Delegating Access To AWS Resources Using IAM and STS (Switch Roles)

Amazon secure most of their services using IAM (Identity Access Management) which is a policy based access mechanism which relies on users, groups, roles, policies etc — by Craig Godden-Payne@beardy.digital

Craig Godden-Payne
Mar 12 · 6 min read ★

Photo by Jose Fontano on Unsplash

AWS take security very seriously. A breach in AWS security could have the impact of a lot of reputational damage, so security is of upmost importance.

Amazon secure most of their services using IAM (Identity Access Management) which is a policy based access mechanism which relies on users, groups, roles, policies etc.

There are four main types of identity and access management, and use an RBAC type model.

Photo by Philipp Katzenberger on Unsplash

What makes up IAM?

- Users can have permissions directly, or belong to a group, where the group stipulates the permissions.
- Groups are used to group a bunch of permissions, for easier assigning to a user or multiple users.
- Roles can be applied to User or an Instance. Roles are meant to be “assumed” rather than granted like you would in a group (i.e. its a temporary inflation of privileges).

- Policies are a collection of permissions that can be applied to users, groups or policies.

Photo by Marc-Olivier Jodoin on Unsplash

How are policies built up?

The policy language you use when creating permissions using IAMs has two elements. Specification (defining access policies) and Enforcement (evaluating policies). Policys are written in JSON, and each statement should contain “PARC” (Principle, Action, Resource, Condition)

- Principal

A principal is an entity which is allowed or denied access to a resource. This can be indicated in different ways e.g.

```
– Anonymous Users
"Principal":"AWS":"*.*"

– Specific Accounts
"Principal":{"AWS":"arn:aws:iam::123456789012:root"}

– Individual IAM User
"Principal":{"AWS":"arn:aws:iam::123456789012:user/name"}

– Federated User
"Principal":{"Federated":"www.amazon.com"}
```

```
- Specific Role
"Principal":{"AWS":"arn:aws:iam::123456789012:role/rolename"}

- Specific Service
"Principal":{"Service":"ec2.amazonaws.com"}
```

- Actions

An action describes the type of access that should be allowed or denied. It must include either an Action or NotAction. e.g.

```
- EC2 Action
"Action":"ec2:StartInstances"

- IAM Action
"Action":"iam:ChangePassword"

- S3 Action
"Action":"s3:GetObject"

- Multiple Action
"Action":["sqs:SendMessage", "sqs:RecieveMessage"]

- Wildcard Action
"Action":"sns:*"

- Wildcard Action (would deny all sns)
"NotAction":"sns:*"

- Using Multiples (this example will allow everything, except for
IAM)
{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Allow",
    "Action": "*",
    "Resource": "*"
  },
  {
    "Effect": "Deny",
    "Action": "iam:*",
    "Resource": "*"
  }]
}
```

- Resource

A resource describes the object or objects that are being requested. It must include either a Resource or NotResource. e.g.

```
- S3 Bucket
"Resource": "arn:aws:s3:::/bucketname/*"

- SQS Queue
"Resource": "arn:aws:sqs:eu-west-2:123456789012:queuename"

- Multiple Dynamo Tables
"Resource": ["arn:aws:dynamodb:eu-west-
2:123456789012:table/tablename","arn:aws:dynamodb:eu-west-
2:123456789012:table/tablename2"]

- EC2 instances for an account in a region
"Resource": "arn:aws:ec2:eu-west-2:123456789012:instance/*"
```

- Conditions

If you add a condition, it must evaluate to true, for the policy to evaluate as true. A condition can contain multiple conditions, and the keys can contain multiple values. e.g.

```
"Condition": {
  "DateGreaterThan": { "aws:CurrentTime": "2018-05-01T00:00:00Z" },
  "DateLessThan": { "aws:CurrentTime": "2018-05-22T00:00:00Z" }
  "IpAddress": { "aws:SourceIp": ["192.0.1.0/24", "192.0.2.0/24"] }
}
```

Photo by chris panas on Unsplash

Example 1

Here is an example of allowing permission for a user to stop, start and terminate any EC2 instance within the account.

