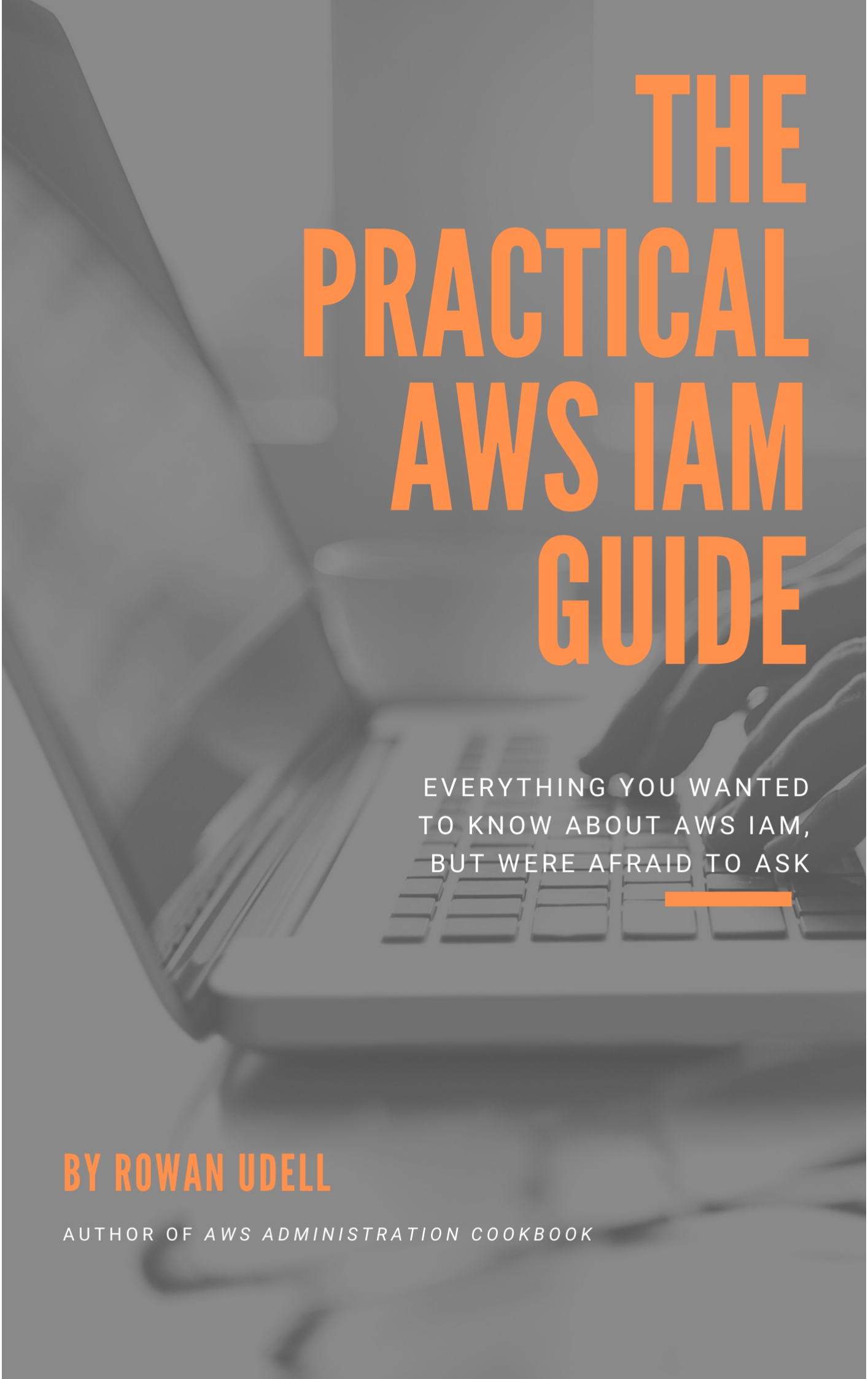


THE PRACTICAL AWS IAM GUIDE

EVERYTHING YOU WANTED TO KNOW ABOUT AWS IAM, BUT WERE AFRAID TO ASK

BY ROWAN UDELL

AUTHOR OF *AWS ADMINISTRATION COOKBOOK*

The Practical AWS IAM Guide

Rowan Udell

Table of Contents

Foreword

IAM is weaved into every corner of AWS. It is a fundamental building block in your journey through the cloud and one with massive consequences if the key concepts or the actions you perform are misunderstood. At times, I've certainly got it very wrong myself! Taking the time to really understand this space means you're not only taking the steps to secure your cloud, but also being able to explain the why behind the designs and decisions you have made - whether that is to a colleague, your manager, or the CISO of your company.

I first met Rowan as a part of the now defunct Cloud Warriors program, an AWS-led program that enables consultants who are employed by AWS partners to develop their technical expertise by engaging with service teams and each other. In the years since it was renamed the AWS Ambassador Program which continues to this day. My favorite part of being in this program is hearing from other experts about what they are working on, the challenges they face, and how they overcome them. Rowan has always been front and center in these discussions and is a clear expert in this field.

It was through discussions like these that I decided to get deeply involved in the IAM space. I often produced open-source projects (some of which are mentioned within this guide) that would attempt to solve the problems I found my customers experiencing, and would often get help from the community in return in the form of bug reports and pull requests. Everyone feels the pain and wants to help.

Reading through this guide was a refreshing experience. Not only is the AWS documentation verbose and often hard to navigate, it's also written by AWS themselves. This means you don't hear about the downsides and gotchas that often appear just as you're going through the steps to secure your cloud. This guide calls it out before you're 3 hours in, pulling your hair out and wondering why your role just turned into a bunch of letters starting with AROA.

For those just starting your cloud journey, don't be afraid of the bluntness of IAM. It's a complicated topic, but Rowan has condensed years of knowledge and experience down into this easy to digest format. Let this guide be your cheat sheet.

Oh, and don't use the root user.

Ian Mckay

AWS Hero, AWS Ambassador, and AWS Community Builder

Introduction

Welcome to *The Practical AWS IAM Guide*. By using this guide, you will learn the most relevant and valuable parts of IAM so that you can get the most out of using Amazon Web Services (AWS).

Though simple at its core, IAM can get complicated. Like any AWS service, IAM has a nontrivial learning curve. If you don't take the time to learn IAM, it's likely you will have issues with your IAM configuration given enough time.

IAM is the only service that's not optional on AWS. If you want to work with any of the other services on AWS, you must talk to IAM.

The biggest problem most builders on AWS have is that IAM is often an afterthought. As soon as they have something that works, they tend to move on to other potentially more interesting challenges. Though this approach might work to start with, it's not sustainable or secure.

The IAM service has extensive official documentation, but that doesn't mean you can find the answers right you need when you need them. Given the scope of IAM and its criticality to your applications, if something goes wrong, it can go wrong *bad*. Just ask companies such as Capital One 🔗. To be clear, issues like this don't mean that IAM is not a secure service. They highlight that using it properly is not easy. What does it mean for you and your company if these large, well-funded, teams and companies struggle to use IAM?

I wrote this guide because I struggled personally with IAM. I also worked with teams and customers professionally who struggled with IAM. I've learned a lot of the lessons shared in this book the hard way, though thankfully not as hard as some. It took years to do so, and I want to save you the time and effort it took me. This book is what I wish I had when I was learning IAM.

Though AWS has always been fantastic in releasing new services and materials to help their customers, there's a limit to even what they can do. Outside the official documentation, there's not a lot of focus on IAM. IAM gets a mention in many courses and other materials, but it's often a side-note or minor detour before covering the main services in focus.

Who This Book is For

> We are drowning in information but starved for knowledge.
>
> — John Naisbitt

This book is for those who want to be confident using IAM. The goal of the book is to save you as much of the pain that comes with the IAM learning process as possible.

The authoritative source about IAM is AWS itself. In my mind, there's no argument that the AWS documentation is *exhaustive*, although sometimes I might tweak that description slightly and say it can be **exhausting**. The official documentation is extensive and of a high quality, it can also be overwhelming a challenging to navigate. It would be hard to argue that everyone needs to know all of it, and know it all of the time.

This guide aims to distill all that content down to concise, helpful, and focused information. By making the key points accessible, you should be able to navigate the rest of the content out there confidently when you need to. Ideally, you will spend as little time on IAM as possible without sacrificing your security and control. That way, you can spend your time doing something else that delivers value to your customers, your business, and you.

I have aimed this book at professionals who will be working with IAM on AWS regularly. By fast-tracking your knowledge and understanding of IAM, you can get more value out of AWS. Your applications and data security are not where you want to be making rookie mistakes.

You do not need to be an expert in AWS to use this book, but I hope you will learn something of value even if you are. Like any sufficiently large and vital AWS service, some of the edges and details may still surprise you. Ideally this guide can be a reference for you and others to make the IAM learning curve more manageable. I thought I knew IAM well before writing this book, and I was shocked by how much more I learned while writing this book.

If you have "DevOps" or "solutions architect" in your job title, or the title you aspire to, then I'm confident you'll get a lot of value from this book. This guide should give you the confidence that you've done your job well and securely. By the time you reach the end of this book, I think you might be surprised when you review the sample policies recommended and used in the real world, including the official AWS documentation and resources.

If you're a developer building applications on AWS, this book will help you develop your applications securely. You'll be able to take your applications for security reviews and penetration testing with a solid reasoning behind your IAM implementation and decisions.

If you work in security, this book will get you up to speed with IAM on AWS quicker and easier than trawling through the official documentation. This book can be a reference about what works for others on AWS, so that you can compare it to your existing approach from other platforms.

A big part of my decision to write this book was realizing that, as AWS says, security is "Job Zero". By improving the security of your applications you make life better for your customers and you, which is what matters in your professional life.

