



Workshop Workbook

Event Driven Security

in AWS with Lambda



Lesson 1

“Restricted Access”

In Lesson 1 we are going to create an event driven workflow the understand, monitor and react within the bounds of an organization's security policies.

You work in an enterprise with over 1,000 employees and a complex set of security rights and roles applied at a user and group level.

Within the organization there is a collection of ‘ADMIN’ users - with the rights to perform any and all account actions.

Policies will always form part of a well-engineered security architecture - but with our model we have a problem.

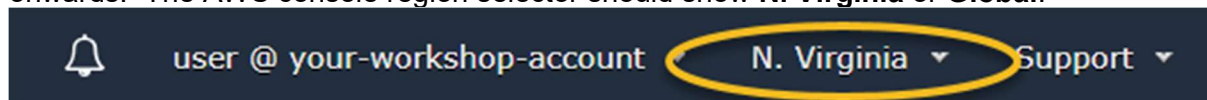
Utilizing a number of AWS products - in this lesson you will implement a serverless, event driven workflow capable of detecting and reacting to policy breach; preventing further damage or unauthorized actions.

NOTE: BE SURE TO CREATE ALL YOUR RESOURCES IN THE N. VIRGINIA REGION (US-EAST-1)

1. SET YOUR REGION TO US. EAST (N. VIRGINIA)

Log in to the AWS console, and make sure you have set your region to US East (N.Virginia).

Please make sure that all resources and services you create are in the same region from here onwards. The AWS console region selector should show **N. Virginia** or **Global**:



Note: IAM is a global service. When working in the IAM console the region will show ‘Global’.

2. CREATE ADMIN IAM ROLE AND GROUP

Let’s start by creating an IAM ROLE called ‘administrator’. We will also create a group ‘admins’ for users that can *assume* that role - we don’t expect that to be a common occurrence, so we want to be able to audit and track when that happens since it would only be necessary when changing IAM permissions. In a real-world scenario, we may have a second role for just “working” that excludes most IAM based permissions.

- Go to the CloudFormation Console

- We are going to “Create Stack”
 - Select “Upload a template to Amazon S3” and “choose file” to Lesson 1 Restricted Access/step-2-admin-iam.yaml
 - Click “Next”
 - Set Stack Name to “AdminIAM” and click “Next”
 - On the Options screen click “Next”
 - Check “I acknowledge that AWS CloudFormation might create IAM resources with custom names.” and click “Create”
- Behind the scenes this CloudFormation template
 - Creates a new IAM Role called ‘administrator’
 - Attaches the AWS Managed Policy ‘AdministratorAccess’
 - Exports the RoleARN and RoleName from the stack in case we ever need them in other stacks
 - Creates an IAM Group called ‘admins’ that can assume the role we created
 - Exports the GroupARN and GroupName from the stack in case we ever need them in other stacks
- You can monitor progress by checking the new stack in the CloudFormation console and reviewing the “events” tab
- If everything goes well you should see CREATE_COMPLETE and four outputs in the “Outputs” tab with the IAM group name and ARN, and the IAM role name and ARN

AWSTemplateFormatVersion: "2010-09-09"

Description: "Event Driven Security - Lesson 1 Step 2"

Resources:

AdminIAMRole:

Type: "AWS::IAM::Role"

Properties:

RoleName: "administrator"

AssumeRolePolicyDocument:

Version: "2012-10-17"

Statement:

-

Effect: "Allow"

Principal:

AWS: !Sub "arn:aws:iam::\${AWS::AccountId}:root"

Action:

- "sts:AssumeRole"

ManagedPolicyArns:

- "arn:aws:iam::aws:policy/AdministratorAccess"

AdminIAMGroup:

Type: "AWS::IAM::Group"

Properties:

GroupName: "admins"

Policies:

- PolicyName: "allowRoleAssume"

PolicyDocument:

Version: "2012-10-17"

Statement:

- Sid: "assumeAdminRole"
Effect: "Allow"
Action: "sts:AssumeRole"
Resource: !GetAtt AdminIAMRole.Arn

3. CREATE IAM USER

To begin the next part of our lesson, let's create a new user **Helen** that is a member of the **admins** group and while they of course are permitted to create IAM users and policy changes we want to audit and notify any time certain changes have been made.

Note: All of the below could also be done with a CloudFormation template, however in the real world IAM users and group memberships are one of the few things that may be managed manually by IAM admins or in entirely different systems like an Active Directory.

- Navigate to the IAM Service
- Click **Users** from the left menu within the IAM console
- Click on **Add User** and enter the user name **Helen**, and in the **AWS access type** section check the box next to **AWS Management Console access**.
- Select the checkbox next to **Custom Password** and enter an easy to remember password (or use your password manager!) - since we are doing a disposable lab we are being a bit lax otherwise you should always pick something strong and assign multi-factor authentication. Untick the checkbox next to **Require password reset** and click **Next: Permissions**
- As above, **Helen** is going to be a member of the **admins** group, so check the box next to **admins** and click **Next: Review** followed by **Create user**
- Note down the **IAM URL** circled on the screenshot below and click **Close**

Add user

1 2 3 4

✓ Success

You successfully created the users shown below. You can view and download user security credentials. You can also email users instructions for signing in to the AWS Management Console. This is the last time these credentials will be available to download. However, you can create new credentials at any time.

Users with AWS Management Console access can sign-in at: <https://your-workshop-account.signin.aws.amazon.com/console>

Download .csv

User	
▶	✓ Helen

4. ROLE ASSUMPTION: TESTING YOUR USER

We are going to test that your user can login and assume a role to be able to do administrative actions. This will walk you through the **ROLE ASSUMPTION** process. Note, you could enforce MFA and IP restrictions on role assumptions making it a powerful way to protect access to roles and accounts.

- Open an incognito browser or a different browser than the one you are logged into the AWS Console on. *AWS only allows you to be logged in as a single user / role across all sessions in a browser*
- Go to the previously noted **IAM URL**
- Enter your IAM User (Helen) and password you created for her
- You should be logged in. *Note, Helen does not by default have many permissions. You may see a lot of red and errors if you try to navigate around the console*

Welcome to Identity and Access Management

We encountered the following errors while processing your request:

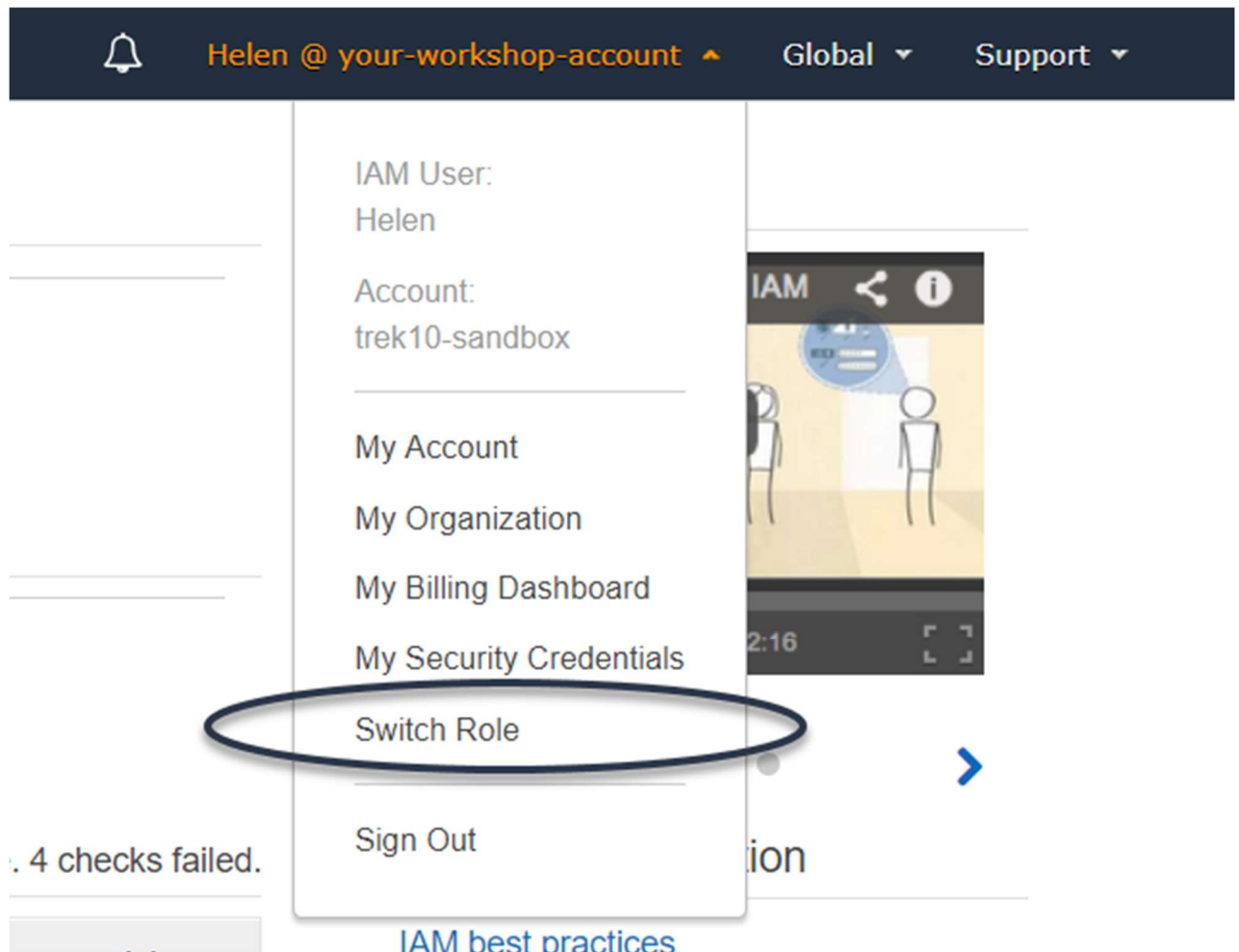
- ✖ User: am:aws:iam::491142528373:user/Helen is not authorized to perform: iam:GetAccountSummary on resource: *
- ✖ User: am:aws:iam::491142528373:user/Helen is not authorized to perform: iam:ListAccountAliases on resource: *

IAM Resources

We encountered the following errors while processing your request:

- ✖ User: am:aws:iam::491142528373:user/Helen is not authorized to perform: iam:GetAccountSummary on resource: *
- ✖ User: am:aws:iam::491142528373:user/Helen is not authorized to perform: iam:ListAccountAliases on resource: *

- Click the User dropdown in the upper right corner and navigate to “Switch Role” in the dropdown



- In the Switch Account screen, enter you AWS Account Id (you can get this from your other logged in AWS Console Session under “Support”)
- Enter “administrator” for the role
- Your screen should look similar to the below

Account* ⓘ

Role* ⓘ

Display Name ⓘ

Color a a a a a

*Required

Cancel

Switch Role

- Click “Switch Role”
- If everything went well, you should be logged in to the console as a role and able to navigate around the console with full administrative capabilities!

5. INSTALL AND CONFIGURE THE AWS CLI

To complete the rest of this lesson, and in future lessons, we are going to need the AWS command line interface (AWS CLI). This walks you through the steps, assuming you don't already have it set up for your workshop AWS account. While it is possible to upload a Lambda function through the web console, it is significantly less work in terms of upload and lifecycle management when using CloudFormation and the Serverless Application Model (AWS SAM).

If you have already set up the CLI, you may skip this step.