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Allow",
    "Action": ["ec2:TerminateInstances"],
    "Resource": "arn:aws:ec2:eu-west-2:123456789012:instance/*"
  }]
}
```

Example 2

Here is an example of creating a “limited” IAM administrator.

The first statement shows allowing creating users, managing keys and setting passwords. The second statement shows this is limited to only attaching on the policy AmazonDynamoDBFullAccess

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Sid": "ManageUsersPermissions",
    "Effect": "Allow",
    "Action":
["iam:ChangePassword","iam:CreateAccessKey","iam:CreateLoginProfile"
,"iam:CreateUser",

"iam:DeleteAccessKey","iam:DeleteLoginProfile","iam:DeleteUser","iam
:UpdateAccessKey",
                "iam:ListAttachedUserPolicies","iam:ListPolicies"],
    "Resource": "*"
  },
  {
    "Sid": "LimitedAttachmentPermissions",
    "Effect": "Allow",
    "Action": ["iam:AttachUserPolicy", "iam:DetachUserPolicy"],
    "Resource": "*",
    "Condition": {
      "ArnEquals": {
        "iam:PolicyArn": [
```

```
            "arn:aws:iam::aws:policy/AmazonDynamoDBFullAccess"
          ]
        }
      }
    }]
  }
```

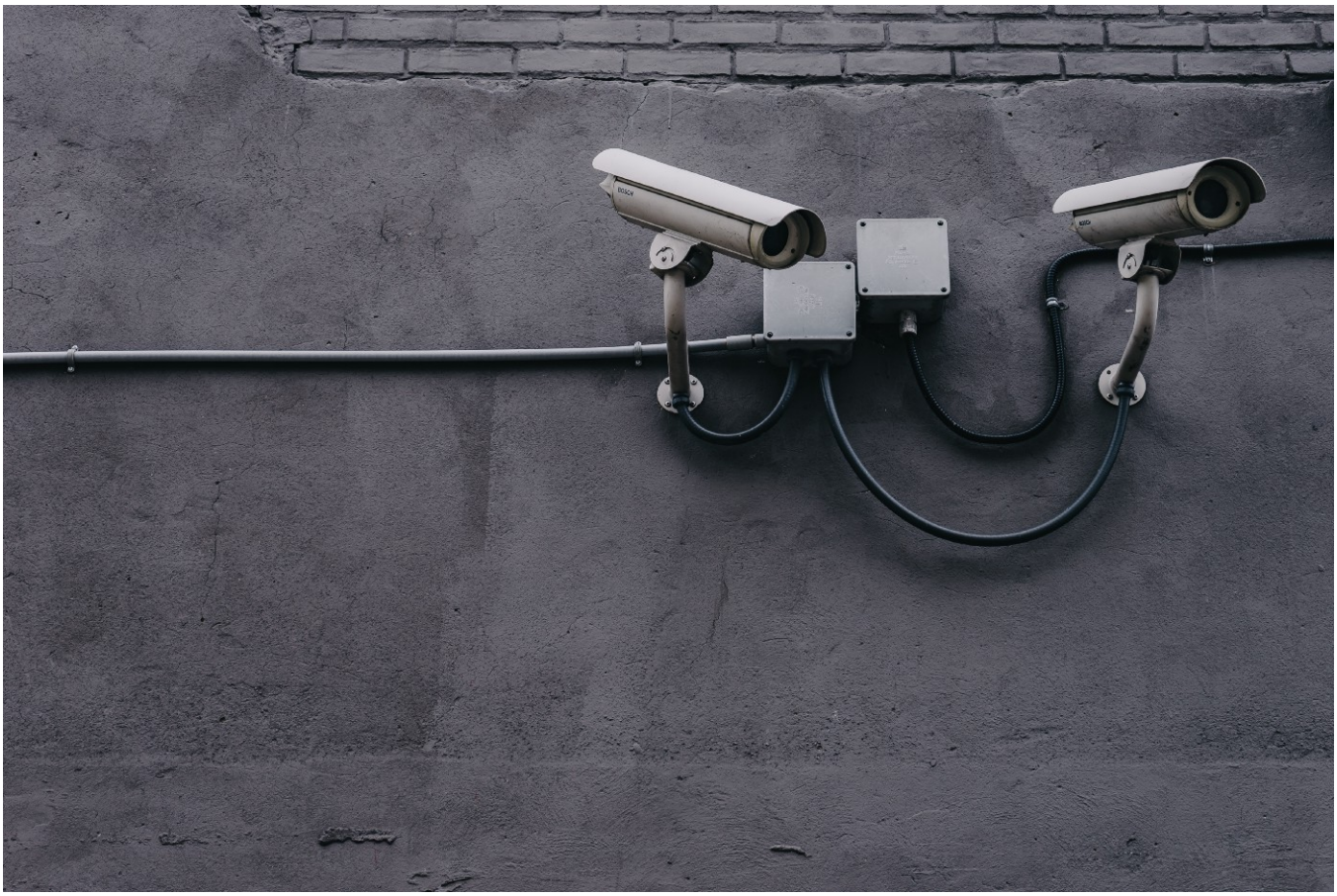


Photo by Scott Webb on Unsplash

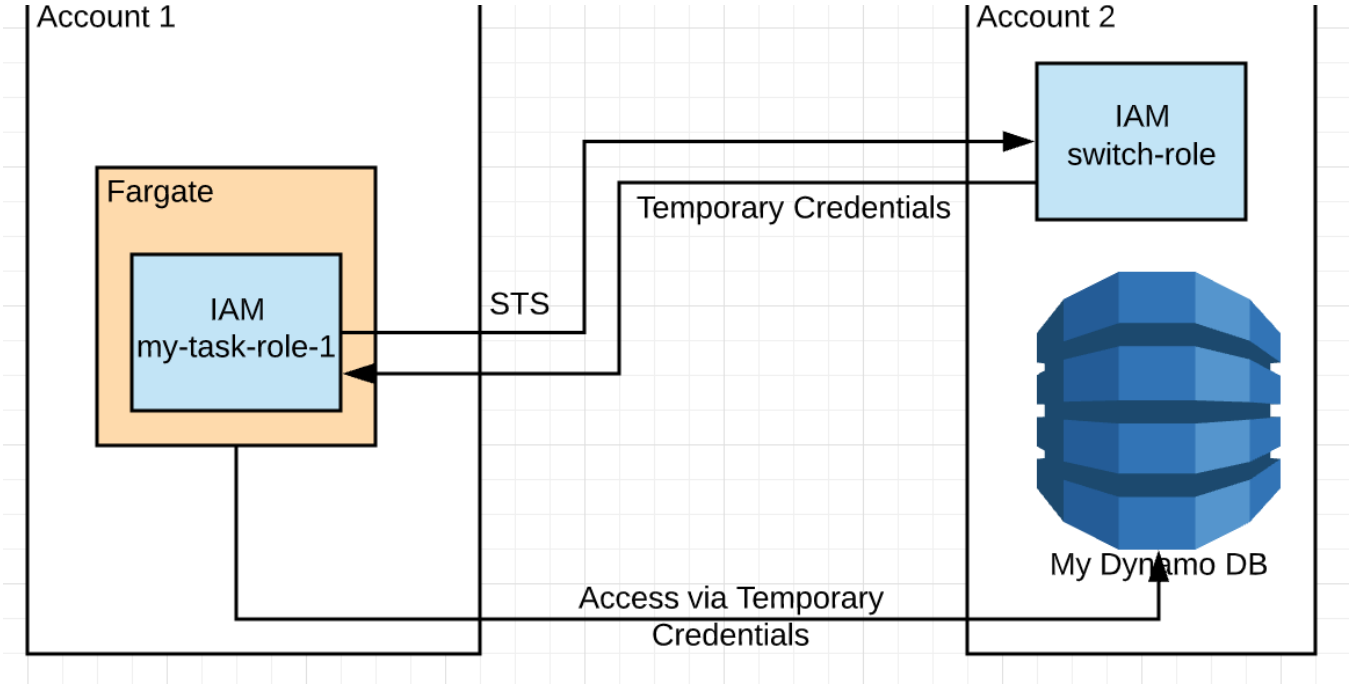
So now we know what IAM can do, how can I now delegate access to an AWS resource using STS?

Lets explain this in more detail.

Say you have a Role (in account 1) which you want to be able to access a DynamoDB table (in account 2)

You may not want to, or can't give the Role (account 1) access to the DynamoDB, instead you may choose for a role within account 2, to have access to the DynamoDB resource, and then allow the Role (account 1) to assume Role (account 2) temporarily.

