include::attributes.txt[]

// Attributes

[.topic] [#define-iam-l2] = Define permissions for L2 constructs with the {aws} CDK :info\_titleabbrev: Define permissions for L2 constructs :keywords: {aws} CDK, {aws} Construct Library, {aws} CloudFormation, IAM, Permissions, Roles, Policies,

== [abstract]

## Define {aws} Identity and Access Management (IAM) roles and policies for L2 constructs when using the {aws} Cloud Development Kit ({aws} CDK).

// Content start

Define {aws} Identity and Access Management (IAM) roles and policies for L2 constructs when using the {aws} Cloud Development Kit ({aws} CDK).

[#define-iam-l2-grant] == Use grant methods to define permissions

When you define your infrastructure using L2 constructs from the {aws} Construct Library, you can use the provided *grant methods* to specify the permissions your resources will require. The {aws} CDK will automatically create the IAM roles needed for all {aws} resources that require them.

The following is an example that defines permissions between an {aws} Lambda function and Amazon Simple Storage Service (Amazon S3) bucket. Here, the grantRead method of the Bucket L2 construct is used to define these permissions:

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — import \* as cdk from ‘aws-cdk-lib’; import { Construct } from ‘constructs’; import \* as s3 from ‘aws-cdk-lib/aws-s3’; import \* as lambda from ‘aws-cdk-lib/aws-lambda’; import \* as kms from ‘aws-cdk-lib/aws-kms’;

export class CdkDemoStack extends cdk.Stack { constructor(scope: Construct, id: string, props?: cdk.StackProps) { super(scope, id, props);

…. const key = new kms.Key(this, ‘BucketKey’); const bucket = new s3.Bucket(this, ‘Bucket’, { encryptionKey: key, }); const handler = new lambda.Function(this, ‘Handler’, { runtime: lambda.Runtime.NODEJS\_20\_X, handler: ‘index.handler’, code: lambda.Code.fromAsset(‘lambda’), });

// Define permissions between function and S3 bucket using grantRead method bucket.grantRead(handler); ….

} } —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const { Stack, Duration } = require(‘aws-cdk-lib’); const s3 = require(‘aws-cdk-lib/aws-s3’); const lambda = require(‘aws-cdk-lib/aws-lambda’); const kms = require(‘aws-cdk-lib/aws-kms’);

class CdkDemoStack extends Stack {

constructor(scope, id, props) { super(scope, id, props);

…. const key = new kms.Key(this, ‘BucketKey’); const bucket = new s3.Bucket(this, ‘Bucket’, { encryptionKey: key, }); const handler = new lambda.Function(this, ‘Handler’, { runtime: lambda.Runtime.NODEJS\_20\_X, handler: ‘index.handler’, code: lambda.Code.fromAsset(‘lambda’), });

// Define permissions between function and S3 bucket using grantRead method bucket.grantRead(handler); ….

} }

== // …

Python:: + [source,python,subs=“verbatim,attributes”] — from aws\_cdk import ( Stack, aws\_s3 as s3, aws\_lambda as \_lambda, aws\_kms as kms, ) from constructs import Construct

class CdkDemoStack(Stack):

…. def **init**(self, scope: Construct, construct\_id: str, \*\*kwargs) -> None: super().\_\_init\_\_(scope, construct\_id, \*\*kwargs)

key = kms.Key(self, 'BucketKey')  
bucket = s3.Bucket(self, 'Bucket')  
handler = \_lambda.Function(  
 self,  
 'Handler',  
 runtime = \_lambda.Runtime.NODEJS\_20\_X,  
 handler = 'index.handler',  
 code = \_lambda.Code.from\_asset('lambda'),  
)  
  
# Define permissions between function and S3 bucket using grantRead method  
bucket.grantRead(handler) ----

….

Java:: + [source,java,subs=“verbatim,attributes”] — package com.myorg;

import software.amazon.awscdk.core.App; import software.amazon.awscdk.core.Stack; import software.amazon.awscdk.core.StackProps; import software.amazon.awscdk.services.kms.Key; import software.amazon.awscdk.services.kms.KeyProps; import software.amazon.awscdk.services.s3.Bucket; import software.amazon.awscdk.services.s3.BucketProps; import software.amazon.awscdk.services.lambda.Function; import software.amazon.awscdk.services.lambda.FunctionProps; import software.amazon.awscdk.services.lambda.Runtime; import software.amazon.awscdk.services.lambda.Code; import software.constructs.Construct;

public class CdkDemoStack extends Stack { public CdkDemoStack(final Construct scope, final String id) { this(scope, id, null); }

…. public CdkDemoStack(final Construct scope, final String id, final StackProps props) { super(scope, id, props);

Key key = new Key(this, "BucketKey", KeyProps.builder().build());  
  
Bucket bucket = new Bucket(this, "Bucket", BucketProps.builder()  
 .encryptionKey(key)  
 .build());  
  
Function handler = new Function(this, "Handler", FunctionProps.builder()  
 .runtime(Runtime.NODEJS\_20\_X)  
 .handler("index.handler")  
 .code(Code.fromAsset("lambda"))  
 .build());  
  
// Define permissions between function and S3 bucket using grantRead method  
bucket.grantRead(handler);

}

public static void main(final String[] args) { App app = new App();

new CdkDemoStack(app, "CdkDemoStack");  
  
app.synth();

} } —- ….

