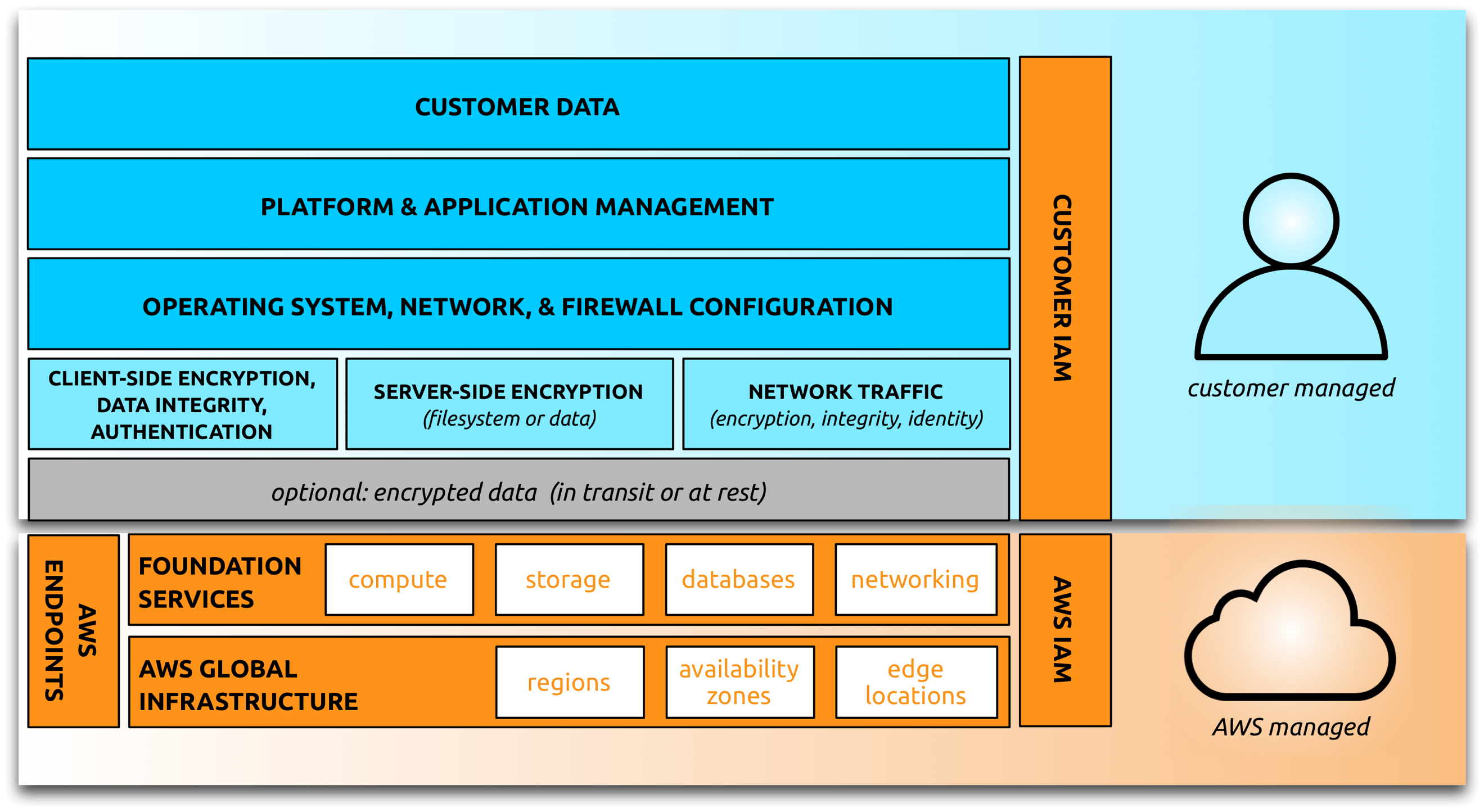
# Chapter 3. Access Management and Security Groups

In all of the previous examples, we have been using access keys that have root-level access to our AWS accounts. This means they can perform any action—including actions that potentially cost thousands of dollars in resource fees—through a few simple API calls. The thought of your AWS keys leaking should be a scary one indeed, so now is a good time to look at some of the tools Amazon provides to securely deploy and manage your applications.[1](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch03.html#idm140367897945616)

# The AWS Security Model

AWS infrastructure services rely on a [shared responsibility model](http://media.amazonwebservices.com/AWS_Security_Best_Practices.pdf) for security. Unlike in the traditional datacenter, where the full responsibility for the environment’s security falls squarely on the IT team, EC2 customers share this burden with the AWS team in significant ways ([Figure 3-1](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch03.html#respo_model)).

In this shared responsibility model, the user owns the operating system’s login credentials but AWS bootstraps initial access to that same operating system. The end user may or may not have administrative control of the provisioning process and a separate administrator may be in charge of configuring and operating the identity management system that provides access to the user layer of the virtualization stack. The separation between AWS’s sphere of security oversight and the customer’s is clearly defined, but it is entirely up to the customer to delineate the level of access end users and administrators are granted, and whether there is any distinction between the two.



###### *Figure 3-1. The shared responsibility model for IAAS services (adapted from*[*AWS Security Best Practices*](http://media.amazonwebservices.com/AWS_Security_Best_Practices.pdf)*)*

Amazon manages the security of facilities and is obviously responsible for the physical security of all hardware assets and network infrastructure. Beyond its global infrastructure, AWS is responsible for the software foundation underlying its services. In EC2’s case, this includes virtual provisioning infrastructure as well as the issuing of any credentials required to access it.

The customer’s security area of responsibility includes the following:

* Amazon Machine Images (AMIs)
* Operating system(s)
* Applications
* Data in transit
* Data at rest
* Data stores
* Credentials
* Policies and configuration

The attack surface exposed varies conspicuously: leaving an operating system image unpatched may expose a number of instances to attack, while an error handling access credentials could lead to the loss of all infrastructure hosted by the account-or the total loss of data confidentiality.

Securing EC2 images is no different than securing a machine in a traditional datacenter: OS images and running instances need to be patched, applications need to be updated, and AWS provides technology to protect data at rest and in transit. What changes, in implementation if not in spirit, is the way to define processes and levels of access for different classes of users. IAM is the mechanism AWS uses to provide access control and privilege separation, and we will examine it in detail in the next section-right after we tighten the security of your AWS account.

### ACCOUNT SECURITY CHECKLIST

* **Do not use root credentials.** Production use of root credentials breaks auditing in the shared responsibility model: were anything untoward to happen, you may not be able to reliably track down the real user responsible. You should secure the account with MFA and consider [“Throwing Away the Root Password”](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch03.html#sidebar_throwing_away_root) for the most aggressive approach to securing the root account.
* **Do use IAM users, user groups, and roles.** This chapter will teach you how to minimize the attack surface of compromised credentials by assigning separate keys to users and compute tasks. Credentials can further be limited in what actions they allow on which resources. Roles are used to eliminate the need to bake credentials into AMIs, eliminating a major leak vector for credentials.
* **Leverage password policy.** You can set a password policy to define complexity requirements for user passwords. More importantly, you should use this feature to define password expiration windows while also implementing a matching key rotation policy.
* **Enable AWS CloudTrail.** CloudTrail logs each AWS API call executed by the entire account, including what credentials were used to authenticate it. These records permit monitoring user behavior, and enable auditing after a security incident has occurred.

### MULTI-FACTOR AUTHENTICATION

Multi-factor authentication (MFA) adds a layer of security to your AWS account. When signing in to AWS, you will need to enter an authentication code in addition to your username and password. This authentication code can be generated by a physical device or by an application running on your computer or smartphone.

Adding a second factor to the authentication process (your password being the first one) gives you a lot of protection against unauthorized account access. It is no magic bullet, but it certainly prevents a lot of common attacks that rely on a single password giving access to your account.

To the cheers of sysadmins everywhere, Amazon decided to base its multifactor implementation (known as AWS MFA) on an open standard. The Time-Based One-Time Password Algorithm (TOTP—RFC 6238) is a method of generating passwords based on a shared secret. These passwords are valid for only a short period of time, and are typically regenerated every 30 seconds or so.

Google has made multi-factor authentication an option for logging in to its services, and as a result, published Google Authenticator. This is a smartphone application—available for Android, iOS, and BlackBerry—that acts as a virtual multi-factor authentication device. Because it is also based on the TOTP algorithm, it works perfectly with AWS MFA, giving you a quick way to increase your AWS account security without any monetary cost.

There is, of course, a small-time cost, as you will need to look up your access code whenever your AWS session expires. From a security perspective, it seems like a cost worth paying.

###### NOTE

If you have purchased a hardware multi-factor authentication device or downloaded a virtual device for your smartphone, visit the [AWS Multi-Factor Authentication page](http://aws.amazon.com/mfa) to tie it into AWS.

# Identity and Access Management

Identity and Access Management (IAM) is the name given to the suite of features that let you manage who and what can access AWS APIs using your account. This permissions-based system can be somewhat overwhelming at first, but resist the temptation to give in and grant all permissions to all users. Having a well-planned policy based on IAM is an important part of AWS security, and fits in well with the defense in depth strategy.

IAM makes a distinction between authentication (“who is this person?”) and authorization(“are they allowed to perform this action?”). Authentication is handled by users and groups, whereas authorization is handled by IAM policies.

###### TIP

Amazon’s [CloudTrail](http://aws.amazon.com/cloudtrail/) service keeps track of the API calls made by users in your account. You can use this to review the full history of AWS API calls that have been made by your account, whether they came from the Management Console, cli tools, or services like CloudFormation. This service is the preferred method to audit user actions, and is invaluable when it comes to diagnosing permissions problems.

## Amazon Resource Names

You may already know that S3 bucket names must be unique across the whole of S3. Have you ever wondered how S3 bucket names must be unique, while there are surely many IAM users named mike or admin?

The answer lies with ARNs and how they are formatted.

To identify IAM users and other resource types, AWS uses an Amazon Resource Name (ARN). An ARN is a globally unique identifier that references AWS objects. Most AWS resource types have ARNs, including S3 buckets and IAM users. ARNs take the following format:

arn:aws:*service*:*region*:*account\_ID*:*relative\_ID*

For example, here is the ARN for Mike’s’ IAM account (with the 12-digit account ID replaced by Xs):

arn:aws:iam::*XXXXXXXXXXXX*:user/mike

Notice that the region is not specified in the user’s ARN. This means that this ARN is a global resource, not tied to any specific region.

Some resources, such as S3 buckets, also omit the account ID in the ARN. S3 buckets use this ARN format:

arn:aws:s3:::*bucket\_name*

For example:

arn:aws:s3:::mike-image-resize

Notice that the only variable is the bucket name. Because S3 ARNs do not include the account number, creating two S3 buckets with the same name would result in a duplicate ARN, so this is not allowed.

## IAM Policies

The idea behind IAM is to separate users and groups from the actions they need to perform. You do this by creating an IAM policy, which is a JSON-formatted document describing which actions a user can perform. This policy is then applied to users or groups, giving them access only to the services you specifically allowed.

The best way to show the flexibility of IAM policies is with an example. Let’s say you use a tagging strategy described in the previous chapter, and have given all of your images a state tag that represents its current status, such as production or retired. As a good sysadmin who dislikes repetitive tasks, you have decided to automate the process of deleting retired images—AMIs that have been replaced by newer versions and are no longer required.

[Example 2-2](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch02.html#boto_delete_retired_images) shows a simple Boto script that deletes any AMIs that are in the retiredstate (according to our [“Tagging Strategy”](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch02.html#tagging_strategy)).

This script calls a number of Boto functions, which, in turn, call AWS APIs to perform the requested actions. If you were to run this script, it would connect to the API using the access key and secret that are stored in your AWS\_ACCESS\_KEY\_ID andAWS\_SECRET\_ACCESS\_KEY environment variables. While convenient, those access keys have far more permissions than are strictly required to run this script. Using them to authorize this script is overkill, and comes with a potential security risk: the more places in which your keys are stored, the more likely they are to be accidentally exposed to someone who does not need them.

There is another downside of reusing the same keys for multiple roles: it becomes very difficult to change them. Good security practices dictate that security credentials should be regularly rotated. If your AWS access credentials are reused in multiple scripts, keeping track of where each access key is being used becomes problematic. Replacing a key involves making the old one inactive so it can no longer be used to access APIs. If you accidentally deactivate the credentials that are used for making your database backups, you have a rather serious problem. If you do not segregate your IAM roles, you will end up being scared to deactivate old access keys because some vital component of your infrastructure will stop working.

A better solution would be to create a set of access credentials that are authorized to perform only the specific actions required by the script. Then you have a set of access credentials specific to each identifiable role—for example, AMI-cleaner, database-backups, and so on.

Let’s create an AMI policy with enough permissions to run the script that cleans old images and snapshots. Looking at the code, we see four Boto function calls. In most cases, Boto’s functions map quite well to AWS action types. Here are the four function calls and the action invoked by each one:

| **Function call** | **Action invoked** |
| --- | --- |
| connect\_to\_region | ec2:DescribeRegions |
| get\_all\_images | ec2:DescribeImages |
| delete\_snapshot | ec2:DeleteSnapshot |
| delete\_image | ec2:DeregisterImage |

A permission is a combination of two items: an action and one or more resources. AWS will check to see whether the authenticated user is allowed to perform the requested action on a specific resource—for example, is the user allowed to create a file (the action) in an S3 bucket (the resource)?

Actions are namespaced strings that take the form *service\_name*:*Permission*. All EC2-related permissions are prefixed with ec2:, such as ec2:DeleteSnapshot.

Because policies can reference highly granular, dynamic permissions across all AWS services, they can be time-consuming to write. When you create a policy, Amazon’s web interface gives you a list of permissions from which you can pick to save some time, but unfortunately no tool can magically remove the time it takes to plan out exactly which permissions each of your users or groups will require.

Using the Management Console is a great way of becoming familiar with the available permissions. Even if you are a hardcore command-line user, we suggest taking a few clicks around the interface to discover which actions are available. Because we already know which permissions to use for this script, we can use the command-line tools to create a user and attach a new policy to it, using the aws iam create-user and iam create-access-key commands.

First, we create a new user for this role, named ami-cleaner:

$ **aws iam create-user --user-name ami-cleaner**

{

"User": {

"UserName": "ami-cleaner",

"Path": "/",

"CreateDate": "2016-06-01T03:18:35.032Z",

"UserId": "AIDAILRZI2G4XH3QC6J4W",

"Arn": "arn:aws:iam::740376006796:user/ami-cleaner"

}

}

$ **aws iam create-access-key --user-name ami-cleaner**

{

"AccessKey": {

"UserName": "ami-cleaner",

"Status": "Active",

"CreateDate": "2016-06-01T03:19:02.919Z",

"SecretAccessKey": "wSelXh56SYP0f5ZxPkpSNL+kThTqU0nc3JeBNsC2",

"AccessKeyId": "AKIAJBYS5AQKKUN7MZJQ"

}

}

The iam create-access-key command generates an access key ID and secret access key for our new user. Store these somewhere safe, as we will need them later.

