentication type	Managed identity
* Managed identity	System-assigned managed identity
Audience	Enter audience

Defining Authorization Policies

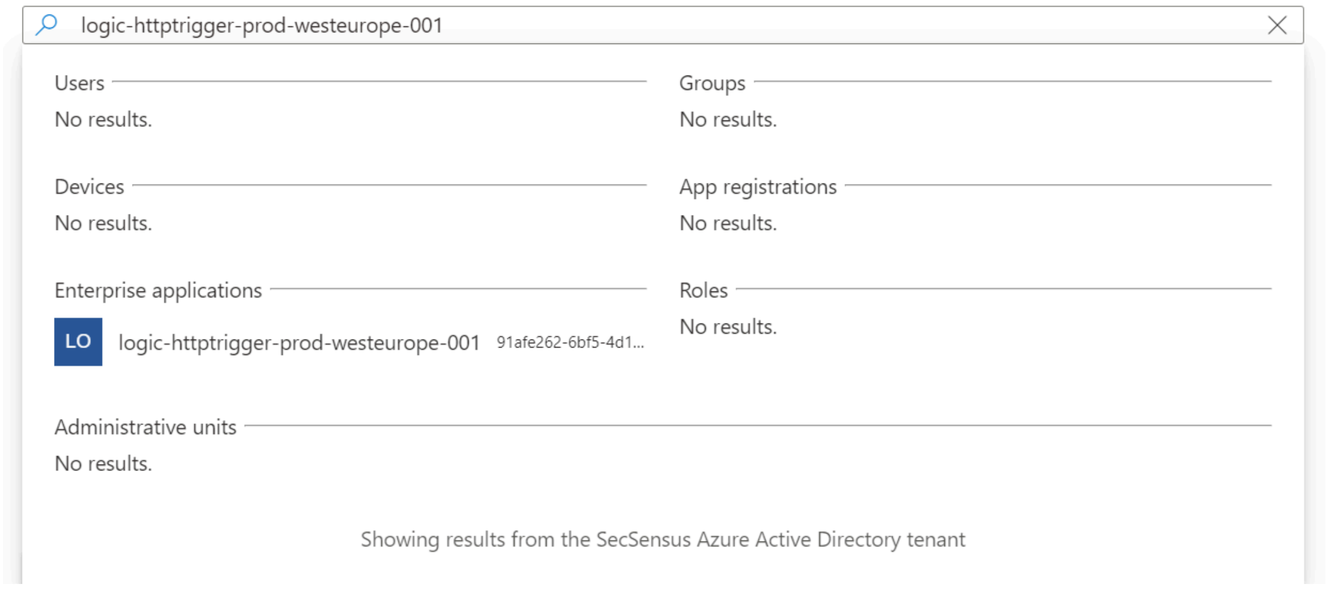
When using OAuth, the **application owner is responsible for configuring proper authorization conditions in the applications**.

Therefore, we still need to create authorization policies in the Logic App so only known managed identities are able to authenticate to the HTTP endpoint. These policies can be created by going to the 'Authorization' tab in the Logic App:



Here we need to define policies that checks if certain Claims in the tokens are valid. For single-tenant applications, the following Claims should be checked:

- Issuer Claim, which makes sure that only tokens issued by our Entra ID tenant are allowed to access the trigger: `https://sts.windows.net/<tenantid>/`.
- The Audience Claim, which checks the intended consumer of the token. For Managed Identities this should be `https://management.azure.com/`.
- The appid Claim, which is the App ID of the Enterprise Application of the Managed Identity used by the sending Logic App (can be found by searching the name of the Managed Identity in Azure AD)


logic-httptrigger-prod-westeurope-001

Users
No results.

Groups
No results.

Devices
No results.

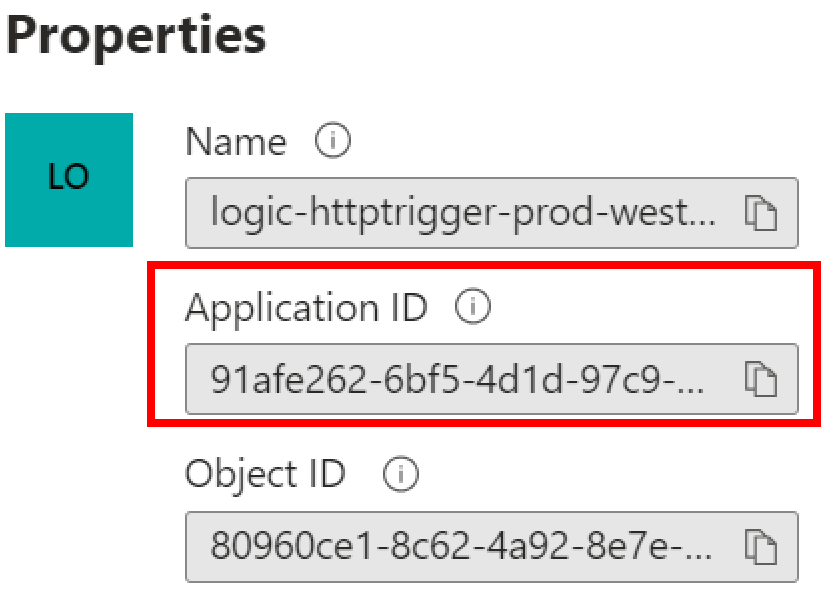
App registrations
No results.