It's a complex thing to describe, so hopefully this diagram should help.



This is a typical flow, and describes what happens in the diagram above

1. A Fargate task running under *my-task-role-1*
2. Uses STS to assume the role of *switch-role*
3. Temporary credentials are returned
4. Authentication chain is updated to use the temporary credentials
5. Access is now available for *switch-role,* inside Fargate container

Lets see how this would work in the AWS CLI

When using the aws cli, you can call sts assume-role. You can test out returning these credentials, and this is really handy when testing access for a particular role outside of AWS.

This is a handy command which will generate temporary credentials for an assume role, in the format of Environment Variables, which I tend to use quite a lot when testing role permissions manually from my machine.

```
aws sts assume-role \
--duration-seconds 3600 \
--role-arn "arn:aws:iam::000000000000:role/switch-role-my-dynamo-
role" \
--role-session-name craig-session-1 | \
jq '.Credentials.SecretAccessKey , .Credentials.AccessKeyId ,
.Credentials.SessionToken' | \
```

```
xargs sh -c 'echo AWS_ACCESS_KEY_ID=$1 AWS_SECRET_ACCESS_KEY=$0
AWS_SESSION_TOKEN=$2'

AWS_ACCESS_KEY_ID=XXXXXXXXXXXXXXXXXXXX
AWS_SECRET_ACCESS_KEY=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
AWS_SESSION_TOKEN=xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxYSKzATOB2E/k4SYji5
R7nR/tPUOtPCfchVa7pLJQQ1x7fMiH3veJIxdj7RhQ//2q+cxpx9naJRR06H7HpikBpO
5gE/iAOM2/hDmg3m28cT1oGj/hKsyb9lW9jeg8RTvsDeekXWFn88T+RGT/KJjdvj4+/y
T3e/qVelBNTZxqBx9MWN6bxgXKVMYyyCmOT1iZBYSWr5VHC2smDmDRmJMm0Rmw1PUrtY
ioG2c/CFDHXLfRtbmPvwq+KOK+p/MFMi2j/IJz4HSGoANwylhylByYNY1GAkqVKvQ2qZ
KfKEOTHxxxxxxvxxxxxxxx/x=
```