- Open the AWS console by going to aws.amazon.com.
- Click **IAM**.
- Click **Users** and then **Add User**.
- Set the User name as **cli-admin**.
- Enable the **Programmatic access** checkbox.
- Click **Next: Permissions**.

Add user



Set user details

You can add multiple users at once with the same access type and permissions. [Learn more](#)

User name*

[Add another user](#)

Select AWS access type

Select how these users will access AWS. Access keys and autogenerated passwords are provided in the last step. [Learn more](#)

- Access type*
- ☒ **Programmatic access**
Enables an **access key ID** and **secret access key** for the AWS API, CLI, SDK, and other development tools.
 - ☐ **AWS Management Console access**
Enables a **password** that allows users to sign-in to the AWS Management Console.

Add user



Set user details

You can add multiple users at once with the same access type and permissions. [Learn more](#)

User name*

[Add another user](#)

Select AWS access type

Select how these users will access AWS. Access keys and autogenerated passwords are provided in the last step. [Learn more](#)

- Access type* ☒ **Programmatic access**
 Enables an **access key ID** and **secret access key** for the AWS API, CLI, SDK, and other development tools.
- ☐ **AWS Management Console access**
 Enables a **password** that allows users to sign-in to the AWS Management Console.

* Required

[Cancel](#) [Next: Permissions](#)

- Click **Attach existing policies directly**.
- Search for **AdministratorAccess** and select it.

Attach one or more existing policies directly to the user or create a new policy. [Learn more](#)

[Create policy](#) [Refresh](#)

Filter: Policy type Showing 5 results

	Policy name	Type	Attachments	Description
<input checked="" type="checkbox"/>	AdministratorAccess	Job function	6	Provides full access to AWS services and resources.
<input type="checkbox"/>	AmazonAPIGatewayAdministrator	AWS managed	0	Provides full access to create/edit/delete APIs in Amazon API Gateway via the AWS Ma...
<input type="checkbox"/>	DatabaseAdministrator	Job function	0	Grants full access permissions to AWS services and actions required to set up and con...
<input type="checkbox"/>	NetworkAdministrator	Job function	0	Grants full access permissions to AWS services and actions required to set up and con...
<input type="checkbox"/>	SystemAdministrator	Job function	0	Grants full access permissions necessary for resources required for application and dev...

[Cancel](#) [Previous](#) [Next: Review](#)

- Click **Next: Review**.
- Click **Create user**.
- Click the **Download .csv** button to download the Access Key ID and Secret Access Key of the user. Alternatively, you can copy and paste these keys to a safe place. **You will need both of these keys later, so don't lose them.**
- Click **Close** when you are finished.

Open a terminal and install the AWS CLI if isn't already installed. You can confirm that you have it installed by running:


```
aws --version
```

If that returns an error please read the AWS CLI User Guide on installing the CLI
Installing the AWS Command Line Interface found at
docs.aws.amazon.com/cli/latest/userguide/installing.html

Once you have the CLI installed, run:

```
aws configure
```

And use the inputs from the .csv that was downloaded previously.

Verify the install:

```
aws sts get-caller-identity --output json
```

If it worked, you will see similar output to the below:

```
{
  "Account": "123456789101",
  "UserId": "AROAJVGWCJWDJWRZRE2KC",
  "Arn": "arn:aws:sts::123456789101:user/serverless-admin"
}
```

Continue onto the next step to create the Lambda function and to deploy it using the CLI.

6. CREATING AND DEPLOYING LAMBDA FUNCTION WITH AWS SAM

Great! We've made it this far. Now our original premise was that we want to know when anyone assumes the administration role for audit purposes. This is going to be introducing some new concepts and AWS services to make this happen, let's go through step by step. The first thing we are going to introduce is the AWS Serverless Application Model (SAM). While you can certainly go through and do everything in the console, AWS SAM provides a rapid way to create and deploy serverless applications with a simplified syntax for CloudFormation for common serverless building blocks.

Note: AWS SAM is an extension of CloudFormation. Anything you can do in CloudFormation you can do in SAM.

The folder Lesson 1/ Step4 in the workshop repository contains a pre-built SAM template and some source code. Let's break it down.

The below three lines are required in any SAM template. The key is the **Transform** line, which tells CloudFormation that this is in fact a SAM template that will need to be transformed with the SAM specification.

```
AWSTemplateFormatVersion: '2010-09-09'  
Transform: "AWS::Serverless-2016-10-31"  
Description: "Event Driven Security - Lesson 1 Step 4"
```

Our next set of lines is the **Globals** section. This is SAM specific functionality not available in normal CloudFormation, and it allows you to set all sorts of default information to keep your templates DRY. Learn more at <http://bit.ly/sam-globals>.

```
Globals:  
  Function:  
    Timeout: 6  
    Runtime: "python3.6"  
    # Environment:  
    # Variables:  
    #   EXAMPLE: value
```

Moving on, we have **Parameters**. Parameters is one of the building blocks of being able to add configurations and values to CloudFormation stacks and your functions without needing to edit or maintain various versions of your template everything you deploy. Parameters can power all sorts of additional conditionals in your templates, making them extremely powerful and flexible.

```
Parameters:  
  WebhookUrl:  
    Description: "What webhook should we alert. Testing? Try  
https://webhook.site/"  
    Type: "String"  
  AlertRoles:  
    Description: "Roles that would should alert when someone assumes them"  
    Type: "String"
```

In this example, we have a configurable WebhookUrl and AlertRoles. When you upload a template with parameters to the AWS Console

When working with the AWS CLI, you pass in parameters with a `--parameter-overrides` option. One of the safest ways is to wrap each Key=Value pairs with quotes. So, for example: `--parameter-overrides "Foo=Bar" "Baz=qux"`

The final bit of our template, and the most exciting part, is the **Resources** section. This is where we define all the elements of our project we actually want to create as infrastructure in AWS. In our case, because of the simplicity offered by SAM we have a single resource to define and create our Lambda function.

```
Resources:  
  AssumptionAlertFunction:  
    Type: "AWS::Serverless::Function"  
    Properties:  
      Handler: "handler.lambda_handler"
```

```
CodeUri: "./src"
Environment:
Variables:
  WEBHOOK_URL: !Ref "WebhookUrl"
  ALERT_ROLES: !Ref "AlertRoles"
```

The key lines include the **Handler**, which defines which file and “function” serves as our entry point to this functions code execution, and the **CodeUri** which defines the actual folder path (which can be relative) for CloudFormation to zip up and *package* to S3.

7. CREATING A CLOUDFORMATION PACKAGING S3 BUCKET

The code needs to be deployed to S3, and Lambda will need to know where that code is. Luckily for us, the AWS CLI can do that easily.

First, we will set up a bucket to store the code and other artifacts generated as part of the packaging process. This can either be done in the AWS console or using the CLI (`aws s3 mb s3://bucketname`). We highly recommend you standardize this bucket naming conventions across your account (and further your organization). Some combination of region and account id will guarantee a unique bucket name. For example: `s3://cloudformation-us-east-1-123456789101`

8. PACKAGING YOUR TEMPLATE AND LAMBDA FUNCTION

The AWS CLI provides an easy way to package code and write templates with references to S3 objects lambda needs.

Note: The AWS SAM CLI offers this exact same functionality, it proxies the SAM “package” and “deploy” commands to the aws cli cloudformation commands.

The command is as follows:

```
aws cloudformation package --template-file template.yaml \
                           --output-template packaged.yaml \
                           --s3-bucket \
                           cloudformation-us-east-1-123456789101
```

This will generate a `packaged.yaml` template with the `CodeUri` and other artifacts replaced with references to your S3 bucket, as well as having upload all the artifacts to the aforementioned locations.

9. DEPLOYING YOUR TEMPLATE AND LAMBDA FUNCTION

Having previously packaged your template, let's deploy it! This is a simple command; the trickiest part is remembering to define our parameters and the flag for defining them from the beginning of this step.

Tip: A quick way to get a webhook to test that your function is making an external call is <https://webhook.site>. Visit the URL and it will give you a long unique URL to plug in and show you all the requests going to that endpoint. You'll likely want to point Slack or other integration and alerting systems in the real world.

```
aws cloudformation deploy --template packaged.yaml \
    --stack-name RoleAssumptionAlert \
    --parameter-overrides \
        "WebhookUrl=https://webhook.site/foobar" \
        "AlertRoles=administrator" \
    --capabilities CAPABILITY_NAMED_IAM
```

The most surprising part here is likely the `--capabilities CAPABILITY_NAMED_IAM`. Without delving into too much detail, realize that this option is necessary for any template that creates IAM related resources. It's a bit of a failsafe so you can say "I realize this is going to create or change things that can impact the correct security access landscape of my account". While you may not see anything of the `AWS::IAM` type in this template, SAM is creating an implicit role under the hood for us.

In fact, after you deploy, go check out your CloudFormation stack in the console and you can "View processed template" to see what all was done for you!

10. TESTING OUR LAMBDA FUNCTION

The final step before we move on is a quick test to make sure we can successfully invoke our function and get output to our webhook. Navigate to the Lambda service in the AWS Console and click the **Test** button. If this is your first time testing a function, it will open a Configure test event dialog. Enter an **Event name**, and copy paste in the content of Lesson 1/Step4/test-event.json to the inline editor, your screen will look like the below. Hit **Create**.

Configure test event

A function can have up to 10 test events. The events are persisted so you can switch to another computer or web browser and test your function with the same events.