Enterprise applications
LO logic-httptrigger-prod-westeurope-001 91afe262-6bf5-4d1...

Roles
No results.

Administrative units
No results.

Showing results from the SecSensus Azure Active Directory tenant

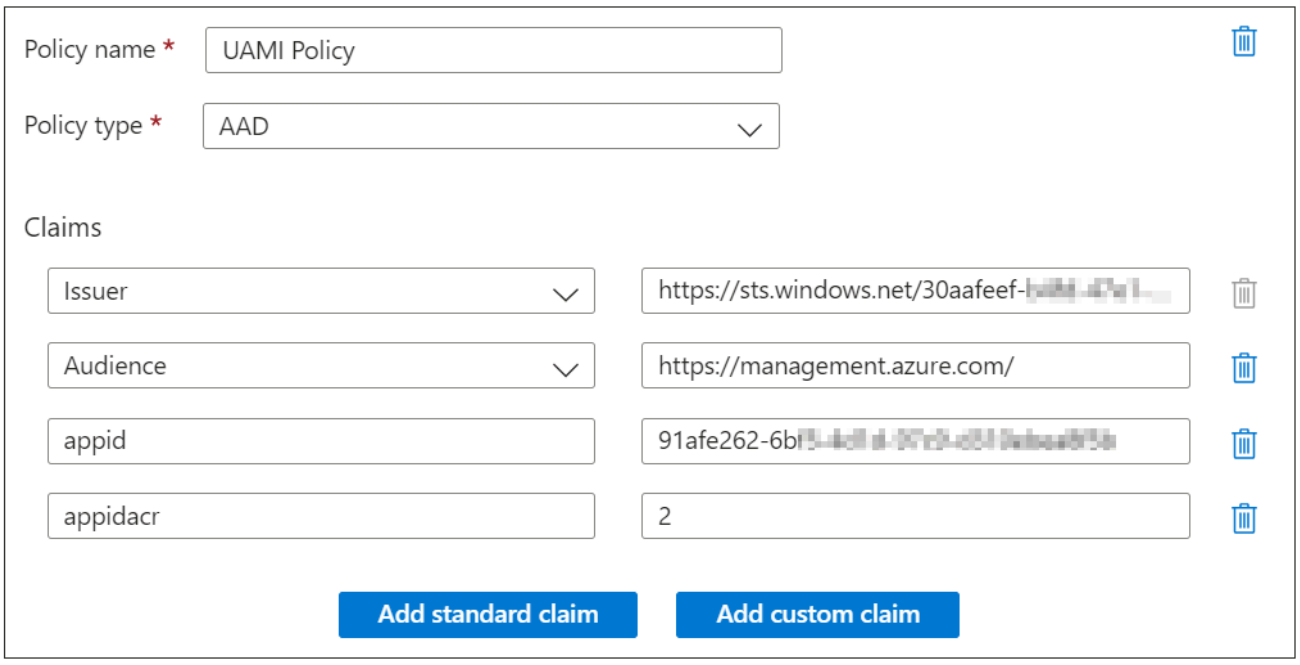


Properties

Name	logic-httptrigger-prod-west...
Application ID	91afe262-6bf5-4d1d-97c9-...
Object ID	80960ce1-8c62-4a92-8e7e-...

- The appidacr Claim, which checks that authentication of the Managed Identity is done via Client Certificates. This should have the value `2`.

The complete policy should be something like the following:



Policy name * UAMI Policy

Policy type * AAD

Claims

Issuer	https://sts.windows.net/30aafeef-[redacted]
Audience	https://management.azure.com/
appid	91afe262-6b[redacted]
appidacr	2

By configuring this policy, we make sure that only tokens issued by our tenant are allowed, only managed identities authenticated via Client

Certificates are authorized, and only the Managed Identity with the configured Application ID is authorized for the HTTP trigger.

Reference: Secure access and data - Azure Logic Apps | Microsoft Learn

When you now test the request, only the Logic App with the configured Managed Identity should be allowed to access the destination Logic App with the configured Authorization policies.