If you have feedback about how to improve this book, please send it to me at rowan@rowanudell.com or on Twitter @elrowan.

Practical Over Complete

Ultimately I want you to be comfortable with IAM so that you can use it proficiently and sleep soundly knowing you've done it well.

Though this book isn't an exhaustive list of all IAM features and explanations of everything you can do with IAM, it is a practical guide to get the most out of AWS and be a launchpad to future learning. IAM is extensive. Its official user guide is over 1,100 pages long. Covering all of the service is impractical and not the purpose of this book.

Conceptually, this book will teach you:

- The building blocks of IAM
- How the aspects of IAM interact with each other, and what happens when they disagree
- How to craft your IAM policies so that you can be confident they are secure
- How to review commonly policies that you can use them for your own needs
- What complimentary AWS services will enhance IAM and improve the effectiveness of your work
- The open-source projects out there that can help you review, assess, and manage your policies

By covering the topics at a practical level, you will have enough knowledge to go forth and discover any level of detail. Where possible, I've included relevant links to the official documentation and references to other quality community sources and tools.

While I would love for this to be the ultimate guide to everything identity related on AWS, that would not be **practical**. This means I have had to limit or skip a few topics that

I would like to have covered, but couldn't fit in the book:

- AWS IAM Roles Anywhere is useful in certain hybrid environments, but s not relevant to all or even most of AWS users, so is not addressed.
- AWS IAM Identity Center has been renamed to be part of the IAM family of services, and was originally its own service called AWS Single Sign-On before that. Federation is a one-off task in an environment, and entails a lot of vendor-specific quirks and requirements, so I have focused on the IAM parts of related to federation.
- Amazon Cognito is an identity service, but for your own applications, and not AWS itself. It has its own set of challenges when it comes to documentation and examples, and would require its own book's worth of content to do justice.

Conventions in This Book

To avoid repetition will refer to the AWS Identity and Access Management service as "IAM". When talking about the general identity concepts that underpin the service I will refer to "identity and access management" in full, rather than abbreviating it as "IAM" as you may see in other material. This is consistent with the official AWS documentation.

I have defined variables in examples by using `SCREAMING_SNAKE_CASE` so that they stand out without invalidating things like ARNs and policies. You should replace these examples with your own values these before using them.

Some examples are:

- An Amazon S3 bucket: `arn:aws:s3:::MY_BUCKET_NAME/*`
- An IAM role: `arn:aws:iam::ACCOUNT_ID:role/ROLE_NAME/*`
- An AWS Lambda function: `arn:aws:lambda:REGION:ACCOUNT_ID:function:FUNCTION_NAME`

Where useful, I have included examples by using the AWS Command Line Interface (AWS CLI) Version 2. The AWS CLI is available for most operating systems, and you will find that the commands closely match the underlying service APIs. If you want to follow along with the example command in the scenarios, use the official AWS getting started documentation.

Links include icons to let you know where you'll go if you click on them. The aws icon links to official documentation, the 🔗 to web-based resources, and links with no icon are to other sections of the book.

What is Identity and Access Management?

One of the biggest challenges with AWS Identity and Access Management (IAM) is that most developers haven't taken the time to learn about the general concepts behind identity and access management. Identity and access management is an area of expertise and specialty on its own. When you use IAM well, your security is transparent to users. When you use IAM poorly, IAM can block, frustrate, and confuse your users. Though possible to work as an AWS developer without appreciating IAM, you can go much further by understanding IAM. I was guilty of this too, and it took me years of working with AWS before I devoted time and effort to understanding the underlying concepts properly. Though this book is about the AWS implementation of identity and access management, the concepts you learn here will help you more broadly with security too. By taking a step back to learn about the concepts that go into the "why" behind identity and access, you can use it to learn IAM better, and make leaps of understanding without needing extensive explanation.

As mentioned in the Conventions in This Book, I will use "IAM" to refer to the AWS Identity and Access Management service, and "identity and access management" when referring to general concepts.

At a high level, two main areas make up identity and access management: **authentication** and **authorization**.

Authentication

The identity management part of IAM is *authentication*. Authentication validates that an entity is what who they claim to be. You might see "authentication" abbreviated as "authn". This section covers how IAM authenticates entities' access to AWS services.

An external **entity** is something that can authenticate with IAM. Once authenticated, the entity has an **identity** with IAM. Identities have **attributes**, some that are unique to them, and others they inherit, which can used to further control how IAM treats them.

The official IAM documentation about terms uses "entity" to describe user and role principals. In this book I will use "entity" to refer to the pure identity and access management entity, and "IAM entity" to refer to a user or role.

To authenticate itself, an entity must provide information that only they and IAM would

know, such as a combination of a unique username and correct password. The IAM can optionally enhance this "shared secret" with extra information such as an Multi-Factor Authentication (MFA) token.

Though a great deal of available content out there covers the nuances of identity and access management for all kinds of systems, you don't need to be an expert to do the right thing, particularly in the context of AWS. The same basic principles consistently and broadly applied to your AWS resources will give you a strong security stance. This guide focuses on the practical implications and leaves the more abstract discussions to other content.

Authentication methods are generally a combination of two or more of the following:

- Something you are
- Something you know
- Something you have

In the context of IAM, these are:

- Something you are: a username
- Something you know: a password
- Something you have: a physical device such as an MFA token generator

The more methods you use, the more secure your authentication.

Identity as the Perimeter

Historically, the most critical perimeter for applications and computer systems was the network perimeter. As the perimeter of an environment, it represented a clear delineation between the trusted internal environment, and untrusted external environments. If you had access to a system, you could generally make it do what you wanted because of relaxed security models, and assumptions about the physical security of computers. These assumptions are not valid in today's cloud-enabled world, where the AWS control pane is on the public internet, and networks are now software-defined constructs. The changes associated with the cloud have made the identity the essential part of the perimeter to control, since in AWS the right identity can change the network perimeter.

Authorization

The access management part of IAM is *authorization*. Authorization validates that an already authenticated entity is permitted to perform the action that it is requesting. You

might see "authorization" abbreviated as "authz". Authorization follows authentication in the identity and access management process.

A n authenticated entity is not enough to grant it full access to the all functionality available, because not all entities have the same level of access to the system. For example an administrator has more access than a normal user. In physical terms, just because you're allowed into the bank doesn't mean you can get in to the vault.

Even an entity such as an administrator might not want to have all its permissions at their fingertips all the time, so they might intentionally limit themselves to a smaller subset of actions for safety reasons. The *blast radius* of what can go wrong, or the worst-case scenario, is reduced by appropriate authorization. This is because the blast radius of an entity is not related to the *intent* of the entity. The entity might be a "good" user who unintentionally does something terrible, or the entity might be a "bad" user looking to cause damage to your resources. The only practical strategy is to limit the worst-case impact wherever possible.

In IAM, policies represent the authorization level of an identity. Policies use a consistent and unambiguous syntax to define allowed actions for an identity as well as which actions are allowed or denied.

How IAM Works

Now that you know the basics of identity and access management, it's time to get practical with IAM. This chapter is an overview of IAM so that you can navigate both the AWS service and its documentation.

Though it's hard to imagine AWS without IAM today, in the early days all interactions with AWS were performed by the root user. AWS released AWS Identity and Access Management (IAM) in 2011, and this history still shows up sometimes, and can explain some of the rough edges and quirks you might come across.

IAM is one of the global services that AWS offers. Like other AWS services, IAM is "eventually consistent", which means you might experience some delay seeing your changes in other regions of AWS. On a daily basis, this delay is almost unnoticeable unless AWS is suffering an interruption or degradation of service. As with most security-related components of your system, you should limit the amount of change to your IAM configuration as much as you can. This helps protect you from misconfiguration, and also reduces the impact of any propagation delays in IAM.

AWS global services are served from the North Virginia (`us-east-1`) Region, which was the first AWS Region. Though this might make it seem like IAM would be unavailable if there were issues in that Region, AWS designed IAM with this in mind. In practice an issue in `us-east-1` will probably impact your ability to create and change IAM resources, but services reading/using IAM in other AWS Regions will continue to work with already provisioned resources.

As you would hope for the key security service on AWS, the service has achieved most popular compliance frameworks, such as SOC.

IAM and its sub-services, such as AWS IAM Access Analyzer have no cost associated with their functionality. IAM actions are recorded in the user activity and API tracking service AWS CloudTrail, which does charge for recording and storing data beyond a default retention period, which is why AWS offers CloudTrail as a separate service from IAM.

Data and Control

IAM is how you control access to the *control plane* of AWS, allowing you to use other AWS services. The control plane contrasts with the *data plane*, which is where most of your applications and data reside.

Examples of data plane access that *are* IAM controlled include: accessing records in a Amazon DynamoDB table, reading logs from Amazon CloudWatch Logs, and reading

data in Amazon S3 objects. Data plane access that **are not** IAM controlled include: SSHing to an Amazon EC2 instance, and connecting to a Microsoft SQL Amazon RDS instance Generally, the more *cloud native* a service is, the more it will use IAM to authenticate access.