☒ Create new test event
☐ Edit saved test events

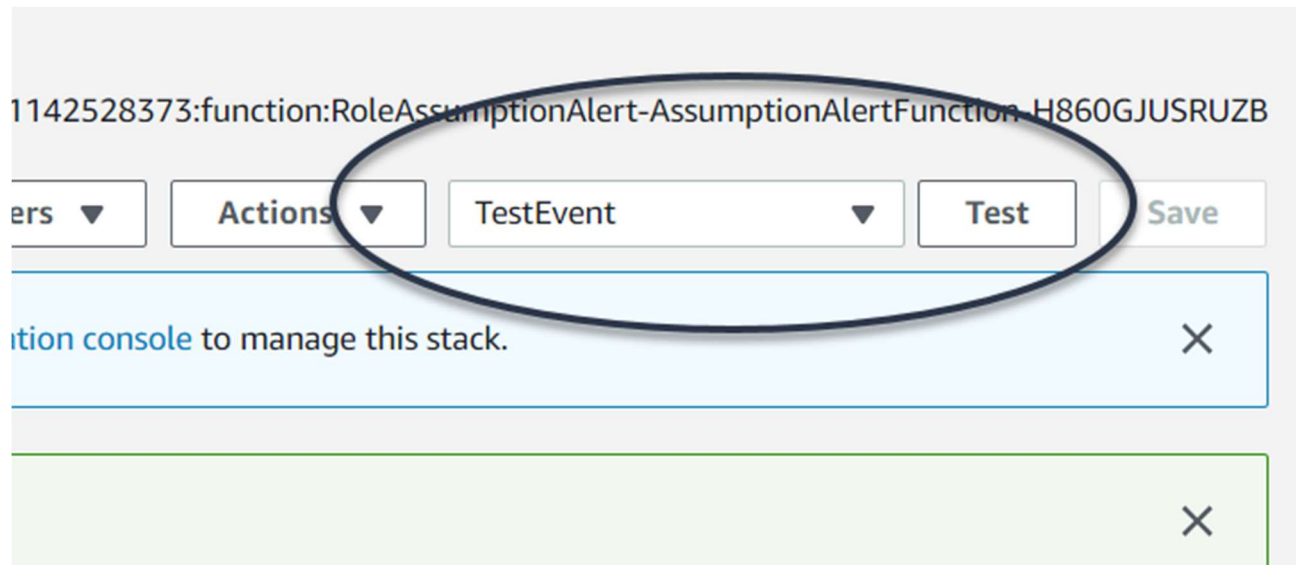
Event template
Hello World ▼

Event name
TestEvent

```
4  "detail-type": "AWS Console Sign In via CloudTrail",
5  "source": "aws.signin",
6  "account": "123456789101",
7  "time": "2018-07-12T18:58:51Z",
8  "region": "us-east-1",
9  "resources": [],
10 "detail": {
11   "eventVersion": "1.05",
12   "userIdentity": {
13     "type": "AssumedRole",
14     "principalId": "AROAI35T4DJ5RPWJUTKU:Helen",
15     "arn": "arn:aws:sts:123456789101:assumed-role/administrator/Helen",
16     "accountId": "123456789101"
17   },
18   "eventTime": "2018-07-12T18:58:51Z",
19   "eventSource": "signin.amazonaws.com",
20   "eventName": "SwitchRole",
21   "awsRegion": "global",
22   "sourceIPAddress": "66.44.45.168",
23   "userAgent": "Mozilla/5.0 (Windows NT 10.0; WOW64; rv:54.0) Gecko/20100101 Firefox/54.",
24   "requestParameters": null,
25   "responseElements": {
26     "SwitchRole": "Success"
27   },
28   "additionalEventData": {
29     "SwitchFrom": "arn:aws:iam::123456789101:user/Helen",
30     "RedirectTo": "https://console.aws.amazon.com/ec2/v2/home?region=us-east-1#Home:"
31   },
32   "eventID": "bd1ccdec-b8fb-49a2-8621-949bbdd96633",
33   "eventType": "AwsConsoleSignIn"
34 }
```

Cancel Create

Select your TestEvent in the dropdown and hit **Test**.



You should see a successful execution and content to your webhook! If you are getting error messages, try reviewing the logs in the console to see what they tell you and see if we can make fixes.

Note: If we were using the SAM CLI, it also makes it possible to test invocations locally (via docker) and pass mock events. Check the SAM CLI docs for details.

11. LOGGING API CALLS

Before you can configure your Lambda function to respond to IAM events you will need to enable CloudTrail to LOG calls to the AWS API endpoints.

- Ensuring you have the **N. Virginia** region select, open the **CloudTrail** console
- Click **Create Trail** - in the **Trail Name** box enter a suitable name - **EventDrivenSecurity** leave the other boxes as defaults (yes/yes).
- Pick a unique name for **S3 bucket** - remember this is unique in all accounts worldwide.

Once completed you've configured AWS to log all API calls - periodically stored within the S3 bucket created above.

12. TRIGGERING THE LAMBDA FUNCTION ON ROLE ASSUMPTIONS

We've created our Lambda function, tested that it works with a test event and sends data to an endpoint. Now, we want to make sure when someone switches roles our lambda function runs in response to that event.

- Ensuring you are still using the **N. Virginia** region - load the **CloudWatch** console.
- Click on **Events** to be taken to **CloudWatch Events**

CloudWatch events is an event or a rule engine. It allows you as an AWS engineer/developer to DO something, in response to certain events happening within supported services. You're going to create a rule to match an event and run your Lambda function.

- Click **Create rule** to create an event rule.
- Ensure **Event pattern** is selected as the event source.
- Change **Service Name** to **AWS Console Sign-in** and **Event Type** to **Sign-in Events** and finally make sure the **Any user** checkbox is selected.

At this point you have indicated to CloudWatch that you want to create an event source which matches a particular pattern when any Console Sign-In is detected via CloudTrail.

The next step is to inform CloudWatch that something should happen when this event occurs - you want to invoke your Lambda function and have CloudWatch pass Lambda details of that event.

- Click on **Add Target** on the right, **Lambda Function** might already be selected, if not, select it from the dropdown.
- From the **Function** drop down menu, select your Lambda function which should start with **RoleAssumptionAlert**.
- Expand **Configure Input** and note that when this event occurs, CloudWatch is sending the Lambda function the entire matched event - and that is how it will know which user is attempting to run the operation.
- Click **Configure Details** and enter a suitable name and description for the rule - since this isn't production neither especially matter - but pick something descriptive such as **SigninEvents**.
- Click **Create rule** to create the rule.

Step 1: Create rule

Create rules to invoke Targets based on Events happening in your AWS environment.

Event Source

Build or customize an Event Pattern or set a Schedule to invoke Targets.

☒ Event Pattern ⓘ ☐ Schedule ⓘ

Build event pattern to match events by service

Service Name

Event Type

☒ Any user ☐ Specific user(s) by ARN

▼ Event Pattern Preview

[Copy to clipboard](#) [Edit](#)

```
{
  "detail-type": [
    "AWS Console Sign In via CloudTrail"
  ]
}
```

Targets

Select Target to invoke when an event matches your Event Pattern or when schedule is triggered.

Lambda function

Function*

▸ Configure version/alias

▸ Configure input

[+ Add target*](#)

13. Code defined events in SAM

If you were questioning the usage of the console to configure the Sign-in Event to Lambda (or even the CloudTrail setup) give yourself a pat on the back! We wanted a little manual interaction with the console for the purposes of the workshop and exploration, but you could certainly do all of this in the SAM templates themselves.

We could update our template to contain the event defined in code like the following:

Resources:

AssumptionAlertFunction:

Type: "AWS::Serverless::Function"

Properties:

Handler: "handler.lambda_handler"

CodeUri: "./src"

Environment:

Variables:

WEBHOOK_URL: !Ref "WebhookUrl"

Events:

SignInEvent:

Type: "CloudWatchEvent"

Properties:

Pattern:

detail-type:

- "AWS Console Sign In via CloudTrail"

For more details on all the available event types in SAM, check out <http://bit.ly/sam-event-types>.

14. TESTING IT ALL TOGETHER

Open a separate browser and load the ***IAM users sign-in link*** if you didn't record this from earlier, its available from the IAM Dashboard and looks something like this:

<https://XXXXXXXXXXXXX.signin.aws.amazon.com/console>

- Login using the ***Helen*** user - you should have recorded the password you picked earlier.
- Go ahead and do the **Switch Role to administrator** that we did previously in **4 ROLE ASSUMPTION: TESTING YOUR USER**

What's happened in the background is that when you assume or Switch Roles, CloudWatch events recognized it and has invoked your Lambda function and passed it an **event** object which contains all relevant information for that call - the type of call, the AWS user performing the call etc.

Since we switched to the administrator role, our Lambda function should have run to completion and called out to our target webhook with a message.

Let's have a look and see if that is the case.

- In the window where you are logged in as your admin user.
- Open the **Lambda** console and click on your **RoleAssumptionAlert** function
- Click on the **Monitoring** tab and **View Logs in Cloudwatch**
- Pick the most recent log stream, you will see if you have more than one that they are sorted by **Last Event Time**
- Your log should look something like that below, the **Event** object passed to the Lambda function contained information identifying **Helen** as the user performing the operation.
- Your Lambda function identified that **Helen** performed a switch role to the administrator role.
- We sent an alert message to the webhook.

▶ 19:54:23	START RequestId: 5fe91e2b-860d-11e8-b455-1d783d0c1b15 Version: \$LATEST
▶ 19:54:23	[INFO] 2018-07-12T19:54:23.702Z 5fe91e2b-860d-11e8-b455-1d783d0c1b15 {"version": "0", "id": "c45c2c17-b322-8fb3-5
▶ 19:54:23	[INFO] 2018-07-12T19:54:23.703Z 5fe91e2b-860d-11e8-b455-1d783d0c1b15 user: Helen, assumed role: administrator
▶ 19:54:23	[INFO] 2018-07-12T19:54:23.703Z 5fe91e2b-860d-11e8-b455-1d783d0c1b15 sending alert notification
▶ 19:54:24	[INFO] 2018-07-12T19:54:24.420Z 5fe91e2b-860d-11e8-b455-1d783d0c1b15 message posted to webhook
▶ 19:54:24	END RequestId: 5fe91e2b-860d-11e8-b455-1d783d0c1b15

It's worth noting at this point - this is a trivial example. Because of the serverless event driven nature of Lambda, this architecture is equally suited to 10, 100, 1000 or even 1,000,000 executions per second and you could be performing much more complex logic.

And there we go! We have built and deployed a fully working "Administration" role assumption auditing and alert system, in the process we learned about AWS Lambda, AWS SAM and CloudFormation, using the AWS CLI, CloudWatch Events, and basic testing and logs usage!

Lesson 2

“Bad IPs”

In Lesson 2 you are going to create an event driven security workflow to proactively protect your website from malicious IP's - in this case your website is a static website hosted from S3.

Initially you will create a website using an S3 bucket in CloudFormation. In front of that, you'll add a CloudFront distribution.

In addition - you'll integrate a Lambda function and AWS' WAF (Web Application Firewall) to block a specific list of unsafe IPs obtained from the SANS database of potentially compromised IP's.

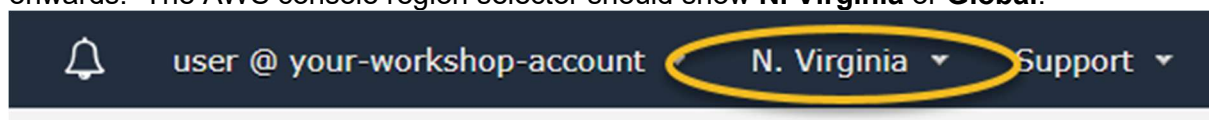
Finally, you'll link your CloudFront distribution and your WAF, dynamically updated by Lambda to produce a serverless event driven security perimeter.

NOTE: PLEASE CREATE ALL YOUR RESOURCES IN THE N. VIRGINIA REGION (US-EAST-1)

1. SET YOUR REGION TO US. EAST (N. VIRGINIA)

Log in to the AWS console, and make sure you have set your region to US East (N.Virginia).

Please make sure that all resources and services you create are in the same region from here onwards. The AWS console region selector should show **N. Virginia** or **Global**:



Note: IAM is a global service. When working in the IAM console the region will show 'Global'.

2. CREATE A STATIC WEBSITE

In this step, you're going to create a static website using S3. The bucket will be created and managed with CloudFormation and over the next few steps, we will continue to add to that template as we secure the website from known unsafe IP's.

Create a new file called website.yaml in the same directory as bad-ip.py. We will start with creating the bucket.

Resources:**SiteBucket:**Type: `AWS::S3::Bucket`**Outputs:****SiteUploadBucket:**Description: `Bucket for the website content (html, javascript, css)`Value: `!Sub s3://${SiteBucket}`

Note that we don't have any other property added. All we are saying is "create a bucket and report the name of the bucket" There are properties like `WebsiteConfiguration` but we are going to ignore those properties. While the bucket is going to host the website, the bucket won't be the endpoint people go to. With `WebsiteConfiguration` turned on, it opens the bucket up over HTTP. We are going to use CloudFront and open the bucket up over HTTPS only. This will make it so there is only a single-entry point (CloudFront instead of CloudFront AND S3).

3. CREATE YOUR CLOUDFRONT DISTRIBUTION

Let's add the CloudFront distribution next so that when we upload the website, we can view it from a URL.