Next, we create an AMI policy and embed it directly in the user:

$ **aws iam put-user-policy --user-name ami-cleaner --policy-name ami-cleaner \**

**--policy-document '{"Version":"2008-10-17","Statement":[{"Effect":"Allow", \**

**"Action":["ec2:DescribeImages","ec2:DeleteSnapshot", \**

**"ec2:DeregisterImage"],"Resource":["\*"]}]}'**

###### WARNING

If you succeed in creating a malformed policy despite IAM’s validation safeguards, it will be waiting for your corrections in the claws of the [policy validator](http://docs.aws.amazon.com/IAM/latest/UserGuide/access_policies_policy-validator.html), in the IAM section of the console.

In this case, the user and policy names are both ami-cleaner. We chose to inline the policy directly in our user for simplicity, but a standalone policy object is more practical for production use as it can be associated with multiple users or even multiple groups of users. Using inlined JSON syntax, we are creating an Allow policy that applies to all resources. We specified a short list of actions that will be allowed, but you can specify as many permissions as you need.

Now we have an IAM user with a policy matching its role, so we can update the script to use the new keys. There are a few ways to do this, depending on your approach to managing your access keys. If no credentials are specified when opening a new connection to an AWS API, Boto will check whether the AWS\_ACCESS\_KEY\_ID andAWS\_SECRET\_ACCESS\_KEY environment variables are set, before falling back to the contents of the ~/.aws/credentials file. Run the script by executing the following commands in a terminal:

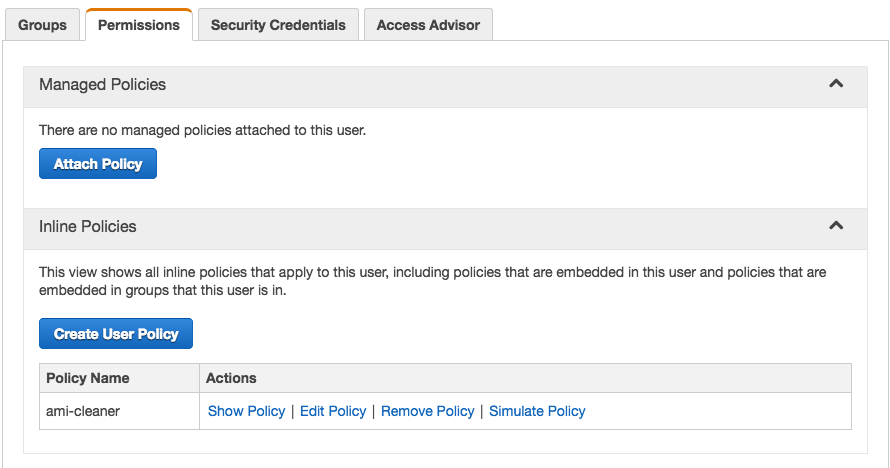
export AWS\_ACCESS\_KEY\_ID='AKIAJBYS5AQKKUN7MZJQ'

export AWS\_SECRET\_ACCESS\_KEY='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'

python delete-retired-amis.py

The script is now being run with the most restrictive set of permissions that will still allow it to function.

By running aws iam put-user-policy, we created a new IAM policy and added it to the user. But what does this policy actually look like? Through the IAM section in the Management Console, you can view the JSON-formatted version of our new AMI. First, find the image-cleaner user and then look in the Permissions tab ([Figure 3-2](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch03.html#policy_detail)).



###### *Figure 3-2. The newly created policy is found in the user’s permissions*

You can also view the body of this embedded policy with the aws iam get-user-policy command:

$ **aws iam get-user-policy --user-name ami-cleaner --policy-name ami-cleaner**

{

"UserName": "ami-cleaner",

"PolicyName": "ami-cleaner",

"PolicyDocument": {

"Version": "2008-10-17",

"Statement": [

{

"Action": [

"ec2:DescribeImages",

"ec2:DeleteSnapshot",

"ec2:DeregisterImage"

],

"Resource": [

"\*"

],

"Effect": "Allow"

}

]

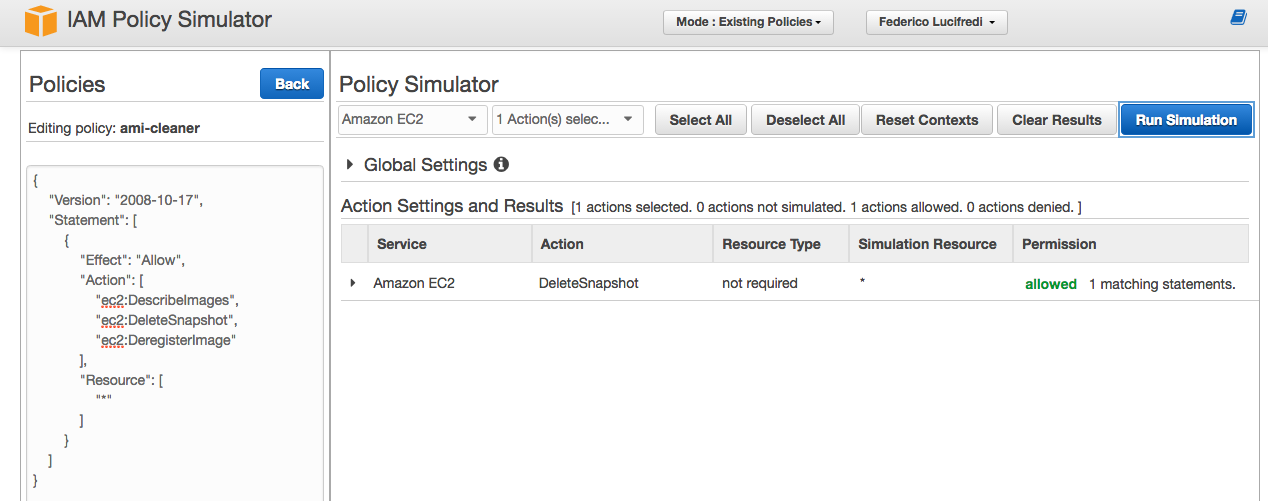
}

}

Readers familiar with JSON formatting will recognize that the Statement attribute is actually a list. Although our example includes only one statement, a single policy can contain multiple statements. Notice that a statement can have only a single Effect. So to create a policy that allows some actions but denies others, we must combine multiple statements with different Effect attributes.

###### WARNING

Determining what set of actions to allow a given policy is not simple, and even more complex is the task of mapping the interactions of multiple sets of permissions on a single user. Amazon provides the [IAM Policy Simulator](https://policysim.aws.amazon.com/) to help you evaluate how multiple policies interact to allow (or deny) a given action. The policy we just created is shown in the simulator undergoing testing in [Figure 3-3](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch03.html#policy_simulator).



###### *Figure 3-3. Evaluating our first policy in the simulator*

### REFERENCING RESOURCES IN IAM POLICIES

The Resource attribute of an IAM policy lets you control exactly which resources an action can be performed on. In the previous example, the policy granted the user permissions to delete any EBS snapshot owned by this account. What if you want a more granular policy that applies only to a subset of resources?

As discussed previously, ARNs are used to globally identify AWS resources. Used in IAM policies, they let you control exactly which resources are included when granting or denying permissions.

Suppose you use S3 buckets across your organization. Although most buckets contain data that could be easily replaced if lost (such as resized versions of images), you have one bucket that contains your database backups—something you certainly don’t want to lose. Your users are creating and deleting buckets via the Management Console, and you would like to make sure nobody accidentally deletes your backup bucket.

The cleanest solution to this problem is to create an IAM policy that allows users to perform any action on S3 buckets, with the exception of the one containing your backups. We do this by creating a policy containing two statements. The first grants the user all S3-related permissions, allowing them to be performed on any resource. The second statement denies all S3-related permissions, but only on the protected bucket.

When conflicting permissions are encountered, Deny takes precedence over Allow. So, users will be able to do anything they want on any bucket, except the one containing your backups.

The policy document describing these permissions looks like this:

{

"Statement": [

{ "Action": [

"s3:\*"

],

"Effect": "Allow",

"Resource": [

"\*"

]

},

{ "Action": [

"s3:\*"

],

"Effect": "Deny",

"Resource": [

"arn:aws:s3:::db-backups"

]

}

]

}

###### WARNING

These examples use the s3:\* action to grant all S3-related permissions. In practice, this is nearly always a bad idea.

When creating an IAM policy to suit a particular role, grant the role as few permissions as possible. It might take awhile (and a bit of trial and error!) to find the fewest permissions you need to fulfill a particular task, but it’s worth spending the time to do so.

Resist the temptation to assign \*:\* permissions!

To implement this policy, first find the ARN of your critical bucket. Assuming that your bucket is named db-backups, the ARN will be as follows:

arn:aws:s3:::db-backups

###### NOTE

When creating a policy, remember to replace the ARNs used in our examples.

Next, create the policy using the command-line tools or Management Console.

If using the Management Console, you can create the policy as follows:

1. Navigate to the IAM service.
2. Select the Policies tab.
3. Click Create Policy.
4. Select Create Your Own Policy.
5. Paste the text into the Policy Document box.

Using aws iam put-user-policy to attach a policy defined with inline JSON is not the most satisfying of user experiences. If you want to create a policy from the command line, you will have a much easier time writing it to a temporary file and uploading it. Assuming you have saved the policy to a file named s3-policy.json, you can create the policy with this command:

aws iam put-user-policy --user-name mike --policy-name s3\_policy \

--policy-document file://./s3-policy.json

Because denying rights to a particular activity is quite a common requirement, Amazon has provided an additional element in its resource policy language to handle this use case. Using NotResource as a shorthand, the preceding policy could be rewritten as follows:

{

"Statement": [

{ "Action": [

"s3:\*"

],

"Effect": "Allow",

"NotResource": [

"arn:aws:s3:::db-backups"

]

}

]

}

This is almost (but not entirely—see [“How Permissions Are Evaluated”](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch03.html#sidebar_permissions_evaluated)) the same as the longer policy we defined previously. NotResource refers to all objects other than the db-backups bucket, so this policy is effectively saying “grant all S3 permissions on all buckets except db-backups.”

The Action element of a policy document also has a negated counterpart, NotAction. Thus, to allow a user to perform all S3 actions except DeleteBucket, you could include this in your policy document:

"NotAction": "s3:DeleteBucket"

##### HOW PERMISSIONS ARE EVALUATED

Whenever you make a request to AWS, Amazon must evaluate the request and decide whether it should be allowed. The logic behind this process means there is a subtle but important difference between the two IAM policies in the previous section.

A request typically includes three bits of information: the user making the request, the action that is to be performed, and the target of the action. For example, Alice wants to terminate an EC2 instance. When receiving the request, AWS will use this information to decide whether it should be allowed by performing the following steps:

1. All policies pertaining to the user making the request are combined, including those applied by way of group membership.
2. If a permission is denied by one of the policies (an explicit deny), the request can be immediately denied.
3. If a permission is explicitly granted by one of the policies (an allow), the request is allowed.
4. Finally, if the permission was not explicitly granted, it is denied (a default deny).

Given this logic, consider the differences between the policies that used Resourceand NotResource as two ways to deny access to a bucket.

The first example includes an explicit deny that ensures no users have permissions to modify the db-backups S3 bucket. The second example, however, merely grants permissions to all S3 buckets except db-backups. There is no explicit deny on the db-backups bucket in the latter case; it is handled by the default deny.

Consequently, if the user were assigned a further policy that granted permissions to all S3 buckets, that user would have permissions to delete the db-backups bucket and all of its contents.

Creating IAM policies is another area where being explicit is definitely better than being implicit.

For more details on how AWS evaluates permissions, see the [AWS Identity and Access Management page](http://docs.amazonwebservices.com/IAM/latest/UserGuide/AccessPolicyLanguage_EvaluationLogic.html) describing its evaluation logic.

### DYNAMIC POLICIES

Conditions can be used to create dynamic IAM policies that behave differently, depending on one or more factors. The attributes of the request (such as the ARN of the requesting user or the source IP address) can be used in Boolean expressions to control whether a request should be allowed or denied.

Some of the available attributes on which you can base your conditions are as follows:

* Time of day
* Source IP address
* Whether the request is being made using HTTP or HTTPS

Of particular use is the SecureTransport attribute, which lets us check whether Secure Sockets Layer (SSL) is being used to make the request. Many of the AWS APIs can be accessed in both secure (HTTPS) and insecure (HTTP) modes. IAM policies provide the only way to force your users to use the secure versions of these APIs.

Let’s say you have an S3 bucket that is used for storing backups of confidential customer information. For regulatory reasons (or perhaps merely because of a healthy level of sysadmin paranoia), you must maintain a remote backup of this data, and the files must be transferred over an SSL connection.

This policy document would ensure that users could not perform any actions against the db-backups bucket if they are connected via plain old HTTP:

{

"Statement":[{

"Effect":"Allow",

"Action":"s3:\*",

"Resource":"arn:aws:s3:::db-backups",

"Condition":{

"Bool":{

"aws:SecureTransport":"true"

}

}

}

]

}

Using conditions, you could further enhance this policy to indicate the following:

* All connections must be secured by SSL.
* Files can be written to the bucket by only the database server (say, IP address 192.168.10.10).
* Files can be read by only the off-site backup server (say, IP address 192.168.20.20).

Of course, you could simply remember to enable the “use SSL” option of the client you use for transferring files from S3, but unless security features are enforced at a technical policy level, they will eventually be accidentally forgotten.

###### NOTE

For a more thorough look at the elements of an IAM policy, have a look at Amazon’s [Access Policy Language](http://docs.amazonwebservices.com/IAM/latest/UserGuide/AccessPolicyLanguage.html) documentation.

### LIMITATIONS OF IAM POLICIES

Although powerful, IAM policies do have some drawbacks that can take some effort to work around. Chief among these is that some AWS services do not yet provide support for ARNs in all actions, and can therefore not be fully managed by IAM policies drawing a distinction between different resource instances.

EC2 instances are a good example. Resource level permissions have been introduced in EC2 since July 2013, but not all ec2: actions support their use yet. Actions without resource level permissions have no way to reference a specific EC2 instance from an IAM policy. Whenever you allow such an an EC2 action in a policy, the resource will be \*, which means it will apply to every instance owned by your AWS account. AWS maintains a continuously updated list of [Supported resource-level permissions for Amazon EC2 API actions](http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ec2-supported-iam-actions-resources.html). A number of other things are simply not yet possible with IAM.