Allowing Multiple Identities

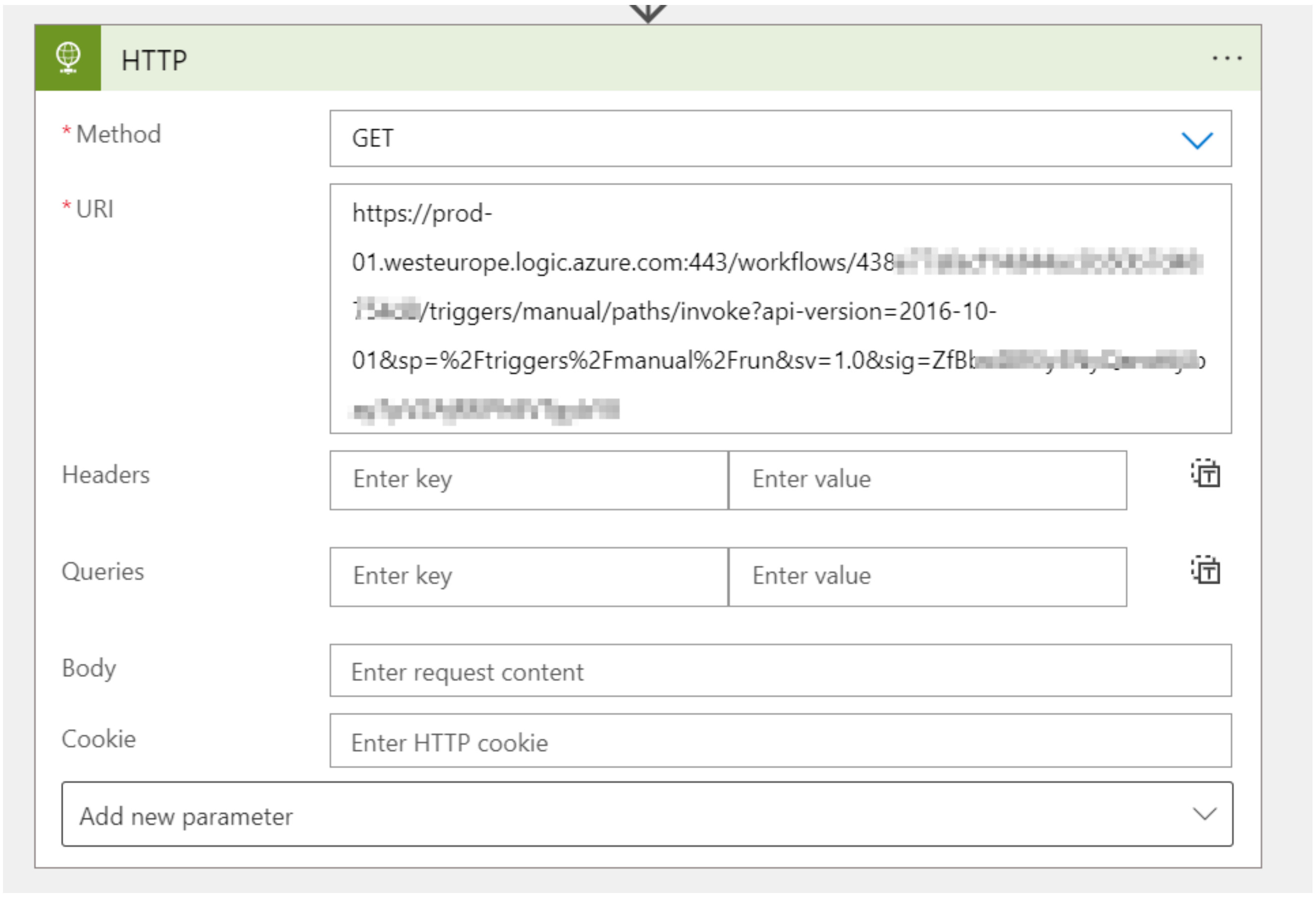
Allowing multiple identities to your Logic App trigger can be done using various ways. The most easy one is to **configure the source applications to use the same User Assigned Managed Identity**, and allow that UAMI in your Authorization Policy.

Another way of doing it is by **creating multiple Authorization policies**. When a token complies with one of the multiple configured policies, the token will pass the authorization process (there is an or relationship between the policies). The only thing you have to change in the new policy is the appid of the new Managed Identity.

Error Codes

When testing unauthorized access, you should run into the following Error Codes:

- When configuring a request without a Managed Identity but with the SAS token URI, you should get the error code like described below.