We will need to do three things for the CloudFront Distribution. First you will add the distribution, and then secondly, add a policy and an origin access identity so that CloudFront will use to speak with the private bucket above. By doing it this way, and not turning the bucket into an HTTP website using `WebsiteConfiguration`, we can keep the bucket private and locked down. The third thing to do is to report the URL to the CloudFront distribution so that it is easy to find.

Within the Resources section:

SiteDistribution:Type: `AWS::CloudFront::Distribution`**Properties:****DistributionConfig:**Enabled: `true`DefaultRootObject: `index.html`**Origins:**- `DomainName: !GetAtt SiteBucket.DomainName``Id: !Sub ${AWS::StackName}-S3Origin`**S3OriginConfig:**`OriginAccessIdentity: !Sub origin-access-identity/cloudfront/${OriginAccessIdentityID}`**DefaultCacheBehavior:**`TargetOriginId: !Sub ${AWS::StackName}-S3Origin`DefaultTTL: `0`MaxTTL: `0`MinTTL: `0`**ForwardedValues:**`QueryString: 'false'`**ViewerProtocolPolicy:** `redirect-to-https`

There are two core parts of the site distribution, the origin and the default cache behavior. The origin tells CloudFront where to resolve routes to. As for the default cache behavior, we want to disable that while working on this so that it is easier to get real feedback on whether something is working or not and not having to worry that we are looking at a cached version.

```
# This policy restricts S3 bucket read access to only Cloudfront, to force
access through the CDN.
SiteBucketPolicy:
  Type: AWS::S3::BucketPolicy
  Properties:
    Bucket: !Ref SiteBucket
    PolicyDocument:
      Statement:
        - Sid: "Grant a CloudFront Origin Identity access to support private
content"
          Action:
            - "s3:GetObject"
          Effect: "Allow"
          Resource: !Sub arn:aws:s3:::${SiteBucket}/*
          Principal:
            CanonicalUser: !GetAtt OriginAccessIdentityID.S3CanonicalUserId

OriginAccessIdentityID:
  Type: AWS::CloudFront::CloudFrontOriginAccessIdentity
  Properties:
    CloudFrontOriginAccessIdentityConfig:
      Comment: Website OAI - Origin Access Identity will allow CloudFront to
speak with S3 while keeping the bucket private
```

The site bucket policy will allow CloudFront to read the contents of the bucket and the origin access identity is used in both the policy and the site distribution to tie the two together.

In the Outputs section:

```
Endpoint:
  Description: Application Address
  Value: !Sub https://${SiteDistribution.DomainName}
```

This will give you the url for the distribution. This URL can also be found in the CloudFront section of the console and it will be xxxxx.cloudfront.net in the Domain Name column.

4. CREATE YOUR WEBSITE

Now that we have the infrastructure up to be able to view a static website, let's put up the simplest website of all.

Create index.html in the same directory as website.yaml and add:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Sample Static Site</title>
</head>
<body>
  <p>Sample static website on S3</p>
</body>
</html>
```

Let's upload that to the bucket. You can either use the AWS CLI and the value found in the output SiteUploadBucket or go into the AWS console and upload the html file to the bucket through the S3 console.

The CLI command is:

```
aws s3 cp ./index.html s3://<bucket-name>
```

If you rather do it via the AWS Console:

- Open the CloudFormation Console
- Browse to S3
- Select the bucket for the site
 - If you see multiple buckets and aren't sure which to use, head over CloudFormation and check the output variable SiteUploadBucket
- Upload index.html to the bucket

You should now be able to view the site at the **Endpoint** URL.

5. CREATE THE LAMBDA FUNCTION

Now that the CLI is configured, we can create and deploy the Lambda function. The code is in the bad-ip.py and shouldn't need any modifications. You will be adding some lines to the website CloudFormation template that will set up the Lambda function and the schedule that will run the Lambda function.

Let's start by adding these two lines at the very top:

```
AWS::TemplateFormatVersion: '2010-09-09'
Transform: AWS::Serverless-2016-10-31
```

This will let the CLI know that you are using the Serverless Application Model (SAM) that is built into the AWS CLI.

We then want to add the Lambda function itself. Start by adding the infrastructure for the Lambda function in the Resources section:

```
BapIp:
  Type: AWS::Serverless::Function
  Properties:
    Handler: bad-ip.handler
    Runtime: python2.7
    Timeout: 60
    MemorySize: 1024
```

Handler is the folder, file, and function that is the entry point for when the Lambda is run. **Runtime** can be Go, Java, .NET Core, Node.js, Python, or you can upload an executable to run. **Timeout** is in seconds and we want to let this run for as long as 60 seconds before the function times out and reports back an error. **Memory size** can be as low as 128 MB and as high as 3008 MB. CPU, network bandwidth, and disk I/O are proportional to the memory size.

Now that we have the basic function in place, let's schedule the function to run once an hour. Copy the Events section and add it to the BadIp function you created in Properties.

```
UpdateBadIpList:
  Type: Schedule
  Properties:
    Schedule: rate(1 hour)
```

We now have the basic of the Lambda function set up but we still have a little bit more before we can test the function.

6. ADD WAF ACL TO CLOUDFRONT

We will need to add a Web ACL to CloudFront so that the Lambda function can do its work.

To create a Web ACL, we will need to create three new pieces of infrastructure in CloudFormation:

```
IPSet:
  Type: AWS::WAF::IPSet
  Properties:
    Name: SANS_IPs
Rule:
  Type: AWS::WAF::Rule
  Properties:
    Name: SANS Rule
    MetricName: sansRule
    Predicates:
      - DataId: !Ref IPSet
        Type: IPMatch
        Negated: 'false'
WebACL:
```



```
Type: AWS::WAF::WebACL
Properties:
  Name: WebACL
  DefaultAction:
    Type: ALLOW
  MetricName: WebACL
  Rules:
    - Action:
        Type: BLOCK
      Priority: 1
      RuleId: !Ref Rule
```

IPSet is the list of IP addresses that will be blocked. It is used by the rule (note the !Ref IPSet in Rule) and the rule is used by WebACL. Your lambda function will also need access to the IP set to be able to add and remove IP addresses.

Add these lines to the BadIp function under Properties:

```
Environment:
Variables:
  IPSET: !Ref IPSet
```

We will also need to add the policy that will allow your Lambda function to talk with WAF. Add these lines to the BadIP function under Properties as well:

```
Policies:
  Statement:
    - Effect: Allow
      Action:
        - waf:GetChangeToken
        - waf:GetChangeTokenStatus
      Resource: "*"
    - Effect: Allow
      Action:
        - waf:GetIPSet
        - waf:UpdateIPSet
      Resource: !Sub arn:aws:waf::${AWS::AccountId}:ipset/${IPSet}
```

Lastly, we need to tie the web ACL to the distribution.

Under Properties of SiteDistribution add:

```
WebACLId: !Ref WebACL
```

7. DEPLOY THE FUNCTION

Previously, we could easily deploy the CloudFormation template from the console because everything required was in the template itself. Now, because of the BadIp function and associated code, the code needs to be deployed to S3, and Lambda will need to know where that code is. Luckily for us, the AWS CLI can do that easily.

We will use the S3 bucket in Lesson 1 for storing the Lambda code. To reiterate another way, there are two buckets. One bucket will store the Lambda code, and the other bucket will store the website and that bucket will be used to serve the website to the public.

```
aws cloudformation package --template-file website.yaml --output-template-file packaged.yaml --s3-bucket cloudformation-us-east-1-123456789101
```

Package will do two things. First it will zip and upload the BadIp source code to S3 in the bucket you created. Secondly, the output template file created will have a new property called CodeUri within BadIp. The URL is the location of the code that was just uploaded to S3.

