Strategies for Managing Access to AWS Resources in AWS Marketplace

July 2016

Notices

Contents

Abstract

This paper discusses how applications in AWS Marketplace that require access to AWS resources can use AWS Identity and Access Management (IAM) roles for authentication, to help protect customers from potential security vulnerabilities.

Overview

Applications in AWS Marketplace that require access to Amazon Web Services (AWS) resources must follow security best practices when accessing AWS to help protect customers from potential security vulnerabilities. Typically, application authors will use a combination of access and secret keys to authenticate against AWS resources. However, for AWS Marketplace,[1] we require application authors to use AWS Identity and Access Management (IAM)[2] roles and do not permit the use of access or secret keys.

This requirement affects two types of applications: applications that interact with AWS resources to operate, and applications that interact with AWS resources on behalf of specific users, either in the same or in different AWS accounts.

When an application requires access to AWS resources to operate, temporary credentials can be obtained by using IAM roles for Amazon Elastic Compute Cloud (Amazon EC2) instances.[3] Applications can then interact with AWS resources without needing to store, secure, and manage a user's access keys.

When an application needs to access AWS resources on behalf of different users, either in the same or in different AWS accounts, the same technique can be applied. IAM roles can be used to access both the resources required by the application, and the resources the application may access on behalf of a user.

By using IAM roles instead of IAM users for both application-specific and user-specific access, you can remove the need for customers to distribute and manage access keys. The following sections explain how you can adopt this strategy.

Accessing Application-Specific Resources

When an application needs to interact with AWS resources, access should be provided by using IAM roles and not IAM users. For example, if an application needs to access an Amazon DynamoDB[4] database and an Amazon Simple Storage Service (S3)[5] bucket, access to these resources are not user-specific.

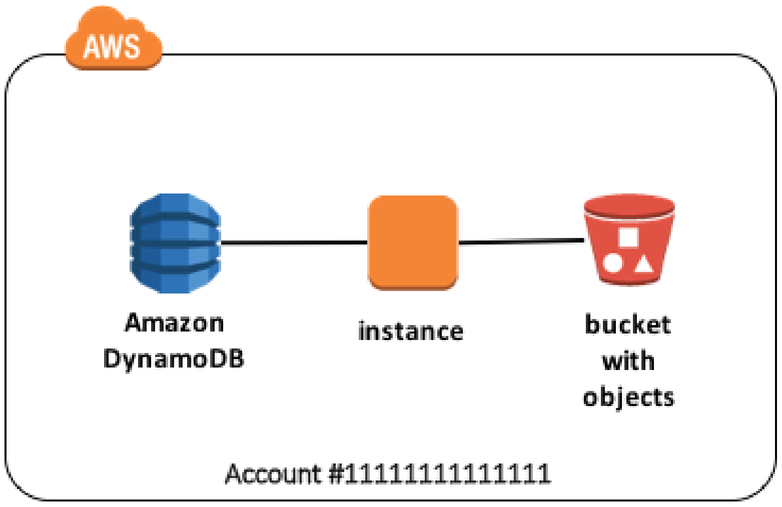

Figure 1: Sample architecture for accessing application-specific resources

The EC2 Instance Role

The EC2 instance is started with an instance role attached. This role has a policy that grants access to the DynamoDB database and the S3 bucket within the same account.

When making API calls to Amazon S3, your application must retrieve the temporary credentials from the IAM role and use those credentials. You can retrieve these credentials from the instance metadata (http://169.254.169.254/latest/meta-data/iam/security-credentials/rolename). If you are using an AWS SDK, the AWS Command Line Interface (AWS CLI),[6] or AWS Tools for Windows PowerShell,[7] these credentials will be obtained automatically.

Using roles in this way has several benefits. Because role credentials are temporary and rotated automatically, you don't have to manage credentials, and you don't have to worry about long-term security risks.

To create and use an IAM instance role:

1. Create a new instance role.
2. Add a trust relationship that allows ec2.amazonaws.com to assume the role.

3. Create a new policy that specifies the permissions required.
4. Add the new policy to the new instance role.
5. Create a new EC2 instance that specifies the IAM role.
6. Build your app by using one of the AWS SDKs. Do not specify credentials when calling methods, because temporary credentials will be automatically added by the SDK.

For more detailed instructions, see IAM Roles for Amazon EC2 in the IAM documentation.[8]

Note You can also configure launch settings used by Auto Scaling groups to use IAM roles.