HTTP

*Method: GET

*URI: https://prod-01.westeurope.logic.azure.com:443/workflows/438[redacted]/triggers/manual/paths/invoke?api-version=2016-10-01&sp=%2Ftriggers%2Fmanual%2Frun&sv=1.0&sig=ZfBb[redacted]

Headers: Enter key | Enter value

Queries: Enter key | Enter value

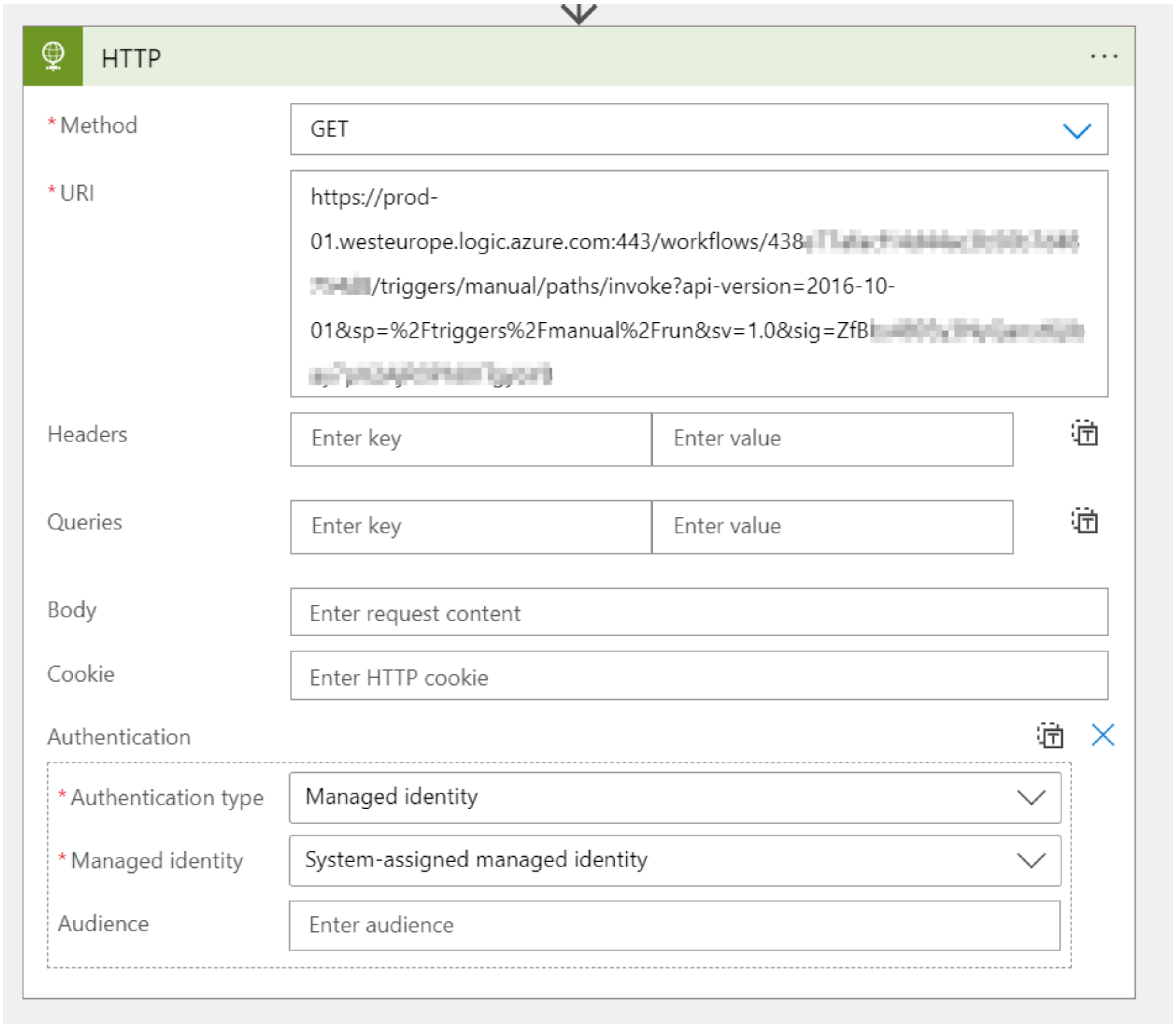
Body: Enter request content

Cookie: Enter HTTP cookie

Add new parameter

```
{
  "error": {
    "code": "InvalidTemplate",
    "message": "The template language expression evaluation failed: 'The template la
  }
}
```

- When configuring a request with the SAS token and a Managed Identity (being the wrong or correct identity), you should get the error code like described below.