```
aws cloudformation deploy --template-file ./packaged.yaml --stack-name website --capabilities CAPABILITY_IAM
```

Make sure to deploy the packaged.yaml that was created from package. You will deploy it against the same stack as before and you need to specify `--capabilities CAPABILITY_IAM` to acknowledge that you are intentionally created an IAM role (for the Lambda function).

After **deploy** runs, you've now updated the website stack to include a Lambda function that will run once an hour that will update the IP set with the most recent CIDR ranges of bad IPs.

Note: Since we have only scheduled runs once an hour, you may just want to `Test` the lambda function (with any event) to force a first run before moving to the next step.

8. REVIEW WAF CONFIG

- Open the **WAF and Shield Console**
- Change the filter to **Global (CloudFront)** if it isn't already.
- You should see **WebACL** which is the WAF access control list which is now in use within your CloudFront distribution.
- Click on **WebACL**, and then on the **Rules** Tab.
- You should see one Rule **SANS Rule** which is the rule created by the CloudFormation template earlier.

SANS Rule

[Edit rule](#)

When a request originates from an IP address in [SANS IPs](#)

IP Addresses in SANS IPs

208.100.26.0/24

196.52.43.0/24

71.6.146.0/24

- Click **SANS Rule** and list the IP addresses in the Rule.

If everything worked as expected, you should see IP addresses in the rule. Either way, let's review the logs for the Lambda function in step 9.

9. LAMBDA FUNCTION REVIEW

- Open the **Lambda Console** and click **Functions**
- Click the **BadIp** function
- Click **Test**. Then select **Scheduled Event** from the **Event template** box. Enter **WAFEvent** in the **Event name** box. Click **Create**. Now click **Test** again to perform a test run of the Lambda function.
- Expand the **Execution result Details** and review the **Log Output** - notice how the Lambda function is obtaining the IP's from SANS and using this list to populate the WebACL created within WAF and Assigned to your CloudFront distribution.
- Click **Monitoring** then **View logs in CloudWatch** - locate the most recent log entry and click it to open the log view.

- Assuming everything is ok ... you should see log entries confirming the IP downloads and entry into the WAF WebACL.

	Time (UTC +00:00)	Message
	2017-04-07	
		<i>No older events found at the momer</i>
▶	08:03:49	Starting update...
▶	08:03:49	Found 20 CIDRs from SANS
▶	08:03:49	['80.82.70.0/24', '185.35.62.0/24', '208.100.26.0/24', '163.172
▶	08:03:50	Found 20 CIDRs from current IPSet
▶	08:03:50	[u'91.211.0.0/24', u'64.125.239.0/24', u'163.172.99.0/24', u'20
▶	08:03:50	There are 0 CIDRs to REMOVE from current IPSet
▶	08:03:50	[]
▶	08:03:50	There are 0 CIDRs to add ADD to current IPSet
▶	08:03:50	[]
▶	08:03:50	START RequestId: bb59e987-1b68-11e7-a907-3f19c6f26713
▶	08:03:50	Starting update...
▶	08:03:50	Found 20 CIDRs from SANS
▶	08:03:50	['80.82.70.0/24', '185.35.62.0/24', '208.100.26.0/24', '163.172
▶	08:03:50	Found 20 CIDRs from current IPSet
▶	08:03:50	[u'74.82.47.0/24', u'196.52.43.0/24', u'208.100.26.0/24', u'45.
▶	08:03:50	There are 0 CIDRs to REMOVE from current IPSet
▶	08:03:50	[]
▶	08:03:50	There are 0 CIDRs to add ADD to current IPSet
▶	08:03:50	[]
▶	08:03:50	END RequestId: bb59e987-1b68-11e7-a907-3f19c6f26713

Lesson 3

“Compromised Servers”

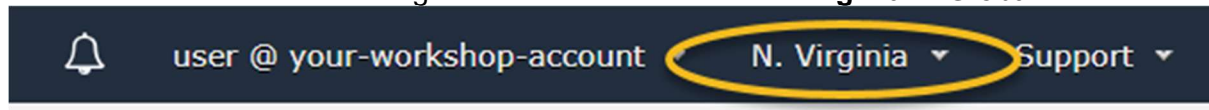
In lesson 3 you’re going to create an event driven workflow which reacts to a suspected compromised server by detecting and reacting to suspect traffic.

NOTE: PLEASE CREATE ALL YOUR RESOURCES IN THE N. VIRGINIA REGION (US-EAST-1)

1. SET YOUR REGION TO US. EAST (N. VIRGINIA)

Log in to the AWS console, and make sure you have set your region to US East (N.Virginia).

Please make sure that all resources and services you create are in the same region from here onwards. The AWS console region selector should show **N. Virginia** or **Global**:



Note: IAM is a global service. When working in the IAM console the region will show ‘Global’.

2. CREATE AN SSH KEY

In this step you’re going to create an SSH key - which can be used to access the web app instances created below if absolutely necessary.

- From the N.Virginia region, load the **EC2** Console
- Click **Key Pairs** from the left-hand menu, followed by **Create Key Pair**
- In the **Key pair name** box, enter **Compsservers** and click **Create**
- You will be asked to save an SSHKey, save it somewhere safe just in case we need it later - if all goes well, we won't.

3. CREATE YOUR BASIC WEBAPP & INFRASTRUCTURE

To begin you're going to create a small web app infrastructure. This infrastructure will consist of a VPC, containing 2 subnets, each in a different availability zone. In each of these subnets will be an initial EC2 instance, for a total of two - each connected to an elastic load balancer acting as an endpoint to your application.

As this lesson is about event driven security - rather than normal infrastructure setup in AWS you will be using a cloud formation template to create this initial basic infrastructure.

- Open the **CloudFormation** console.
- Click **Create Stack**
- Select **Upload a template to Amazon S3** and click **Browse / Choose File**
- Select the **webapp.yaml** template and click **Open & Next**
- Enter **webapp** as the Stack name, select the SSH key you created in the previous step (**CompServers**), leave other values as defaults and click **Next, Next** and **Create**

Specify Details

Specify a stack name and parameter values. You can use or change the default parameter values, which are defined in the AWS CloudFormation template. [Learn more.](#)

Stack name

Parameters

SSHKey

CompServers

SSH Key to use for EC2 instance access

SubnetAIPRange

10.0.0.0/24

IP Range to use for SubnetA

SubnetBIPRange

10.0.1.0/24

IP Range to use for SubnetB

VPCCidr

10.0.0.0/16

VPC Cidr Range to use

The stack creation process may take 5-10 minutes - but assuming everything goes well you should end up with a complete entity as below:

Create Stack ▾

Actions ▾

Design template

Filter: Active ▾ By Stack Name

	Stack Name	Created Time	Status
<input checked="" type="checkbox"/>	webapp	2017-04-09 05:45:49 UTC+1000	CREATE_COMPLETE

Events ▾

2017-04-09	Status	Type	Logical ID	Statu
▶ 05:47:50 UTC +1000	CREATE_COMPLETE	AWS::CloudFormation::Stack	webapp	
▶ 05:47:46 UTC +1000	CREATE_COMPLETE	AWS::AutoScaling::AutoScaling Group	AutoScalingGroup	
▶ 05:47:00 UTC +1000	CREATE_COMPLETE	AWS::EC2::SubnetRouteTableAssociation	SubnetRouteTableAs sociationA	
▶ 05:46:59 UTC +1000	CREATE_COMPLETE	AWS::EC2::SubnetRouteTableAssociation	SubnetRouteTableAs sociationB	

Outputs ▾

Key	Value	Description	Export Name
Endpoint	http://webapp-ELB-25FMOD3R295E-1364027572.us-east-1.elb.amazonaws.com/webapp.php	Application Address	
VPCId	vpc-e42bda9e	VPC ID to be used later by FN:...	webapp:Lesson3-VPC

Click on the dropdown triangle next to **events** and select **Outputs**

This will show your application endpoint URL - click on it to open the application (**Please Note** it may take a few minutes to work correctly - due to instance provisioning and DNS propagation) and the VPCId. The VPCId is used in step 5. **If you change the name of the stack from webapp, please note the new Export Name for the VPC Id.**

If everything is working OK - the application should display the Serverlessconf logo and a pair of IPv4 addresses - one internal, the other external.



Try refreshing a few times (*by 'few' this may mean '57,000' depending on how co-operative the ELB is being :)*), the IP's should change, there are 2 x EC2 instances serving the application behind an elastic load balancer.

This is your simple 'webapp' - clearly it does nothing but show the IP - if this were real, it could be the Serverlessconf agenda and planning application - but for this lesson's purposes its fine.

Under normal circumstances this application should service incoming requests from the internet **only**, together with a VERY LIMITED set of outgoing communications initiated from the server pair - this lesson looks at how you as a security professional can utilize a serverless event driven workflow to audit outgoing connections and notify an entity (you?) of any suspect ones.

To audit these connections, you're going to use a combination of event-driven capable products - **VPC Flow Logs**, **CloudWatch Logs** and **AWS Lambda**.

4. CONFIGURE NOTIFICATION

First up, you need a way that the Lambda function can communicate with you - notifying you when an event happens and detailing the actions taken to mitigate the risk of that event.

To do this, you're going to utilize AWS Simple Notification Service - or SNS.

Create a new file called **alerts.yaml in the same directory as webapp.yaml**. In this CloudFormation template, we will create an SNS topic and add a configurable parameter to set the email used in SNS at time of deployment. This could possibly go into the webapp.yaml but by having this in its own template, it will allow you to easily add alerts to similar stacks while only maintaining one template rather than having the alerting infrastructure in multiple CloudFormation templates.

Parameters:

VPCImportName:

Description: This name will match the exported value from webapp.yaml.

Type: String

Default: webapp:Lesson3-VPC

Email:

Description: Your email address that will be attached to the SNS Topic

Type: String

Default: you@example.com

Resources:

SNSTopic:

Type: AWS::SNS::Topic

Properties:**DisplayName:** Lesson3**Subscription:**- **Endpoint:** !Ref Email**Protocol:** email

- Open the **CloudFormation** console.
- Click **Create Stack**
- Select **Upload a template to Amazon S3** and click **Browse / Choose File**
- Select the **alerts.yaml** template and click **Open & Next**
- Enter **alerts** as the Stack name, enter your email address and click **Next, Next** and **Create**
- At this point you should get an email in your inbox asking you to confirm the subscription

Specify Details

Specify a stack name and parameter values. You can use or change the default parameter values, which are defined in the AWS CloudFormation template. [Learn more.](#)

Stack name alerts

Parameters

Email you@example.com

Your email address that will be attached to the SNS Topic

SNS uses a 'pub' / 'sub' model - topics have subscribers who receive notifications which are published to the topic. In this case, Lambda will do the publishing, and YOU will be a subscriber - using email to receive the notifications.

SNS dashboard

Topics

Applications

Subscriptions

Text messaging (SMS)

Topics

Publish to topic

Create new topic

Actions ▾

Filter alerts

<input type="checkbox"/>	Name	ARN
<input type="checkbox"/>	alerts-SNSTopic-1AM9PM...	arn:aws:sns:us-east-1:491142528373:alerts-SNSTopic-1AM9PM5UEC5RU

After the stack completes, if everything is fine, you should be able to browse to the SNS dashboard, look at topics and have a topic which looks something like above. Note that the name is **alerts-SNSTopic-random characters**. This is because we didn't name our topic. Unless required, it is generally best practice to let CloudFormation determine a name and manage the topic through CloudFormation. We will see how to load this randomly generated name into Lambda using CloudFormation in step 7.

Confirm the subscription in your email (check spam) before moving on.

5. ENABLE VPC FLOW LOGS

VPC flow logs provide 'not-quite-real-time' way of investigating traffic TO and FROM a VPC - and provides IP, PORT and protocol level granularity of that incoming and outgoing traffic. You're going to enable VPC flow logs on the entire Lesson3 VPC - and configure your Lambda function to react to those logs.

In **alerts.yaml** add the following snippets. First, we will add a **parameter** to **pass in the Export Name** from **webapp.yaml**. Please leave the email parameter's configuration alone.

```
NewLogGroupName:
  Description: The name of the log group to be created
  Type: String
  Default: EDS-Lesson3
VPCImportName:
  Description: This name will match the exported value from webapp stack
  Type: String
  Default: webapp:Lesson3-VPC
```

Now add the below snippet to the **Resources** section.

```
FlowLogs:
  Type: AWS::EC2::FlowLog
  Properties:
    DeliverLogsPermissionArn: !GetAtt FlowLogsRole.Arn
    LogGroupName: !Ref NewLogGroupName
    ResourceId:
      Fn::ImportValue: !Ref VPCImportName
    ResourceType: VPC
    TrafficType: ALL
FlowLogsRole:
  Type: AWS::IAM::Role
  Properties:
    AssumeRolePolicyDocument:
      Version: "2012-10-17"
      Statement:
        - Effect: Allow
          Principal:
            Service:
              - vpc-flow-logs.amazonaws.com
          Action: sts:AssumeRole
    Policies:
      - PolicyName: flowlogs-policy
        PolicyDocument:
          Version: "2012-10-17"
          Statement:
```

