How to Enable Cross-Account Access to the AWS Management Console

by Mike Pope | on 13 MAY 2014 | in [AWS Identity And Access Management (IAM)](https://aws.amazon.com/blogs/security/category/security-identity-compliance/aws-identity-and-access-management-iam/), [How-To](https://aws.amazon.com/blogs/security/category/how-to/) | [Permalink](https://aws.amazon.com/blogs/security/how-to-enable-cross-account-access-to-the-aws-management-console/) | [Comments](https://commenting.awsblogs.com/embed.html?disqus_shortname=aws-security-blog&disqus_identifier=850&disqus_title=How+to+Enable+Cross-Account+Access+to+the+AWS+Management+Console&disqus_url=https://aws.amazon.com/blogs/security/how-to-enable-cross-account-access-to-the-aws-management-console/) | [Share](https://aws.amazon.com/es/blogs/security/how-to-enable-cross-account-access-to-the-aws-management-console/)

July 26, 2017, update: We recommend that you use [cross-account access by switching roles](https://aws.amazon.com/blogs/security/enable-a-new-feature-in-the-aws-management-console-cross-account-access/) in the AWS Management Console. Also see the related documentation: [Switching to a Role (AWS Management Console)](http://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles_use_switch-role-console.html).

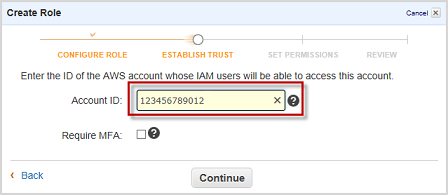
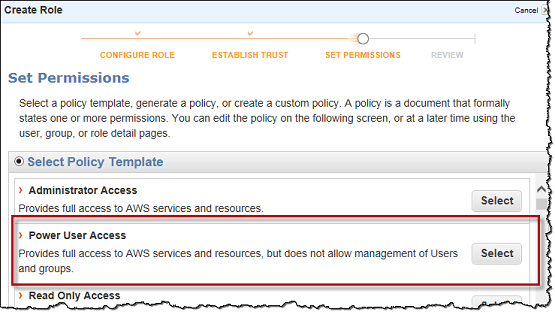
Last December we described how you can [delegate access to your AWS account using IAM roles](http://blogs.aws.amazon.com/security/post/TxC24FI9IDXTY1/Delegating-API-Access-to-AWS-Services-Using-IAM-Roles). Using IAM roles, you can take advantage of [cross-account access](http://docs.aws.amazon.com/IAM/latest/UserGuide/cross-acct-access.html) to give users access across AWS accounts when they need it. That’s great for API or CLI calls. But people have asked us whether they can also use cross-account roles for console access, not just API or CLI access—is it possible to let an IAM user sign in to the console to manage resources in *any* account that belongs to the organization? The answer is yes, and it’s actually pretty easy.

Imagine that you have two AWS accounts, which we’ll refer to as Prod and Dev. You want to give IAM users in the Dev account temporary and limited access to the Prod account via the console. In this post, we’ll show you how to do this using a short script written in Python.

We’ll begin by walking you quickly through the usual preliminaries for establishing cross-account access, namely creating a role in one account to allow access and granting permissions to users in a different account who should get access to the first account. Then we’ll show you the script and explain what it does.

**Task 1: Create an IAM role in the Prod account (the account that users want to sign into)**

To begin, you create a role in the Prod account that users from the Dev account can assume in order to get temporary security credentials.

1. Make sure you have the account ID for the Dev account.
2. Sign in to the Prod account as a user with administrator privileges.
3. In the IAM console, create a new role and name it CrossAccountSignin. Choose the wizard option for creating cross-account access between accounts that you own. For details, see [Creating a Role for Cross-Account Access](http://docs.aws.amazon.com/IAM/latest/UserGuide/create-role-users-assume.html).When you create the role, specify the account ID of the Dev account (the account where the users are defined).  
     