HTTP

* Method: GET

* URI: https://prod-01.westeurope.logic.azure.com:443/workflows/438[redacted]/triggers/manual/paths/invoke?api-version=2016-10-01&sp=%2Ftriggers%2Fmanual%2Frun&sv=1.0&sig=ZfBI[redacted]

Headers: Enter key | Enter value

Queries: Enter key | Enter value

Body: Enter request content

Cookie: Enter HTTP cookie

Authentication

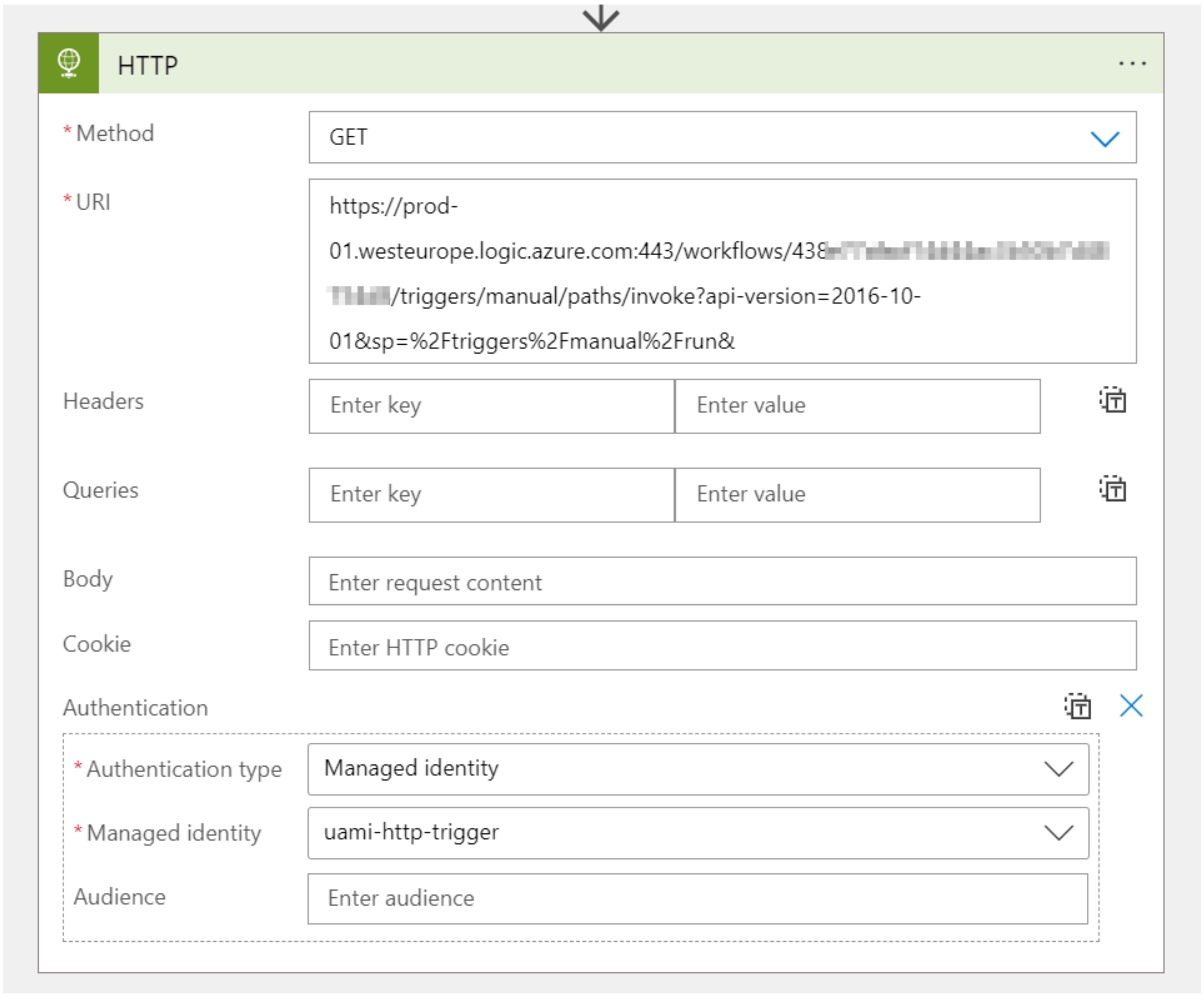
* Authentication type: Managed identity

* Managed identity: System-assigned managed identity

Audience: Enter audience

```
{
  "error": {
    "code": "InvalidTemplate",
    "message": "The template language expression evaluation failed: 'The template la
  }
}
```

- When configuring a request with the URI omitting the SAS token but providing a wrong Managed Identity, you should get the error code like described below.