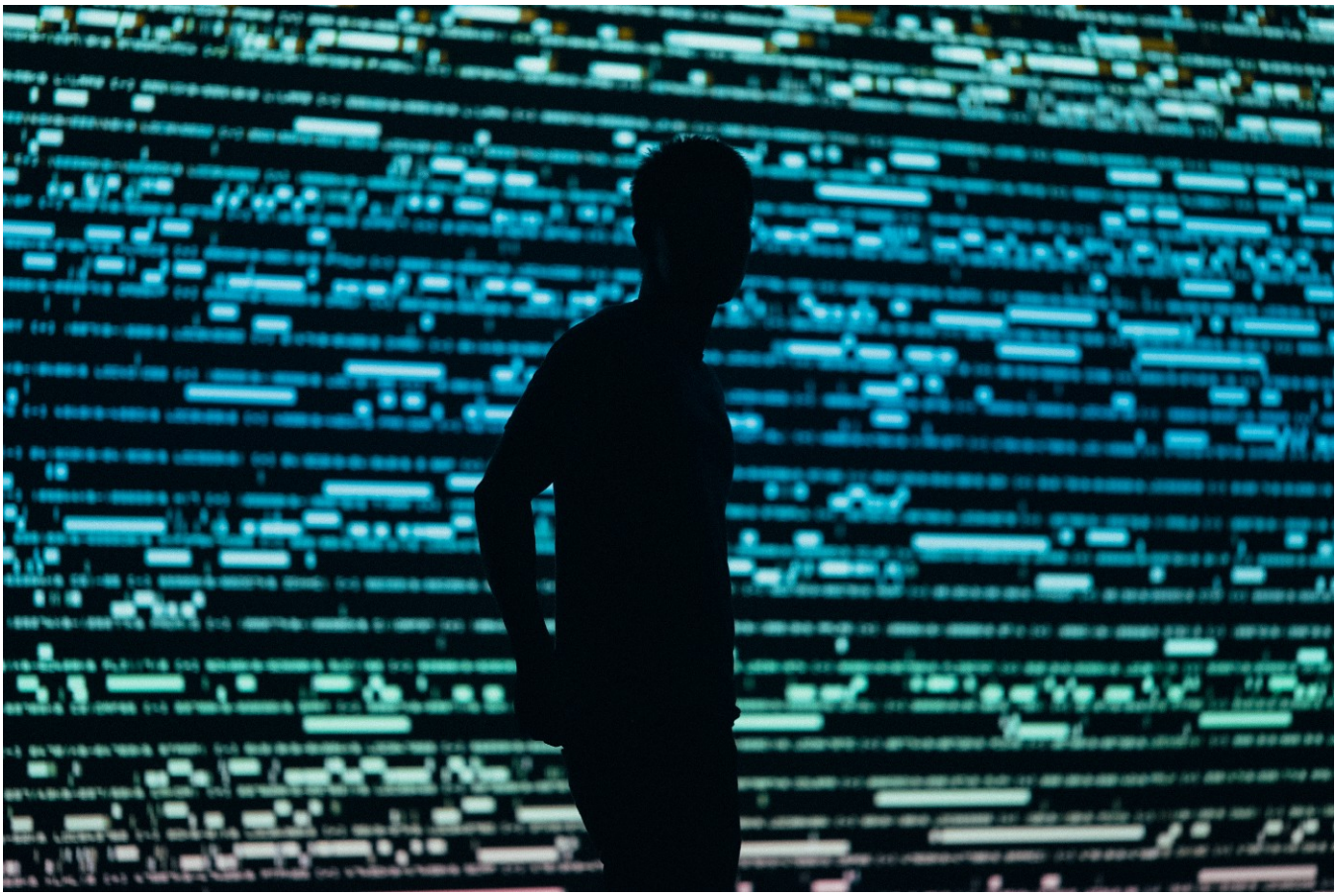


Photo by Chris Yang on Unsplash

How can we setup access between the two roles?

Here is an example of the resources you need to set this up, I have used terraform so how the configuration, but you could just do this in the console if you prefer.

1. Create a role in account 1. This is the role which will be assuming role 2.

```
resource "aws_iam_role" "task_iam_role" {
  name               = "my-fargate-task-role"
  assume_role_policy = "${file("${path.module}/policies/iam/ecs-
```

```
task-service-trust.json")}"
}
```

2. Since in my example I am using Fargate, I am allowing access for ECS to assume the role

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "",
      "Effect": "Allow",
      "Principal": {
        "Service": [
          "ecs-tasks.amazonaws.com"
        ]
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

3. Create a role for the service you want to consume in Account2, I have named this switch-role for ease. This is the role, which will have permissions to be assumed by role 1, and have permission to access my resource.

```
resource "aws_iam_role" "switch_role" {
  name               = "switch-role"
  assume_role_policy =
"${data.template_file.trust_policy_file.rendered}"
}
```

4. The policy should look something like this, it is allowing role 1 from account 1 to assume it.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": [
          "arn:aws:iam::${ACCOUNT1_ID}:role/my-fargate-task-role"
        ]
```

```
        },
        "Action": "sts:AssumeRole"
      }
    ]
}
```

5. Add a second policy, for the resources that role 2 needs to access. Notice the resource is specifying Account2_ID, because it is actually role 2, in account 2 that is doing the work, it is just role 1 which is assuming role 2

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": ["dynamodb:*"],
      "Resource": "arn:aws:dynamodb:eu-west-
1:${ACCOUNT2_ID}:table/*"
    }
  ]
}
```