To work around this limitation, some people have taken to operating multiple AWS accounts—one for each department. IAM roles, covered later in this chapter, give you a way to securely share resources between accounts, making this a viable option in the remaining cases where resouce-level permissions are not yet available.

Operating with multiple accounts is a valid advanced security strategy in its own right, the table lists a few sensible strategies for your consideration. To be clear, we are not taking the position that multiple accounts are a recommended default strategy for every user, we believe a supporting rationale needs to be identified to justify the cost in time and effort of the additional complexity. Whether it is hosted in a single AWS account or multiple ones, we recommend you consider the impact of your security structure on the team’s velocity. The insight found in [Conway’s Law](https://en.wikipedia.org/wiki/Conway%27s_law) is critical to designing effective security boundaries for any team: “any organization [..] will inevitably produce a [product] design whose structure is a copy of the organization’s communication structure.”

| **Account strategy** | **Consequences** |
| --- | --- |
| Single AWS account | Centralized management with minimum overhead. Secure with tailored users limited in access to actions and resources. |
| Separate production, development, and testing AWS accounts | Supplement single account capabilities with separation between multiple AWS accounts. Additional effort required staging resources between accounts. |
| Multiple AWS accounts, one per department | Access to actions and resources can follow radically different procedures for each organization. Cooperation on private shared resources marginally more complex. |
| Multiple AWS accounts, one per function | Functionally centralized management with different accounts for DNS, DBMS, CDN, CMS or any other services. |

## IAM Users and Groups

Because users and groups need to deal only with authentication, they are relatively simple compared to other AWS facilities. If you are familiar with how Unix or Windows handles user and group permissions, you already know the core principles behind IAM users and groups.

A user can be a human who logs in to the Management Console with a username and password, or a program that uses a set of access credentials to interact with AWS APIs. The user can be assigned one or more IAM policies, which specify the actions the user is allowed to perform.

To ease administration, users can be placed in groups. When an IAM policy is assigned to a group, all members of that group inherit the permissions designated by the IAM policy. It is not possible to nest groups; a group cannot contain other groups.

IAM is a global AWS service, meaning it is not tied to any particular region. An IAM user will be able to access APIs in any region, if allowed by its assigned IAM policies.

You should create a separate user for each person who will access your account, rather than sharing the master password for your AWS account. As people leave and join the organization, it will be a lot easier to revoke old security keys and assign permissions to new accounts.

Assigning permissions to specific users has its purposes, but it is often a sign that tasks and knowledge are not being shared across the team. If Alice is the only person with CreateSnapshot permissions, how is Bob going to handle backups while she is on vacation?

Aim to map AWS groups to specific roles within your organization, and apply the policy to the group instead. Managing updates to permissions is also a lot easier, as they will need to be made in only one place.

##### THROWING AWAY THE ROOT PASSWORD

Throwing away the root password for your AWS account is an increasingly popular security best practice: you cannot lose control of credentials you do not know and have no access to. If for any reason you ever needed it, you could always regain access to your root account identity as long as you have access to the e-mail address the account is bound to. [Eric Hammond](https://alestic.com/2014/09/aws-root-password/) of Alestic shared this security best practice:

1. Create an IAM user with full administrative privileges, including access to account billing information. This grants the user ability to update payment methods and most account information.
2. Change the AWS root account password to a long, locally generated random string which you will retain no copies of. On Ubuntu, you can use pwgen -s 24 1 to generate such a password.

Although an intruder who manages to gain rogue access to the AIM user with full administrative privileges could still do a lot of damage -running up costs, destroying resources and data, etc.-at least they could not lock you out and change the associated e-mail address to prevent your team to recover access. When you detect the intrusion, you can recreate the root password and lock out the intruder.

Eric has identified the following seven exceptions as the only AWS functionality you may ever need your root account for:

* Changing the e-mail and password of the AWS root account
* Transfering a Route53 Domain registration
* Canceling AWS services, like support
* Closing the account
* Submitting a penetration testing inquiry form
* Setting up consolidated billing
* Activating (or deactivating) IAM user access to billing information

### ORGANIZING USERS AND GROUPS WITH PATHS

If you are coming from a Lightweight Directory Access Protocol (LDAP) or Active Directory background, you might be used to a little more flexibility in user and group layout. In particular, the inability to nest groups within groups can feel like a big limitation when moving from one of these systems to IAM.

Paths are an optional feature of IAM users that can be used to implement more complicated user and group scenarios. In combination with IAM policies, they can be used to create a hierarchical structure for your users and groups.

Suppose you have several departments in your organization, and you would like each department to manage its own users. No one likes resetting passwords or setting up new accounts, so delegating this to a group of trusted users within that department will save time and headaches on all sides.

Start by creating two new groups: one to hold the normal users of the group and one for the group admins. Let’s use a development team as an example and create groups named dev\_admins and dev\_users:

$ **aws iam create-group --group-name dev\_admins --path "/dev/"**

{

"Group": {

"Path": "/dev/",

"CreateDate": "2016-06-05T14:03:57.681Z",

"GroupId": "AGPAIHLR2VSAFC2VNVXCQ",

"Arn": "arn:aws:iam::740376006796:group/dev/dev\_admins",

"GroupName": "dev\_admins"

}

}

$ **aws iam create-group --group-name dev\_users --path "/dev/"**

{

"Group": {

"Path": "/dev/",

"CreateDate": "2016-06-05T14:05:45.019Z",

"GroupId": "AGPAI7PNSRYTT573CEYH6",

"Arn": "arn:aws:iam::740376006796:group/dev/dev\_users",

"GroupName": "dev\_users"

}

}

Next, create two users. Alice is the most responsible member of the dev team, so she will be in the dev\_admins and dev\_users groups. Bob, being slightly less responsible (or at least feigning irresponsibility to avoid being assigned additional tasks), is only in the dev\_users group:

$ **aws iam create-user --user-name alice --path "/dev/"**

{

"User": {

"UserName": "alice",

"Path": "/dev/",

"CreateDate": "2016-06-05T22:48:26.833Z",

"UserId": "AIDAJKWL3DGB6E4OHBTYK",

"Arn": "arn:aws:iam::740376006796:user/dev/alice"

}

}

$ **aws iam add-user-to-group --user-name alice --group-name dev\_admins**

$ **aws iam add-user-to-group --user-name alice --group-name dev\_users**

$ **aws iam create-user --user-name bob --path "/dev/"**

{

"User": {

"UserName": "bob",

"Path": "/dev/",

"CreateDate": "2016-06-05T23:27:59.704Z",

"UserId": "AIDAIVVPCRZA4V26N4J52",

"Arn": "arn:aws:iam::740376006796:user/dev/bob"

}

}

$ **aws iam add-user-to-group --user-name bob --group-name dev\_users**

We can verify that the users and groups have been created with the correct paths by issuing the aws iam list-users and aws iam list-groups commands:

$ **aws iam list-users --output text**

USERS arn:aws:iam::740376006796:user/dev/alice 2016-06-05T22:48:26Z /dev/ AIDAJKWL3DGB6E4OHBTYK alice

USERS arn:aws:iam::740376006796:user/dev/bob 2016-06-05T23:27:59Z /dev/ AIDAIVVPCRZA4V26N4J52 bob

[...]

$ **aws iam list-groups --output text**

GROUPS arn:aws:iam::740376006796:group/dev/dev\_admins 2016-06-05T14:03:57Z AGPAIHLR2VSAFC2VNVXCQ dev\_admins /dev/

GROUPS arn:aws:iam::740376006796:group/dev/dev\_users 2016-06-05T14:05:45Z AGPAI7PNSRYTT573CEYH6 dev\_users /dev/

Now that the users and groups are set up, we need to create an IAM policy next.

Notice that the ARNs for our new users and groups include /dev as part of the identifier. This is the magic that makes it all work. Because we can use wildcards when specifying resources in IAM policies, we can simply grant the user permission to execute IAM actions on resources that exist under the /dev hierarchy. As before, the asterisk indicates “all resources”:

{

"Statement": {

"Effect": "Allow",

"Action": "iam:\*",

"Resource": [

"arn:aws:iam::740376006796:group/dev/\*",

"arn:aws:iam::740376006796:user/dev/\*"

]

}

}

Save the preceding policy document in a text file and apply it to Alice’s group with the the aws iam put-group-policy command, taking care to replace the ARN with your own account’s:

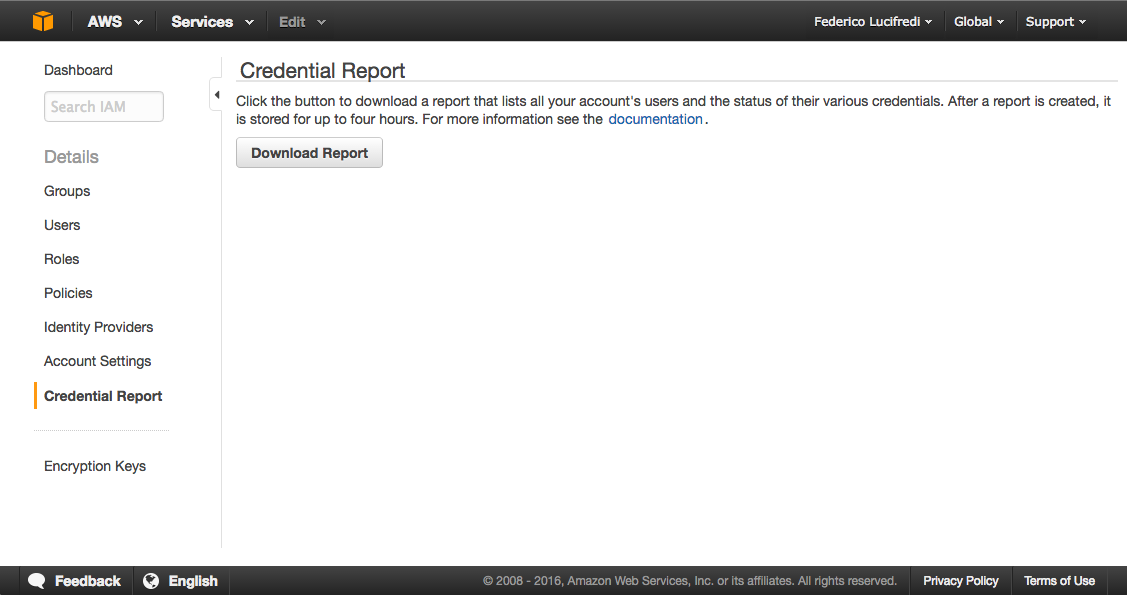
$ **aws iam put-group-policy --group-name dev\_admins --policy-name dev\_admin \**

**--policy-document file://./dev\_admin.json**

Once this policy is applied, Alice will be able to reset Bob’s password or create a new user in the /dev hierarchy, but she will not be able to create a new user in the /supporthierarchy.

### AUDITING AND ROTATING ACCESS KEYS

Security best practices include the removal of keys no longer in use, as well as regularly changing keys, a practice known as key rotation. IAM assists the administrator in this tedious task by providing ready access to all the information that is needed to promptly identify accounts or keys no longer in use. The IAM console reports when access keys were last used, in what region, and for what AWS service. Details concerning a user’s password last use complement this data to form a complete picture of when an account was last active.



###### *Figure 3-4. The credential report is only one click away in the IAM console*

The Credential Report is found in the IAM console ([Figure 3-4](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch03.html#credential_report)) and can be generated from the CLI as follows:

$ **aws iam generate-credential-report**

{

"State": "STARTED",

"Description": "No report exists. Starting a new report generation task"

}

$ **aws iam get-credential-report | jq -r '.Content' | base64 -d > report.csv**

$ **more report.csv**

user,arn,user\_creation\_time,password\_enabled,password\_last\_used [...]

alice,arn:aws:iam::740376006796:user/dev/alice,2016-06-05T22:48:26+00:00 [...]

[output truncated]

The report is returned in base64 encoding wrapped in a JSON envelope, requiring the use of jq and base64 to save the comma-separated CSV file’s contents.

Once your audit has identified any idle accounts to be deactivated, the task turns to rotating the old keys of the remaining accounts to newly generated ones. This is a well-defined process requiring a number of well-planned steps. Let’s say our audit identified that Bob’s key is due for rotation: our first step would be adding a second access key to the account (up to two active access keys are allowed for each user).

$ **aws iam list-access-keys --user-name Bob**

{

"AccessKeyMetadata": [

{

"UserName": "bob",

"Status": "Active",

"CreateDate": "2015-06-05T00:41:52Z",

"AccessKeyId": "AKIAI33YRI5D4IIJOANA"

}

]

}

$ **aws iam create-access-key --user-name Bob**

{

"AccessKey": {

"UserName": "bob",

"Status": "Active",

"CreateDate": "2016-06-21T04:27:41.291Z",

"SecretAccessKey": "CFgIptAtrqUKhfe/p3v1OBHciK5sY/n8EBX8JyO/",

"AccessKeyId": "AKIAIL5TWIUC2M76F3PQ"

}

}

$ **aws iam list-access-keys --user-name Bob**

{

"AccessKeyMetadata": [

{

"UserName": "bob",

"Status": "Active",

"CreateDate": "2016-06-21T04:27:41Z",

"AccessKeyId": "AKIAIL5TWIUC2M76F3PQ"

},

{

"UserName": "bob",

"Status": "Active",

"CreateDate": "2015-06-05T00:41:52Z",

"AccessKeyId": "AKIAI33YRI5D4IIJOANA"

}

]

}

The new secret access key needs to be forwarded to Bob securely, preferably in some automated fashion. Key IDs (AKIDs) will help you track a key through its lifecycle; once you are satisfied the new key has replaced the old one in all uses and everything is still functioning as expected, you can retire the old key as inactive:

$ **aws iam update-access-key --access-key-id AKIAI33YRI5D4IIJOANA \**

**--status Inactive --user-name Bob**

$ **aws iam list-access-keys --user-name Bob**

{

"AccessKeyMetadata": [

{

"UserName": "bob",

"Status": "Active",

"CreateDate": "2016-06-21T04:27:41Z",

"AccessKeyId": "AKIAIL5TWIUC2M76F3PQ"

},

{

"UserName": "bob",

"Status": "Inactive",

"CreateDate": "2015-06-05T00:41:52Z",

"AccessKeyId": "AKIAI33YRI5D4IIJOANA"

}

]

}

Functionally equivalent to being deleted, an inactive key has the salient property of being ready to return to service immediately upon a single command being issued. This makes turning a key inactive the safest way to retire it from production, with the comfort of an immediate “undo” option should anything go wrong. The final step of actually deleting the key is performed by passing the AKID and user name to aws iam delete-access-key. There is no urgency to taking this action, but it will be required before the next key rotation is performed as AWS limits users to two access keys.

