Enforcing strict EC2 instance tagging policies with Lambda

MONDAY. MAY 01, 2017 - 12 MINS

AWS LAMBDA PYTHON

TL;DR

I coded something in order to enforce strict tagging policies on AWS EC2 instances using Python and a bunch of AWS services (Lambda, Cloudtrail, SNS, and S3). If you keep reading, I'm going to talk to you about AWS Lambda and Serverless computing, or FaaS (Function as a service).

You can check the source code and permission related template files here:

```
https://github.com/sebiwi/broom
```

If you want to use it, create a Cloudtrail trail, activate it on every region, create an S3 bucket to store the Cloudtrail logs, create a SNS topic for notifications, create a Lambda function using the Python code (adapted to your resources and use case), an IAM role using the policy that comes with it, and finally activate events from the S3 bucket to the Lambda function on object creation.

I can teach you how to do all of these things if you don't know how. Just keep reading.

Why?

I like to be thorough and organized with my cloud resources. I usually work with and manage several shared AWS accounts for different teams. Without a proper tagging workflow, the EC2 resource set can become a mess. You'll probably find yourself asking your team who created an untagged instance, and if they are still using it. That feels kinda dull, since we're in 2017 and all.

What do I want in order to fix this? I want to analyze each instance that is created on my AWS account, and destroy it if it is not properly tagged. I also want to get a notification when this happens, with useful information about the instance creation request, such as the instance type, or the username of the user that created the resource.

Keep your stuff clean!

I decided to call this thing Broom, since I use it to keep my things clean. Get it?

To be honest, I also wanted to take Lambda for a ride, since Serverless is pretty trendy nowadays. Let us discuss it.

Serverless and Lambda

Serverless is another paradigm in the Something-as-a-Service universe. It allows you to execute code (functions) on the Cloud, without worrying about the underlying servers, middleware or configuration. The basic premise is simple: your code is executed only when it is needed. You're also billed to the execution time of your function, so you actually pay (almost) exactly what you use.

Even though the paradigm is called Serverless, this doesn't mean that your code is executed without any servers. It means that you don't have to provision, configure or manage the servers that are used to run your code yourself, and that your provider is going to do it for you. A corollary to the paradigm is the fact that your code can scale automatically (both up and down) indefinitely according to the demand. The only constraint is that your code must be stateless.

Since Serverless is right now the new fad of IT, there are many new frameworks adopting the model. For example, Fission is a framework for serverless functions on Kubernetes. If you don't know what Kubernetes is, or how it works, you can check it out over here.

And Lambda? Well, Lambda is AWS serverless computing service. It originally came out in 2014, and only supported NodeJS at the time. Nowadays it has Python, Java and C# support, and it can also run native Linux executables, if you can run them from within one of the supported languages/frameworks. This allows you to run Go and Haskell binaries on Lambda, for example, if you do some hacking. In terms of billing, Lambda is metered in increments of 100 milliseconds, which is almost negligible compared to the EC2 minimum usage fee of one hour.

How does this work? How can AWS spin up EC2 instances, deploy your code in them, and answer requests that fast? *How*?

Get the hint?

It's all about containers, really. When a Lambda function is triggered, Lambda launches a container with your code and the necessary dependencies in it. This takes some time, but its amount is negligible compared to the normal server provisioning delay. The container is also reused (on a best-effort basis) for further invocations of the function, if no changes have been made to either the code or the configuration.

Exciting, isn't it? You wanna know how to set it up? Let's go!

Just so you know what we're going to do next: we're going to log every API call made on AWS, put these logs on an S3 bucket, launch a Lambda function that analyzes newly-created instances for certain tags whenever a new log file is created on the bucket, and publish notifications on an SNS topic whenever an instance is destroyed due to lack of tags.

Note: we're going to be creating a list of AWS resources. Remember to create them on the same region whenever it is possible.

First up, see the trail

We need to be able to trace the AWS API calls in the target AWS account so we can see when EC2 instances are actually created. You can do this with Cloudtrail.