   And when you set permissions, choose the Power User Access policy template. This lets users from the Dev account work with all services in the Prod account *except* IAM. For example, they won’t be able to change the permissions for the role they’re assuming, and they won’t be able to create new users or roles.  
   

When you’re finished, take note of the role Amazon Resource Name (ARN), which will look like this, except it will contain the actual account ID for the Prod account: arn:aws:iam::*Prod-account-ID*:role/CrossAccountSignin.

**Task 2: Give users in the Dev account permission to assume the role in the Prod account**

The CrossAccountSignin role you created in the Prod account grants access to the Dev account, but the owner of the Dev account still needs to grant access to individual users in that account before the users can access the Prod account.

1. Sign in to the Dev account as a user with administrator privileges.
2. Edit the permissions for a user (or group of users) who are allowed to sign in to the Prod account and grant sts:AssumeRole permissions. As the resource for the action, specify the ARN of the CrossAccountSigninrole you created earlier. Here’s an example of a policy that you can attach to a user or group:
3. {
4. "Version": "2012-10-17",
5. "Statement": [{
6. "Effect": "Allow",
7. "Action": ["sts:AssumeRole"],
8. "Resource": "arn:aws:iam::*Prod-account-ID*:role/CrossAccountSignin"
9. }]

}

If the user or group already has a policy attached, you can attach this example as an additional policy.

**Task 3: Create a script to allow the user to sign into the Prod account console**

At this point, users in the Dev account who have the example policy attached can assume the CrossAccountSigninrole in the Prod account. To assume a role, a user (or a program) calls the AssumeRole API. This API returns a set of temporary security credentials that can be used to access the Prod account with the permissions specified in the CrossAccountSignin role.

As noted earlier, assuming a role is useful for API or CLI access. But we want to convert the AssumeRole call directly into a console sign-in. To do that, you can create a script that takes advantage of a feature in AWS known as the[federation endpoint](http://docs.aws.amazon.com/STS/latest/UsingSTS/STSMgmtConsole-manualURL.html) (*https://signin.aws.amazon.com/federation*). You can make a request to this endpoint and pass it temporary security credentials that you get from AssumeRole. The endpoint returns a sign-in token that you can then use to construct a console URL. This console URL lets a user sign in to the console without having to supply a username and password, because the URL contains a token that indicates that the user is already authenticated.

We’ve created an example script that shows how to do all this. Our script will roll up all the tasks that are required in order to implement this scenario:

1. Determine which role to assume. For purposes of our example, we’ll have the user pass the account ID and role name on the command line, which lets us construct a role ARN.
2. Make the [AssumeRole](http://docs.aws.amazon.com/STS/latest/APIReference/API_AssumeRole.html) call, passing the ARN of the role to assume.
3. Extract the resulting credentials.
4. Call the federation endpoint, passing the credentials in the format that the endpoint requires. The request looks something like this:https://signin.aws.amazon.com/federation?Action=getSigninToken&Session=*temporary-credentials*where Session contains the credentials formatted into a JSON block that’s also URL-encoded. Before encoding, the JSON block looks like this (wrapped here for readability):
5. {
6. "sessionId":"*temporary-access-key-ID*",
7. "sessionKey":"*temporary-secret-access-key*",
8. "sessionToken":"*security-token*"

}

1. Get the return value from the federation endpoint, which is a JSON block that contains a sign-in token.
2. Create a URL that includes the parameter Action=login and the sign-in token. This URL is good for 15 minutes from the time it was created. The URL will look something like this:https://signin.aws.amazon.com/federation?Action=login&Issuer=*your-domain*&Destination=*console-URL*&SigninToken=*signin-token*where:
   * Issuer can be null. This value is actually designed for [federation use cases](https://aws.amazon.com/iam/details/manage-federation/)—in those cases, it’s the domain of the federating organization, and if the session times out, this value is displayed as a link so the user can return to that domain to sign in again. In our case, there’s not really anyplace to return to if the session times out, so we’re just passing a null value for this parameter.
   * Destination is the URL-encoded URL of the console you want to present to the user. That can be the AWS Management Console ([*https://console.aws.amazon.com/*](https://console.aws.amazon.com/)) or a service-specific console (like *https://console.aws.amazon.com/ec2/v2/home* for EC2).
   * SigninToken is the token returned by the previous call to the federation endpoint.
3. Call the default browser and pass it the generated URL.