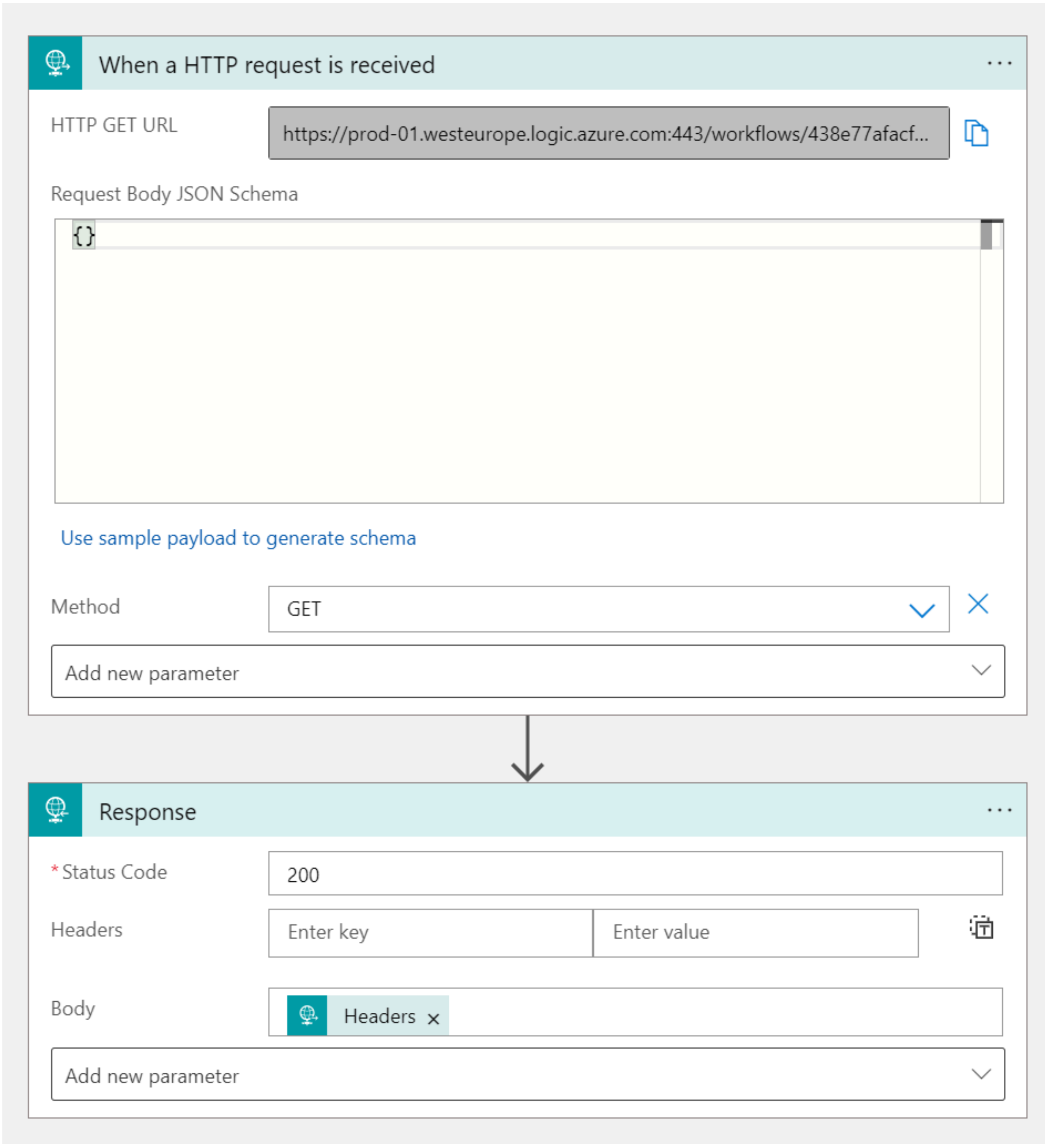
* Method	GET
* URI	https://prod-01.westeurope.logic.azure.com:443/workflows/438[redacted]/triggers/manual/paths/invoke?api-version=2016-10-01&sp=%2Ftriggers%2Fmanual%2Frun&
Headers	Enter key / Enter value
Queries	Enter key / Enter value
Body	Enter request content
Cookie	Enter HTTP cookie
Authentication	
* Authentication type	Managed identity
* Managed identity	uami-http-trigger
Audience	Enter audience

```
{
  "error": {
    "code": "MisMatchingOAuthClaims",
    "message": "One or more claims either missing or does not match with the open au
  }
}
```

Debugging

What if you are facing other issues, or want to check which Claims are present in the tokens you are sending? I will tell you how I debugged some of my issues.