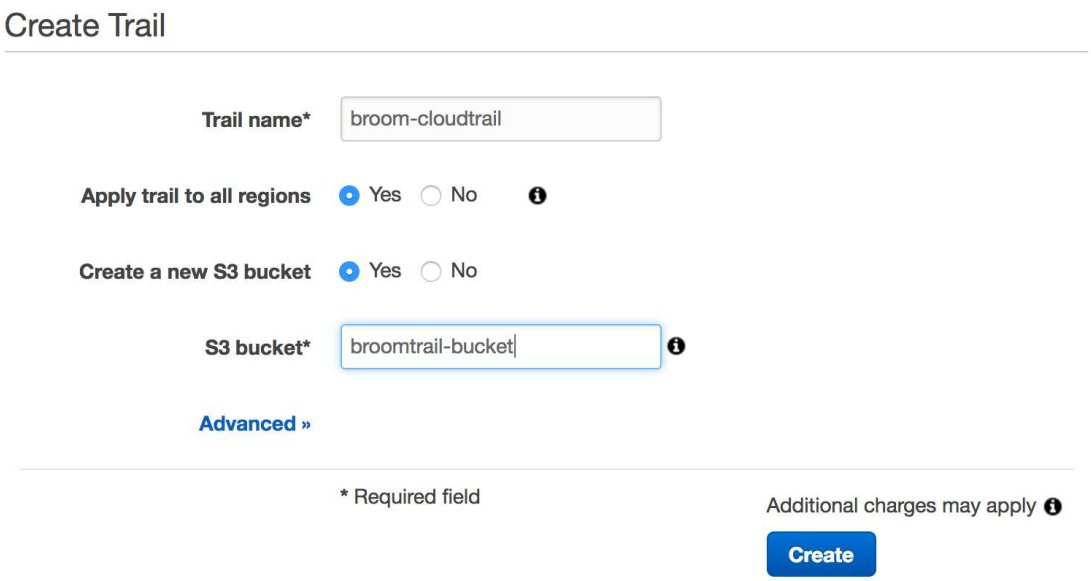
From the Cloudtrail FAQs:

```
AWS CloudTrail is a web service that records API calls made on your account and del
log files to your Amazon S3 bucket
```

You can create a Cloudtrail trail either using the AWS cli tool:

```
aws cloudtrail create-trail --name broom-cloudtrail --s3-bucket-name broomtrail-buc
```

Or the AWS web console:



You can create a new S3 bucket in the same single step if you're using the console. You'll have to do it separately if you're using the CLI. You can also configure an SNS topic in order to receive notifications each time a log object is stocked in the S3 bucket. This is hell, and I encourage you not to do it.

Remember to create a multi-region trail, since we want to be able to audit instances in all regions.

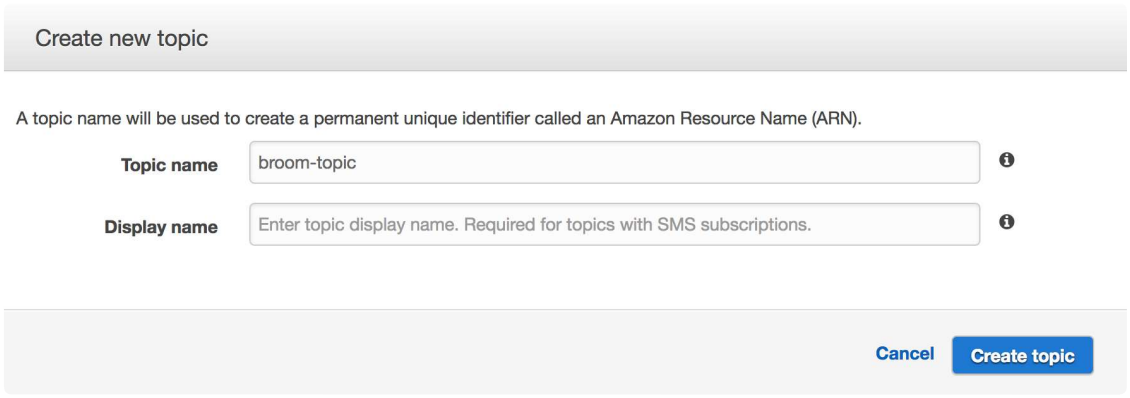
Notify me then!