C#:: + [source,csharp,subs=“verbatim,attributes”] — using Amazon.CDK; using Amazon.CDK.{aws}.KMS; using Amazon.CDK.{aws}.S3; using Amazon.CDK.{aws}.Lambda;

namespace CdkDemo { public class CdkDemoStack : Stack { internal CdkDemoStack(Construct scope, string id, IStackProps props = null) : base(scope, id, props) { var key = new Key(this, “BucketKey”);

…. var bucket = new Bucket(this, “Bucket”, new BucketProps { EncryptionKey = key });

var handler = new Function(this, "Handler", new FunctionProps  
 {  
 Runtime = Runtime.NODEJS\_20\_X,  
 Handler = "index.handler",  
 Code = Code.FromAsset("lambda")  
 });  
  
 // Define permissions between function and S3 bucket using grantRead method  
 bucket.GrantRead(handler);  
}

} } —- ….

Go:: + [source,go,subs=“verbatim,attributes”] — package main import ( “github.com/aws/aws-cdk-go/awscdk/v2” “github.com/aws/aws-cdk-go/awscdk/v2/awskms” “github.com/aws/aws-cdk-go/awscdk/v2/awss3” “github.com/aws/aws-cdk-go/awscdk/v2/awslambda” “github.com/aws/constructs-go/constructs/v10” “github.com/aws/jsii-runtime-go” ) // … func NewCdkDemoStack(scope constructs.Construct, id string, props \*CdkDemoStackProps) awscdk.Stack { stack := awscdk.NewStack(scope, &id, &props.StackProps) key := awskms.NewKey(stack, jsii.String(“BucketKey”), nil) bucket := awss3.NewBucket(stack, jsii.String(“Bucket”), &awss3.BucketProps{ EncryptionKey: key, }) handler := awslambda.NewFunction(stack, jsii.String(“Handler”), &awslambda.FunctionProps{ Runtime: awslambda.Runtime\_NODEJS\_20\_X(), Handler: jsii.String(“index.handler”), Code: awslambda.Code\_FromAsset(jsii.String(“lambda”), &awss3assets.AssetOptions{}), }) bucket.GrantRead(handler) return stack } // … — ====

When you use grant methods of L2 constructs to define permissions between resources, the {aws} CDK will create roles with least privilege policies based on the method you specify. As a security best practice, we recommend that you use the method that applies only the permissions that you require. For example, if you only need to grant permissions for a Lambda function to read from an Amazon S3 bucket, use the grantRead method instead of grantReadWrite.

For each method that you use, the CDK creates a unique IAM role for the specified resources. If necessary, you can also directly modify the policy that will be attached to the role. The following is an example:

==== [role=“tablist”] TypeScript:: + [source,javascript,subs=“verbatim,attributes”] — import { aws\_iam as iam } from ‘aws-cdk-lib’;

handler.addToRolePolicy(new iam.PolicyStatement({ actions: [‘s3:GetObject’, ’s3:List\_\_’], resources: [ bucket.bucketArn, bucket.arnForObjects(’\_\_’), ] })); —

JavaScript:: + [source,javascript,subs=“verbatim,attributes”] — const iam = require(‘aws-cdk-lib/aws-iam’);

handler.addToRolePolicy(new iam.PolicyStatement({ actions: [‘s3:GetObject’, ’s3:List\_\_’], resources: [ bucket.bucketArn, bucket.arnForObjects(’\_\_’), ] })); —

Python:: + [source,python,subs=“verbatim,attributes”] — from aws\_cdk import aws\_iam as iam

handler.add\_to\_role\_policy(iam.PolicyStatement( actions=[‘s3:GetObject’, ’s3:List\_\_’], resources=[ bucket.bucket\_arn, bucket.arn\_for\_objects(’\_\_’), ] )) —

Java:: + [source,java,subs=“verbatim,attributes”] — import software.amazon.awscdk.services.iam.PolicyStatement; import software.amazon.awscdk.services.iam.PolicyStatementProps;

handler.addToRolePolicy(PolicyStatement.Builder.create() .actions(Arrays.asList(“s3:GetObject”, “s3:List\_\_“)) .resources(Arrays.asList( bucket.getBucketArn(), bucket.arnForObjects(”\_\_“) )) .build()); —

C#:: + [source,csharp,subs=“verbatim,attributes”] — using Amazon.CDK.{aws}.IAM; using Amazon.CDK.{aws}.S3; using Amazon.CDK.{aws}.Lambda;

handler.AddToRolePolicy(new PolicyStatement(new PolicyStatementProps { Actions = new[] { “s3:GetObject”, “s3:List\_\_” }, Resources = new[] { bucket.BucketArn, bucket.ArnForObjects(“\_\_“) } })); —

Go:: + [source,go,subs=“verbatim,attributes”] — package main

import ( // … “github.com/aws/aws-cdk-go/awscdk/v2/awsiam” // … )

// …

func NewCdkDemoStack(scope constructs.Construct, id string, props \*CdkDemoStackProps) awscdk.Stack { // …

…. handler.AddToRolePolicy(awsiam.NewPolicyStatement(&awsiam.PolicyStatementProps{ Actions: jsii.Strings(“s3:GetObject”, “s3:List*”), Resources: jsii.Strings(bucket.BucketArn(), bucket.ArnForObjects(”*”)), }))

// … } —- ==== ….