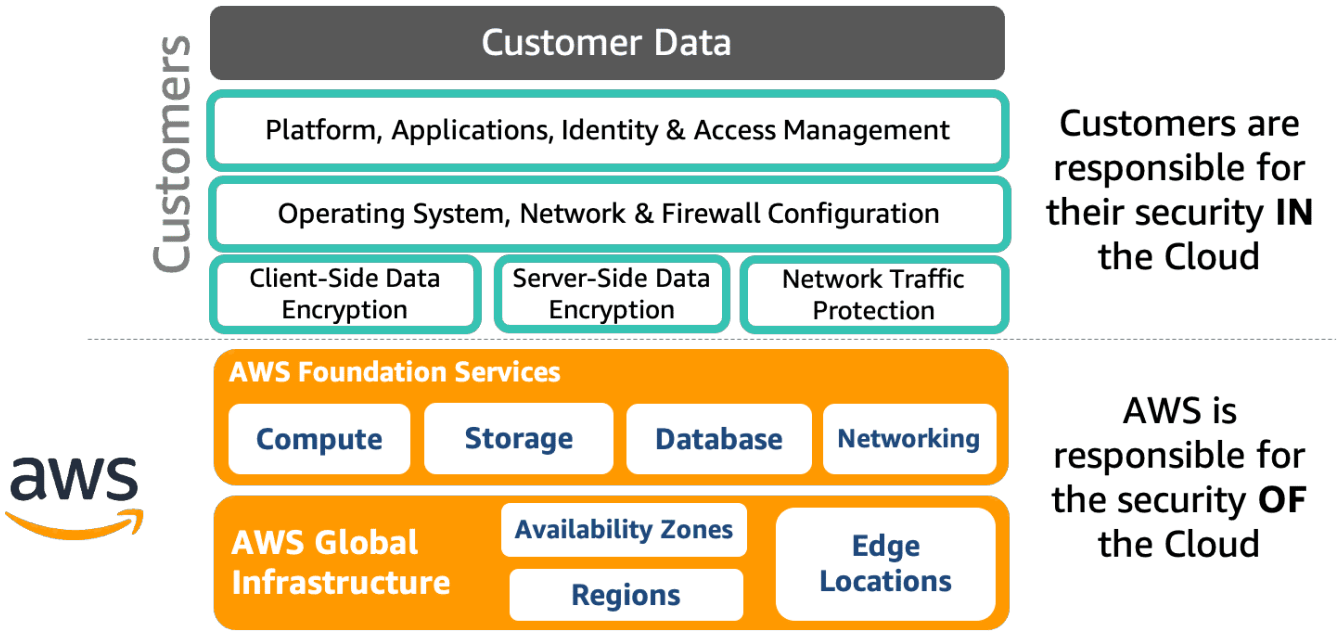
The Shared Responsibility Model

The AWS Shared Responsibility Model underpins the way you use AWS, and is relevant to how you use IAM. The model defines the following tenants of cloud usage:

- AWS is responsible for the "security **of** the cloud"
- Customers are responsible for their "security **in** the cloud"

There's only one small word difference between those two tenants, but it has significant implications. Though AWS will do everything they can to provide you secure services by default, you must take responsibility for how you use them.

AWS provides a visualization that shows you the separation of concerns in action:



Source: Deep Dive with Security: AWS Identity and Access Management

As you can see, "Identity & Access Management" sits in the customer's area of responsibility. IAM is the first opportunity to secure your AWS environment and keep up your side of the Shared Responsibility Model. Another way to express the model is:

- AWS provides you secure services
- How you use those services is up to you

The underlying implication is that just because an application runs on AWS doesn't mean it's *automatically* secure. There are actions that you can and should take to ensure the security of your application in the cloud, and IAM is a big part of that.

This model has implications for meeting compliance framework compliance such as PCI-DSS, HIPA, and others. If these apply to you, check out the lesser-known service, AWS Artifact, where AWS keeps the latest versions of compliance reports for their services.

Requests

Your interaction with AWS comes in the form of a request to AWS service API, and all requests get evaluated by IAM. This evaluation happens in a *context*, which helps IAM decide if to allow or deny the request. This context include the requested *action* which represents the activity you want the AWS service to perform. If IAM allows a request, the request is passed to the target service for action. If IAM denies a request, the target service never receives the request.

At a high level, the following rules apply to all requests:

- **Default deny**: By default, all actions are denied, which is how AWS is "secure by default".
- **Explicit allow required**: An action is allowed if explicitly mentioned somewhere in the policy chain. A policy might use a *wildcard* to allow an action, which means that the action might not be visible in any policies.
- **Explicit deny wins**: If there is an explicit denial of the action requested in your evaluation chain, IAM denies the action. Though an explicit allow must be present for an action to have any chance of succeeding, it's still not always enough.

This is a simplified version of the rules that you should always keep in mind when thinking about requests. In Policy Evaluation you will see how different policy types interact and exceptions to the rules.

Making Requests

You can send requests to AWS APIs by several different methods. The most common ways are:

- The AWS Management Console
- The AWS CLI tool

- A language-specific AWS SDK
- Directly to an AWS HTTPS API endpoint

Regardless of the method you use, all requests ultimately result in a signed request to an HTTPS API endpoint. The different methods only just change how much heavy lifting you have to do for yourself, and how much is done for you.

Signing Requests

Almost all requests made to AWS must be signed by using the AWS Signature Version 4 process, often abbreviated as "SigV4". Though the ins and outs of signing are beyond the scope of this book, it's still good to know about the process at a high level.[1]

This signing process uses the access key and secret key you receive as part of the authentication process from IAM or AWS STS. Signing requests ensures that the requests are being made by an authenticated entity; for a specific service, AWS Region, and time; and without sending secrets in the request. If you use the AWS CLI or SDKs, this process is taken care of for you. You only need to sign your requests if you use a programming language that lacks an official SDK.

Request Context

All requests to AWS happen in a context. This context becomes important when you start to include conditions in your policies, because the conditions check the context decide if policy statement applies to a request.

The request context is a superset of the actual request made by the client, since it includes more than the information submitted as part of the request. IAM builds the request context from the following sources of information:

- **Actions**: What the request wants to accomplish (for example, invoke a Lambda function).
- **Resources**: The resources targeted by the action (for example, the invoked Lambda function).
- **Principal**: The entity that is making the request (for example, a role).
- **Environment data**: Additional data about the request (for example, the IP address or the date and time).
- **Resource data**: Additional data about the target resource (for example, tags on the resource).

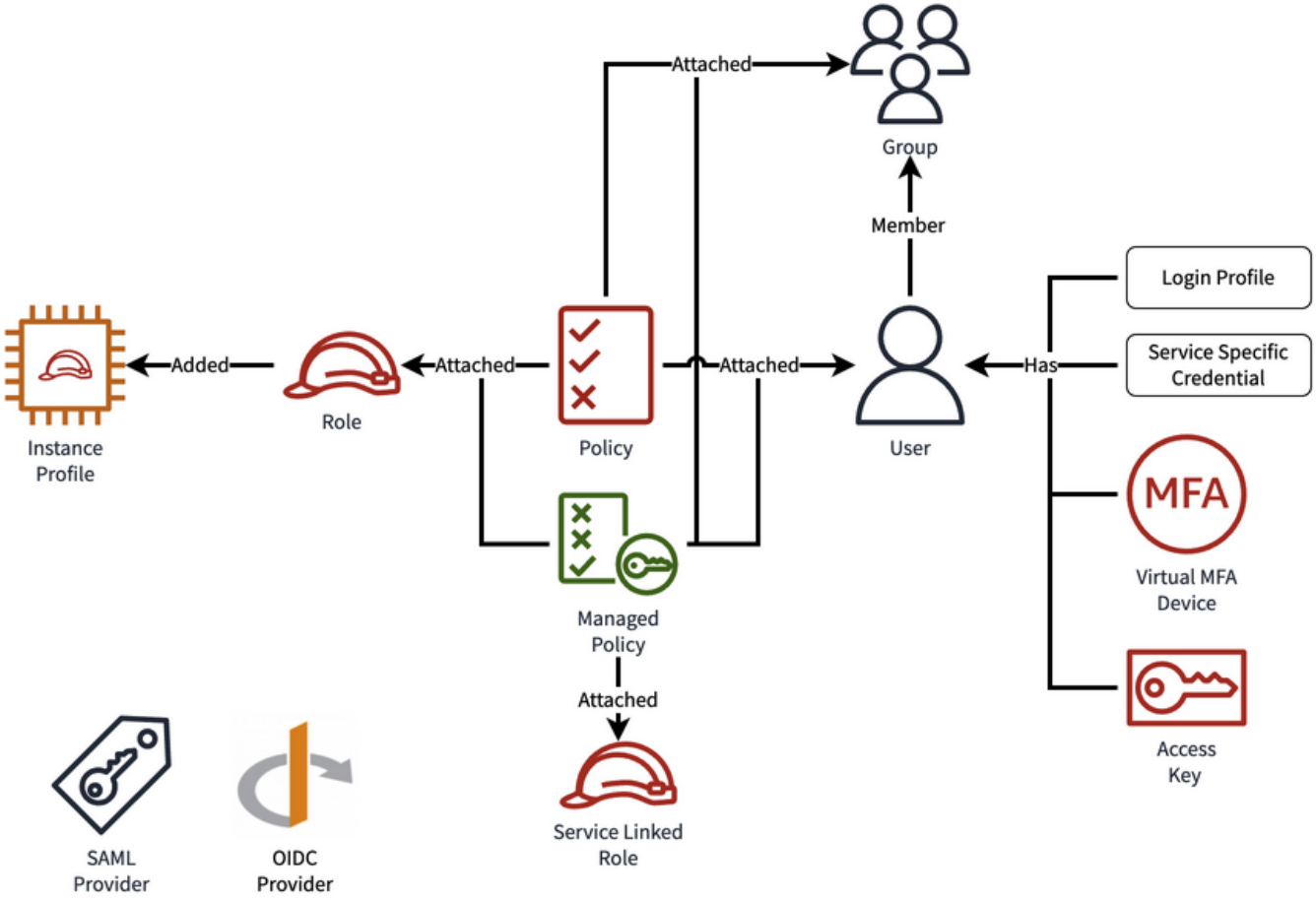
These sources of information are layered together so that IAM can decide to allow or deny

each request, as illustrated in the following diagram.