In our example, we'll create the instance role with the following trust relationship:

```
{
  "Version": "2008-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "ec2.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

Add the **AmazonDynamoDBFullAccess** and **AmazonS3FullAccess** policies to the IAM role, and then create the EC2 instance by specifying the role.

Accessing Resources on Behalf of Users

To illustrate the scenario of accessing AWS resources on behalf of specific users, consider an application that processes images stored in S3 buckets on behalf of a user. The application itself might use services such as DynamoDB for storing configuration and job status. The following diagram shows the architecture.

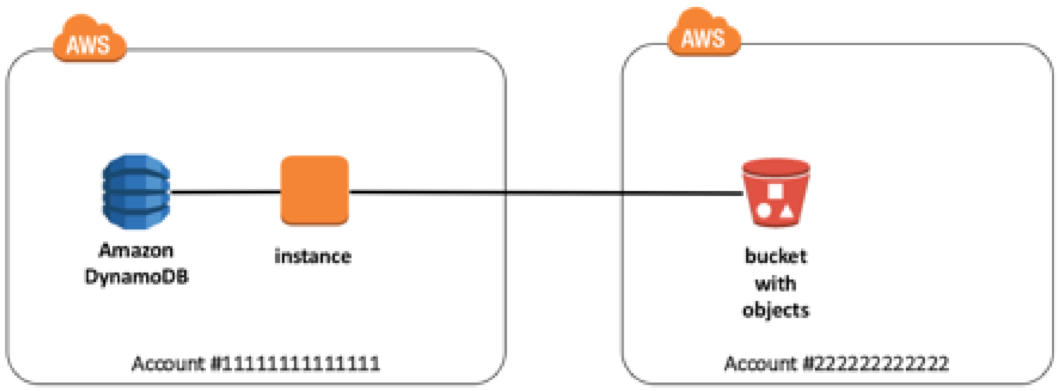

Figure 2: Sample architecture for accessing AWS resources on behalf of users

In this scenario, the EC2 instance hosting the application would use an instance profile that gives specific permissions to DynamoDB.

When accessing Amazon S3 resources on behalf of the user, the application would switch to a different IAM role: a role that was set up by the user with specific permission to access the S3 buckets. This method would allow an application to access resources on behalf of different users, without the need to store credentials. Users would still need to create IAM policies and IAM roles, but this is no different from creating IAM users and IAM roles for the same reason.

There are two IAM roles in play:

- **EC2 instance role** (application role) – This is the role the application uses to obtain temporary credentials to access application-specific resources, such as the DynamoDB database.
- **Account access roles** (user roles) – These are the roles the application uses to obtain temporary credentials to access resources for specific users of the application.

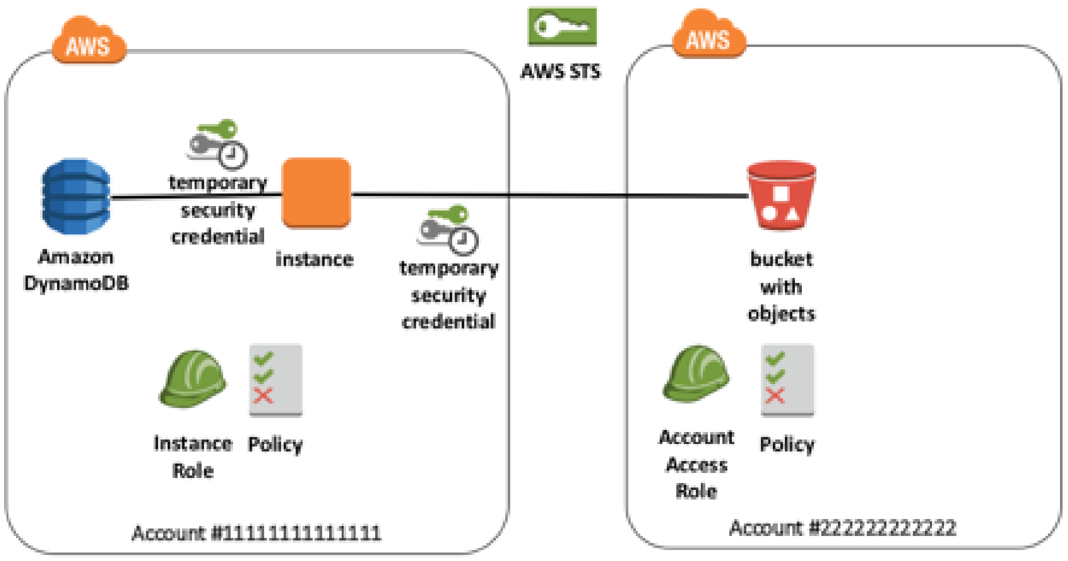

Figure 3: Roles and policies

The EC2 Instance Role

The EC2 instance role would be configured in the same way as in the first scenario.

The Account Access Role

Since the application can also access S3 buckets and objects from other AWS accounts, it is tempting to maintain a list of credentials to access these AWS resources; however, the same technique of using roles and temporary credentials is preferred. This strategy again removes the need for the application to store anything but benign information, or handle key rotation scenarios.

Using roles across accounts is no more difficult to set up than creating users and assigning policies, but it requires a few extra steps:

1. In the target account (the account that contains the AWS resources):
   a. Create a new IAM role.
   b. Add a trust relationship that specifies the root of the application hosting account as the principal. Include a condition that specifies an external ID.

   c. Create a new policy that specifies the permissions required, and attach it to the role.

2. In the application hosting account (the account where the application is hosted):

   a. Create a new policy that specifies that the `sts:AssumeRole` action is allowed to the role defined in the target account.

   b. Attach the new policy to the instance role.

In the target account, we can create a role named `my-user-role` with the following trust relationship:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": [
          "arn:aws:iam::111111111111:root"
        ]
      },
      "Action": "sts:AssumeRole",
      "Condition": {
        "StringEquals": {
          "sts:ExternalId": "myapp"
        }
      }
    }
  ]
}
```