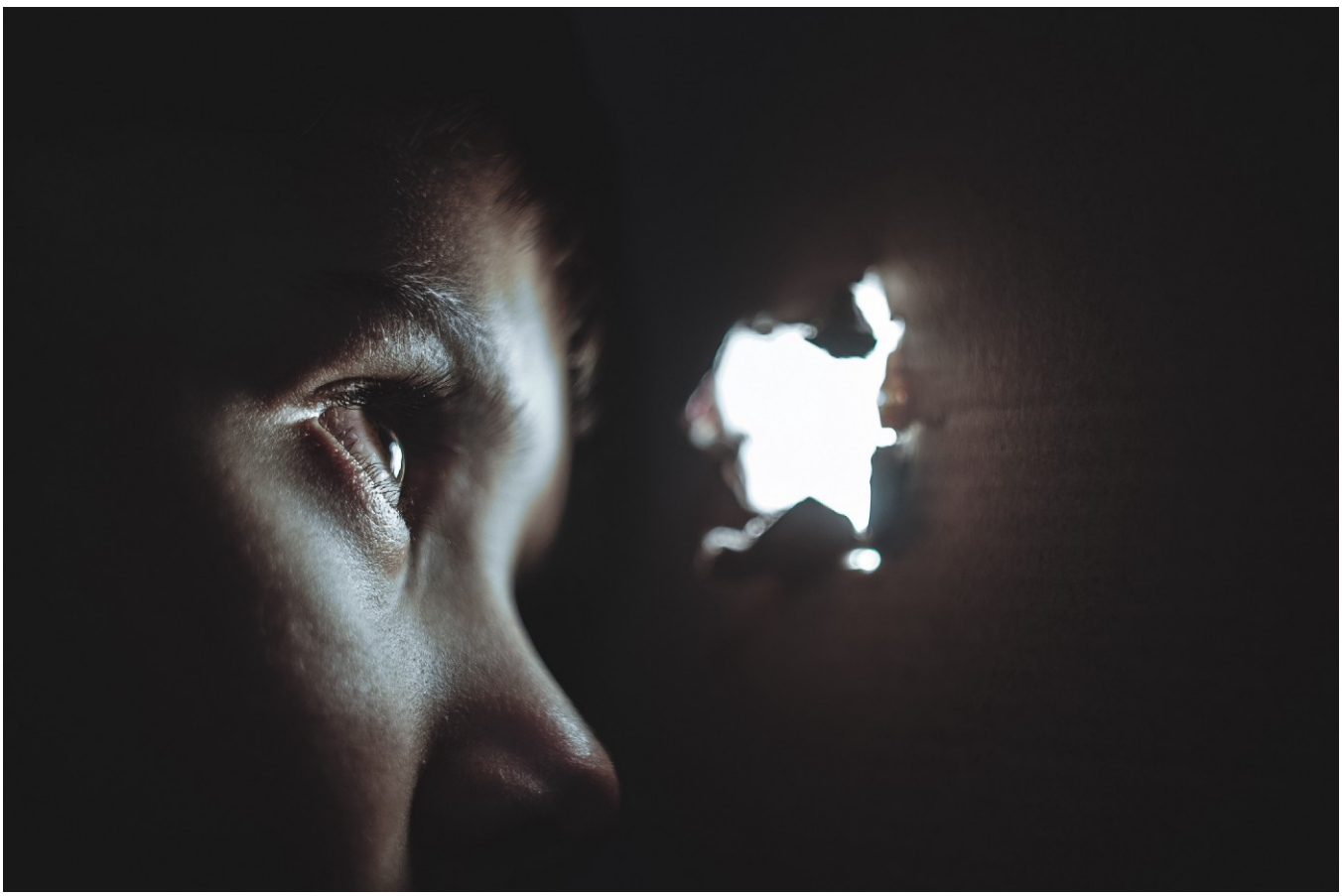


Photo by Dmitry Ratushny on Unsplash

Testing

You could test this out now, using the CLI commands above, or look at the implementation I have below, where I am returning an authenticated session in boto3 using Python.

```
def assumed_role_session(role_arn: str):
    role = boto3.client('sts').assume_role(RoleArn=role_arn,
RoleSessionName='switch-role')
    credentials = role['Credentials']
    aws_access_key_id = credentials['AccessKeyId']
    aws_secret_access_key = credentials['SecretAccessKey']
    aws_session_token = credentials['SessionToken']
    return boto3.session.Session(
        aws_access_key_id=aws_access_key_id,
        aws_secret_access_key=aws_secret_access_key,
        aws_session_token=aws_session_token)

def call_dynamo(*args, **kwargs):
    assumed_session =
assumed_role_session('arn:aws:iam::ACCOUNT2_ID:role/switch-role')
    dynamo_client = assumed_session.client('dynamodb')
    response = dynamo_client.list_tables(Limit=100)
    print(str(response))
```

Access is granted!

DevOps Security AWS Python Developer