The Request Context

Resources

Managing IAM well starts with your IAM resources. By knowing at a high level which resources are and are not part of IAM, you'll be more easily able to manage your resources. Some of these resources are used only once or not at all in an AWS account. Resources you should know about have specific sections in the book.



IAM Resources

There are a few IAM resources that don't feature heavily or interact with other resources, so I have left them off this diagram. For example, account aliases are less relevant today because they don't take advantage of newer service features, and while you can manage server certificates in IAM you should use AWS Certificate Manager to manage certificates. A quick introduction to each of the IAM resources will help you understand what resources you can manage with IAM.

Access Keys

Long-term access keys are associated with a user. These keys are part of the request-signing process that allows them to make requests to AWS APIs. Though you can see their age, these credentials are *long-lived* because they don't expire after a set time. Because these keys never expire, you need to carefully monitor them to ensure they're not abused or misused.

Account Aliases

An account alias is a label that saves you from having to remember your 12-digit AWS account ID. Its only use is to create a human-friendly URL that you can use to sign in to a specific AWS account.

Aliases are a holdover from the early days of AWS when you signed in as a user to multiple accounts. These days you should avoid using alias URLs to sign in to your accounts directly, and instead use federation to centralize your authorization for ease of use and visibility.

Groups

Groups allow you to associate identity policies with multiple users. If you use users in your environment, then you should be using groups as much as possible. Groups will help you standardize and manage permissions across your users.

Instance Profiles

A profile allows you to assign a role to your Amazon EC2 instance.

Instance profiles exist so that you don't need to create users with access keys or use them on your long-lived servers, which would be a significant security risk. If you have long-term access keys in your EC2 environment, you should replace them with instance profiles as soon as possible.

The AWS CLI and SDKs can automatically query the EC2 metadata service via a reserved IP address and retrieve short-term access keys. Profiles can have one role associated with them at a time, and this limit cannot change.

If you update a role associated with a profile it will be eventually consistent. You can force the update by *disassociating* the profile and reassociating it, or by stopping and starting the instance.

Login Profiles

Login profiles are to allow users to access the AWS Management Console. They do this by assigning a password that the user can enter to gain access.

OpenID Connect Providers

This resource represents an OpenID Connect (OIDC) identity provider (IdP). Creating a provider configures a trust relationship between the account and the external IdP. Once created, you must create an IAM role that trusts the provider and allows it to assume the role.

Policies

Policies are JSON documents that grant permissions to identities. You will spend most of your time with IAM crafting policies, and much of this book is devoted to helping you get them just right. These policies are distinct from resource policies which are attached to resources and are defined and managed by the resources' service, not IAM. Policies can be AWS managed, standalone, or inline.

Policy Versions

Different versions of customer managed policies are stored by IAM so that you can change back to a previous version of your policy quickly and easily. Up to five versions of customer managed policies are automatically kept. Inline policies will not have any versions recorded.

Roles

One of the most common IAM resources, roles are a principal that other principals can *assume*. Roles can have identity policies associated with them for authorization. A role does not have its own credentials, so it cannot be used for authentication. Roles should be the primary way that you and your applications and you interact with IAM.

SAML Providers

The SAML provider resource represents a SAML 2.0-compliant IdP in your AWS environment. With this definition you can federate with the IdP, and the identities can assume roles in the environment.

Service-Linked Roles

These roles permit AWS services to access your environment. The roles represent your explicit approval for the service to operate in your environment. The service controls the policies attached to the role so that the service can perform the required functions, including configuring other AWS services. Only the service defined in the trust relationship can assume the role.

Service Specific Credentials

These credentials are created by IAM and associated with a specific user, just like access keys or a login profile. You can only use them to access AWS CodeCommit and Amazon Keyspaces.

Like you shouldn't be using users, you shouldn't use service-specific credentials directly either. This functionality exists because the applications these services work with (`git` and Cassandra, respectively) don't understand IAM natively. Use federation or an authentication plugin instead.

Users

This resource represents a principal that can access your AWS environment. Users can have login profiles, long-term credentials in the form of access keys, and service-specific credentials associated with them.

You shouldn't be using IAM users to access your environment! They are a holdover from *olden times* and should be used only for non-federated principals, such as machines and services that don't understand IAM natively.

Virtual MFA Device

These software-based multi-factor authentication (MFA) devices can be associated with a user, and used for authenticating them. An MFA device can be a dedicated device such as a hardware-based token generator, but more commonly they are virtual devices with an app to generate tokens. Though apps are not as secure as a dedicated piece of hardware, they are good enough to satisfy the "something you have" authentication method, and increase the security of logins.

MFA devices are different from the verification method used with when creating new AWS accounts, when AWS sends a verification code to your phone number. This validates that you control a specific phone number, and is not an authentication method.

Amazon Resource Names

Amazon Resource Names are a human and machine-friendly way to designate and identify resources in AWS. Though ARNs are not specific to IAM, they are used extensively in IAM resources. Policies rely heavily on ARNs, and multiple policy fields accept one or more ARNs. Much of your time will be spent getting the `Principal` and `Resource` element ARNs just right.

Internally to IAM, most ARNs get converted to unique IDs, but this only becomes visible in a specific scenarios. Reading, writing, and understanding ARNs are critical to using IAM effectively.

Segments

There are six segments in ARNs, each separated by a colon `:`.

```
arn:<PARTITION>:<SERVICE>:<REGION>:<ACCOUNT_ID>:<RESOURCE_ID>
```

An ARN can have more than 5 `:` in it, but any after the 5 shown above are considered part of the `RESOURCE_ID`. Some resources do not require values for all segments, but all ARNs must have the correct number of separators. Segments are also sometimes referred to as "components" of an ARN.

The first segment is the letters `arn`, so you know this is an ARN and not a similar-looking string of characters.

Partition is the AWS administrative domain of the resource. The valid values for this are `aws` for standard AWS Regions, `aws-cn` for China-based Regions, and `aws-us-gov` for US Gov Cloud-based regions. In the majority of cases, this will be `aws`. If you are operating in one of the other partitions, you might already know about the partition because it comes with a variety of limitations and quirks. If you're not using these other partitions, you can ignore them.

Service is the short name for the AWS service the resource belongs to, such as `iam` for IAM. This short name matches the prefix used for a service's IAM actions that you will specify in the `Action` element of your policies.

Region is the short name for the AWS Region in which the resource exists in (for example, `ap-southeast-2` for Sydney). This segment is blank for the resources of global services, such as IAM.

Account ID is the 12-digit AWS account number that owns the resource. The exception are resources such as AWS Managed Policies that have an account segment of `aws`, because they are present in your account but are not owned by your account.

Resource ID represents a specific resource ID and must be unique in an account. Usually, this will include the resource type followed by a unique identifier, such as a name for a user or role. Some IAM resources also support an optional path that is appending to the resource ID, separated by a forward slash ("/").

Wildcards

ARNs support two wildcard characters: * and ?. These are glob characters W. The star (*) wildcard matches any string of characters. The question mark (?) wildcard matches any

single character.

Wildcards are essential to writing policies, but you should use them with caution. Using wildcards means you don't need to hard-code specific references to actions, resources, and principals in your policies. However, if you use wildcards without thinking through their usage, they can have unintended consequences, the most common being overprovisioned access.

IAM Resource ARNs

Being familiar with the common IAM ARNs will make it easier for you to configure IAM itself, and the resources of other AWS services.

The various IAM resource ARNs look like the following.

```
arn:aws:iam::ACCOUNT_ID:root
arn:aws:iam::ACCOUNT_ID:user/USER_NAME_WITH_PATH
arn:aws:iam::ACCOUNT_ID:group/GROUP_NAME_WITH_PATH
arn:aws:iam::ACCOUNT_ID:role/ROLE_NAME_WITH_PATH
arn:aws:iam::ACCOUNT_ID:policy/POLICY_NAME_WITH_PATH
arn:aws:iam::ACCOUNT_ID:instance-profile/INSTANCE_PROFILE_NAME_WITH_PATH
arn:aws:sts::ACCOUNT_ID:federated-user/USER_NAME
arn:aws:sts::ACCOUNT_ID:assumed-role/ROLE_NAME/ROLE_SESSION_NAME
arn:aws:iam::ACCOUNT_ID:mfa/VIRTUAL_DEVICE_NAME_WITH_PATH
arn:aws:iam::ACCOUNT_ID:u2f/U2F_TOKEN_ID
arn:aws:iam::ACCOUNT_ID:server-certificate/CERTIFICATE_NAME_WITH_PATH
arn:aws:iam::ACCOUNT_ID:saml-provider/PROVIDER_NAME
arn:aws:iam::ACCOUNT_ID:oidc-provider/PROVIDER_NAME
```

The IAM Identifiers documentation has more examples.

IAM Paths

A common scenario in IAM is giving similar permissions to a group of entities. Paths represent a namespace you can provide to your entities to group them. Common groupings are the team or division entities correspond to, or a specific application stack. This information is included in the ARN after the resource specifier, but before the resource's unique ID.

Paths don't seem to be used or recommended much these days. This kind of functionality is probably better implemented using attribute-based access control, because it has wider support and better visibility than paths.

Paths give another dimension of control. For example, a resource policy might apply to all roles with a specified path. The following example could be part of the resource policy of a shared resource, and it would match any of the roles that have the path `ApplicationA` regardless of the role name used.

```
"Principal": "arn:aws:iam::111111111111:role/ApplicationA/*"
```

You can set and edit paths only via API calls and not in the web console. Paths are not visible as a separate field in the console, but you can see them as part of a resource's ARN.

The following entities support paths:

- Roles
- Policies
- Instance profiles
- MFA devices
- Server certificates
- Users

Unique IDs

Even though you see ARNs in the policies and API calls you use with IAM, the service uses a different unique ID for the resources internally. These unique IDs are not set by user input and cannot be reused. Generally, you don't need to worry about these IDs, but they can be exposed in certain scenarios.