Next up, we create an SNS (Simple Notification Service) topic in order to receive notifications whenever instances are destroyed. Same as before, you can do it using either the CLI or the console:

```
aws sns create-topic --name broom-topic
```

Remember to note the ARN of your topic, since you will be using it for the next step:

```
{
    "ResponseMetadata": {
        "RequestId": "1469e8d7-1642-564e-b85d-a19b4b341f83"
    },
    "TopicArn": "arn:aws:sns:region:weird_number:topic_name"
}
```



Console version

Don't forget to subscribe to this topic if you want to receive notifications. You probably do, so do it.

Know your role

The Lambda function needs to be able to access some things:

- Cloudwatch, in order to stock execution logs
- The Cloudtrail S3 bucket, in order to recover the log files
- The EC2 API, in order to list instance tags and destroy instances when necessary
- The previously created SNS topic, in order to send notifications.

In order to do all of these things, we'll create a policy using the template policy file that comes with the project. You can find the template policy file here. You can use it almost as it is, just remember to replace the SNS ARN with the one you got in the previous section:

```
{
    "Effect": "Allow",
    "Action": [
        "sns:Publish"
    ],
    "Resource": arn:aws:sns:region:weird_number:topic_name
}
```

Then, you need to create a role, and attach the previously created policy to the role. The Lambda function will then be executed using this role, and

it will have access to all the necessary resources.

Lambda for all

Finally, let's create the actual Lambda function. You can create a package for your function and then upload it directly to AWS using the CLI. This is useful when you are using dependencies that are not standard and that are not already present in the AWS Lambda environment. This is not the case with the Broom function, so you can just use the AWS console to create it. I used Python and boto3 for Broom. You can see the source code here.

Basically, I declare the SNS topic ARN, and the codes that I'll be using to tag our instances. In this case, it can be the codes assigned to each person that is currently working on the team. I always refer to myself as 'LOL', for example:

```
# Values
SNS_TOPIC_ARN = 'arn:aws:sns:region:weird_number:topic_name'
CODES = ['LOL', 'GGG', 'BRB', 'YLO']
```

Then, I declare some helper functions, the first to decompress the Cloudtrail S3 log files:

```
def decompress(data):
    with gzip.GzipFile(fileobj=io.BytesIO(data)) as f:
        return f.read()
```

And the second one to generate a report on the destroyed instance:

```
def report(instance, user, region):
    report = "User " + user + " created an instance on region " +  region + " witho
    report += "Instance id: " + instance['instanceId'] + "\n"
    report += "Image Id: " + instance['imageId'] + "\n"
    report += "Instance type: " + instance['instanceType'] + "\n"
    report += "This instance has been destroyed."
    return report
```

Then, the lambda_handler function, which is the actual function that is going to be triggered by the S3 event. First, I recover the bucket name and the object key:

```
def lambda_handler(event, context):
    bucket = event['Records'][0]['s3']['bucket']['name']
    key = urllib.unquote_plus(event['Records'][0]['s3']['object']['key'].encode('ut
```

This might look a little bit weird. It's only because the event message structure is kinda complex. Then, I recover the actual log (s3_object), decompress it, and create a JSON object with it:

```
s3_object = s3.get_object(Bucket=bucket, Key=key)
s3_object_content = s3_object['Body'].read()
s3_object_unzipped_content = decompress(s3_object_content)
json_object = json.loads(s3_object_unzipped_content)
```

A log object may contain many API calls. I'm only interested in the ones that create EC2 instances. That means the ones with the "RunInstances" event name. If I find one of these events, I'll connect to that region and recover the created instances:

```
for record in json_object['Records']:
    if record['eventName'] == "RunInstances":
        user = record['userIdentity']['userName']
        region = record['awsRegion']
        ec2 = boto3.resource('ec2', region_name=region)
        for index, instance in enumerate(record['responseElements']['instancesSet']
            instance_object = ec2.Instance(instance['instanceId'])
```

