Only you can see this message

This story's distribution setting is on. You're in the Partner Program, so this story is eligible to earn money. Learn more

How to securely use AWS Secrets Manager to inject secrets into ECS using Terraform.

Its important to make sure the security of your secrets is as tight as can be, and this guide should help to inject secrets into ECS containers.

Craig Godden-Payne
May 12 · 4 min read ★

Generate a fairly strong password using terraforms built in random password generator

```
resource random_password db_password {
  length            = 56
```

```
    special           = true
    min_special       = 5
    override_special  = "!#$%^&*()-_=+[]{}<>:?"
    keepers           = {
      pass_version  = 1
    }
  }
```

Next, create a secret in secrets manager, I have used the name `my-password` and attach the policy

```
resource aws_secretsmanager_secret my_password {
   name   = "my-password"
   policy = data.aws_iam_policy_document.my_password_policy.json
 }
```

The Only Step You Need to Progress in Your Career as a Developer or Engineer.

I don’t usually like to write articles about myself, as it feels a bit self-indulgent, but I thought it would be useful…

medium.com

Here is the policy I used, I want to allow the execution role of the fargate orchestrator to be able to access the secret.

```
data aws_iam_policy_document my_password_policy {
   statement {
     effect = "Allow"
     principals {
       identifiers = ["arn:aws:iam::XXXXXXXXXXXX:role/sample-app-
execution-role"]
       type = "AWS"
     }
     actions = [
       "secretsmanager:GetSecret",
       "secretsmanager:GetSecretValue"
     ]
     resources = ["*"]
   }
 }
```

Next, add the generated password into the secrets manager

```
resource aws_secretsmanager_secret_version my_password {
   secret_id     = aws_secretsmanager_secret.my_password.id
   secret_string = random_password.my_password.result
 }
```

Now we need to allow access from the calling side. Create an IAM policy, which allows the execution role to access the secret

```
 resource aws_iam_policy secrets_access {
   policy = <<POLICY
 {
     "Version": "2012-10-17",
     "Statement": [
         {
             "Effect": "Allow",
             "Action": [
                 "secretsmanager:GetResourcePolicy",
                 "secretsmanager:GetSecretValue",
                 "secretsmanager:DescribeSecret",
                 "secretsmanager:ListSecretVersionIds"
             ],
             "Resource": "arn:aws:secretsmanager:eu-west-
2:XXXXXXXXXXXX:secret:*"
         }
     ]
 }
 POLICY
}
```

Creating a Simple, Low-Cost Twitter Bot, utilising Serverless Technologies.

People have a love-hate relationship with twitter bots.

medium.com

Then attach the policy to the execution role

```
resource aws_iam_role_policy_attachment secret_access {
   role       = "my_execution_role_name"
   policy_arn = aws_iam_policy.secrets_access.arn
 }
```

This should be enough to allow access to the secretsmanager, and the secret we want to inject.

Now we want to actually inject the secret into the running container, and this can be done using the task definition.

I decided to use a template for my task definition, here is an example of the template I used. I wanted to abstract some of the interpolation out

```
{
  "cpu": ${cpu},
  "image": "${image}",
  "memoryReservation": ${memoryReservation},
  "name": "${name}",
  "networkMode": "awsvpc",
  "environment": ${environment},
  "secrets": ${secrets},
  "portMappings": [
   {
    "containerPort": ${containerPort}
   }
  ],
  "logConfiguration": {
   "logDriver": "awslogs",
   "options": {
    "awslogs-group": "${awslogs-group}",
    "awslogs-region": "eu-west-2",
    "awslogs-stream-prefix": "fargate"
   }
  },
  "essential": true
 }
```

Then using variables, I instantiated the template with the values I wanted

```
data "template_file" "container_definition" {
   template = file("${path.module}/container-definition.json.tpl")

   vars = {
     cpu            = var.cpu_allowance
     image          = "XXXXXXXXXXXX.dkr.ecr.eu-west-
2.amazonaws.com/my-sample-app"
     memoryReservation  = var.memory_allowance == "" ? 512 :
var.memory_allowance
     name           = "my-sample-app"
     containerPort  = var.service_port == "" ? 80 : var.service_port
     awslogs-group  =
aws_cloudwatch_log_group.cloudwatch_log_group.name
```

```
    environment    = jsonencode(var.environment_variables)
    secrets       = jsonencode(var.secrets)
  }
}
```

I then supply the secrets like this:

```
secrets                         = [{
  name: "my_password",
  valueFrom: data.aws_secretsmanager_secret.my_password.arn
}]
```