```
- Effect: Allow
  Action:
    - logs:CreateLogGroup
    - logs:CreateLogStream
    - logs:PutLogEvents
    - logs:DescribeLogGroups
    - logs:DescribeLogStreams
  Resource: "*"

```

This will consist of a flow log object and a IAM role that FlowLogs will use so that it can know what AWS permissions it has. The Flow Logs role will only need access to create and describe log groups and streams and the ability to put log events into CloudWatch Logs.

DeliverLogsPermissionArn is what ties the flow log to the role **FlowLogsRole**. The log group name EDS-Lesson3 is where the logs will live that will be generated from flow logs.

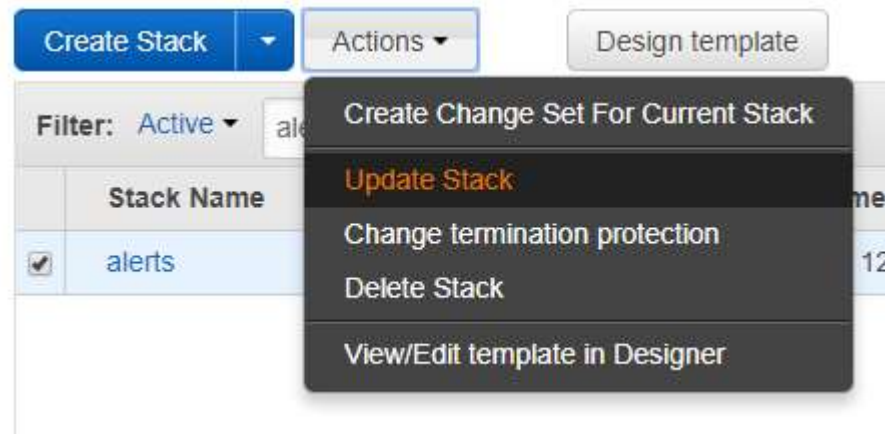
Let's unpack what is going on in the **ResourceId** section. There is a **!Ref** being used on the parameter **VPCImportName**. When CloudFormation runs, the **Ref** intrinsic function will resolve, putting the text from the parameter in it's place, and then **Fn::ImportValue** will run with **webapp:Lesson3-VPC**. As long as that matches the export name from the webapp stack, the VPC ID that was generated in the webapp stack will now be imported here and used as the resource id.

ResourceType is set to VPC because the **ResourceId** provided is a VPC Id. **TrafficType** is set to ALL so that Lambda will be exposed to accepted or rejected traffic.

FlowLogsRole is the boiler plate required to allow FlowLogs to do what it needs to do. If you were to create this flow log manually, the console would ask if you would like to make a role with these permissions.

Please note, because an IAM role is being created, there is now an extra step in running the CloudFormation template.

Let's select the alerts stack in CloudFormation and run update.



- Open the **CloudFormation** console.

- Select the **alerts** stack
- Click **Actions > Update Stack**
- Select **Upload a template to Amazon S3** and click **Browse / Choose File**
- Select the **alerts.yaml** template and click **Open & Next**
- If you didn't rename the webapp stack, leave the parameters as default, otherwise set VPCImportName to the Export Name as seen when selecting the webapp in the CloudFormation console.
- Click **Next, Next, acknowledge that you are creating an IAM resource,** and **Update**

Capabilities



The following resource(s) require capabilities: [AWS::IAM::Role]

This template contains Identity and Access Management (IAM) resources that might provide entities access to make changes to your AWS account. Check that you want to create each of these resources and that they have the minimum required permissions. [Learn more.](#)

☒ I acknowledge that AWS CloudFormation might create IAM resources.

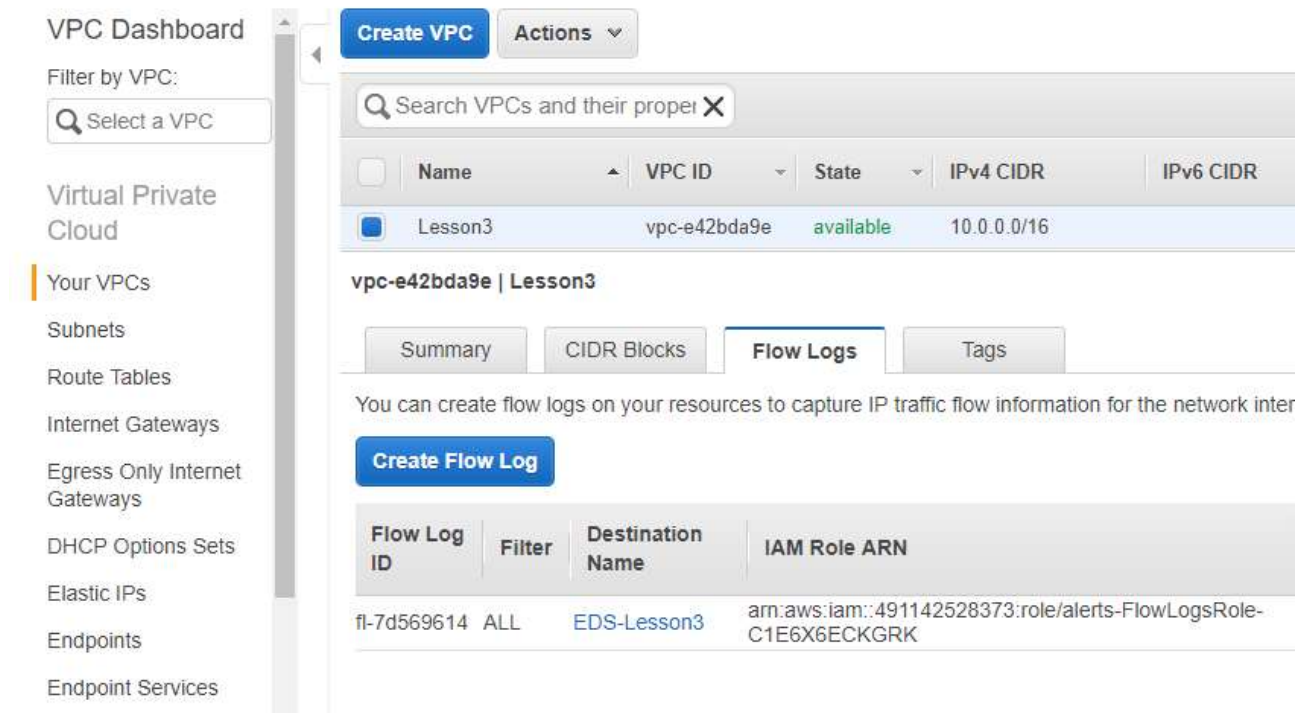
Preview your changes

Based on your input, CloudFormation will change the following resources. For more information, choose [View change set details](#).

Action	Logical ID	Physical ID	Resource type	Replacement
Add	FlowLogs		AWS::EC2::FlowLog	
Add	FlowLogsRole		AWS::IAM::Role	

[Cancel](#)[Previous](#)[Update](#)

When complete, you should have an active flow log attached your the Lesson 3 VPC. This means that metadata for any incoming and outgoing traffic will be captured within CloudWatch logs.



The screenshot shows the AWS VPC Dashboard. On the left is a navigation menu with options like 'Virtual Private Cloud', 'Your VPCs', 'Subnets', 'Route Tables', 'Internet Gateways', 'Egress Only Internet Gateways', 'DHCP Options Sets', 'Elastic IPs', 'Endpoints', and 'Endpoint Services'. The main area shows a table of VPCs with columns: Name, VPC ID, State, IPv4 CIDR, and IPv6 CIDR. One VPC, 'Lesson3' (vpc-e42bda9e), is highlighted. Below the table, there's a section for 'vpc-e42bda9e | Lesson3' with tabs for 'Summary', 'CIDR Blocks', 'Flow Logs', and 'Tags'. The 'Flow Logs' tab is active, showing a 'Create Flow Log' button and a table of flow logs. The table has columns: Flow Log ID, Filter, Destination Name, and IAM Role ARN. One flow log is listed with ID 'fl-7d569614', filter 'ALL', destination 'EDS-Lesson3', and IAM Role ARN 'arn:aws:iam::491142528373:role/alerts-FlowLogsRole-C1E6X6ECKGRK'.

Right click on the **EDS-Lesson3** link in the Destination Name column and open in a new tab - this will open the log group for this flow log.

Every Elastic Network Interface or **ENI** captured by the flow log will have its own **Log Stream** in this **EDS-Lesson3** log group - meaning you can parse logs for a specific ENI using a log stream or aggregate across all ENIs by scoping to the Log Group.

You will see anywhere from zero to four log streams when you look - and these are two EC2 instances and the listener interfaces for the elastic load balancer.

NOTE: It can take up to 10 minutes for things to show up in the logs. Amazon does a capture window.

```

▼ 20:38:02      2 031722217824 eni-98a46b6f 10.0.0.234 10.0.0.32 80 60305 6 2 112 1491683882 1491683939 ACCEPT OK
2 031722217824 eni-98a46b6f 10.0.0.234 10.0.0.32 80 60305 6 2 112 1491683882 1491683939 ACCEPT OK

```

Feel free to open a specific log stream and have a look inside...

...it will look something like above - showing account ID, ENI ID, source and destination IP's, ports, protocols as well as number of bytes, start and end times and whether the data was ACCEPTED or REJECTED.

6. CREATE THE LAMBDA FUNCTION

Let's create and deploy the Lambda function. The code is in the `instance_killer` folder and shouldn't need any modifications. You will be adding some lines to the alerts CloudFormation template that will set up the Lambda function, the trigger that kick off the Lambda function, the environment variable so that the code knows the SNS topic, and a role with restricted permissions for that function to use.

Let's start by adding these two lines at the very top:

```
AWSTemplateFormatVersion: '2010-09-09'
```

```
Transform: AWS::Serverless-2016-10-31
```

```
Parameters:
```

```
...
```

This will let the CLI know that you are using the Serverless Application Model (SAM) that is built into the AWS CLI.

Next, we want to add a parameter to pass in the subscription filter pattern. If you did not use the default VPC CIDR range in the web app CloudFormation template, please adjust the `10.0.*` in Default as makes sense. Add the **SubscriptionFilterPattern** into the **Parameters** section.

```
SubscriptionFilterPattern:
```

```
  Description: Amazon VPC Flow Logs Filter Pattern to track OUTGOING traffic  
(instance -> internet) from the VPC (10.0.*)
```

```
  Type: String
```

```
  Default: '[version, account_id, interface_id, srcaddr != "-", dstaddr !=  
"10.0.*", srcport != "-", dstport != "-", protocol, packets, bytes, start, end,  
action, log_status]'
```

We then want to add the Lambda function itself. Start by adding the infrastructure for the Lambda function in the **Resources** section:

```
InstanceKiller:
```

```
  Type: AWS::Serverless::Function
```

```
  Properties:
```

```
    Handler: instance_killer/instance_killer.lambda_handler
```

```
    Runtime: python2.7
```

```
    Timeout: 60
```

```
    MemorySize: 1024
```

Handler is the folder, file, and function that is the entry point for when the Lambda is run.

Runtime can be Go, Java, .NET Core, Node.js, Python, or you can upload an executable to run. **Timeout** is in seconds and we want to let this run for as long as 60 seconds before the function times out and reports back an error. **Memory size** can be as low as 128 MB and as high as 3008 MB. CPU, network bandwidth, and disk I/O are proportional to the memory size.

Now that we have the basic function in place, let's wire the function up to trigger when a specific text pattern is seen within the flow logs log group. **Copy the Events section and add it to the Properties of the InstanceKiller function you created.**


```
InstanceKiller:
  Type: AWS::Serverless::Function
  Properties:
    ...
    Events:
      FlowLogLogged:
        Type: CloudWatchLogs
        Properties:
          LogGroupName: !Ref NewLogGroupName
          FilterPattern: !Ref SubscriptionFilterPattern
```

Add Environment at the same level as Events.

```
Environment:
  Variables:
    SNSARN: !Ref SNSTopic
```

SNSARN is an environment variable that gets loaded into the Lambda's operating system. The **Ref intrinsic function** will allow you to get the ARN of the SNS topic that was created in step 4.

At the **same level as Environment and Events**, add Policies:

```
Policies:
  Statement:
    - Effect: Allow
      Action:
        - ec2:DescribeNetworkInterfaces
        - ec2:DescribeVolumes
        - ec2:CreateSnapshot
        - ec2:StopInstances
        - ec2:TerminateInstances
      Resource: "*"
    - Effect: Allow
      Action:
        - sns:Publish
      Resource: "*"

```

The Policies section will let your Lambda function know what AWS services it can access. By default, access to the different services are denied. This policy will allow the Lambda function to do those six specific actions. Those actions are needed by the `instance_killer.py` Python script.

As a bonus exercise, can you scope the statement down to the specific resources instead of using a wildcard?

After everything has been added, it should look like:

```
InstanceKiller:
  Type: AWS::Serverless::Function
  Properties:
    Handler: instance_killer/instance_killer.lambda_handler
    Runtime: python2.7
    Timeout: 60
    MemorySize: 1024
    Events:
      FlowLogLogged:
        Type: CloudWatchLogs
        Properties:
          LogGroupName: !Ref NewLogGroupName
          FilterPattern: !Ref SubscriptionFilterPattern
    Environment:
      Variables:
        SNSARN: !Ref SNSTopic
    Policies:
      Statement:
        - Effect: Allow
          Action:
            - ec2:DescribeNetworkInterfaces
            - ec2:DescribeVolumes
            - ec2:CreateSnapshot
            - ec2:StopInstances
            - ec2:TerminateInstances
          Resource: "*"
        - Effect: Allow
          Action:
            - sns:Publish
          Resource: "*"

```

7. DEPLOY THE FUNCTION

Previously, we could easily deploy the CloudFormation template from the console because everything required was in the template itself. Now, because of the `instance_killer` function and associated code, the code needs to be deployed to S3, and Lambda will need to know where that code is. Luckily for us, the AWS CLI can do that easily.

First, we will set up a bucket to store the code. This can either be done in the AWS console or using the CLI (`aws s3 mb s3://bucketname`).

After that, we need to use the CLI and we will run the package command.