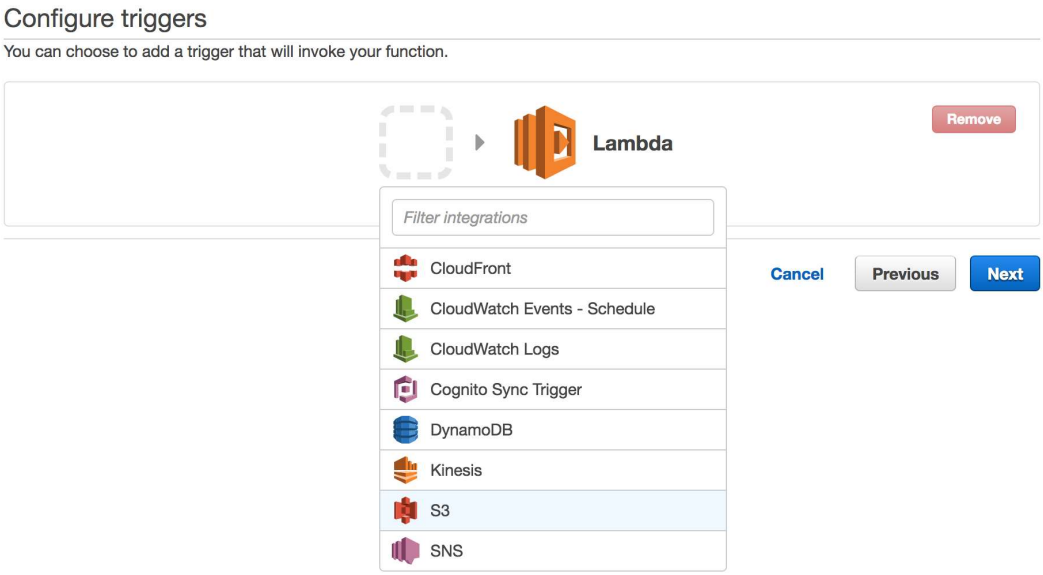
I'll check if the tag is present in the instance, and if it exists in the list of valid codes that I defined previously. If it is not, I'll destroy the instance, and publish a report to the SNS topic:

```
tags = {}
for tag in instance_object.tags:
    tags[tag['Key']] = tag['Value']
if('Code' not in tags or tags['Code'] not in CODES):
    instance_object.terminate()
    sns.publish(TopicArn=SNS_TOPIC_ARN, Message=report(instance, user, region))
```

That's it! You can adapt this in order to check your instances in many different ways, like a different tag, or a combination of many.

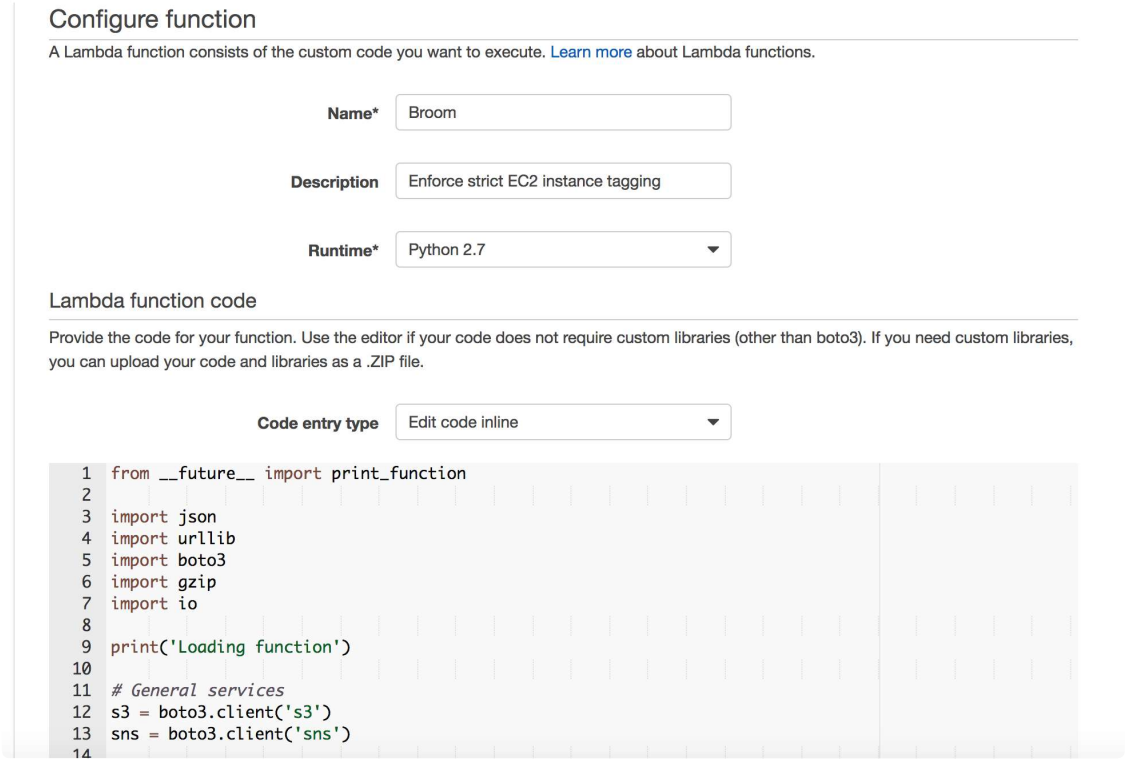
Now, go to the Lambda section on the AWS console, and click on "Create a Lambda function". Many function blueprints will be proposed to you then. Just go with "Blank Function", since you already have everything that's needed.