The deployed task definition looks like this:

```
{
  "ipcMode": null,
  "executionRoleArn": "arn:aws:iam::XXXXXXXXXXXX:role/my-app-
execution-role",
  "containerDefinitions": [
    {
      "dnsSearchDomains": null,
      "logConfiguration": {
        "logDriver": "awslogs",
        "secretOptions": null,
        "options": {
          "awslogs-group": "/fargate/my-sample-app",
          "awslogs-region": "eu-west-2",
          "awslogs-stream-prefix": "fargate"
        }
      },
      "entryPoint": null,
      "portMappings": [
        {
          "hostPort": 80,
          "protocol": "tcp",
          "containerPort": 80
        }
      ],
      "command": null,
      "linuxParameters": null,
      "cpu": 512,
      "environment": [
        {
          "name": "my_username",
          "value": "username"
        }
      ],
      "resourceRequirements": null,
      "ulimits": null,
      "dnsServers": null,
      "mountPoints": [],
      "workingDirectory": null,
      "secrets": [
        {
          "valueFrom": "arn:aws:secretsmanager:eu-west-
2:XXXXXXXXXXXX:secret:my-password-gD4Hqa",
          "name": "my_password"
        }
      ],
      "dockerSecurityOptions": null,
      "memory": null,
      "memoryReservation": 1024,
      "volumesFrom": [],
      "stopTimeout": null,
      "image": "XXXXXXXXXXXX.dkr.ecr.eu-west-2.amazonaws.com/my-
sample-app:latest",
      "startTimeout": null,
      "firelensConfiguration": null,
      "dependsOn": null,
      "disableNetworking": null,
      "interactive": null,
      "healthCheck": null,
      "essential": true,
      "links": null,
      "hostname": null,
      "extraHosts": null,
      "pseudoTerminal": null,
      "user": null,
      "readonlyRootFilesystem": null,
      "dockerLabels": null,
      "systemControls": null,
      "privileged": null,
      "name": "my-sample-app"
    }
  ],
  "placementConstraints": [],
  "memory": "1024",
  "taskRoleArn": "arn:aws:iam::XXXXXXXXXXXX:role/my-sample-app-task-
role",
  "compatibilities": [
    "EC2",
    "FARGATE"
  ],
  "taskDefinitionArn": "arn:aws:ecs:eu-west-2:XXXXXXXXXXXX:task-
definition/my-sample-app:9",
  "family": "my-sample-app",
  "requiresAttributes": [
    {
      "targetId": null,
      "targetType": null,
      "value": null,
      "name": "com.amazonaws.ecs.capability.logging-driver.awslogs"
    },
    {
      "targetId": null,
      "targetType": null,
      "value": null,
      "name": "ecs.capability.execution-role-awslogs"
      },
      {
        "targetId": null,
        "targetType": null,
        "value": null,
        "name": "com.amazonaws.ecs.capability.ecr-auth"
      },
      {
        "targetId": null,
        "targetType": null,
        "value": null,
        "name": "com.amazonaws.ecs.capability.docker-remote-api.1.19"
      },
      {
        "targetId": null,
        "targetType": null,
        "value": null,
        "name": "ecs.capability.secrets.asm.environment-variables"
      },
      {
        "targetId": null,
        "targetType": null,
        "value": null,
        "name": "com.amazonaws.ecs.capability.docker-remote-api.1.21"
      },
      {
        "targetId": null,
        "targetType": null,
        "value": null,
        "name": "com.amazonaws.ecs.capability.task-iam-role"
      },
      {
        "targetId": null,
        "targetType": null,
        "value": null,
        "name": "ecs.capability.execution-role-ecr-pull"
      },
      {
        "targetId": null,
        "targetType": null,
        "value": null,
        "name": "com.amazonaws.ecs.capability.docker-remote-api.1.18"
      },
      {
        "targetId": null,
        "targetType": null,
        "value": null,
        "name": "ecs.capability.task-eni"
      }
    ],
    "pidMode": null,
    "requiresCompatibilities": [
      "FARGATE"
    ],
    "networkMode": "awsvpc",
    "cpu": "512",
    "revision": 9,
    "status": "ACTIVE",
    "inferenceAccelerators": null,
    "proxyConfiguration": null,
    "volumes": []
  }
```

Using Microservice Patterns in an increasingly Serverless world

When working on an application domain, it is beneficial to use the microservices software design pattern.

medium.com

AWS Terraform DevOps