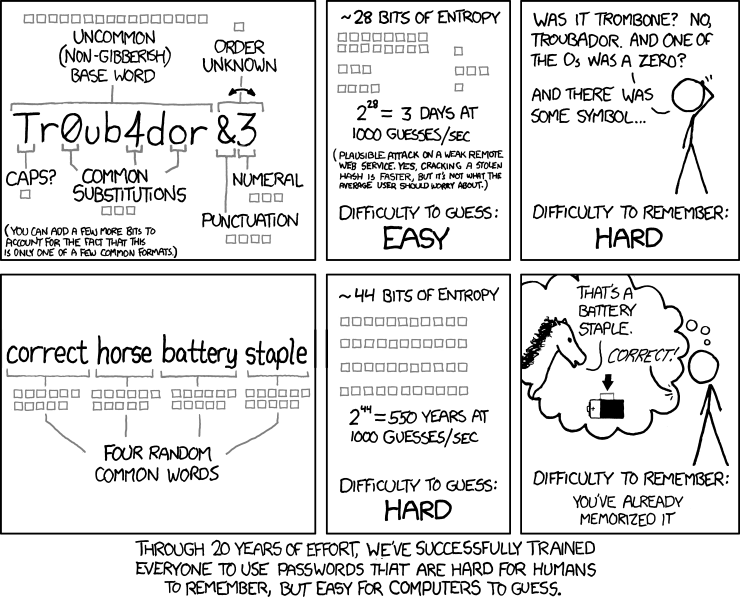
### PASSWORD POLICY

A password policy enables you to define length and complexity requirements for user passwords account-wide. Equally important and perhaps more interesting, it permits setting an expiration window for user passwords matching your key rotation policy. For example:

$ **aws iam update-account-password-policy --allow-users-to-change-password \**

**--max-password-age 90 --minimum-password-length 14**

This will require that users change their passwords every 90 days, while simultaneously granting them permission to actually change their passwords. Federico tipically requires 14-character minimum length passwords (AWS defaults to 6), without forcing their character composition. The intent is to nudge users towards the use of safe and more easily recalled passphrases-see XKCD’s common-sense summary of password security in [Figure 3-5](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch03.html#xkcd_passwords). You will need to weigh the inconvenience to your users against your organization’s threat profile as you alter these settings. Note that the aws iam update-account-password-policy command does not support partial updates: no parameters are required, but those not supplied will silently return to their default values-this can be both convenient when resetting defaults, and confusing if you are not aware of this command’s behavior.



###### *Figure 3-5.*[*XKCD’s uniquely elegant explanation of password strength*](https://xkcd.com/936/)*, courtesy of Randall Munroe*

### CLOUDTRAIL

CloudTrail can log all activity occurring in the account by creating a record of every API call in a specified region or globally, irrespective of the tool originating the call-CLI, console, and even other AWS services. These records greatly enhance your ability to determine what user performed what action at a given time, and is essential to reconstruct what has really happened in the event of a security incident.

###### WARNING

CloudTrail will include in its logs all API calls generated by any AWS service on the user’s behalf. You will be startled when you notice this for the first time, and may wonder if another, possibly rogue, user is carrying out some kind of unauthorized activity. To determine if an API call was generated automatically by another service, examine the invokedBy field of the CloudTrail record in question.

CloudTrail generates log files recording all activity occurring in the account (or in a specific AWS region), storing them in an S3 bucket. Trails can be created using the aws cloudtrail create-trail command. This commands requires a pre-existing S3 bucket to store logs to-remember that as the namespace of the s3 service is flat, you will need to find a unique name for your trail’s bucket:

$ **aws s3 mb s3://global-trail**

make\_bucket: s3://global-trail/

$ **aws s3api put-bucket-policy --bucket global-trail --policy file://cbp.json**

The second command sets a bucket policy that grants CloudTrail all the permissions it requires to perform its logging. The file defines the same policy the AWS console would set automatically if you choose to initialize your trail in the UI instead:

{

"Version": "2012-10-17",

"Statement": [

{

"Sid": "AWSCloudTrailAclCheck20150319",

"Effect": "Allow",

"Principal": {

"Service": "cloudtrail.amazonaws.com"

},

"Action": "s3:GetBucketAcl",

"Resource": "arn:aws:s3:::global-trail"

},

{

"Sid": "AWSCloudTrailWrite20150319",

"Effect": "Allow",

"Principal": {

"Service": "cloudtrail.amazonaws.com"

},

"Action": "s3:PutObject",

"Resource": "arn:aws:s3:::global-trail/AWSLogs/740376006796/\*",

"Condition": {

"StringEquals": {

"s3:x-amz-acl": "bucket-owner-full-control"

}

}

}

]

}

Next, we create a trail logging the account’s activity in all AWS regions:

$ **aws cloudtrail create-trail --name global-trail --s3-bucket-name global-trail\**

**--is-multi-region-trail**

{

"IncludeGlobalServiceEvents": true,

"Name": "global-trail",

"TrailARN": "arn:aws:cloudtrail:us-east-1:740376006796:trail/global-trail",

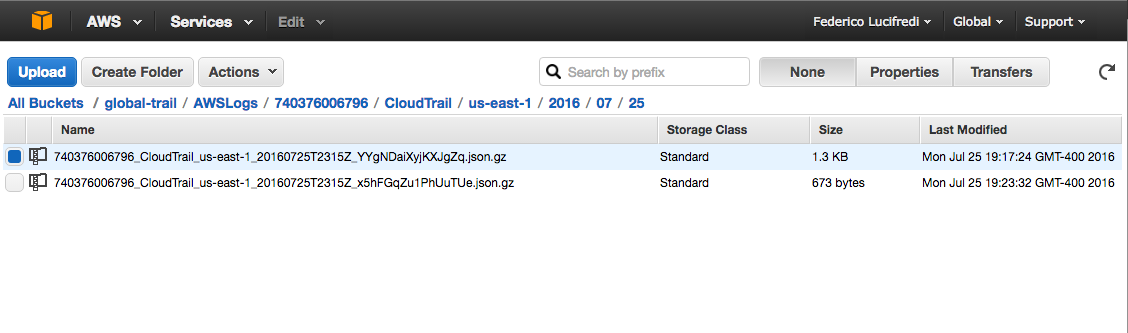
"LogFileValidationEnabled": false,

"IsMultiRegionTrail": true,

"S3BucketName": "global-trail"

}

$ **aws cloudtrail start-logging --name global-trail**



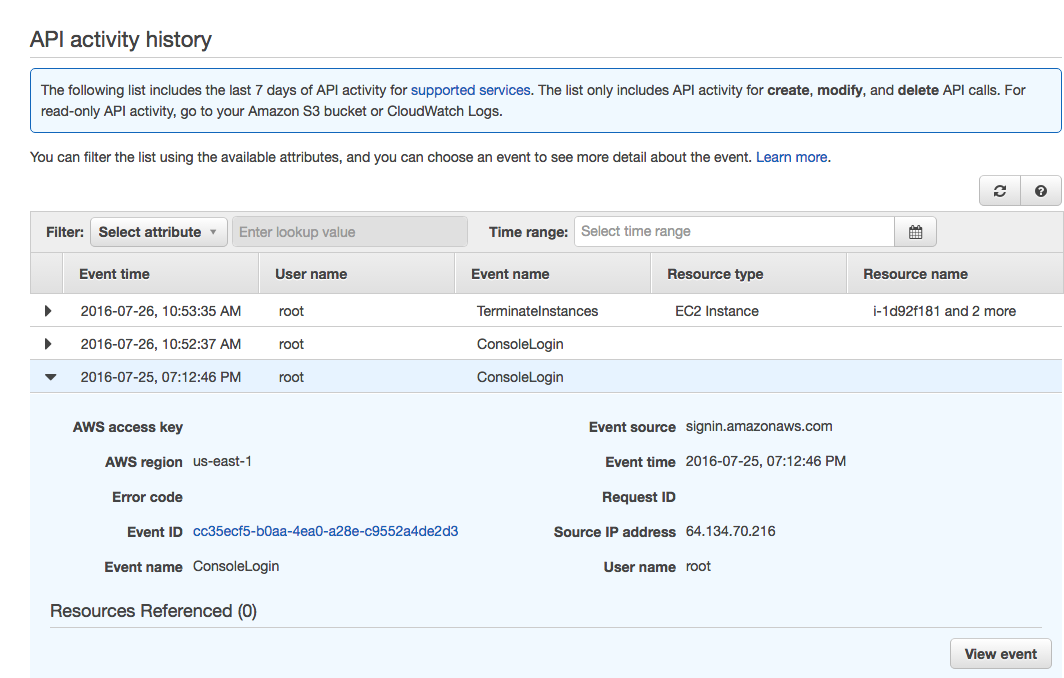
###### *Figure 3-6. Navigating to a log file in the trail’s S3 bucket.*

Examining a trail’s contents requires sifting through the S3 bucket configured earlier, and retrieving the log file for the region and day of interest. This is most easily accomplished in the AWS console, in particular if your browser is equipped with an extension like Chrome’s[JSONview](https://chrome.google.com/webstore/detail/jsonview/chklaanhfefbnpoihckbnefhakgolnmc?hl=en), which makes cursory examination of JSON files that much more efficient ([Figure 3-7](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch03.html#examine_trail)).



###### *Figure 3-7. A CloudTrail login record examined inline in Chrome with JSONview-*someone*logged in as*root*, contrary to our policy!*

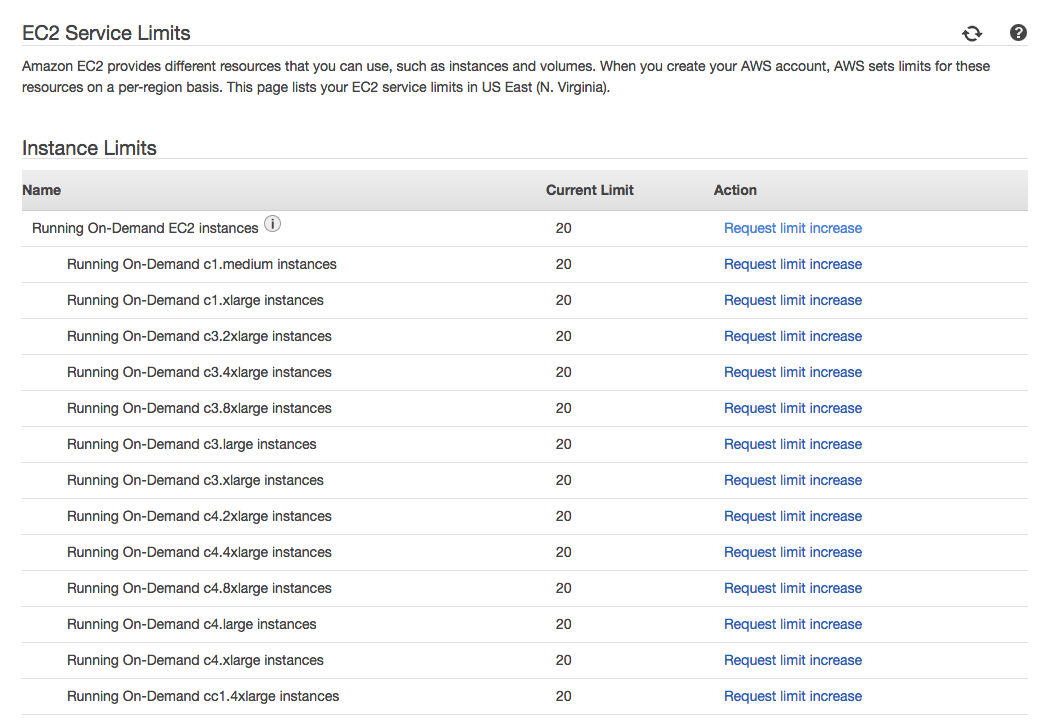
Navigating to CloudTrail S3 bucket in the AWS Console ([Figure 3-6](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch03.html#navigating_trail)) also provides download access to the trail’s files for offline analysis in your tool of choice. The granularity of single files may hinder your analysis when examining sequences of events spanning longer than a single day or more than one AWS region, which is the reason why the AWS console provides the [API Activity](https://console.aws.amazon.com/cloudtrail/home?region=us-east-1#/events) view. Consolidating events for the last seven days, this interface selectively visualizes create, delete, and modify events. The interface provides convenient direct access to the API call metadata as well as limited search functionality ([Figure 3-8](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch03.html#api_activity)).



###### *Figure 3-8. The API Activity History interface to CloudTrail, visualizing the same login record we reviewed earlier.*

##### RESOURCE LIMITS

Denial of Service is one of the most common security attack vectors: what if a user logged into AWS and launched the public cloud equivalent of a fork bomb? Launching thousands upon thousands of instances would quickly exhaust the region’s capacity and disrupt provisioning for other users. For this reason, AWS accounts have default [service limits](https://console.aws.amazon.com/ec2/v2/home?region=us-east-1#Limits:) enforced on a regional basis ([Figure 3-9](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch03.html#resource_limits)). You can request these resource ceilings be raised at any time by opening a support request on the very same page-some of Federico’s teams have access to accounts with spot limits exceeding thousands of instances for large-scale benchmarking, AWS can be very accomodating to customers’ needs if you make a good case, so don’t be shy about your goals.