First of all, I made sure to return the Header of the request in the body of the response when an authentication attempt succeeds:



When a HTTP request is received

HTTP GET URL: https://prod-01.westeurope.logic.azure.com:443/workflows/438e77afacf...

Request Body JSON Schema: {}

Use sample payload to generate schema

Method: GET

Add new parameter

Response

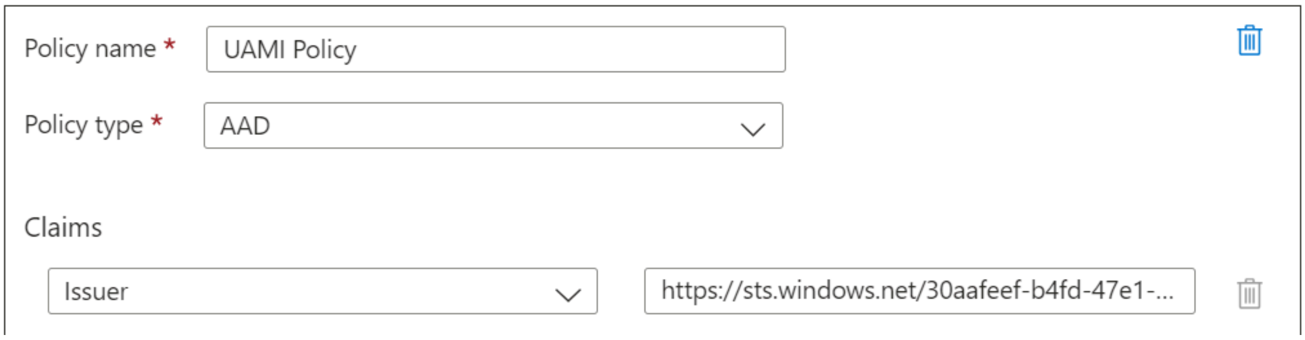
* Status Code: 200

Headers: Enter key | Enter value

Body: Headers

Add new parameter

Then, I created my Authorization policy as such that I only check the Issuer claim. This should always work since the Managed Identity lives in the same tenant as the Logic App (except if you are a hacker from another tenant ;))