On the "Configure triggers" section, choose S3, and select the Bucket you created previously. On "Event type", choose "Object Created", since you want your function to be executed when a new log object is created. You can enable the trigger right away, or afterwards, it's up to you.



Trigger happy

Then name your function, select the runtime, and include your code with it:

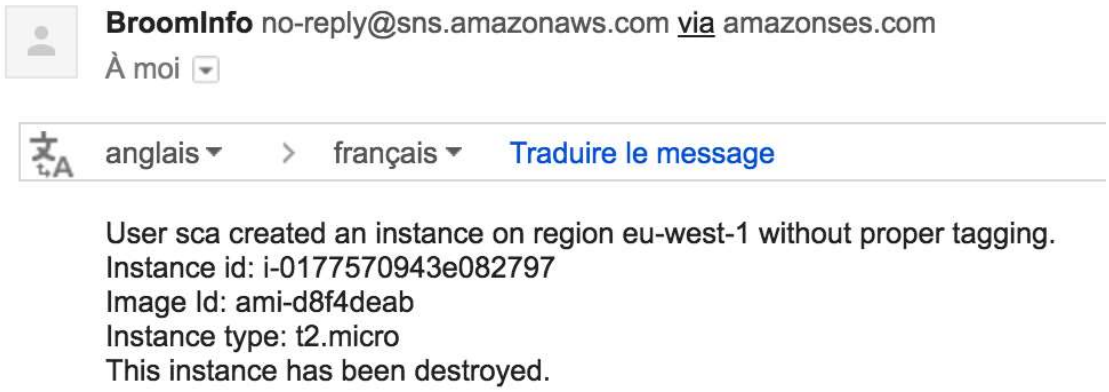


Be sure to specify the role you created earlier in the “Lambda function handler and role” section. If you don’t, the function won’t execute properly.

You can change the assigned resources and the maximal execution time in the advanced settings section. I usually set this up to 128 MB and 20 second timeout, but it might depend on the specific characteristics of your team.

After that, create the function on the review page. That’s it, you’re all set!

If you create an instance without proper tagging, you will receive a notification like this one:



Whoops, sorry!

Final thoughts

You can see all the Lambda limits here. For this use case, none of them are actually blocking. There's a 100 concurrent Lambda function execution limit which might be a problem if you have a single account and people working on many different regions (shrug). This is actually a soft limit, which is meant to avoid excessive charges without your approval. You can request a limit increase if you feel that the default limit is getting in your way.

Mocking the necessary resources in order to test this was slightly cumbersome too. I had to create valid logs, make sure that they had the necessary events in them, and then use the "Test" option in the Lambda dashboard on the AWS console, while specifying the mocked log in the test JSON object. I'd like to have a way to make this simpler somehow.

I could have used a serverless framework to address this issue during development: something like Serverless, or chalice. I'll definitely try one the next time I do some serverless development. These really come in handy, as you can test your code locally, generating input events without setting up the whole infrastructure. They also help you manage and update multiple environments, and create the resources needed for your function, with Cloudformation. You can also automate the creation of these Lambda management functions using a separate infrastructure as code tool, like Terraform.

Anyways, I hope you enjoyed that as much as I did!

« How does it work? Kubernetes! Episode 5
- Master and worker, at last!

Sebiwi
Perfectionist

Tweet Share