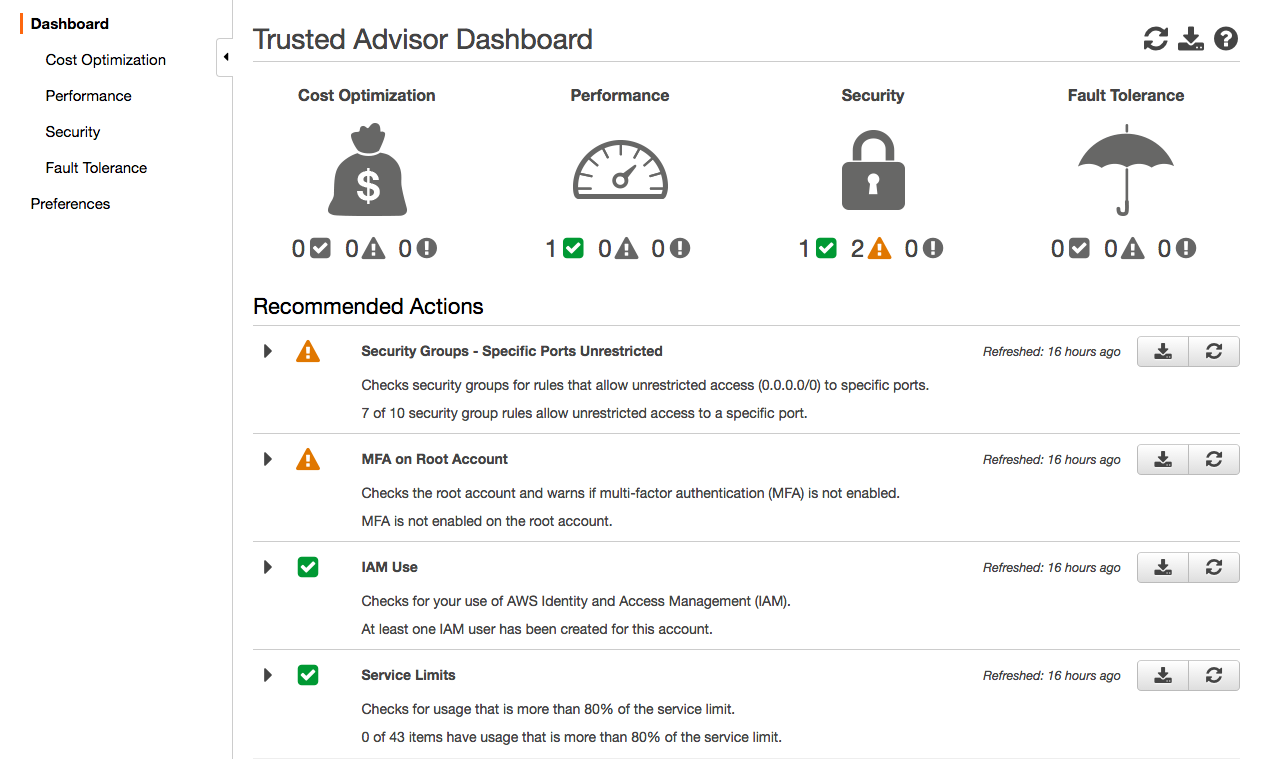


###### *Figure 3-9. Some of the default account limits listed in the AWS console of a new account.*

### TRUSTED ADVISOR

True to its name, the AWS Trusted Advisor is a part of AWS’s support package offering automated advice in the areas of cost reduction, performance, security, and fault tolerance. Full access to Trusted Advisor is only available with an enterprise support package, but some functionality is generally available to all users. The freebies include very useful basic security advice as well as analysis of usage thresholds meant to prevent outages caused by exceeding account resource limits.

The Trusted Advisor dashboard provides access to useful information even to those not subscribing to AWS’ support services, making it a useful resource to monitor the health of your account with minimum effort. In particular, the security scan includes a summary of open ports that is hard to beat for convenience-[Figure 3-11](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch03.html#open_ports) illustrates the point with a section of the spreadsheet produced by auditing an account running this chapter’s examples. The AWS team regularly adds new checks to Trusted Advisor, which are usually announced by Jeff Barr in AWS’ official [blog](https://aws.amazon.com/blogs/aws/).



###### *Figure 3-10. The Trusted Advisor dashboard gives access to the automated analysis tooling as well as a summary of their most recent findings.*

Naturally, Trusted advisor reports can be retrieved programmatically and integrated in your automation-a security dashboard could retrieve the open port numbers on your account for example. A premium support subscription is required for API access, and the lowest-cost option (developer support) is excluded as of this writing. We start by retrieving the identifier of the Trusted Advisor check in question, then forcing a refresh for this particular report:

$ **aws support describe-trusted-advisor-checks --language en | jq \**

**'.checks[] | select(contains({name:"Ports Unrestricted"})).id'**

"HCP4007jGY"

$ **aws support refresh-trusted-advisor-check --check-id HCP4007jGY**

{

"status": {

"checkId": "HCP4007jGY",

"status": "enqueued",

"millisUntilNextRefreshable": 3599988

}

}

$ **aws support refresh-trusted-advisor-check --check-id HCP4007jGY**

{

"status": {

"checkId": "HCP4007jGY",

"status": "success",

"millisUntilNextRefreshable": 270266

}

}

We can now retrieve the results and parse them for local use to our heart’s content (or our CIO’s).

$ **aws support describe-trusted-advisor-check-result --check-id HCP4007jGY**

{

"result": {

"checkId": "HCP4007jGY",

"status": "warning",

"flaggedResources": [

{

"status": "warning",

"resourceId": "4lPCS\_Zw5DRIqfc6yKAu4Vc2r2F96s\_YmiVOpKmqoRA",

"region": "us-east-1",

"isSuppressed": false,

"metadata": [

"us-east-1",

"image-resizing-ImageResizingSecurityGroup-SYAMMMJ89PVX",

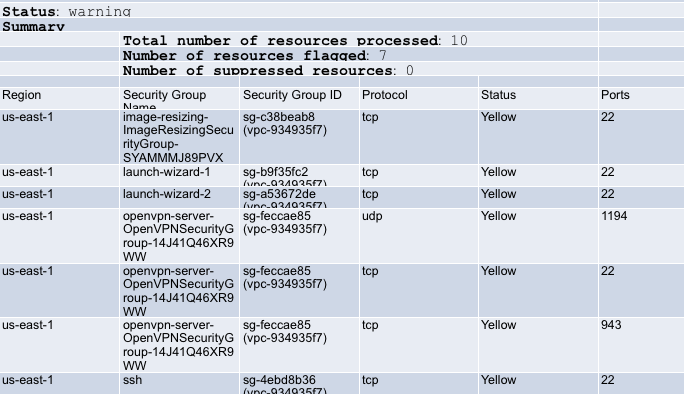
"sg-c38beab8 (vpc-934935f7)",

"tcp",

"Yellow",

"22"

[ output truncated ]



###### *Figure 3-11. Trusted Advisor’s audit: open ports found in the authors’ security groups.*

###### TIP

Users who do not have access to a support subscription will not be able to leverage Trusted Advisor’s most advanced features, but a handy third-party alternative is also available: [Security Monkey](https://github.com/Netflix/security_monkey). Hailing from one of the earliest teams to adopt public cloud as the computing platform of choice in a large enterprise, Security Monkey populates a database tracking configuration changes in your accounts’ critical infrastructure and then runs rule-based checks every time a change occurs. Some noteworthy differences arise due to the different nature of the two tools-one a SaaS solution, the other an Open Source software package: Security Monkey empowers end users with the capability to define custom security checks, which Trusted Advisor does not.

# IAM Roles

Consider the following scenario: you regularly need to resize images stored in an S3 bucket. Knowing a bit about Boto, you write a script that will look in the incoming directory of your S3 bucket for new images, perform the resize operations, and save the resulting images in the processed directory.

You want to launch an EC2 instance that will run this script after it has finished booting. For the script to work, it needs to use a set of AWS credentials with permissions to read the source files and write the output files to the S3 bucket.

In the previous section, we have already created the IAM user, applied the appropriate policy to it, and downloaded the access key and secret. But how will you provide these keys to the instance so they can be used by the log-processing script?

Until June 2012, the process of distributing AWS keys to your EC2 instances was somewhat painful. There were essentially two main options: bake the keys into the AMI so they were available when an instance booted, or provide them to the instance at runtime, perhaps with user data.

Both had their own downsides. If keys were baked into the AMI, replacing keys meant building a new AMI. If you went for the user data option, your unencrypted AWS keys were easily visible in the Management Console and other places. The Amazon team recognized that both options lacked security and simplicity, so they introduced IAM roles in response.

IAM roles almost entirely remove the problems surrounding this issue. Like users and groups, IAM roles can have one or more policies applied to them. When you launch an instance, you assign it a role that you have previously created. AWS will automatically generate access credentials and make them available to the instance. These credentials can then be used to access AWS services, with the permissions specified by the role’s policies.

Best of all, Amazon will regularly rotate the keys during the lifetime of the instance, without requiring any action on your part. This can be a big relief if you are working with long-running instances, as it seriously reduces the time in which compromised keys are usable.

Given these advantages, we can’t recommend using IAM roles highly enough. If all of your AWS scripts use roles, you will never need to worry about rotating these access credentials, even when people leave your organization. Furthermore, you will no longer run the risk of accidentally revoking keys that are still in use in some little-visited corner of your infrastructure (although you could still delete an IAM policy that you need).

###### WARNING

If not properly configured, IAM roles can be used for privilege escalation. Imagine a nefarious user who has permissions to launch instances with IAM roles, but does not have permissions to delete Route 53 DNS records. By launching an instance with a role that does have these permissions, the user could easily SSH into the instance and retrieve the credentials.

IAM policies can be used to control which roles can be assigned by a user when they launch an instance, by explicitly referencing the ARN of the role when granting the user the iam:PassRole permission.

To see IAM roles in action, let’s implement the example just given. To start, we will need an S3 bucket containing an example image. Although we will use aws s3 in the following example, you could, of course, create the S3 bucket and upload an example file via the Management Console.

###### TIP

[s3cmd](http://s3tools.org/s3cmd) is a popular command-line tool for interacting with S3, which Mike has found very useful when creating S3-based backup systems and Federico’s team uses to test compatibility with the S3 interface. It is readily available in the default package repositories of many Linux systems.

If you are using Ubuntu, you can install this alternative cli tool with sudo apt install s3cmd. Before using s3cmd, you will need to run s3cmd --configure. This will write a file in your home directory containing your AWS credentials, along with some other settings.

First, create a new S3 bucket. Because S3 bucket names must be unique, you will need to choose your own name for it:

$ **aws s3 mb s3://mike-image-resize**

make\_bucket: s3://mike-image-resize/

Download an example image file and copy it to the S3 bucket. We will use the O’Reilly logo for this example:

$ **wget -q http://cdn.oreillystatic.com/images/sitewide-headers/ml-header-home-blinking.gif**

$ **aws s3 cp ml-header-home-blinking.gif s3://mike-image-resize/incoming/**

upload: ./ml-header-home-blinking.gif to s3://mike-image-resize/incoming/ml-header-home-blinking.gif

$ **aws s3 ls --recursive s3://mike-image-resize**

2016-06-05 23:21:43 9067 incoming/ml-header-home-blinking.gif

$

Now that we have a bucket, we can create a policy that will allow the script to create and delete the contents of the bucket. As with most tasks involving IAM, the first step is to think about which permissions the script will require. Thankfully, our example script is quite simple—the tasks it performs map to the following actions:

1. List the contents of the bucket: s3:ListBucket.
2. Get the original file: s3:GetObject.
3. Store the resized images in the bucket: s3:PutObject.
4. Delete the processed files: s3:DeleteObject.

To make our application as secure as possible, this role will have access only to the bucket we created earlier, so malicious users who managed to access these credentials would not be able to affect other parts of your application. To do this, we need to know the ARN of the bucket.

As we saw near the beginning of this chapter, the ARN for an S3 bucket takes the format arn:aws:s3:::*name-of-bucket*. The consecutive colons are not merely decoration: other resource types use these fields to store additional attributes that are not used by S3 ARNs.

Because permissions can apply to either the contents of the bucket or the bucket itself, we actually need to specify two ARNs:

* arn:aws:s3:::*name-of-bucket*
* arn:aws:s3:::*name-of-bucket*/\*

###### WARNING

You might consider saving some typing and simply specifying the ARN asarn:aws:s3:::*my-bucket*\*.But if you have a bucket named, say, my-bucket-secure, you will be granting this role permissions on this bucket too. To quote the Zen of Python, “explicit is better than implicit”—even if it does sometimes involve more typing.

The first ARN references the bucket itself, and the second references any keys stored within that bucket. If we wanted to, we could assign an even more stringent set of permissions that allows the application to read and delete files in the incoming directory, but only write to the processed directory.

We now have all the information we need to create a role and assign a policy to it. We do this with the aws iam create-role command, creating a role named image-resizing:

$ **aws iam create-role --role-name image-resizing --assume-role-policy-document \**

**file://./ec2-assume-role.json**

{

"Role": {

"AssumeRolePolicyDocument": {

"Version": "2012-10-17",

"Statement": {

"Action": "sts:AssumeRole",

"Effect": "Allow",

"Principal": {

"Service": "ec2.amazonaws.com"

}

}

},

"RoleId": "AROAIJSFDM5WJNR2M5ZVI",

"CreateDate": "2016-06-07T05:22:34.145Z",

"RoleName": "image-resizing",

"Path": "/",

"Arn": "arn:aws:iam::740376006796:role/image-resizing"

}

}

The last line of the output is the ARN of the newly created role; we will use this later. The trust policy document controls which services may assume this role—at the time of writing, EC2, AWS Data Pipeline, Amazon Elastic Transcoder, or AWS OpsWorks are the only services that can assume roles.

We used the following trust policy document to create the image-resizing role:

{

"Version": "2012-10-17",

"Statement": {

"Effect": "Allow",

"Principal": {"Service": "ec2.amazonaws.com"},

"Action": "sts:AssumeRole"

}

}

Once the role has been created, we can create a policy and embed it into the role:

{

"Version": "2012-10-17",

"Statement": [

{

"Sid": "Stmt1465279863000",

"Effect": "Allow",

"Action": [

"s3:DeleteObject",

"s3:GetObject",

"s3:ListBucket",

"s3:PutObject"

],

"Resource": [

"arn:aws:s3:::mike-image-resize/\*",

"arn:aws:s3:::mike-image-resize"

]

}

]

}

$ **aws iam put-role-policy --role-name image-resizing \**

**--policy-name image-resizing --policy-document file://image-resizing.json**

The role and policy are both named image-resizing. If you wish to distinguish between them (or you just like Hungarian notation), you might want to call the latter policy-image-resizing, but making consistent naming choices will help you maintain your sanity as you navigate AWS. If this command completed without any errors, the policy has been created and applied to the role.

Finally, we need to create an IAM instance profile, which will let us launch an EC2 instance using the role:

$ **aws iam create-instance-profile --instance-profile-name image-resizing \**

**--output text**

INSTANCEPROFILE arn:aws:iam::740376006796:instance-profile/image-resizing 2016-06-07T06:33:09.622Z AIPAIAUTOO3GIHBKVG67G image-resizing /

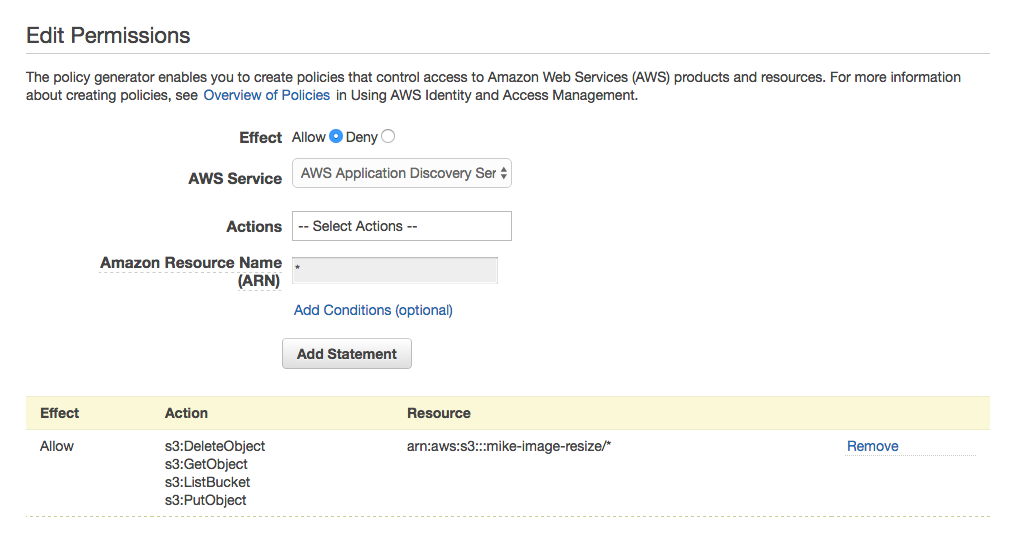
$ **aws iam add-role-to-instance-profile --instance-profile-name image-resizing \**

**--role-name image-resizing**

###### TIP

As an automation-loving sysadmin, your attachment to the cli interface should not deter you from exploring the IAM facilities found in the console. A good example of this is the policy generator wizard found in the permissions section of the users, groups, or roles tab in the IAM console ([Figure 3-2](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch03.html#policy_detail)).