The following is an example role ID.

```
AROA22MSEQDJU4ZEAXMUK
```

The most common ID prefixes you will see are `AROA` for roles and `AIDA` for users. A full list of unique ID prefixes for each resource is in the official documentation aws.

Unique IDs are most commonly seen when a principal listed in a policy is deleted. All references to the deleted principal show the underlying ID and not the ARN. This prevents an entity from either escalating or hiding their privileges by deleting and recreating a resource with the same name, such as a user or role. To fix the reference, you must edit the reference to use the new ARN, which reconfirms the trust relationship to the new principal.

Roles

Being able to switch your level of permissions is important feature of access management, so you can follow good security practices like the principle of least privilege. Roles are an IAM principal type that make it easy for you to define, group, and change the active permissions in your AWS environment. By defining roles you can craft a level of authorization that is appropriate to the requirements. Roles are the most common IAM principal you should be using to interact with IAM.

Because roles are an identity, you need to associate identity policies with them to give them permission to access resources. These policies can be either inline to the role definition, or managed policies (by the customer or AWS) that are attached to the role. Roles are commonly used by more than one entity.

Even though roles can be specified as the `Principal` in a policy, they can't make requests to AWS directly, and have no long-term credentials. Roles must be *assumed* by another principal, which results in an *assumed-role session* which provides short-term credentials.

Though you can only assume one role at a time, you can use a role to assume another role. This is known as *role chaining*. Even when chaining roles like this, only one role is active at a time, and the permissions granted are not cumulative.

Assuming a role in another account is called *cross-account role assumption*, and is how you access resources in other accounts that don't have their own resource policies. See Using Multiple AWS Accounts for more details.

Trust Policies

As a resource of the IAM service, roles have their own resource policies. These are referred to as trust policies, because they establish a *trust relationship* between the role and the entity that allowed to assume the role.

Some common entities that a role might trust are:

- An IAM user
- Another IAM role
- An AWS account
- An organization
- An identity provider (web-based or federated)

- An AWS service
- An Amazon EC2 instance

Unlike many resource policies for other services that are optional, you **must** define a role's trust policy.

Service Roles

A service role is a role that has a trust relationship with an AWS service. You must create and manage the role in your account, and specify it when you configure the service.

This is what a trust policy for an Amazon EC2 service role looks like.

```
{
  "Version": "2012-10-17",
  "Statement": {
    "Effect": "Allow",
    "Principal": {
      "Service": "ec2.amazonaws.com"
    },
    "Action": "sts:AssumeRole"
  }
}
```

When attached to a role, this allows the Amazon EC2 service principal to call `sts:AssumeRole` to assume the role, and use the permissions granted by the policies attached to the role. If the service role does not have all the required privileges, then the associated AWS service will probably not work as intended in your account. Services that require a service role will detail it in their documentation, and generally provide AWS Managed Policies that you can use to grant the correct permissions. Because you have created the role, you manage its lifecycle including updating and deleting the role.

Service-Linked Roles

Using normal service roles permit services to work in your accounts, but they require effort on your part to create and maintain them. Service-linked roles are a special service role that is defined and managed by the AWS service in your account for you.

You can delete service-linked roles only after you delete all the related resources. Without this restriction, it would be possible to break the functionality of the service in your AWS account.

In newer AWS accounts, service-linked roles are usually created for you automatically.

For existing AWS accounts, services will generally prompt you before creating the service-linked role as part of a one-off account configuration step. You cannot change or delete service-linked roles, and they cannot have permissions boundaries applied to them.

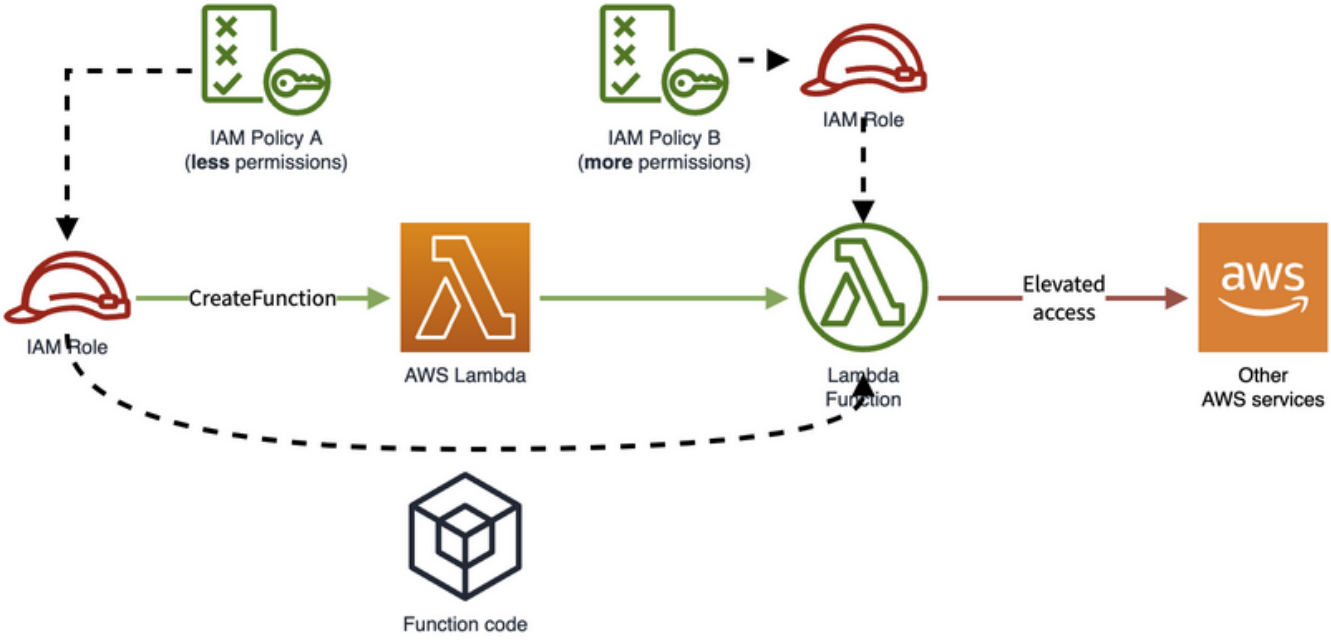
IAM PassRole

The action `iam:PassRole` can be a point of confusion when using roles in AWS. Some API calls to AWS can fail with the error message "... is not authorized to perform: iam:PassRole on resource ..." even though the caller has permission for the API method they are calling. This can even impact the AWS CloudFormation service, which can be a surprise. Though the error message does a good job of telling what the problem is, it doesn't attempt to tell you why it's a problem. The error is as much about the API call as it is the caller and their permissions, and is a protection put in place to protect against the *confused deputy* problem.

The confused deputy problem is where a legitimate service is used to do something that was not intended, and potentially avoiding security controls that are in place. This issue can happen when you are calling an API method that will assign a role to a resource on your behalf, such as creating a Lambda function, or provisioning an EC2 instance with an instance profile (which includes an associated role).

The risk is that this kind of call can be legitimate, but it can also be used to circumvent controls and perform an *escalation of privilege* attack on an AWS environment. As the name suggests, this kind of attack is performed by an entity that already has some level access, and are able to trick the confused deputy in to giving them more than intended.

Here's an example of the confused deputy problem in action with AWS Lambda when provisioning a function.



AWS Lambda and the confused deputy problem

In the diagram above, the principal has some permissions granted to them by policy A, and has legitimately asked AWS Lambda to create a function on their behalf. As part of this legitimate call the principal assigns a role with policy B to the newly created function, one which has **more** permissions than their own role and policy, for example administrator privileges. At the same time they are uploading function code that will execute in the context of the more privileged role, and be able to call other AWS services with the elevated privileges. The confused deputy problem is not unique to AWS or IAM.

The main point of confusion is that unlike most actions, there is no "PassRole" API method for IAM that you can call. The principal creating resources with roles will need this action, even though they will never make a pass role API call. The action `iam:PassRole` is a "permission only" action, as called-out on the SAR page for IAM. The action supports specifying specific role resources that the principal is allowed to pass to other resources, which should be the same or less than what they themselves are allowed to use.

Granting the `iam:PassRole` permission helps mitigate the confused deputy problem in the context of services in your account, except for cross-account and organization role assumption. For that, you will need to use `sts:ExternalId` condition.

Sessions

Assuming a role starts an assumed role *session*. Sessions are a resource of the AWS Security Token Service (STS), as you can see service segment of the ARN.

```
arn:aws:sts::ACCOUNT_ID:assumed-role/ROLE_NAME/ROLE_SESSION_NAME
```

When you make an assume role call you must supply the role ARN to assume, and well as a unique name to identify session. This makes it possible to differentiate different entities assuming the same role.

The response returned will contain a JSON payload that includes the information needed to use the assumed role session to make requests of AWS services. The interesting fields returned are `AccessKeyId` and `SecretAccessKey` which make up the session key pair, and a `SessionToken`. Though the access key ID and secret access key look like a normal user *key pair*, the difference is these will expire and that the session token must be sent with all requests, otherwise requests made with the key pair will be rejected. There are a few other fields included in the response payload that are interesting, but not required to use the session, such as ARN of the session, and when the session expires.

Since sessions only have short-term credentials they limit the blast radius of compromised credentials in your environment, because there is a limited window that they can be used for. Assumed role sessions last for 1 hour by default, but can be configured to last from as short as 15 minutes, and up to 12 hours. The range of times accepted varies depends on the type of the session, as you can see in the official documentation. Session duration represents the classic trade-off between security and convenience, and the session duration you should choose will depend upon the task at hand. A shorter session is more secure, but might cause interruption if your session expires while you are in the middle of a task. A longer session will be more convenient because of less interruptions, but gives an attacker a longer window to use credentials.