Note that the account number 111111111111 is used in the principal Amazon Resource Name (ARN) to ensure that only IAM users and roles from that account can assume this role. Furthermore, the inclusion of an `sts:ExternalId` condition means that the caller also needs this information to complete the `AssumeRole` function. See the code sample later in this paper for information on how this condition is used.

The permissions added to the role permit access to specific S3 buckets. It is good practice to be explicit in permissions rather than using wildcards. The following is an example of the permissions added:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "s3:ListBucket"
            ],
            "Resource": [
                "arn:aws:s3:::myBucket1",
                "arn:aws:s3:::myBucket2"
            ]
        }
    ]
}
```

Back in the application hosting account, we need to add a new permission to the role to allow it to assume the role in the target account:

```
{
    "Version": "2012-10-17",
    "Statement": {
        "Effect": "Allow",
        "Action": "sts:AssumeRole",
        "Resource": "arn:aws:iam::222222222222:role/my-
user-role"
    }
}
```

You can use a wildcard in the application hosting account, since the permissions need to be explicitly defined in the target account. This also allows you to access roles across multiple AWS accounts.

```
{
    "Version": "2012-10-17",
    "Statement": {
        "Effect": "Allow",
        "Action": "sts:AssumeRole",
        "Resource": "arn:aws:iam::*:role/my-user-role"
    }
}
```

Switching Roles

In the application, we do not need to code anything special to use the instance role and the permissions that gives us. However, to access the S3 buckets in the other AWS accounts, we will need to assume the new role and use the temporary credentials for that role in our SDK calls. The following code snippet shows a Node.js example:

```
var accounted = '222222222222';
var rolename = 'my-user-role';
var externalId = 'myapp';
var sts = new AWS.STS();
var stsparams = {
    RoleArn: 'arn:aws:iam::'+ accountid + ':role/' + rolename,
    RoleSessionName: 'myappsession',
    ExternalId: externalId,
    DurationSeconds: 3600
};

AWS.config.credentials = new AWS.EC2MetadataCredentials();
var tempCredentials = new AWS.TemporaryCredentials(stsparams);
var options = {
    credentials: tempCredentials
}
var s3 = new AWS.S3(options);
```

Lines 5–10 define the parameters (`stsparams`) for obtaining the temporary credentials on line 13. We build the `RoleArn` from parameters defined in lines 1 and 2, along with the `externalId` in line 3.

Once we have the temporary credentials, we use these in line 17 to access the S3 resource.

AWS Marketplace Considerations

There are a few things to consider when using IAM roles for AWS Marketplace.

Using External IDs

It is important not to just rely on the role name; you must specify an external ID to be used by the application. Furthermore, you should allow the customer deploying your application to define the external ID value. You should use a different external ID for each AWS account to limit exposure.

Using Wildcards for IAM Roles

Since users will be supplying roles in different accounts, you can use wildcards to designate target accounts in the application hosting account. You should use a well-known role name, but you can substitute a wildcard for the account number.

The following example is a good use of a wildcard:

```
arn:aws:iam::*:role/my-user-role
```

The following example is not an acceptable use of a wildcard:

```
arn:aws:iam::*
```

Great Documentation

Customers need to create IAM roles and polices in the AWS accounts they want to access, so you should provide explicit documentation to walk customers through creating the correct roles and policies.

Summary

Applications in AWS Marketplace that require access to AWS resources must implement authentication using IAM roles, as discussed in this guide. This helps reduce the potential vulnerabilities within a customer's AWS account by providing access only to temporary credentials.

Contributors

The following individuals and organizations contributed to this document:

- David Aiken, partner solutions architect, AWS Marketplace

Notes

1 https://aws.amazon.com/marketplace/

2 https://aws.amazon.com/iam/

3 https://aws.amazon.com/ec2/

4 https://aws.amazon.com/dynamodb/

5 https://aws.amazon.com/s3/

6 https://aws.amazon.com/cli/

7 https://aws.amazon.com/powershell/

8 https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/iam-roles-for-amazon-ec2.html