The JSON policies we are using in this book were seldom hand-written, and even in such cases IAM wizards conventintly assisted with validation. We invite you to follow our example.



###### *Figure 3-12. Defining a JSON policy using the IAM wizard*

To see it in action, we can now launch a new EC2 instance that assumes this role. We do this by passing the name of the profile when running ec2-run-instances (or by selecting from the list of IAM roles in the Launch Instance Wizard). Note that IAM roles can only be assigned to new instances and that it is not possible to assign a different role to a running instance.

To launch an instance using this role, execute the following command—remembering to update the security group and the name of your SSH key pair. Note the last argument, which specifies the name of the instance profile we just created:

$ **aws ec2 run-instances --image-id ami-c80b0aa2 --region us-east-1 *--key mike \***

***--security-groups ssh* --instance-type t2.micro \**

**--iam-instance-profile Name="image-resizing"**

{

"OwnerId": "740376006796",

"ReservationId": "r-6e6b93cc",

"Groups": [],

"Instances": [

[...]

"IamInstanceProfile": {

"Id": "AIPAIAUTOO3GIHBKVG67G",

"Arn": "arn:aws:iam::740376006796:instance-profile/image-resizing"

},

[...]

Once the instance has booted, open an SSH session to it. In the previous section, we used the ec2metadata tool to display the instance’s metadata. At the time of writing, ec2metadata does not have support for IAM roles, so we must use the common Unix curl command to display the credentials. The limited-access URL to access the credentials with curl is always http://169.254.169.254/latest/meta-data/iam/security-credentials/*instance-profile-name*.

$ **curl http://169.254.169.254/latest/meta-data/iam/security-credentials/image-resizing**

{

"Code" : "Success",

"LastUpdated" : "2016-06-11T03:44:36Z",

"Type" : "AWS-HMAC",

"AccessKeyId" : "ASIAJ5HOGZLCDMXIMARA",

"SecretAccessKey" : "hPaw27itOOxc2Sq2fZxKKgzM9pbctYg5GjnAsUbI",

"Token" : "",

"Expiration" : "2016-06-11T09:55:33Z"

}

Fortunately, the authors of the AWS SDKs have decided to make things easy for us when using the client libraries. Boto, for example, has built-in support for IAM roles. If you connect to an AWS service without specifying any credentials, Boto will check to see whether it is running on an EC2 instance and whether this instance has an IAM role. If so, Boto will use the credentials supplied by IAM to authorize your API requests.

[Example 3-1](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch03.html#example_s3_resize) shows a simple Boto script that looks in the S3 bucket’s incoming directory and resizes all the files it finds. Processed files will be stored in the bucket’s processed directory.

##### ***Example 3-1. Script accessing S3 buckets***

*#!/usr/bin/python*

**import** **tempfile**

**from** **PIL** **import** Image

**import** **shutil**

**import** **sys**

**from** **boto.s3.connection** **import** S3Connection

**from** **boto.s3.key** **import** Key

IMAGE\_SIZES = [

(250, 250),

(125, 125)

]

bucket\_name = sys.argv[1]

*# Create a temporary directory to store local files*

tmpdir = tempfile.mkdtemp()

conn = S3Connection()

bucket = conn.get\_bucket(bucket\_name)

**for** key **in** bucket.list(prefix='incoming/'):

filename = key.key.strip('incoming/')

**print** 'Resizing %s' % filename

*# Copy the file to a local temp file*

tmpfile = '%s/%s' % (tmpdir, filename)

key.get\_contents\_to\_filename(tmpfile)

*# Resize the image with PIL*

orig\_image = Image.open(tmpfile)

*# Find the file extension and remove it from filename*

file\_ext = filename.split('.')[-1]

**for** resolution **in** IMAGE\_SIZES:

resized\_name = '%s%sx%s.%s' % (filename.rstrip(file\_ext), resolution[0], resolution[1], file\_ext)

**print** 'Creating %s' % resized\_name

resized\_tmpfile = '%s/%s' % (tmpdir, resized\_name)

resized\_image = orig\_image.resize(resolution)

resized\_image.save(resized\_tmpfile)

*# Copy the resized image to the S3 bucket*

resized\_key = Key(bucket)

resized\_key.key = 'processed/%s' % resized\_name

resized\_key.set\_contents\_from\_filename(resized\_tmpfile)

*# Delete the original file from the bucket*

key.delete()

*# Delete the temp dir*

shutil.rmtree(tmpdir)

This script has a few dependencies, which can be installed on Ubuntu “Trusty” systems as follows:

sudo apt install gcc python-dev python-pip

sudo pip install --upgrade boto

sudo apt install libtiff5-dev libjpeg8-dev zlib1g-dev libfreetype6-dev \

liblcms2-dev libwebp-dev tcl8.6-dev tk8.6-dev python-tk

sudo pip install pillow

###### TIP

Make sure you are using a recent version of Boto which has support for IAM roles.

Notice that the script contains no mention of access credentials. Boto will fall back to using those provided by the IAM metadata.

Save the program to a file on the instance and execute it, passing the name of your S3 bucket as a command-line argument. If everything goes according to plan, you should see something similar to the following:

$ **python image-resizing.py *your-bucket-name***

Resizing l-header-home-blinking.gif

Creating l-header-home-blinking.250x250.gif

Creating l-header-home-blinking.125x125.gif

$

On your computer (not the instance we just launched), you can now use s3cmd to show the contents of the bucket and verify that the resized images were indeed created:

$ **aws s3 ls --recursive s3://*your-bucket-name***

2016-06-11 22:32:01 8001 processed/l-header-home-blinking.125x125.gif

2016-06-11 22:32:01 26902 processed/l-header-home-blinking.250x250.gif

$

Once you have finished with the example script and your own experiments, remember to terminate the instance.

###### TIP

There are a number of problems with the previous example that would prevent you from using it in a high-traffic application, primarily because it would be impossible to scale out by launching multiple instances to process the files. Because multiple instances would be processing the bucket simultaneously, race conditions could emerge when two instances try to process the same image.

If you are building something like this for production use, Simple Queue Service (SQS) would be a better bet.

By using IAM roles, we removed the need to manually distribute and manage AWS credentials. Although there was a human behind the keyboard executing the image-resizing script, it is easy to see how IAM roles can save a lot of administration overhead, particularly when building applications based on Auto Scaling or CloudFormation.

## Using IAM Roles from Other AWS Accounts

In [“Limitations of IAM policies”](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch03.html#aws_iam_policy_limitation), we mentioned that it is not always possible to define an IAM policy that—for example—allows users from the Development department to perform an action on certain instances, while preventing them from performing the same action on instances launched by the Marketing team. For most situations, this limitation is not a huge problem; an instance might be accidentally clobbered on occasion, but that is not the end of the world.

However, in some situations you might want to have a strict border between your AWS resources. You might need to guarantee for regulatory purposes that only members of the Support team are allowed to perform any action at all on production instances. The solution in those cases where resource permissions currently fall short is to create separate AWS accounts.

A side effect of maintaining separate AWS accounts is that you will receive a separate bill for each one, separating costs automatically-with a single account, the same can be accomplished through a careful tagging strategy. AWS Consolidated Billing lets you combine the bills from multiple AWS accounts, while still seeing exactly which account is responsible for each line item. This can save a lot of time in budget meetings, as arguments over who launched those m4.10xlarge instances for testing and forgot to terminate them become a thing of thepast.

In November 2012, Amazon released a feature called cross-account API access to help customers who have gone down this route. As the name suggests, cross-account API access provides a framework for securely sharing resources between AWS accounts. Today, roles are the mechanism used to grant cross-account access. This feature is described more fully in IAM’s [common scenarios](http://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles_common-scenarios_aws-accounts.html) documentation.

# Using IAM in CloudFormation Stacks

IAM policies are a powerful tool on their own, but they become even more useful when combined with CloudFormation. Creating IAM policies and users from CloudFormation templates means that your CloudFormation stack contains everything required to run your application. The more self-contained your application is, the easier it will be to deploy and manage.

Building on the previous steps that introduced IAM roles, we will now create a CloudFormation stack that runs the image-resizing application. Everything—the S3 bucket, the IAM policy and role, and the EC2 instance that does the work—is contained within the stack template.

As always, a little planning is required before we start writing the CloudFormation stack template. First, consider the resources we need to create:

* IAM role
* IAM policy
* S3 bucket
* EC2 instance

These resources need to reference each other—for example, the stack will create a new S3 bucket and an IAM policy that references the bucket. When CloudFormation creates resources, they are given automatically generated names, which are not human-friendly and cannot be predicted in advance. Because of this, we must use the Ref function whenever we want to reference another resource in the stack.

Ref is an intrinsic function of CloudFormation templates. Passing it the logical name of a CloudFormation resource (the name you specify for the resource in the stack template) will return a value that can be used to reference that resource in other parts of the same template. It can be thought of as similar to variables in programming terms: the logical name of the resource will be replaced with the actual value at runtime.

###### NOTE

The CloudFormation template language supports built-in functions that can add a bit of flexibility to your stack templates. The full list of available functions is available on the AWS [Intrinsic Functions](http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/intrinsic-function-reference.html) page.

This example uses the Join and Ref functions to refer to other resources in the stack. Although not quite as flexible as the domain-specific language included in tools like Chef or Puppet, this small set of functions can be combined to add some interesting features to your stack templates.

With that in mind, let’s begin creating the stack template. Create a file named image-resizing.json and add the preliminary boilerplate common to all templates:

{

"AWSTemplateFormatVersion" : "2010-09-09",

"Description" : "Image resizing stack",

"Resources" : {

The first resource we will define is the S3 bucket, which is the simplest:

"ImageResizingBucket": {

"Type": "AWS::S3::Bucket"

},

This creates a simple S3 bucket with a logical name of ImageResizingBucket.

Next, we create the IAM role, profile, and policy:

"ImageResizingRole" : {

"Type": "AWS::IAM::Role",

"Properties": {

"AssumeRolePolicyDocument": {

"Statement": [ {

"Effect": "Allow",

"Principal": {

"Service": [ "ec2.amazonaws.com"]

},

"Action": ["sts:AssumeRole"]

} ]

},

"Path": "/"

}

},

"ImageResizingPolicies": {

"Type": "AWS::IAM::Policy",

"Properties": {

"PolicyName": "root",

"PolicyDocument": {

"Statement": [ {

"Effect": "Allow",

"Action": [

"s3:ListBucket", "s3:GetObject",

"s3:PutObject", "s3:DeleteObject"

],

"Resource": [

{"Fn::Join": [ "", [ "arn:aws:s3:::", {"Ref": "ImageResizingBucket"} ] ] },

{"Fn::Join": [ "", [ "arn:aws:s3:::", {"Ref": "ImageResizingBucket"}, "/\*" ] ] }

]

} ]

},

"Roles": [ {

"Ref": "ImageResizingRole"

} ]

}

},

"ImageResizingProfile": {

"Type": "AWS::IAM::InstanceProfile",

"Properties": {

"Path": "/",

"Roles": [ {

"Ref": "ImageResizingRole"

} ]

}

},

The ImageResizingRole is an IAM role that will be assigned to our instance. ImageResizingPolicies contains IAM policies (or, as in this case, a single policy) defining which actions the user is allowed to perform. Note the use of the Fn::Join andRef intrinsic functions. Ref lets us assign ImageResizingBucket, a logical name for the S3 bucket, to an actual bucket name, such as simage-resizing-imageresizingbucket-86q5y1qzusge. This is necessary as the actual bucket name will become available only at runtime.

This value is, in turn, passed to the Fn::Join function. Join combines a list of strings into a single string, separated by the given delimiter character. In this case, we use an empty delimiter ("") and join two strings to create a valid ARN for the new S3 bucket.

The second use of Fn::Join also appends /\* to the bucket’s ARN, which is used to declare actions that reference the bucket’s contents, rather than the bucket itself.

By combining Ref and Fn::Join, we can dynamically create the ARN string used in IAM policies.

The ImageResizingProfile simply acts as a container, allowing us to assign the role to an instance.

The next step is to declare an EC2 instance and a security group that will let us SSH into this instance:

"ImageResizingInstance" : {

"Type" : "AWS::EC2::Instance",

"Properties" : {

"InstanceType": "t2.micro",

"ImageId": "ami-c80b0aa2",

"KeyName": "your-ssh-key-name",

"SecurityGroups" : [

{"Ref": "ImageResizingSecurityGroup"}

],

"IamInstanceProfile": {

"Ref": "ImageResizingProfile"

},

"Tags" : [

{"Key" : "role", "Value": "image-resizing"}

],

"UserData" : {

"Fn::Base64": {"Ref": "ImageResizingBucket"}

}

}

},

"ImageResizingSecurityGroup" : {

"Type" : "AWS::EC2::SecurityGroup",

"Properties" : {

"GroupDescription" : "Allow SSH from anywhere",

"SecurityGroupIngress" : [ {

"IpProtocol" : "tcp",

"FromPort" : "22",

"ToPort" : "22",

"CidrIp" : "0.0.0.0/0"

}

]

}

}

This section creates a micro instance and assigns to it the newly created IAM instance profile. It also populates the user data with the name of the S3 bucket.

The ImageResizingSecurityGroup is a simple security group that allows SSH access from any IP address—not the most secure of firewalls, but it will serve for this example.

Remember to update the ImageID and KeyName attributes to refer to a valid AMI and SSH key pair name.

The final step is to add an Outputs section:

},

"Outputs" : {

"BucketName" : {

"Description" : "The name of the S3 bucket",

"Value" : { "Ref" : "ImageResizingBucket" }

},

"InstanceIP" : {

"Description" : "Public IP address of the newly created EC2 instance",

"Value" : { "Fn::GetAtt" : [ "ImageResizingInstance", "PublicIp" ] }

}

}

}

While not strictly required, outputs can be useful, especially when debugging new stacks. Outputs are visible in the Management Console, and can also be accessed from the command line with aws cloudformation describe-stacks. We define two outputs so we can easily see the IP address of the instance and the name of the S3 bucket.