When assuming a role, the session can have extra policies applied to it which further reduce the permissions that the session has authorization to perform. See session polices for more information. This is another way that sessions reduce the blast radius, and are superior to users from a security perspective. Session policies cannot be used to grant additional access, only limit the permissions already authorized by associated identity policies. Sessions can have extra tags set on them as part of the assumption process. Like other AWS resource tags, these are key-value pair attributes that appear in the request context, and can be checked with the `aws:PrincipalTag` key in a `Condition` policy element.

Session tags can be optionally marked as *transitive*, which means that they will be passed on to future sessions when role-chaining is used. Only the tags set when assuming a role can be made transitive, and cannot be overridden by later sessions. Session tags are often used as part of an attribute-based access control (ABAC) approach to dynamically control what level of access an assumed role has to AWS resources. See the ABAC scenario for an example of session tags.

Users and Groups

The practical advice for IAM users and groups these days is simple: Don't. Users and groups are a effectively a **legacy** feature of IAM, and you should not use them unless you have a need that can't be met otherwise. The main reason why users should be strongly avoided is because they rely on long-term credentials. Though you can force users to use MFA devices, access keys and passwords that never expire are a security risk that should be avoided.

Users in IAM are both principals and identities. A user can be the `Principal` of a resource policy, and they can have identity policies associated to them, either defined inline, or attached to them. You should strongly prefer attached policies over inline policies.

Including identity policies to a user directly inline is an anti-pattern buried in an anti-pattern, and you should avoid it because it makes the policies harder to track and maintain. Always use groups if you use users, and attach policies to the groups.

While there might be a few legitimate use cases for users, they are generally a lot fewer than people first think when they start using AWS. The fact that many AWS tutorials on the internet start with "1. Create an IAM user with Administrator privileges" doesn't help the situation, and this includes an unfortunate number of tutorials in the official AWS documentation.

The best use case for IAM users today is not for human users, but **machine** users. For the right way to grant your human users access to your AWS environment, use federation.

Humans vs Machines

Machine users that interact with AWS generally require long-term credentials, such as username and password, or access key and secret access key. This is because it is complicated to use features such as federation and MFA with machine users, and doesn't greatly improve the security of a system by doing so. This is why using IAM users is appropriate for machine users.

Some examples of machine users that use long-term user credentials to interact with AWS are deployment pipelines that make automated changes, or older software that isn't aware of IAM roles. All of the humans in your AWS environment should be assuming roles, which means they can use short-term credentials via sessions.

If you're paying for a third-party solution that only supports user-based access to your AWS environment, you should ask when they're going to support role-based access with the `sts:ExternalId` condition

Machine users should also be performing consistent and known tasks in your environment, which means that you should be able to lock their permissions to what they require, and no more than that. An automated system that requires a lot of permissions (for example, administrator access) needs to be broken in to separate systems or tasks, so that access can be limited. See the least privilege scenario for crafting a policy without wildcards for your machine users, and reducing the blast radius of any compromised long-term credentials.

Groups

If you find yourself in a situation where you have to use users, then you should use IAM groups as well. Groups allow you to apply policies to a collection of users, which encourages you to think about the users' activities at a higher-level, and more requirements-focused way, which will result in better policies.

Though groups are an identity (because you can attach identity policies to them) they are not a principal, so cannot be directly specified as a resource policy `Principal`. Ideally you use groups to assign the bare minimum permissions required to your users, and then provide roles specific to the job function or task at hand for those users to assume. Making your users assume a specific role gives you additional levels of control and visibility in to your environment and usage, and gives you the ability to use conditions more effectively.

The Root User

When you create an AWS account manually, only the account's root user is available for use. This user is a special user that predates IAM and is not actually part of IAM. The root user is a special type of principal that can also be used to reference all the principals in an AWS account.

A root user is identified by the email address associated with the AWS account, and it has its own sign-in option in the AWS Management Console. The root user is the ultimate power in the context of an account: It can close the account itself! However, even the ability to close the account can be denied via service control policies.

Your AWS account and its root user are linked because this is how you accessed AWS before IAM was available to create users and roles. This is why the ARNs for your AWS account and the root user of the account are the same.

You should not be using the root account unless you are performing tasks that only the root user can perform. Always remember that the potential impact of changes made by the root account is everything in their AWS account, as they are not limited by SCPs, identity policies, or permissions boundaries. These tasks are important but infrequent actions, and they don't justify regular usage of the root account.

You should never, ever use the AWS root account to create long-term credentials.

Any access keys created won't expire, so these long-term credentials for your root user represent a security risk that isn't worth the convenience of programmatic access.

Quotas

All AWS services such as IAM have their own sets of quotas. Quotas used to be known as "limits" but were renamed with the introduction of Service Quotas to manage them. Before the introduction of Service Quotas, all quotas were tracked on a per-service basis and you had to log a Support request to have quotas increased.

Quotas can be either *hard* or *soft* limits on how you can use a service. You can request that AWS raise the soft limits in a given AWS account, but only to a maximum. Hard limits in your accounts cannot be raised no matter how nicely you ask.

Quotas are account based, which is another reason to follow a multi-account strategy. By using multiple AWS accounts your resources will spread more evenly through your environment, and reduce the likelihood you will be limited by quotas.

You can view a summary of your IAM quotas that have been applied and used quotas at the account level with the `GetAccountSummary` API call, which looks like the following in the AWS CLI.

```
aws iam get-account-summary --output table
-------------------------------------------------
|                GetAccountSummary               |
+------------------------------------------------+
||                  SummaryMap                   ||
|+-----------------------------------+---------+|
||  AccessKeysPerUserQuota           |  2      ||
||  AccountAccessKeysPresent         |  0      ||
||  AccountMFAEnabled                |  0      ||
||  AccountSigningCertificatesPresent|  0      ||
||  AssumeRolePolicySizeQuota        |  2048   ||
||  AttachedPoliciesPerGroupQuota    |  10     ||
||  AttachedPoliciesPerRoleQuota     |  10     ||
||  AttachedPoliciesPerUserQuota     |  10     ||
||  GlobalEndpointTokenVersion       |  1      ||
||  GroupPolicySizeQuota             |  5120   ||
...
```

Because quota values can change over time, the official IAM documentation is always your best source to learn about default and maximum quotas. You can view the hard and soft quotas for IAM in the IAM page in Service Quotas console, and this also is where you can request increases.

If you're unlucky enough to run into limits during a critical or time-sensitive event, you can upgrade the level of support on your AWS account to get faster turn around on your support requests, for example to developer or business level. You can drop the level of support back down after your limit request has been actioned to reduce your ongoing costs.

The most common hard quotas you might run in to are the managed policy size limit and the role name. Managed policies cannot exceed 6,144 characters, which includes all the associated JSON syntax but not white space characters. Role names cannot exceed 64 characters. These limits are probably in place to reduce the load on IAM. Even though these limits are frustrating to encounter, hitting these is often an indicator that you're trying to do too much in a single policy and you should consider splitting it into multiple policies.

[1] For an accessible introduction to request signing, this blog post by the unparalleled AWS Distinguished Engineer @colmmacc is a great read.

Policies

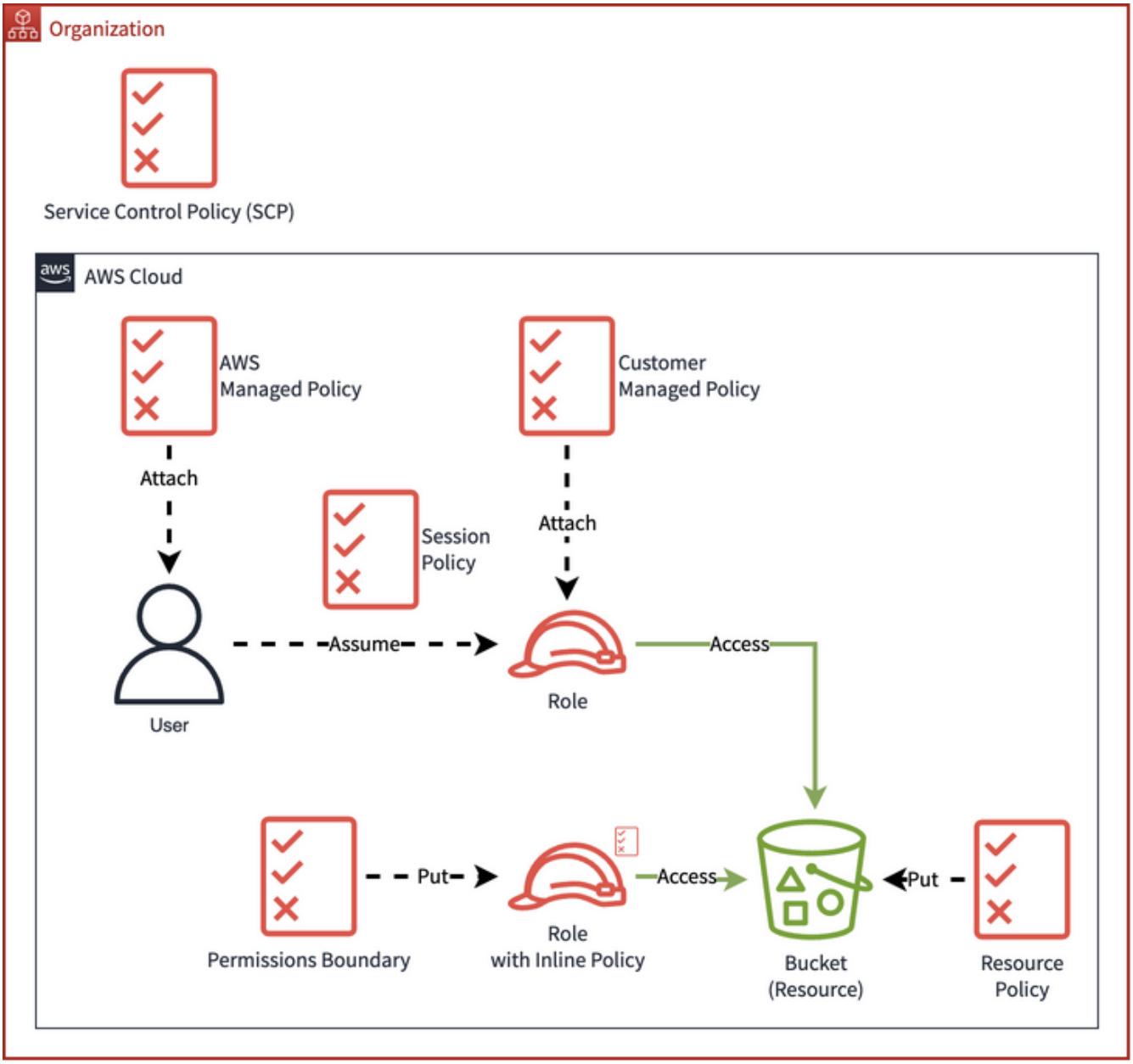
To get the most out of any AWS service, you need to know the resources you're working with. In IAM's case, the resource you will work most with is **policies**. Achieving your security goals in AWS means knowing how to write, use, and troubleshoot policies.