```
aws cloudformation package --template-file alerts.yaml --output-template-file packaged.yaml --s3-bucket cloudformation-us-east-1-123456789101
```

Package will do two things. First it will zip and upload the `instance_killer` source code to S3 in the bucket you created. Secondly, the output template file created will have a new property called `CodeUri` within `InstanceKiller`. The url is the location of the code that was just uploaded to S3.

```
aws cloudformation deploy --template-file packaged.yaml --stack-name alerts --parameter-overrides 'SubscriptionFilterPattern=[version, account_id, interface_id, srcaddr != "-", dstaddr != "10.0.*", srcport != "-", dstport != "-", protocol, packets, bytes, start, end, action, log_status]' --capabilities CAPABILITY_IAM
```

Make sure to deploy the packaged.yaml that was created from package. You will deploy it against the same stack as before and you need to specify `--capabilities CAPABILITY_IAM` to acknowledge that you are intentionally created an IAM role (for the Lambda function).

After **deploy** runs, you've now updated the alerts stack to include a Lambda function that will kill EC2 instances when they try to speak out to the internet.

8. TESTING ALL THE THINGS

The log pattern filter and Lambda function is designed to identify any traffic which doesn't match a few scenarios.

1. Destination address of `10.0.*` is OK - this is generally ELB -> EC2 instance
2. Destination address of `!=10.0.*` is NOT FINE - this could be direct access to the EC2 instance. (normally this would be prevented via a SG ... but to illustrate the lesson, its an easy one to use)
3. Destination address of `!=10.0.*` is NOT FINE - this could be EC2->INTERNET

Browse to the DNS name of the ELB that you noted down earlier and refresh it 10-20 times... this should NOT cause any invocation of the Lambda function - as its case #1 above.

- Open the **EC2** console - click **Instances** and pick one of the instances with the **CompServers Key Name**, it doesn't matter which one.
- Select the instance, note down the **Instance ID** and the **IPv4 Public IP**
- In a new browser tab, browse to <http://IPv4PublicIP/webapp.php>
- The page should load - you will notice its the same format as the page served by the load balancer - that's because it IS the same page/EC2 instance - only accessed directly.
- Open the **CloudWatch** console, click **Logs**, click on the **instance_killer** log group and open the latest log by time stamp in a new tab. (Note that the actual log you want might not be the latest one, but start with the most recent and go back from there)

You should be able to step through, and see that the traffic from YOUR IP, direct to the EC2 instance was flagged as suspicious - and it resulted in instance termination. If you go back to the EC2 console after a little while you'll see that the instance you connected to is terminated, and that a new one is created in its place by the ELB.

▶ 00:25:37	LOG: Instance: i-06d86ee4eadd10bc5 Interface: eni-5eb0aea9 SrcAddr: 10.0.0.218 DstAddr: 203.220.103.228 DstPort: 53017
▶ 00:25:37	LOG: ALERT!! Disallowed traffic 203.220.103.228:53017 by instance i-06d86ee4eadd10bc5
▶ 00:25:37	LOG: Instance: i-06d86ee4eadd10bc5 Interface: eni-5e60aea9 SrcAddr: 10.0.0.218 DstAddr: 203.220.103.228 DstPort: 53025
▶ 00:25:38	LOG: ALERT!! Disallowed traffic 203.220.103.228:53025 by instance i-06d86ee4eadd10bc5
▶ 00:25:38	LOG: Instance: i-06d86ee4eadd10bc5 Interface: eni-5e60aea9 SrcAddr: 10.0.0.218 DstAddr: 203.220.103.228 DstPort: 53016
▶ 00:25:38	LOG: ALERT!! Disallowed traffic 203.220.103.228:53016 by instance i-06d86ee4eadd10bc5
▶ 00:25:38	LOG: Instance: i-06d86ee4eadd10bc5 Interface: eni-5e60aea9 SrcAddr: 10.0.0.218 DstAddr: 203.220.103.228 DstPort: 53022
▶ 00:25:38	LOG: ALERT!! Disallowed traffic 203.220.103.228:53022 by instance i-06d86ee4eadd10bc5
▶ 00:25:39	LOG: Instance: i-06d86ee4eadd10bc5 Interface: eni-5e60aea9 SrcAddr: 10.0.0.218 DstAddr: 203.220.103.228 DstPort: 53019
▶ 00:25:39	LOG: ALERT!! Disallowed traffic 203.220.103.228:53019 by instance i-06d86ee4eadd10bc5
▶ 00:25:39	LOG: Instance: i-06d86ee4eadd10bc5 Interface: eni-5e60aea9 SrcAddr: 10.0.0.218 DstAddr: 203.220.103.228 DstPort: 53023
▶ 00:25:39	LOG: ALERT!! Disallowed traffic 203.220.103.228:53023 by instance i-06d86ee4eadd10bc5
▶ 00:25:39	LOG: Instance: i-06d86ee4eadd10bc5 Interface: eni-5e60aea9 SrcAddr: 10.0.0.218 DstAddr: 203.220.103.228 DstPort: 53018
▶ 00:25:39	LOG: ALERT!! Disallowed traffic 203.220.103.228:53018 by instance i-06d86ee4eadd10bc5
▶ 00:25:39	LOG: Instance: i-06d86ee4eadd10bc5 Interface: eni-5e60aea9 SrcAddr: 10.0.0.218 DstAddr: 203.220.103.228 DstPort: 53028
▶ 00:25:39	LOG: ALERT!! Disallowed traffic 203.220.103.228:53028 by instance i-06d86ee4eadd10bc5
▶ 00:25:40	LOG: Instance: i-06d86ee4eadd10bc5 Interface: eni-5e60aea9 SrcAddr: 10.0.0.218 DstAddr: 203.220.103.228 DstPort: 53021
▶ 00:25:40	LOG: ALERT!! Disallowed traffic 203.220.103.228:53021 by instance i-06d86ee4eadd10bc5
▶ 00:25:40	LOG: Instance: i-06d86ee4eadd10bc5 Interface: eni-5e60aea9 SrcAddr: 10.0.0.218 DstAddr: 203.220.103.228 DstPort: 53020
▶ 00:25:40	LOG: ALERT!! Disallowed traffic 203.220.103.228:53020 by instance i-06d86ee4eadd10bc5
▶ 00:25:40	LOG: Instance: i-06d86ee4eadd10bc5 Interface: eni-5e60aea9 SrcAddr: 10.0.0.218 DstAddr: 203.220.103.228 DstPort: 53029
▶ 00:25:40	LOG: ALERT!! Disallowed traffic 203.220.103.228:53029 by instance i-06d86ee4eadd10bc5
▶ 00:25:40	LOG: Instance: i-06d86ee4eadd10bc5 Interface: eni-5e60aea9 SrcAddr: 10.0.0.218 DstAddr: 203.220.103.228 DstPort: 53026
▶ 00:25:40	LOG: ALERT!! Disallowed traffic 203.220.103.228:53026 by instance i-06d86ee4eadd10bc5
▶ 00:25:40	LOG: Instance: i-06d86ee4eadd10bc5 Interface: eni-5e60aea9 SrcAddr: 10.0.0.218 DstAddr: 203.220.103.228 DstPort: 53027
▶ 00:25:41	LOG: ALERT!! Disallowed traffic 203.220.103.228:53027 by instance i-06d86ee4eadd10bc5
▶ 00:25:41	LOG: There are 1 instances on the kill list!
▶ 00:25:41	LOG: Killing instance i-06d86ee4eadd10bc5
▶ 00:25:41	LOG: Sending stop message to instance. i-06d86ee4eadd10bc5
▶ 00:25:41	LOG: Snapshot for instance i-06d86ee4eadd10bc5 volume vol-06430ce9bb6ef26d7 snapshot snap-0f88c61e22422873e
▶ 00:25:41	LOG: Sending terminate message to instance. i-06d86ee4eadd10bc5

N.b you could also see direct traffic from other sources cause the Lambda function to terminate instances....and this is (as far as the scope of this lesson goes) by design.

You can also SSH to the public IP of an instance ... and from that instance attempt to initiate outbound communications - ping an IP, install curl and download HTTP from a website (amazon.com) all of this is regarded as suspicious and will cause instance termination.