Save all of these combined sections to image-resizing.json and create the stack:

$ **aws cloudformation create-stack --stack-name image-resizing \**

**--template-body file://image-resizing.json --capabilities CAPABILITY\_IAM**

{

"StackId": "arn:aws:cloudformation:us-east-1:740376006796:stack/image-resizing/4dc95850-3375-11e6-8a60-50d5cd148236"

}

You can watch the progress of the stack creation in the Management Console.

###### NOTE

If this command fails, make sure you have set up your command-line tools correctly. Also, check that the IAM credentials you are using have permissions to launch new instances with IAM roles and create all of the resources referenced in the stack template.

Now that the instance is running, you can connect to it and run the image-resizing script. Copy the script in [Example 3-1](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch03.html#example_s3_resize) to a file named image-resize.py and install the requirements listed in [“IAM Roles”](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch03.html#aws_access_iam_roles).

The last time we ran the script, we had to pass the bucket name as an argument. This time, we parse the bucket name from the output of the ec2metadata command. Alternatively, you could update the script to read the value directly from user data instead of a command-line argument.

As before, place an example image in the incoming/ directory of your S3 bucket and then run the following commands to resize your test image:

$ **BUCKET=$(ec2metadata --user-data)**

$ **python image-resizing.py $BUCKET**

Resizing l-header-home-blinking.gif

Creating l-header-home-blinking.250x250.gif

Creating l-header-home-blinking.125x125.gif

Although we had to log in to the instance to run the script manually, it is clear that combining all of the resources required for a particular business task into a single CloudFormation stack has benefits. Later, we will look at methods of running tasks automatically when an instance boots.

# Security Groups

Given the dynamic nature of EC2, which launches and terminates instances in response to changes in demand, it would be difficult to easily manage firewall rules with a traditional firewall, such as iptables or pf. Defining rules when you know neither the hostname nor the IP address in advance could be tricky.

AWS provides security groups as an alternative (or sometimes, a supplement) to normal firewall software. Security groups consist of a series of access rules. When launching an instance, one or more security groups are assigned to it. Their combined rulesets define which traffic is allowed to reach the instance.

VPC security groups operate on inbound and outbound network traffic, and don’t provide all the features you might be used to. If you want quality of service or deep packet inspection, or if you use your firewall logs for bandwidth reporting, you will need to combine security groups with your own firewall software. Security groups do, however, have some bells and whistles of their own, which we will look at in this chapter.

###### WARNING

When you first launch an instance from the AWS console, a security groupnamed launch-wizard-1 will be created automatically. The wizard will apply this security group to the instance being launched and, amidst prominent warnings, instruct AWS to provide unimpeded two-way ssh connectivity to them.

It can be tempting to add your custom rules to this default group and use it for all of your instances. This leads to a maintenance and security nightmare, where the most disparate services rely on the same security group’s policy, and making changes to the group itself effectively means risking breaking potentially unknown services in production.

Having a well-planned security group strategy from the beginning of a project can save a lot of headaches later.

The rules that make up a security group combine a source, a destination port, and a protocol. As in a traditional firewall, the source can be a single IP address (192.168.1.10) or a network block in Classless Inter-Domain Routing (CIDR) notation (192.168.0.0/24). Using this, you can define rules that allow your office IP address access to SSH on your EC2 instances, for example. A default rule allows all outbound traffic, but this can be deleted and replaced with more granular controls.

The source can also be the name of another Security Group, which is where they really begin to shine. Suppose you have a PostgreSQL server running on port 5432, which should be accessible only to a particular subset of your EC2 instances. Because instances are launched dynamically, you do not know their IP addresses in advance, so you cannot create rules using that method. To solve this problem, you can create security groups and dynamically assign instances to groups as the instances are created.

Also of note is the stateful nature of security groups, which permits replies to allowed traffic to flow without impediment in either direction-see [connection tracking](http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-network-security.html#security-group-connection-tracking) in the official AWS documentation for the complete details.

For this example, first create a security group. We give it a custom name, db\_clients:

aws ec2 create-security-group --group-name db\_clients --description "Database client security group"

Next, create a security group named db\_servers:

aws ec2 create-security-group --group-name db\_servers --description "Database server security group"

Finally, create a rule in the db\_servers group that allows members of the db\_clientsgroup access on TCP port 5432:

aws ec2 authorize-security-group-ingress --group-name db\_servers --protocol tcp --port 5432 --source-group db\_clients

When launching new instances, you will need to assign the newly created security groups as appropriate—for example, PostgreSQL servers in the db\_servers group. With this setup, you can easily ensure that all of your database clients can access PostgreSQL, while locking it down to protect it from external access.

###### NOTE

This method also works across AWS accounts—that is, you can reference security groups belonging to other AWS accounts within your own group rules, provided the two VPCs are first [peered](http://docs.aws.amazon.com/AmazonVPC/latest/PeeringGuide/Welcome.html).

Security groups can also reference themselves—that is, allow members of a security group to access other members of that group. To see this in action, update the db\_servers security group to allow itself access on TCP port 5432:

aws ec2 authorize-security-group-ingress --group-name db\_servers --protocol tcp --port 5432 --source-group db\_servers

Now, if you have two instances in the db\_servers group, they will each be able to access the other’s PostgreSQL instance—perfect for streaming replication.

This design pattern of role-based security group pairs is a good way of controlling access to your instances. It is likely that many types of instances will need to access your database server, such as web servers, monitoring systems, and reporting tools. Placing all of the database-access rules in a single db\_servers group gives you only one thing to change if you, for example, change the port your database is running on.

###### NOTE

At the time of this writing, AWS allows users to create 500 security groups per VPC, each spanning up to 50 rules. Up to five security groups can be assigned to each network interface, providing plenty of flexibility for the design of your security posture.

One capability that has received little notice in the transition from EC2 Classic to VPC networking is that the new model allows changing at runtime what security groups are assigned to a running instance. Security group changes are no longer limited to boot time. This has interesting applications, particularly in security incidents. Admins can now quarantine a running instance and remove it from production during forensic analysis without having to connect to it to modify its internal state in any way. This permits immediate recovery of production to proceed with new instances while incident analysis is still being conducted.

Mark Nunnikhoven has demonstrated an automated workflow that responds to security events (e.g. the detection of malware) by changing the security group of the affected instance to isolate it for analysis, then forcing the health check of an autoscaling group to fail to automatically replace the compromised instance with a new one. This process cannot resolve the underlying cause of the vulnerability, which requires human intervention, but the automated process frees the operations team to perform the analysis, knowing that automation is taking care of restoring production. Mark’s [presentation](https://markn.ca/2014/11/sec313-updating-security-operations-for-the-cloud/) is Federico’s AWS re:Invent all-time favorite, and food for thought when designing the infosec process of your production workloads. While a persistent attacker may continue to breach vulnerable instances until a fix is manually introduced, the prompt removal of the compromised instance poses a formidable obstacle to any attacker trying to expand their foothold deeper in your infrastructure.

###### TIP

As your operations grow more sophisticated, the sheer number of security groups in your account can become a challenge. While creating different security groups on a per-application basis will limit the number of rules one needs to track in a single task, security oversight becomes increasingly complex.

Anay Nayak has developed [aws-security-viz](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch03.html) as a way to manage this difficulty. Anay’s tool visualizes the rules defining a collection of security groups, enabling you to examine an entire account’s ruleset (or a select subset), and the relationships existing between your security groups.

aws-security-viz requires Ruby 2.0, and works equally well on Mac OS X and Ubuntu hosts. Setup is quite simple:

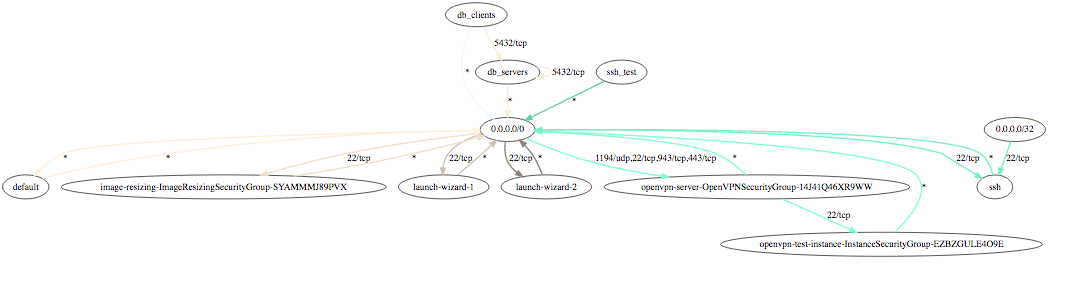
apt install graphviz

gem install aws\_security\_viz

Once installed aws-security-viz can inspect an account’s current security groups directly by invoking the AWS CLI, or import a previously saved JSON dump. Visualize an entire account’s setup with the following:

aws\_security\_viz -a *AWS access key* -s *aws secret key* -f viz.svg --color=true

Output takes the form of an SVG image ([Figure 3-13](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch03.html#security_viz)) or a small website. Consider integrating an automatically refreshing such view in your dashboards.



###### *Figure 3-13. View of security groups in the authors’ account generated with aws\_security\_viz*

# Protecting Instances with SSH Whitelists

Defense in depth is one of the key principles of successful IT security. SSH has an amazing track record when it comes to security, but there is no reason to let the whole world look for insecurities in your SSH setup. Security groups can be used to limit which IP address can access SSH on your instances, creating a whitelist of trusted addresses.

Depending on your organization, this whitelist might not change frequently, and might be small enough for you to recognize each IP address. In larger teams, maintaining the whitelist can become a chore, and it becomes difficult to know which address belongs to whom.

Implementing your SSH whitelist as a CloudFormation-managed security group can alleviate this headache and provide other benefits in the process. First, consider the alternative—manually adding and removing addresses via the Management Console. This is undesirable for a number of reasons, chiefly that there is no audit trail. If someone accidentally deletes a rule, there is no way of tracking down who did this and reverting the change.

##### MAINTAIN STRONG SECURITY POLICIES WHEN MOVING TO THE CLOUD

Securely setting up your infrastructure takes time and effort. It will sometimes be an inconvenience. The trade-off between security and convenience is well understood in the IT industry. You will need to choose the right position on this spectrum for your company’s specific situation.

One common mistake is to add a rule to your default security group that allows SSH traffic from anywhere (0.0.0.0/0). This makes it convenient to access your servers remotely, but will also result in your SSH port being probed 24 hours a day. Given how easy it is to manage SSH whitelists with security groups, there is no excuse for not taking the time to set it up.

We have seen people who should know better take some horribly insecure shortcuts on AWS, including the one just mentioned. These are things that they would never consider doing on physical hardware. Just because we are on the cloud does not mean we should forget security best practices.

AWS provides a lot of tools to securely operate your infrastructure, but it does not enforce their use—that’s up to your organizational policies.

The text-based nature of CloudFormation templates means we have an easy way of tracking changes to the whitelist—committing them to a source-control system such as Git when updating the list. This immediately gives us an audit trail, a change log, and an easy way to revert unwanted changes.

There is, however, one downside to managing whitelists in this way: the CloudFormation template syntax. Here is the section required to allow ingress from a single IP address:

"InstanceSecurityGroup" : {

"Type" : "AWS::EC2::SecurityGroup",

"Properties" : {

"GroupDescription" : "A test IP address",

"SecurityGroupIngress" : [ {

"IpProtocol" : "tcp",

"FromPort" : "22",

"ToPort" : "22",

"CidrIp" : "192.168.1.10/32"

} ]

}

}

Most of this template must be repeated for every IP address you want to whitelist. Typing this stanza over and over again will quickly get repetitive, so some people like to automate this. One common method is to have a CSV file containing IP addresses, which is used to generate the CloudFormation stack template file.

A security group created as part of a CloudFormation stack will have a name like *xxx*-ssh-whitelist. Resources created by CloudFormation have automatically generated names, which can make them a little difficult to reuse in other stacks. You will need to remember this name and reference it in your other CloudFormation stacks to assign instances to this security group. Also, if you replace this stack (i.e., delete it and re-create it), the security group will have a new name. This limitation can be worked around by using a two-stage approach to creating security groups.

Our current stack template performs two actions: creating the security group and populating it with addresses. Breaking this into two stages makes it much easier to manage security group whitelists with CloudFormation.

There are two ways to define which security group an ingress rule (as inbound security group rules are known) belongs to. In the previous example, we specified a list of ingress rules as an attribute on the AWS::EC2::SecurityGroup resource type. Thus, the rules are children of the security group, so CloudFormation implicitly knows that they belong to the parent.

The other method involves creating AWS::EC2::IngressRule resources and explicitly listing which security groups they belong to. So we can create the security group outside of CloudFormation (i.e., with the Management Console or command-line tools) and then use CloudFormation to populate the list of IP addresses.

Either way, two-stage definitions give the best of both worlds. We can control which name is assigned to our security group and still store the stack template in Git.

Now, you might be already jumping ahead and planning an even better security group layout. What about having an ssh\_whitelist group that contains further security groups such as devs\_whitelist, support\_whitelist, and so on? Unfortunately, this is not supported: security groups cannot be nested, so this will not work as expected.

# Virtual Private Networks and Security Groups

What if a whitelist is not enough? The overhead of adding and removing IP addresses from the list is not particularly grueling, but there is an overhead. If you are frantically trying to SSH into an instance to diagnose a problem that is causing production downtime, the last thing you want is to waste time updating CloudFormation stacks before you can get to work fixing things.

Or perhaps you would like an additional layer of security in front of SSH, such as a VPN server that requires client-side certificates before allowing connections.

In these scenarios, a solution based solely on security groups won’t quite cut it; we need a dedicated VPN server running within EC2. The VPN server acts as a bastion host: the secure entry point to your other instances.

This means your public servers do not need to accept SSH connections from the public Internet. Instead, their security groups can be configured to allow only those SSH connections that originate from the VPN instance. You no longer need to worry about script kiddies probing your ports, and there is no SSH whitelist to manage.

Because the instances will not need to receive any inbound connections from the public Internet, we can use Amazon’s Virtual Private Cloud service in this example.

##### AMAZON VIRTUAL PRIVATE CLOUD

Amazon Virtual Private Cloud (VPC) is the current iteration of EC2’s networking feature set, designed to improve the security of your EC2 instances. When EC2 was launched, the default network model (now referred to as “EC2 Classic” and no longer available to new accounts) automatically assigned a public IP address to each instance. Although instances were still protected by a security group, they were routable from the public Internet by default.

In 2009, Amazon introduced VPC. The VPC model allows users to create logically isolated sections in their network architectures, rather than having a single section containing all of their EC2 instances.