This section of the book covers all you need to know about policies: what kinds there are, how they fit together, and how you craft them.

Types of Policies

IAM policies come in various types, and knowing how they differ from each other is the difference between *using* IAM and *mastering* it. Knowing how to read policies, write them, and how they interact with each other will improve your daily confidence and speed as you work on AWS.

The following diagram shows the different IAM policy types and how they relate at a high level.



Different Types of IAM Policies

The policies applied to a request determine if the requested action is allowed or denied, so it's critical to get your policies right. The following sections describe each of the different types of policies in the order they are used when evaluating requests.

Service Control Policies

Service control policies (SCPs) are a policy that applies at the organization level of an AWS environment. Though they are technically a feature of the AWS Organizations service, they use the same policy syntax as IAM policies. Due to their role in policy evaluation and the impact they have on other policies you can consider them part of IAM, but remember that you will manage them separately.

SCPs are attached to organizational units (OUs) in your environment, and apply to the AWS accounts in the OU. By default, an organization has an SCP called `FullAWSAccess`

applied to the root OU of the organization that allows all actions on all resources.

Though you can allow or deny actions with SCPs they do not grant permissions by themselves, you still need an identity or resource policy to do that.

Effective actions with SCPs

If you want to allow most (but not all) actions, an SCP based on a list of denied actions is the most efficient way. If you want to deny most IAM actions, then you should use an SCP based on an list of allowed actions.

Some older AWS environments may not be using AWS Organizations, or not have the full feature set enabled. In these environments, SCPs do not exist, and will not impact requests. Given the extra security and functionality that being in an organization provides, I strongly recommend all AWS customers use AWS Organizations with all features enabled.

SCPs have the ability to restrict the root user of an AWS account. Other policy types that exist in account either don't apply to the root user, or cannot restrict them, such as identity policies and permissions boundaries. SCPs can be used to prevent the root account from doing things such as leaving the organization without approval from the organization itself. This is a good *mitigating control* for if you can't or don't want to set up MFA on all of your member accounts in an organization. It is still important to setup MFA on the root user of the organization management account, since SCPs do not apply to the management account!

Don't be confused by the error message you get when a SCP denies an action; It will say "due to an explicit deny in a service control policy" even when the denial is **implicit**.

You cannot see the SCPs that are applied to an account from within the account. If you try to use an action denied by SCPs, you will see a note mentioning it in the error message. There are some things that cannot be prevented by SCPs, which will need to be managed via identity policies.

Identity policies

Identity policies are the most common type of IAM policy, and they are what most people mean when they say "policy" in the context of AWS. These are the policies you will spend most of your time with, since they are required to grant authorization to actions on AWS resources. Sometimes these are referred to as "permissions policies" because they authorize principals to have permission to use certain IAM actions. Identity policies can apply to users, roles, and groups, even though groups are not an identity.

Don't forget that identity policies **cannot** be applied to the root user. Though not mentioned explicitly in the official documentation, this is probably because root users predate the IAM service.

Identity policies can be defined in two places: *inline policies* that are part of other resources, or as standalone *managed policies*. Managed policies come in two forms: *AWS managed* and *customer managed*. Regardless of where they are defined, all identity policies follow the same syntax.

Inline Policies

IAM policies that can be defined as part of other resources are known as inline policies. The resources that support these embedded policies are users, groups, and roles. Inline policies are helpful when you have a simple, one-off scenario and want to avoid additional complexity such as deploying multiple resources.

You should prefer managed policies in general, because of the improved control, and visibility they have over inline policies.

Because they are defined in the context of an identity, you don't need to specify a `Principal` element for these identity policies. Versions of inline policies are not saved automatically, like they are for managed policies. Since groups are not an IAM principal, the effective principal of a group policy is a user in the group.

AWS Managed Policies

When getting started with AWS or a new AWS service there can be a lot to learn. From an IAM policy point of view, you need to know what service actions you must allow to perform common tasks with service. This can be hard when you're starting with a new service and resources. To reduce this barrier of entry, and avoid customers using administrator access all the time, AWS provides AWS managed policies as a focused, but still flexible, starting point to use a service.

To see a sorted list of AWS managed policies in your account, you can scope and query a call to the `ListPolicies` API of IAM. Using the AWS CLI, the abbreviated list looks like the following.

```
aws iam list-policies --scope AWS --query 'Policies[].PolicyName | sort(@)'
```

```
[
    "APIGatewayServiceRolePolicy",
    "AWSAccountActivityAccess",
    "AWSAccountManagementFullAccess",
    "AWSAccountManagementReadOnlyAccess",
    "AWSAccountUsageReportAccess",
    "AWSAgentlessDiscoveryService",
    "AWSAppMeshEnvoyAccess",
    "AWSAppMeshFullAccess",
    "AWSAppMeshPreviewEnvoyAccess",
    ...
```

As shown by these examples, AWS managed policies generally start with "AWS" or "Amazon", then the name of service they apply to, and then the task or level of access they provide. These documents are managed by AWS, so even though they can change over time you cannot customize them or change their default version. The exact policy names and contents of AWS managed policies are continually changing, and at the time of writing this chapter there are 1044 AWS managed policies in my account.

Because they exist in your account already, and need to work with resources that haven't been created yet, AWS managed polices are generic. To work as intended, they will have a `Resource` element defined that covers all the resources in the AWS account (usually `*`).

Customer Managed Policies

AWS managed policies are a good starting point, but to make them specific to your applications and needs you will need to create your own policies. To do this you create customer managed policies in your account. By crafting your own policies you can grant just enough, but not too much, access to AWS services and resources in your environment. Customer managed policies are essential for practicing least privilege, because unlike AWS managed policies they can be scoped to specific requirements and resources. Creating an managing these policies is where you will spend most of your time administering IAM, and is also the most important part of IAM to get right.

These standalone policy documents exist as separate resources to the identities, groups, and roles that they are eventually associated with. As a result, these policies can used for multiple identities and reduce duplication. These policies can be managed with their own lifecycle of review and change, which gives you better visibility than AWS managed

policies.

Changes to managed policies are versioned automatically by IAM, and up to five different versions of a single customer managed policy are kept automatically. This version is not the same as the `Version` element in the policy itself, which refers to the policy language syntax version. The *default* version is the active version of the policy used by all the identities the policy is attached to. By changing the default version of a policy, you can quickly roll-back to a previous version of policy without having to deploy resources, or update any of the associated resources. Versions are automatically named for you, for example `v1`, `v2`, `v3`, and so on. You can delete old policy versions, but only if the version is not the current default. Version names do not change once they have been created, so if you have deleted older versions to make space for new versions then names may not be sequential.

Resource Policies

Resource policies control access to resources of many services in AWS. Not all resources support resource policies, only those that work with IAM. The official IAM documentation has the complete list of resources that support resource policies. You define resource policies on the resource to which they apply, so they do not exist as separate documents like identity policies can.

Resource policies differ from other policy types in that they define the principal that can access them. This means they can specify principals that do not exist in the same account, thus enabling cross-account access. This is because an AWS account operates as a *zone of trust*, and resources in that zone implicitly trust any principal in the same zone. Outside that zone, permissions must be explicitly granted by a resource policy. Within the same zone of trust resource policies are not required, only identity policies.

Resource policies can short-circuit the policy evaluation logic that most people find surprising. If a resource policy allows a principal an action, and the principal's identity policy allow that action, then permissions boundaries and session policies with an **implicit** deny do not effect the request.

To directly access an AWS resource from outside its account, the target resource must support resource policies. If the target resource does not support resource policies you must instead use cross-account role assumption to access it. This means you assume a role in the same account as the resource, and then make the request from within the resource's zones of trust. This is possible because IAM roles are a resource that support resource policies. See the Using Multiple AWS Accounts scenario for examples.

Permissions Boundaries

Permissions boundaries are an advanced feature of IAM that let you put limits on how actions can be used by a principal. This functionality is most useful when you are delegating access to the IAM service itself to principals in your environment. Without any kind of limitations, a principal with access to create and update IAM resources can escalate their own privileges. For example, an entity with the ability to create IAM resources can create a new role that has more permissions than their current principal, and then assume that role.