Important! Because the URL is already authenticated (via the token that it contains), you should treat it with as much care as you treat the actual credentials that you used to generate the URL. (That’s why it’s good for only 15 minutes at a time.)

**Our sign-in script**

To write our example script, we chose Python and the [AWS SDK for Python](http://boto.readthedocs.org/en/latest/), also known as *boto*. We chose Python because it’s cross-platform, so this code will run on Windows, a Mac, and Linux. We also believe that Python is relatively easy for programmers in any language to read and understand. If you don’t want to use Python, you can perform the same tasks using any of the [AWS SDKs](http://aws.amazon.com/tools/).

If you want to run this script, you’ll need to do the following:

1. [Download](https://www.python.org/downloads/) and install Python for your platform. We’re using Python 2.7 for this example.
2. Install boto, which is on GitHub. For details, see the [installation instructions](https://github.com/boto/boto#installation).
3. Install the Python [requests](https://pypi.python.org/pypi/requests) package, which is used in the example script to make a web request to the *https://signin.aws.amazon.com/federation* endpoint. A convenient way to install Python packages it to use [pip](https://pip.pypa.io/en/latest/installing.html), which gets packages from the Python package index site. You can then install requests by running pip install requests at the command line.

Here’s the Python script. After the code listing you’ll find a few notes about how we coded the script.

#!/usr/bin/env python

import requests # "pip install requests"

import sys, os, urllib, json, webbrowser

from boto.sts import STSConnection # AWS Python SDK--"pip install boto"

# Step 1: Prompt user for target account ID and name of role to assume

if len(sys.argv) == 3:

account\_id\_from\_user = sys.argv[1]

role\_name\_from\_user = sys.argv[2]

else:

print "ntUsage: ",

print os.path.basename(sys.argv[0]), # script name

print " <account\_id> <role\_name>"

exit(0)

# Create an ARN out of the information provided by the user.

role\_arn = "arn:aws:iam::" + account\_id\_from\_user + ":role/"

role\_arn += role\_name\_from\_user

# Step 2: Connect to AWS STS and then call AssumeRole. This returns

# temporary security credentials.

sts\_connection = STSConnection()

assumed\_role\_object = sts\_connection.assume\_role(

role\_arn=role\_arn,

role\_session\_name="AssumeRoleSession"

)

# Step 3: Format resulting temporary credentials into a JSON block using

# known field names.

access\_key = assumed\_role\_object.credentials.access\_key

session\_key = assumed\_role\_object.credentials.secret\_key

session\_token = assumed\_role\_object.credentials.session\_token

json\_temp\_credentials = '{'

json\_temp\_credentials += '"sessionId":"' + access\_key + '",'

json\_temp\_credentials += '"sessionKey":"' + session\_key + '",'

json\_temp\_credentials += '"sessionToken":"' + session\_token + '"'

json\_temp\_credentials += '}'

# Step 4. Make a request to the AWS federation endpoint to get a sign-in

# token, passing parameters in the query string. The call requires an

# Action parameter ('getSigninToken') and a Session parameter (the

# JSON string that contains the temporary credentials that have

# been URL-encoded).

request\_parameters = "?Action=getSigninToken"

request\_parameters += "&Session="

request\_parameters += urllib.quote\_plus(json\_temp\_credentials)

request\_url = "https://signin.aws.amazon.com/federation"

request\_url += request\_parameters

r = requests.get(request\_url)

# Step 5. Get the return value from the federation endpoint--a

# JSON document that has a single element named 'SigninToken'.

sign\_in\_token = json.loads(r.text)["SigninToken"]