VPC makes it possible to create a variety of network topologies within AWS. This has benefits for people using AWS as an extension of their own datacenter, or those with specific network security requirements that cannot be satisfied by security groups alone.

VPC introduced components to EC2 that emulate features found when running your own network outside AWS. These include subnets and routing tables, network access control lists (ACLs), and the ability to specify IP address ranges for your EC2 instances and other AWS resources.

In 2013, VPC became the default for new AWS accounts. Upon using an EC2 region for the first time, a default VPC is automatically created, including a default subnet, routing table, and other required components of a VPC. This process is essentially invisible to the user: new EC2 instances will be launched in the default VPC automatically, as we have seen in [Chapter 2](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch02.html#chap_first_steps)

Our example will include a VPC consisting of two subnets. The first subnet will be accessible via the public Internet, and will be home to our bastion host. The second subnet will be private: it will not be routable from the Internet, and will be accessible only to the bastion host after we implement the required routing and access control rules.

As a demonstration, we will use the free OpenVPN as our VPN server, although the same general procedure will apply to other VPN servers as well. Luckily, the makers of OpenVPN have published an AMI that contains everything you need to run an OpenVPN server. To cut down on the installation time, we will use this ready-to-go AMI instead of installing it from scratch ourselves.

In the security groups section, we looked at some strategies for defining Security Groups for client/server applications such as PostgreSQL. In this section, we will be creating two security groups:

openvpn

This will be assigned to OpenVPN instances, and will allow the public Internet to access the OpenVPN server.

protected\_ssh

This will be assigned to all other instances, and will allow SSH access only from the OpenVPN instance(s).

These security groups will be created as part of the CloudFormation stacks.

The instance could be launched in any number of ways. We are going to use CloudFormation so we can have one stack that contains the VPN instance and security group, and another that contains a protected EC2 instance and its own security group.

You can find the ID of the OpenVPN AMI by searching for it in the Launch Instance Wizard in the Management Console. Alternatively, OpenVPN maintains a list of AMIs for each EC2 region in the [EC2 Appliance (AMI) Quick Start Guide](http://docs.openvpn.net/how-to-tutorialsguides/virtual-platforms/amazon-ec2-appliance-ami-quick-start-guide/).

At the time of writing, the ID for the OpenVPN AMI in us-east-1 is ami-7ab25917. You will need to replace this if you are using a different region, or if a new OpenVPN AMI has since been created.

The OpenVPN AMI has two configuration phases. First, the OpenVPN installation process requires some configuration data such as the instance’s hostname and the admin user’s password. This takes place when the instance is booting. This configuration data can be provided as user data or entered using the OpenVPN installer on the command line.

The second configuration phase takes place after OpenVPN is installed and running, and is done through the OpenVPN web interface. It is at this point that you can create additional users who will be allowed to access the VPN.

In this example, we will perform the first configuration stage manually using the OpenVPN installer. Although using user data is more convenient, it will leave the OpenVPN administrative account password visible in both the CloudFormation and EC2 web consoles, which is not desirable.

[Example 3-2](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch03.html#openvpn_cloudformation) shows the CloudFormation stack we will use to create the OpenVPN instance and associated resources.

##### ***Example 3-2. OpenVPN CloudFormation stack***

{

"AWSTemplateFormatVersion" : "2010-09-09",

"Description" : "OpenVPN EC2 Instance and Security Group",

"Parameters" : {

"KeyName": {

"Description" : "EC2 KeyPair name",

"Type": "String",

"MinLength": "1",

"MaxLength": "255",

"AllowedPattern" : "[\\x20-\\x7E]\*",

"ConstraintDescription" : "can contain only ASCII characters."

},

"AllowedIPRange" : {

"Description" : "IP Range allowed to access OpenVPN via SSH and HTTP(S)",

"Type": "String",

"MinLength": "9",

"MaxLength": "18",

"Default": "0.0.0.0/0",

"AllowedPattern": "(\\d{1,3})\\.(\\d{1,3})\\.(\\d{1,3})\\.(\\d{1,3})/(\\d{1,2})",

"ConstraintDescription": "Must be a valid IP CIDR range of the form x.x.x.x/x."

},

"AMI" : {

"Description" : "OpenVPN AMI ID",

"Type": "String"

}

},

"Resources" : {

"OpenVPNInstance" : {

"Type" : "AWS::EC2::Instance",

"Properties" : {

"InstanceType" : "t2.micro",

"SecurityGroups" : [ { "Ref" : "OpenVPNSecurityGroup" } ],

"KeyName" : { "Ref" : "KeyName" },

"ImageId" : { "Ref" : "AMI"},

"SourceDestCheck" : "false"

}

},

"OpenVPNSecurityGroup" : {

"Type" : "AWS::EC2::SecurityGroup",

"Properties" : {

"GroupDescription" : "Allow SSH, HTTPS and OpenVPN access",

"SecurityGroupIngress" : [

{

"IpProtocol" : "tcp",

"FromPort" : "22",

"ToPort" : "22",

"CidrIp" : { "Ref" : "AllowedIPRange"}

},

{

"IpProtocol" : "tcp",

"FromPort" : "443",

"ToPort" : "443",

"CidrIp" : { "Ref" : "AllowedIPRange"}

},

{

"IpProtocol" : "tcp",

"FromPort" : "943",

"ToPort" : "943",

"CidrIp" : { "Ref" : "AllowedIPRange"}

},

{

"IpProtocol" : "udp",

"FromPort" : "1194",

"ToPort" : "1194",

"CidrIp" : { "Ref" : "AllowedIPRange"}

}

]

}

}

},

"Outputs" : {

"InstanceId" : {

"Description" : "InstanceId of the OpenVPN EC2 instance",

"Value" : { "Ref" : "OpenVPNInstance" }

},

"OpenVPNSecurityGroup" : {

"Description" : "ID of the OpenVPN Security Group",

"Value" : { "Fn::GetAtt" : [ "OpenVPNSecurityGroup", "GroupId" ] }

},

"PublicIP" : {

"Description" : "Public IP address of the newly created EC2 instance",

"Value" : { "Fn::GetAtt" : [ "OpenVPNInstance", "PublicIp" ] }

}

}

}

Save this stack template to a file named openvpn.json and create the stack with the CloudFormation command-line tools:

aws cloudformation create-stack --stack-name openvpn-server \

--template-body file://./openvpn.json --region=us-east-1 \

--parameters ParameterKey=KeyName,ParameterValue=*your-key-name* \

ParameterKey=AllowedIPRange,ParameterValue=*0.0.0.0/0* \

ParameterKey=AMI,ParameterValue=*ami-7ab25917*

This stack template introduces a new feature of CloudFormation: parameters. Parameterscan be thought of as variables within your stack template. They can be given a default value that can be overridden when creating the stack. They are not quite as flexible as variables, as they can be set only once (when launching or updating the stack), but they do allow for a certain amount of reusability within your stack templates.

The parameters required by this stack are specified on the command line with the syntax ParameterKey=*KeyName*,ParameterValue=*value*. You will need to replace the SSH key name and IP range with your own values.

If you are not launching the stack in the us-east-1 region, you will also need to change the AMI parameter to match the OpenVPN AMI ID for your region. You can find the ID on the OpenVPN [AWS AMI Marketplace](https://aws.amazon.com/marketplace/pp?sku=a6vjvrelz10rgvvemklxv2dow) page.

After the stack has been created, we can find the IP address of the instance by querying its outputs:

aws cloudformation describe-stacks --stack-name openvpn-server

This command will output a full description of the stack, including the outputs and their values. Instead of searching through this information manually, you could use the jq tool to filter the JSON and print only the required values. We use the following filter:

aws cloudformation describe-stacks --stack-name openvpn-server | \

jq '.Stacks[0].Outputs[] | select(.OutputKey=="OpenVPNSecurityGroup" or .OutputKey=="PublicIP").OutputValue'

This command will parse the JSON and print the OutputValue for the OpenVPNSecurityGroup and PublicIP outputs, for example:

"54.164.47.241"

"sg-feccae85"

Now that the instance is running, it must be configured. Begin by connecting to the instance via SSH as the openvpnas user. After logging in to the connection, an OpenVPN configuration process will be automatically started. The default choices presented by the application are suitable for our uses, so press the Enter key on each line to accept them.

Once this process exits, you will need to set the password for the openvpn user, used to configure OpenVPN through its web interface. Generate a password and set it by executing the following:

sudo passwd openvpn

You can now open the configuration page in your web browser. The address will be displayed after the OpenVPN configuration process completes, and will be something like https://54.77.153.76:943/admin.

When you open this address in your web browser, you should see the OpenVPN welcome page. Using the [OpenVPN Quick Start Guide](https://docs.openvpn.net/how-to-tutorialsguides/virtual-platforms/amazon-ec2-appliance-ami-quick-start-guide/), you can now configure the OpenVPN server according to your requirements.

###### NOTE

It is possible to create DNS records from CloudFormation templates, so we could, in fact, set up a CNAME so we can access this instance by visiting, for example, vpn.example.com.

After the VPN server has been configured, you should now be able to connect to the VPN, using the OpenVPN documentation for your platform of choice.

###### NOTE

This OpenVPN sever is a single point of failure, which is not desirable when it is the only way you can SSH into your other instances. Before using a solution like this in production, you should explore methods of making this system more robust. For example, you could run an Auto Scaling group with one or more instances of OpenVPN so that failed instances are automatically replaced.

Now that the VPN server is working, we can verify that it is working as expected when it comes to protecting our instances. We will do this by launching a new instance and assigning it to the protected\_ssh security group. [Example 3-3](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch03.html#cloudformation_protected_instance) shows a simple CloudFormation stack template that declares a single instance using this security group.

##### ***Example 3-3. Launching a protected instance with CloudFormation***

{

"AWSTemplateFormatVersion" : "2010-09-09",

"Description" : "Example EC2 instance behind an OpenVPN server",

"Parameters" : {

"KeyName": {

"Description" : "EC2 KeyPair name",

"Type": "String",

"MinLength": "1",

"MaxLength": "255",

"AllowedPattern" : "[\\x20-\\x7E]\*",

"ConstraintDescription" : "can contain only ASCII characters."

},

"AMI" : {

"Description" : "AMI ID",

"Type": "String"

},

"OpenVPNSecurityGroup" : {

"Description" : "OpenVPN Security Group ID",

"Type": "String"

}

},

"Resources" : {

"Ec2Instance" : {

"Type" : "AWS::EC2::Instance",

"Properties" : {

"InstanceType" : "t2.micro",

"SecurityGroups" : [ { "Ref" : "InstanceSecurityGroup" } ],

"KeyName" : { "Ref" : "KeyName" },

"ImageId" : { "Ref" : "AMI"}

}

},

"InstanceSecurityGroup" : {

"Type" : "AWS::EC2::SecurityGroup",

"Properties" : {

"GroupDescription" : "Allows SSH access from the OpenVPN instance",

"SecurityGroupIngress" : [

{

"IpProtocol" : "tcp",

"FromPort" : "22",

"ToPort" : "22",

"SourceSecurityGroupId" : { "Ref" : "OpenVPNSecurityGroup"}

}

]

}

}

},

"Outputs" : {

"PrivateIP" : {

"Description" : "Private IP address of the EC2 instance",

"Value" : { "Fn::GetAtt" : [ "Ec2Instance", "PrivateIp" ] }

},

"PublicIP" : {

"Description" : "Public IP address of the EC2 instance",

"Value" : { "Fn::GetAtt" : [ "Ec2Instance", "PublicIp" ] }

}

}

}

Save this file as protected\_instance\_cloudformation.json and execute the following command to create the stack:

aws cloudformation create-stack --stack-name openvpn-test-instance \

--template-body file://protected\_instance\_cloudformation.json \

--region=us-east-1 --parameters ParameterKey=KeyName,ParameterValue=*federico* \

ParameterKey=AMI,ParameterValue=*ami-c80b0aa2*

ParameterKey=OpenVPNSecurityGroup,ParameterValue=*sg-feccae85*

As before, you will need to adjust some of the parameters to match your environment. The value for the OpenVPNSecurityGroup should be the value retrieved from the describe-stacks command executed earlier.

Find out the public and private IPs of the instance by running the following:

aws cloudformation describe-stacks --stack-name openvpn-test-instance |\

jq '.Stacks[0].Outputs[]'

Once the stack has been created, make sure your VPN is disconnected and try to SSH to the public IP of the instance. This should time out, because your public IP address is not allowed to access instances in this security group.

Connect to the VPN and then try to SSH to the instance’s private IP address. This time you should be presented with the familiar Ubuntu prompt, confirming that your security groups are doing their job and traffic is being routed over the VPN.

Setting up the OpenVPN server and performing ongoing maintenance adds overhead that is not present when working with security groups on their own. However, for some companies, the additional layer of security is a must. Managing this stack with CloudFormation will keep the maintenance to a minimum.

This is just an introduction to running OpenVPN with CloudFormation. TurnKey Linux has published an example CloudFormation template that includes a VPN and separate subnets for the OpenVPN server and protected instances. Available on their [GitHub page](https://github.com/turnkeylinux-apps/openvpn/blob/master/contrib/cloudformation_vpc_openvpn.template), this is a great starting point for building highly available and secure VPNs in Amazon’s cloud.

# A Security State of Mind

Security is a job by its very nature never complete, and subject to some of the strongest trade-offs in IT. Too little security may be as bad as too much security-the former leading to catastrophic incidents, the latter preventing your team from being effective. We have laid a solid foundation for your security thinking in a public cloud environment, and we offer a few key takeaways:

* Most of the security concepts you already know have direct parallels in AWS.
* Manually maintaining security groups and whitelists is time-consuming and error prone—don’t do it. Automate these processes using CloudFormation or your own custom scripts.
* It is a lot easier to build a secure infrastructure from the beginning than it is to improve security on a running infrastructure.
* Time spent defining a sensible IAM policy at the beginning of a project will pay off in reduced headaches later in the project.
* The cloud is not a magic place where security rules do not apply. Just as your datacenter managers would not prevent you from deploying a physical server with a misconfigured firewall, AWS will not prevent you from building an insecure infrastructure.
* Public cloud security may be stronger than a private datacenter’s when properly architected around a clear understanding of the shared responsibility model, particularly for smaller IT teams that are invariably resource-constrained.
* A VPN-based bastion host can add a security and auditing layer, at the cost of increased maintenance.
* Allowing SSH from 0.0.0.0/0 is nearly always a bad idea.

[1](https://www.safaribooksonline.com/library/view/aws-system-administration/9781449342562/ch03.html#idm140367897945616-marker)A sad cautionary tale about AWS security is that of [Code Spaces](https://threatpost.com/hacker-puts-hosting-service-code-spaces-out-of-business/106761/).