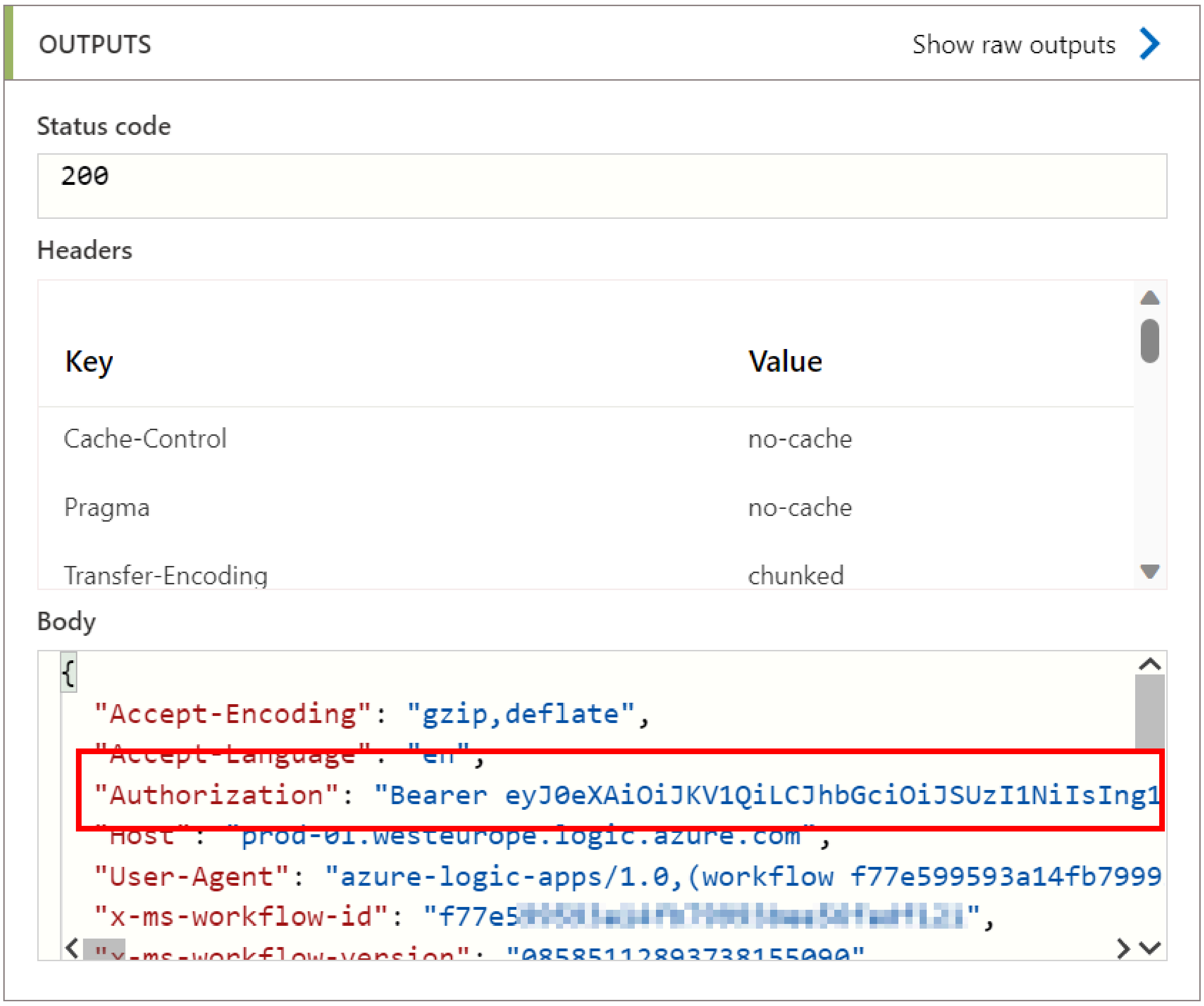


Policy name *	UAMI Policy
Policy type *	AAD

Claims

Issuer	https://sts.windows.net/30aafeef-b4fd-47e1-...

If this request works, you know that your issue resides in the Claim checks you are doing in the Authorization policies. To check which claims you are sending, you can now copy your Bearer token from the Sending Logic App (since we responded in the listening Logic App with the received Header).



OUTPUTS Show raw outputs

Status code

200

Headers

Key	Value
Cache-Control	no-cache
Pragma	no-cache
Transfer-Encoding	chunked

Body

```
{
  "Accept-Encoding": "gzip,deflate",
  "Accept-Language": "en",
  "Authorization": "Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiIsIng1
  "Host": "prod-01.westeurope.logic.azure.com",
  "User-Agent": "azure-logic-apps/1.0,(workflow f77e599593a14fb7999
  "x-ms-workflow-id": "f77e5[redacted]",
  "x-ms-workflow-version": "08585112893738155090"
```

Now you can decode your Bearer token with jwt.io, and check the Claims that are present. Using this, creating your Authorization policies should go much easier.

Things to keep in mind

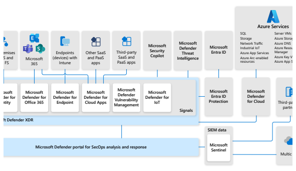
First of all, I would like to mention that SAS tokens are not perse bad. I just want to create awareness since the URL that gets generated in an HTTP trigger is in fact confidential, since it contains the SAS token for authentication. This is something application developers need to keep in mind. The URL should be kept in a safe place, or ideally, SAS tokens should be dynamically loaded from an Azure Key Vault for example. **Anyone with the original URL generated in the HTTP trigger is able to make calls to the trigger**.

Secondly, using OAuth in Logic Apps is not a bulletproof solution. In the Authorization policies there is no option to check for roles present an array of roles, which means that **you cannot build a mechanism for role-based authorization**. There is also no possibility to validate the authenticity of the token by checking the signature in the Logic App Authorization policies. This means (but I still have to test this), that **miscreated tokens might be able to pass the Authorization policies since there is no check on the signature of the token**. For **mission-critical APIs, it is better to use a Function App with a programming language that supports a library for validating OAuth tokens**. This will make sure all the necessary checks can be done and will be done like they are supposed to be done. Examples of Microsoft authentication libraries can be found here: Microsoft identity platform authentication libraries - Microsoft Entra | Microsoft Learn.

Lastly, I wanted to mention **mitigating controls can also be used for securing HTTP triggers**. For example, you can use Azure API Management or Access Controls to limit incoming calls by source IP. More info on Secure access and data - Azure Logic Apps | Microsoft Learn.

Conclusion

In this blog post, you should have learned how to properly migrate your Logic App with an HTTP trigger to use Azure OAuth instead of SAS tokens. We used another Logic App for triggering the HTTP trigger Logic App, but this process should be the same if you use other resources that support Managed Identities or OAuth as authentication to HTTP triggers.



Transition from Microsoft Sentinel to Defender XDR -

Detecting non-privileged Windows Hello abuse

MDE Device Discovery - Improving the monitored

Practical challenges

Introduction Microsoft announced on the…

Jul 4, 2025 12 min read

Introduction I recently followed a live session of Dirk…

Apr 26, 2025 16 min read

network page

Introduction This blogpost is probably the first …

Mar 19, 2025 6 min read