# Step 6: Create the URL that will let users sign in to the console using

# the sign-in token. This URL must be used within 15 minutes of when the

# sign-in token was issued.

request\_parameters = "?Action=login"

request\_parameters += "&Issuer="

request\_parameters += "&Destination="

request\_parameters += urllib.quote\_plus("https://console.aws.amazon.com/")

request\_parameters += "&SigninToken=" + sign\_in\_token

request\_url = "https://signin.aws.amazon.com/federation"

request\_url += request\_parameters

# Step 7: Use the default browser to sign in to the console using the

# generated URL.

webbrowser.open(request\_url)

**Notes about the script**

The script calls AssumeRole using the following code. In the Python SDK, you make a connection to an AWS service and then call a method (here, assume\_role) in order to call that API.

sts\_connection = STSConnection()

assumed\_role\_object = sts\_connection.assume\_role(

role\_arn=role\_arn,

role\_session\_name="AssumeRoleSession"

)

The AssumeRole call requires a parameter for the session name. This value is used in two places. First, it becomes part of the name that identifies the user in the navigation bar of the console. Second, if AWS CloudTrail is enabled for the Prod account, the session name is included in the log record for the AssumeRole call.

To keep things short, the script doesn’t validate user input and doesn’t handle exceptions. If you run the script and pass an invalid account ID or role name, you’ll see the raw error returned by AWS. That’s also true if the script is run using the credentials of an IAM user who isn’t allowed to assume the role. Similarly, in the interests of keeping the code easy to read, we used simple concatenation to build up some strings instead of using a more efficient method. For example, the script creates a JSON block and the final URL using concatenation. We also broke up some lines to avoid wrapping code.

**Running the script**

After you’ve created the role in the Prod account and the user and permissions in the Dev account, you can try out the script. Calling AssumeRole (or the boto equivalent, assume\_role) requires an access key from an IAM user or the temporary security credentials obtained earlier. (You can’t call AssumeRole using the root-level access key for an AWS account.)

To help safeguard access keys, the AWS SDKs let you keep credentials in a configuration file or in environment variables instead of embedding them directly in code. Our code is relying on this automatic lookup of credentials. For details about how to set up credentials for boto, see [Boto Config](http://boto.readthedocs.org/en/latest/boto_config_tut.html) in the Python SDK documentation.

To run the script, copy the code listing from above and save it as a *.py* file—for example, as *ConsoleSignin.py*. If you’re already signed in using your default browser, sign out. Then run the script using a command like this:

python ConsoleSignin.py 123456789012 CrossAccountSignin

If you’re using Linux, you might need to make the file executable by running chmod +x ConsoleSignin.py, and you might need to include path information, such as ./python.

You pass two values on the command line. In the example, *123456789012* is the account ID for the Prod account, and CrossAccountSignin is the name of the role in the Prod account that you want to assume.

If all goes well, the browser will open to the AWS Management Console. In the navigation bar, your identity will be listed as CrossAccountSigning/AssumeRoleSession, which is the combination of the role you’ve assumed and the session name.

Screenshot showing the identity that is the combination of the role you've assumed and the session name

If you do see an error, examine the error listing to determine what happened. The most typical error is that you’re using credentials for an IAM user who doesn’t have permissions to assume the role in the Prod account. Don’t forget that if you previously were signed in to the console, you should sign out before you run this script.

You can test that the role permissions are working by trying to go to the IAM console. If you used the Power User Accesspolicy template, the IAM console will display a bunch of “is not authorized” errors, which is exactly what we intended when we used that policy template.

**Beyond this script**

We wrote this script to show one possible way to perform cross-account console sign-in. The script illustrates the tasks you need to perform: how to assume a cross-account role, how to call the federation endpoint feature to get a sign-in token, and how to create a URL for cross-account console access. This script works on the command line, but you can see how the technique illustrated here could be built into a desktop-based or web-based application, and we encourage you to expand the ideas presented here for your own requirements.

As always, if you have questions about anything you read in our blog, please post a note to the [IAM forum](https://forums.aws.amazon.com/forum.jspa?forumID=76&start=0).