

An AWS Fargate use case

In our use case, we focus on the activating event and create an AWS event rule pointing to the AWS Fargate Cluster we are using:

```
{
  "detail": {
    "clusterArn": ["arn:aws:ecs:eu-central-1:1234567890:cluster/clustername"],
    "lastStatus": ["ACTIVATING"]
  },
  "detail-type": ["ECS Task State Change"],
  "source": ["aws.ecs"]
}
```

The target for this event rule is an AWS Lambda function with the appropriate IAM permissions:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "ec2:CreateTrafficMirrorSession",
        "ec2:DescribeTrafficMirrorSessions",
        "ec2:CreateTags"
      ],
      "Resource": "*"
    },
    {
      "Effect": "Allow",
      "Action": "logs:CreateLogGroup",
      "Resource": "arn:aws:logs:eu-central-1:1234567890:*"
    },
    {
      "Effect": "Allow",
      "Action": "ecs:StopTask",
      "Resource": "arn:aws:ecs:eu-central-1:1234567890:task/*"
    },
    {
      "Effect": "Allow",
      "Action": [
        "logs:CreateLogStream",
        "logs:PutLogEvents"
      ],
      "Resource": [
        "arn:aws:logs:eu-central-1:1234567890:log-group:/aws/lambda/
loggroupname:*"
      ]
    }
  ]
}
```

Configuring a trusted entity

To allow the Lambda to run with the IAM role, we require a trusted entity:

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

The AWS Lambda function also needs to know the previously created traffic mirror filter and the traffic mirror target.

We configured our Lambda to have these values available as environment variables:

Environment variables (2) Edit	
The environment variables below are encrypted at rest with the default Lambda service key.	
Key	Value
TrafficMirrorFilter	tmf-011415269e9c93569
TrafficMirrorTarget	tmt-08c911877e8e9c51c

The Lambda source code

With this setup, we can finally prepare the Lambda source code itself. It retrieves the required data — like the elastic network interfaces of the newly created container — and uses it to make a traffic mirror session with this interface.

The source code for that is:

```
import json
import boto3
import logging
import os
from botocore.exceptions import ClientError

log = logging.getLogger()
log.setLevel(logging.INFO)

def lambda_handler(event, context):
    TrafficMirrorFilter = os.environ.get('TrafficMirrorFilter')
    TrafficMirrorTarget = os.environ.get('TrafficMirrorTarget')
    client = boto3.client('ec2')
    client_ecs = boto3.client('ecs')
    #print(json.dumps(event))

    if event[«source»] != «aws.ecs» or event[«detail-type»] != «ECS Task State
Change»:
        log.info(f'Function only supports input from events with a source type of: aws.
ecs')
        raise ValueError(«Function only supports input from events with a source type
of: aws.ecs»)
    log.info(f'laststatus: {event[«detail»][«lastStatus»]}')
    TaskId = event[«resources»][0].split('/')[2]
    Cluster = event[«resources»][0].split('/')[1]
    log.info(f'Container Task Id: {TaskId} on Cluster: {Cluster}')
    for attachment in event[«detail»][«attachments»]:
        #log.info(f'{json.dumps(event)}')
        if attachment.get(«status») == «ATTACHED» and attachment.get(«type») == «eni»:
            for property in attachment.get(«details»):
                if property[«name»] == «networkInterfaceId» and event[«detail»]
[«lastStatus»] == «ACTIVATING»:
                    print(property[«value»])
                    #check if tms allready exists
                    existing_tms = client.describe_traffic_mirror_sessions(
                        Filters=[
                            {
                                'Name': 'network-interface-id',
                                'Values': [
                                    property[«value»],
                                ]
                            }
                        ],
```

The Lambda source code

```
    ],
    MaxResults=5
).get('TrafficMirrorSessions', [])
if existing_tms:
    log.info(f'TrafficMirrorSession for interface {property[«value»]} already
exists')
else:
    log.info(f'Creating TrafficMirrorSession for interface
{property[«value»]}')
    try:
        response = client.create_traffic_mirror_session(
            NetworkInterfaceId=property[«value»],
            TrafficMirrorTargetId=TrafficMirrorTarget,
            TrafficMirrorFilterId=TrafficMirrorFilter,
            SessionNumber=1,
            Description=»Traffic mirror Source created by lambda«,
            TagSpecifications=[
                {
                    'ResourceType': 'traffic-mirror-session',
                    'Tags': [
                        {
                            'Key': 'Name',
                            'Value': «Session for interface « +
property[«value»]
                        },
                    ]
                },
            ],
            DryRun=False
        )
    except ClientError as e:
        if e.response['Error']['Code'] == 'InvalidNetworkInterfaceID.
NotFound':
            log.info(f'Eni not present yet {property[«value»]}')
            elif e.response['Error']['Code'] ==
'NetworkInterfaceNotSupportedException':
                response = client_ecs.stop_task(
                    cluster=Cluster,
                    task=TaskId,
                    reason='killed by lambda due to not spawned on nitro instance
for trafficmirror reasons'
                )
                log.info(f'Killing Task: {TaskId} on Cluster {Cluster} since the
task was not started on a nitro instance (Traffic mirror not available)')
            else:
                log.info(f'Error for {property[«value»]} is Code:
{e.response[«Error»][«Code»]} Message: {e.response[«Error»][«Message»]}')
```

From now on, this setup will trigger when a new container starts. It will automatically create a traffic mirror session and attach it to the previously created traffic mirror target.

This way, the solution will receive any incoming traffic from all containers running inside the AWS Fargate service.