```
ssh ec2-user@publicipv4 -i CompServers.pem
```

All of the above are extreme examples designed to illustrate that the function works - in production you would likely be a lot more rigorous in how you identify suspicious traffic.

Lesson 4

“Bad Config”

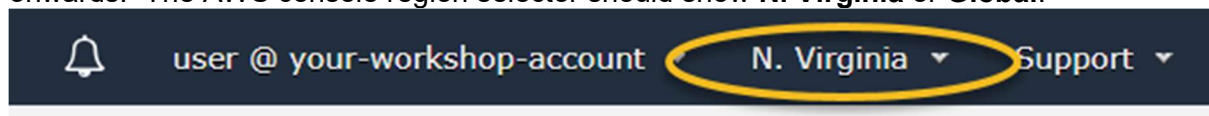
In this lesson you will create an event driven workflow to evaluate configuration of certain AWS resources in near real-time. You'll evaluate the configuration of some Lambda functions and their associated VPC security groups using AWS Config backed by AWS Lambda and custom rules.

NOTE: PLEASE CREATE ALL YOUR RESOURCES IN THE N. VIRGINIA REGION (US-EAST-1)

1. SET YOUR REGION TO US. EAST (N. VIRGINIA)

Log in to the AWS console, and make sure you have set your region to US East (N.Virginia).

Please make sure that all resources and services you create are in the same region from here onwards. The AWS console region selector should show **N. Virginia** or **Global**:



Note: IAM is a global service. When working in the IAM console the region will show ‘Global’.

2. CREATE THE FUNCTIONS USED WITHIN THE LESSON

Let's begin by creating a VPC with four Lambda functions which will be the subject of our evaluations within this lesson.

- Open the **CloudFormation** console.
- Click **Create Stack**
- Select **Upload a template to Amazon S3** and click **Browse / Choose File**
- Select the **functions.yaml** template and click **Open & Next**
- Enter **Lesson4** as the **Stack name** and click **Next, Next**
- Check the box saying **'I acknowledge that AWS CloudFormation might create IAM resources'** and click **Create**
- Wait for CloudFormation to create your stack (this will take a few minutes)

If the process completes successfully you should have a stack with a **CREATE_COMPLETE** status, as below:

Create Stack ▾ Actions ▾ Design template

Filter: **Active** ▾ By Stack Name

	Stack Name	Created Time	Status	Description
<input checked="" type="checkbox"/>	Lesson4	2017-04-09 11:31:21 UTC+1000	CREATE_COMPLETE	


Overview Outputs Resources **Events** Template Parameters Tags Stack Policy Change Sets


2017-04-09	Status	Type	Logical ID
▶ 11:33:21 UTC+1000	CREATE_COMPLETE	AWS::CloudFormation::Stack	Lesson4
▶ 11:33:18 UTC+1000	CREATE_COMPLETE	AWS::EC2::Instance	EC24
▶ 11:33:18 UTC+1000	CREATE_COMPLETE	AWS::EC2::Instance	EC21
▶ 11:33:10 UTC+1000	CREATE_COMPLETE	AWS::EC2::Instance	EC22
▶ 11:33:05 UTC+1000	CREATE_COMPLETE	AWS::EC2::Instance	EC23

3. REVIEW Lambda Functions

- Open the **Lambda** Console and click **Functions**
- You should have four new functions (as shown below)

Lambda > Functions

Functions (41)  Actions ▾ Create function

Q Filter by tags and attributes or search by keyword 

	Function name	Description	Runtime	Code size	Last Modified
<input type="radio"/>	test-badconfig-3		Python 2.7	190 bytes	41 seconds ago
<input type="radio"/>	test-badconfig-4		Python 2.7	190 bytes	41 seconds ago
<input type="radio"/>	test-badconfig-2		Python 2.7	190 bytes	41 seconds ago
<input type="radio"/>	test-badconfig-1		Python 2.7	190 bytes	41 seconds ago

- Select each function in turn and scroll down to the **Network** section to see that each function has one security group. Two functions, bad-config-1 and bad-config-4, share a security group.
- To quickly view the security group rules associated with each Lambda function, scroll down in the **Network** section to the “inbound rules”/“outbound rules” table:

Inbound rules | Outbound rules

Security group ID	Ports	Source
sg-c8bbb583	80	0.0.0.0/0
sg-c8bbb583	22	1.2.3.4/32

- Some SG's may have ***tcp/80*** inbound from ***0.0.0.0/0***. That's fine and is allowed by our corporate security policy.
- Some SG's may have ***tcp/22*** inbound from a specific /32 IP – that's also allowed by policy.
- Some SG's may have ***tcp/22*** inbound from ***0.0.0.0/0***. That's NOT allowed – it's a breach of policy
- Some SG's may have ***tcp/8080*** inbound from ***0.0.0.0/0***. That's NOT allowed – it's an internal management port and access to it from the internet is a breach of security policy.
- Become familiar with which functions have which security configuration from above.

4. THE PROBLEM

The problem that this lesson hopes to address is that if you have 100's or even 1000's of Lambda functions - together with any form of CI/CD system, you might not be able to keep track of the compliance of your resources to a security policy set by the business. We're going to configure a custom event driven way to evaluate resources against this policy, and highlight any issues which occur.

5. LAMBDA AND AWS Config

In this section you're going to configure AWS Config and create a Lambda backed custom rule. This is going to be used to evaluate any security group changes - but do so with a focus on the Lambda functions which utilize those security groups.

- Open the ***AWS Config*** console and if required click on ***Get Started***
- You might need to configure AWS Config initially - if so follow the steps below:
 - Tick ***Record all resources supported in this region*** and ***Include Global Resources***
 - Ensure the option to ***Create a bucket*** is selected.
 - Select ***Stream configuration changes and notifications to an Amazon SNS topic***
 - Select ***Create a topic***
 - Select ***Create a role***
 - Click ***Next*** followed by ***Skip*** and then ***Confirm*** to complete the initial setup of AWS Config

At this point AWS config is gathering information on the resources within the account and global services which are terminated from a logging perspective in us-east-1.


To look at the resources identified by ***AWS Config***:



- Click ***Resources***
- Change ***Resources*** to ***Lambda:Function*** & ***EC2: Security Group***

Lambda: Function

☐ Include deleted resources

Look up

Choose Config timeline  to view a history of configuration details for the resource.

Resource type	Config timeline 	Compliance	Manage resource
Lambda Function	test-badconfig-1	--	

- Click **Look up**

You should see at least four Lambda functions and at least four security groups created by the CloudFormation template you applied as part of this lesson. Note that none of the resources are being evaluated and so have no compliance – that's because we have no rules.

Next, we'll create a custom rule and supporting Lambda function which will evaluate the VPC Lambda functions for compliance on organizational security policy based on the configuration of the security groups which they utilize.

- Click **Rules, Add Rule** and **Add custom rule** - do take a moment to read the other rule types - as you will see there are some standard (non Lambda backed ones).
- Under **Name** enter **Badconfig_rule**
- You don't yet have a Lambda function, so click **Create AWS Lambda function**
- This will open a new tab, and take you to the blueprint screen
- Click **Author from scratch**.
- In the **Name** box, enter **badconfig_Lambda**
- In the **Role** dropdown select **Create a custom role**
- This will open a new tab which will create a new role.
- Change the **Role Name** contents to **badconfig_Lambda_role**
- Click the reveal triangle to open **View policy Document**
- Click **Edit** and accept any warning dialogue
- Replace the contents of the policy document box with the contents of **Lambda_role_policy.json**
- Click **Allow** to create the Role and be taken back to the Lambda tab.
- Click **Create function**.
- Select **Python 2.7** in the Runtime dropdown
- Change **Handler** to be **index.lambda_handler**
- Replace the contents of the code box, with the contents of the **badconfig_Lambda.py** file
- In **Basic Settings** change the **Timeout** value to **1 min 0 sec**.
- Scroll back to the top and click **Save** to complete the creation of your Lambda function.

ARN - arn:aws:lambda:us-east-1:031722217824:function:badconfig_lambda

- Copy the **ARN** from the top right of the screen, it should look something like:
- You need the full line minus the initial '**ARN -** ', so 'arn:aws:lambda.....'
- Copy that, and go back to the **AWS Config** Tab

AWS Lambda function ARN*

arn:aws:lambda:us-east-1:031722217824:function:badconfig_lambda

[Edit AWS Lambda function](#)

- Paste the ARN into the **AWS Lambda function ARN** box
- Select **Configuration Changes** in the **Trigger Type** section
- Ensure **Scope of Changes** is set to **Resources**
- Change **Resources** to include **Lambda::Function** and **EC2: Security Group**

Resources*

Lambda: Function x

EC2: SecurityGroup x

- Finally in the Rule parameters section, configure as follows, with port1 and port2 values:

Key	Value
port1	22
port2	8080

- Click **Save** and the rule will be saved, and the rule will be set to **Evaluating**
- This may take a short while, while the Lambda function evaluates the resources and applies a compliance status. You may need to click the 'refresh' button to check on the status of the evaluations.

[+ Add rule](#)

Rule name	Compliance
Badconfig_rule	Evaluating...

After a short while you should see some compliance issues:

[+ Add rule](#)

Rule name	Compliance
Badconfig_rule	3 noncompliant resource(s)

Click on **Badconfig_rule** for additional information:

Resource type	Config timeline	Compliance
Lambda Function	test-badconfig-1	Noncompliant with 1 rule
Lambda Function	test-badconfig-2	Noncompliant with 1 rule
Lambda Function	test-badconfig-3	Compliant
Lambda Function	test-badconfig-4	Noncompliant with 1 rule

The rule shows that 3 of 4 functions are in non-compliant states - and we can use the expand triangle to reveal additional information as to why. Do that with all four functions.

Resource type	Config timeline	Compliance
Lambda Function	test-badconfig-1	Noncompliant
Annotation Function has non compliant groups sg-8c868fc7		
Lambda Function	test-badconfig-2	Noncompliant
Annotation Function has non compliant groups sg-019a934a		
Lambda Function	test-badconfig-4	Noncompliant
Annotation Function has non compliant groups sg-8c868fc7		

Next, you'll look at why those functions are failing the compliance checks.

6. REVIEWING NON-COMPLIANCE

The AWS Config rule shows that while you have three compliance issues. The issues are caused by only two noncompliant security groups. So let's take a look at those.

- Note down the **Security Group ID's** of the noncompliant groups on YOUR rule.
- In a new Tab, open the **EC2** Console
- On the left menu locate and click **Security Groups**
- For each of the security groups listed as noncompliant, note the name tag of that group (should be.
- Click on the **Inbound** tab for the security group and review the rules.
- Remember - organization policy dictates a number of things
 1. That port 22 shouldn't be globally accessible.
 2. That port 8080 is an internal management port - and shouldn't be accessible from the outside internet.

- Click **Edit** to edit the security group and change any **SSH** line items to have a **Source** of 1.2.3.4/32 and any **Custom TCP Rule** line items where the port range is 8080 - change the **Source** to be 1.2.3.4/32

Edit inbound rules

Type ⓘ	Protocol ⓘ	Port Range ⓘ	Source ⓘ
HTTP	TCP	80	Custom 0.0.0.0/0
SSH	TCP	22	Custom 1.2.3.4/32

Edit inbound rules

Type ⓘ	Protocol ⓘ	Port Range ⓘ	Source ⓘ
HTTP	TCP	80	Custom 0.0.0.0/0
Custom TCP Rule	TCP	8080	Custom 1.2.3.4/32

- Once done, go back to the tab with your **AWS Config** loaded.
- Click **Rules**
- You may see you still have 3 noncompliant resources... if so , just wait a few minutes - utilise the refresh button.

Rule name	Compliance
Badconfig_rule	Compliant



- After a few minutes ... without you having to force an update.. you should notice that the rule compliance status updates to '**Compliant**'

What's happened, is that in the background, **AWS Config** has detected a change to resources it supports - the rule you created matches changes to Lambdas and Security Groups... groups you just edited removing the public port exposure.


AWS config uses Lambda as a backing for this custom rule - so Lambda is invoked by AWS Config.


It determines that the public exposure has been removed .. and updates the compliance status - all in an event driven, serverless way.

- Click the **Badconfig_rule** for additional information.
- You can see - there will be a recent **Successful evaluation** date - and your resources are now all compliant.

Overall rule status Last successful invocation on July 9, 2018 at 2:29:33 PM 
Last successful evaluation on July 9, 2018 at 2:29:34 PM 

Resources evaluated

Click on the  icon to view configuration details for the resource when it was last evaluated with this rule.

Resource type	Config timeline 	Compliance
▼ Lambda Function	test-badconfig-4	Compliant

Annotation

This resource is compliant with the rule.

- (optionally) If you want to test further ... return the SG's to the way they were ... 0.0.0.0/32 on 22 and 8080.

If you want to review what Lambda is doing:

- Open the **Lambda Console** and click **Functions**
- Click **Badconfig_Lambda** and select the **Monitoring** tab
- here you can see successful invocations and errors
- Click **View Logs in CloudWatch** to move across to the logs stored in the **CloudWatch** console.
- Open any of the **Log Streams** to see the changes between COMPLIANT and NON-COMPLIANT.

7. WORKSHOP TEARDOWN

During the course of this workshop we've created a lot of AWS resources, all of which you could be charged for. The easiest, and recommended, way to teardown all of these resources is to **close the account** that you used during this workshop, assuming that you created a dedicated account for this activity.

If you are unable to close the account here are a list of resources you should examine - delete / tear these down as you need:

- Delete IAM User Helen and AdminCLI if you created it
- Delete all CloudWatch Event Rules Manually created (SigninEvents)
- Empty all s3 buckets
- SNS Subscriptions (alert topic)
- Teardown all CloudFormation Stacks (alerts, webapp, your-site, RoleAssumptionAlert, AdminIAM)
- Disable CloudTrail and delete related bucket
- Delete Leftover s3 buckets
 - config-bucket
 - cloudtrail
- Delete IAM Role
 - badconfig_Lambda
 - Config-role-us-east-1
- Delete IAM Policies
 - Config-role-us-east-1-AWSConfigDeliveryPermissions
- SNS Topics
 - config-topic