However, we recommend that you use the grant methods when available.

[#define-iam-l2-manual] == Manually create and use IAM roles

If you prefer not to use the CDK grant methods to create and manage permissions, you must manually create and configure them. You can create IAM roles using the {aws} Management Console, {aws} CLI, or {aws} SDKs. Then, you can pass them into your CDK application manually or use the *role customization* feature.

[#define-iam-l2-manual-role] === Reference and manage all roles manually

Constructs that require a role have an optional role property that you can use to pass in a role object.

*To reference a role manually*:: + . Use Role.fromRoleName() to reference your pre-existing role. The following is an example: + [source,javascript,subs=“verbatim,attributes”] — const existingRole = Role.fromRoleName(stack, ‘Role’, ‘my-pre-existing-role’, { mutable: false // Prevent CDK from attempting to add policies to this role }) — + . Pass the pre-existing role when defining your resource. The following is an example: + [source,javascript,subs=“verbatim,attributes”] — const handler = new lambda.Function(stack, ‘Handler’, { runtime: lambda.Runtime.NODEJS\_20\_XZ, handler: ‘index.handler’, code: lambda.Code.fromAsset(path.join(\_\_dirname, ‘lambda-handler’)), // Pass in pre-existing role role: existingRole, }); —

[#define-iam-l2-manual-customization] === Use the role customization feature

The {aws} CDK *role customization* feature generates a report of roles and policies in your CDK app. You can use this feature to generate a report. Then you can substitute pre-created roles for them.

*To use the role customization feature*:: + . Add Role.customizeRoles() somewhere towards the top of your CDK application. The following is an example: + [source,javascript,subs=“verbatim,attributes”] — const stack = new Stack(app, ‘LambdaStack’);

// Add this to use the role customization feature iam.Role.customizeRoles(stack);

// Define your resources using L2 constructs const key = new kms.Key(stack, ‘BucketKey’); const bucket = new s3.Bucket(stack, ‘Bucket’, { encryptionKey: key, }); const handler = new lambda.Function(stack, ‘Handler’, { runtime: lambda.Runtime.NODEJS\_16\_X, handler: ‘index.handler’, code: lambda.Code.fromAsset(path.join(\_\_dirname, ‘lambda-handler’)), });

// The grantRead() is still important. Even though it actually doesn’t mutate // any policies, it indicates the need for them. bucket.grantRead(handler); — + . When you synthesize your application, the CDK will throw an error, indicating that you need to provide the pre-created role name to Role.customizeRoles(). The following is an example of the generated report: + —++++++(LambdaStack/Handler/ServiceRole) AssumeRole Policy: [ { “Action”: “sts:AssumeRole”, “Effect”: “Allow”, “Principal”: { “Service”: “lambda.amazonaws.com” } } ] Managed Policy ARNs: [ “arn:(PARTITION):iam::aws:policy/service-role/AWSLambdaBasicExecutionRole” ] Managed Policies Statements: NONE Identity Policy Statements: [ { “Action”: [ “s3:GetObject*”, ”s3:GetBucket*”, “s3:List\*” ], “Effect”: “Allow”, “Resource”: [ “(LambdaStack/Bucket/Resource.Arn)”, “(LambdaStack/Bucket/Resource.Arn)/\*” ] } ] —- + . Once the role is created, you can pass it into your application for the resource that it applies to. For example, if the name of the role created for LambdaStack/Handler/ServiceRole is lambda-service-role, you would update your CDK app as follows: + [source,javascript,subs=“verbatim,attributes”] —- const stack = new Stack(app, ‘LambdaStack’); // Add this to pass in the role iam.Role.customizeRoles(stack, { usePrecreatedRoles: { ‘LambdaStack/Handler/ServiceRole’: ‘lambda-service-role’, }, }); —- + The CDK will now use the pre-created role name anywhere that the role is referenced in the CDK application. It will also continue to generate the report so that any future policy changes can be referenced. + You will notice that the reference to the Amazon S3 bucket ARN in the report is rendered as (LambdaStack/Bucket/Resource.Arn) instead of the actual ARN of the bucket. This is because the bucket ARN is a deploy time value that is not known at synthesis (the bucket hasn't been created yet). This is another example of why we recommend allowing CDK to manage IAM roles and permissions by using the providedgrantmethods. In order to create the role with the initial policy, the admin will have to create the policy with broader permissions (for example,arn:aws:s3:::\*`).++++++