As the name suggests, permissions boundaries do not grant any permissions on their own. Instead they specify the maximum permissions that can be allowed to the entity. This means that the effective permissions for a principal with a permissions boundary are the intersection of actions and resources between the permissions boundary and the identity policies applied.

Effective actions with permissions boundaries

To use permissions boundaries, you create a policy that represents the allowed actions.

PermissionsBoundary.json

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "RunInstancesInSydney",
      "Effect": "Allow",
      "Action": "ec2:RunInstances",
      "Resource": "*",
      "Condition": {
        "StringEquals": {
          "aws:RequestedRegion": [
            "ap-southeast-2"
          ]
        }
      }
    }
  ]
}
```

This policy restricts the principal so that they can launch EC2 instances in the Sydney (`ap-southeast-2`) region, but not other regions.

Permissions only boundaries apply to IAM users and roles, which means you cannot apply permissions boundaries to the root user account. For permissions boundaries to be effective, you must also ensure that the boundaries apply to any resource that the principal creates. The `iam:PermissionsBoundary` condition key supports string operators, and checks that the given policy ARN is attached to the principal.

I prefer to rely on the AWS account boundary to limit the actions of principals, rather than permissions boundaries. Managing the account boundary is usually simpler, and is something you need to do regardless of permissions boundaries. If you have a solid multi-account strategy, and don't need to delegate IAM access in your environment, then you will probably not need to use permissions boundaries.

Session Policies

Session policies apply extra controls to an assumed role sessions. They are an optional and advanced feature of IAM that let you create dynamic and fine-grained policies for your session principals, and can be useful when implementing least privilege. These policies are most useful when a session is started on behalf of an entity, such as when you use a federated identity provider to manage your identities.

Session policies can apply restrictions to an assumed role, but not to grant new permissions. This means actions allowed by a session policy must also be allowed by the role's associated identity policy for them to usable. As with other policies, an explicit `Deny` effect for an action in any policy involved will deny the action.

When you start a session with one of the STS API calls to assume a role, you can optionally provide a policy document to further refine the permissions that the session can use.

```
aws sts assume-role \
  --role-arn arn:aws:iam::ACCOUNT_ID:role/ROLE_NAME \
  --role-session-name SESSION_NAME \
  --policy file://SESSION_POLICY_FILENAME.json \
  --policy-arns arn=MANAGED_POLICY_ARN
```

As shown in this `assume-role` command, session policies can be either a JSON document, or you can use the ARN of a managed policy. The session policies can be used for the STS API methods `AssumeRole`, `AssumeRoleWithSAML`, `AssumeRoleWithWebIdentity`, and `GetFederationToken`.

The most common use-case for session policies is applying them to sessions started via an federated identity provider. Though the roles might all share the same IAM role, you might not want them to have exactly the same level of access to resources. Managing an individual role for every entity that might access your AWS environment would be a lot of effort, and might be impossible at scale due to service quotas. You can use a session policy to include resource-level restrictions to a shared role that all identities assume. This means you can manage different levels of similar, but different, access with a single role, reducing the administrative overhead. A practical example would be giving employees access to objects in an S3 bucket, but only allowing them to access and write to their own objects, as determined by information in their session.

Policy Evaluation

When IAM receives a request, it *evaluates* it to decide if it should be allowed or denied. The exact process used depends on the environment and the principal making the request, which is represented in the request context. Even though the process can seem complicated, each step is simple, and this section will break down the policy evaluation flow.

Explicit vs Implicit

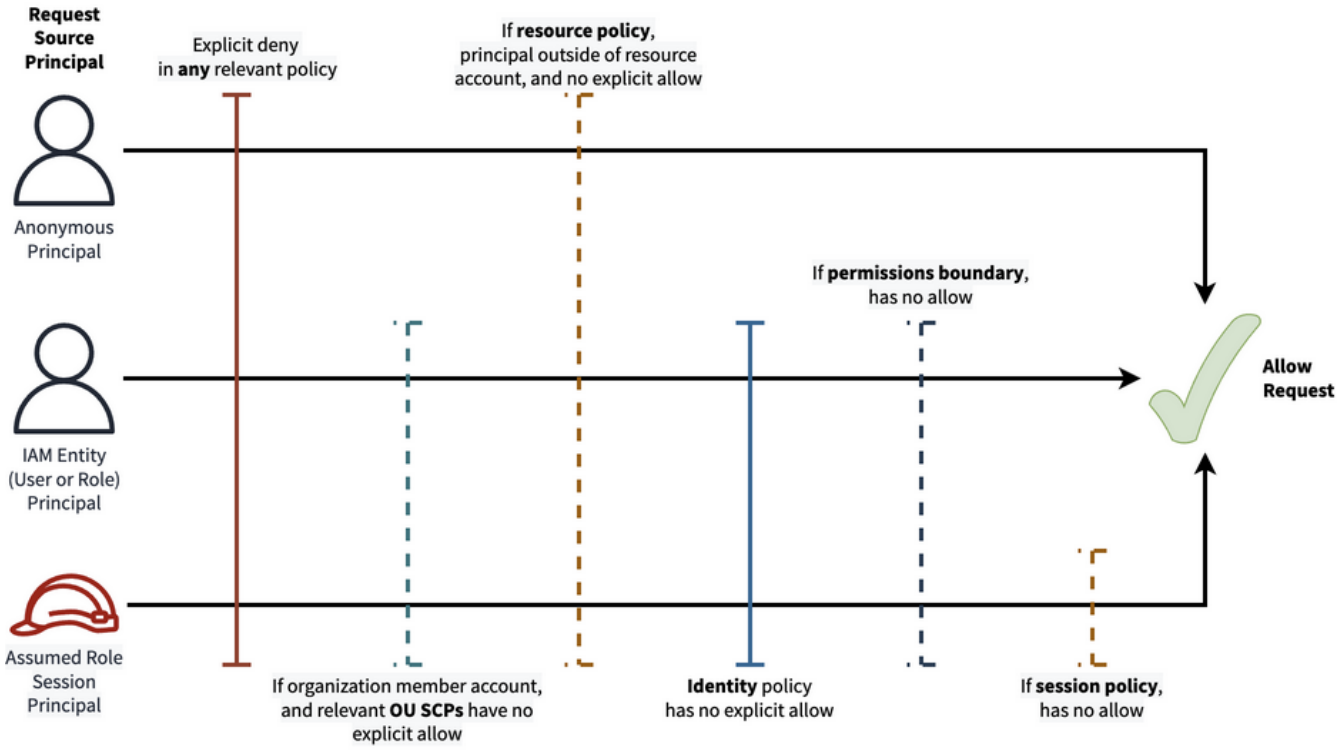
One key point to be clear about is the difference between *implicit* and *explicit* specification of actions. An **explicit** specification is something that is clearly stated. Something which is **implicit** is suggested, but not directly specified.

This can be complicated in the context of IAM policies, because something can be explicitly *specified* without being explicitly *visible*. For example, using the `*` wildcard for the `Action` element explicitly specifies all IAM actions, without writing them all in the policy.

To evaluate a request, IAM starts with an *implicit deny* decision; no actions are allowed on any resources. Based on the action and resources in the request, that must be changed to an *explicit allow* in the evaluation flow for the request to succeed. An *explicit deny* anywhere in the flow will override an explicit allow anywhere else in the evaluation.

Evaluation Flow

It is important to understand the evaluation flow to troubleshoot any issues with your policies; If you don't understand what's going on, it's hard to know what's going wrong! This simplified version of the impressive/intimidating diagram in the official IAM documentation shows the flow for each scenario.



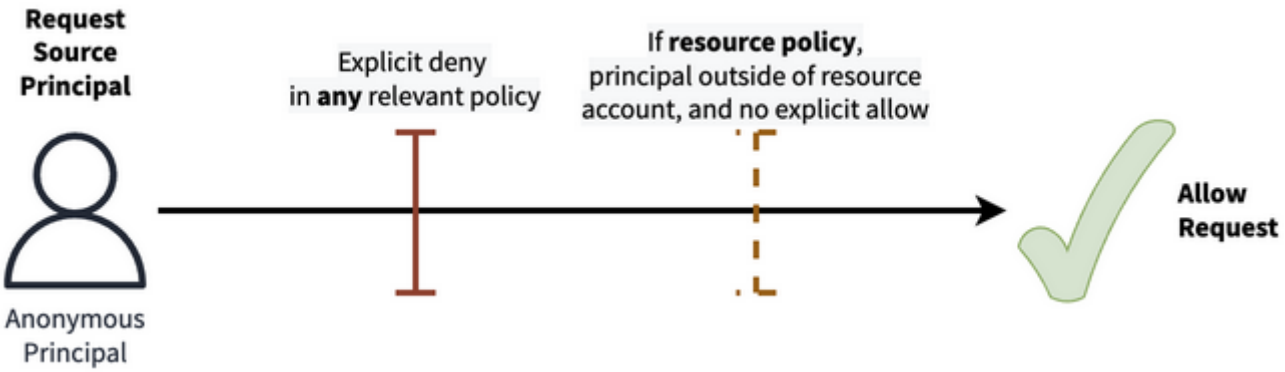
Policy evaluation flow

If **any** of the evaluations represented by vertical lines are true, then the request will be denied. Unless specifically mentioned, an allow or deny can be explicit or implicit. Dotted lines represent evaluations that only happen if the related policy type exists for the identity or resource involved in the request, for example SCPs only apply to requests in accounts that are part of an organization. If the request does not match the evaluation it is not denied, and the request continues to the next evaluation until it is denied or finally allowed.

Anonymous Principal

The anonymous principal includes unauthenticated users on the internet, so you should take extra care when provisioning access for it. Though this is a simple scenario which isn't supported by all services or resources, it is an important one to get right.

The anonymous principal is not authenticated with AWS, so it does not have an identity to attach policies to. This means that only resource policies apply to it.

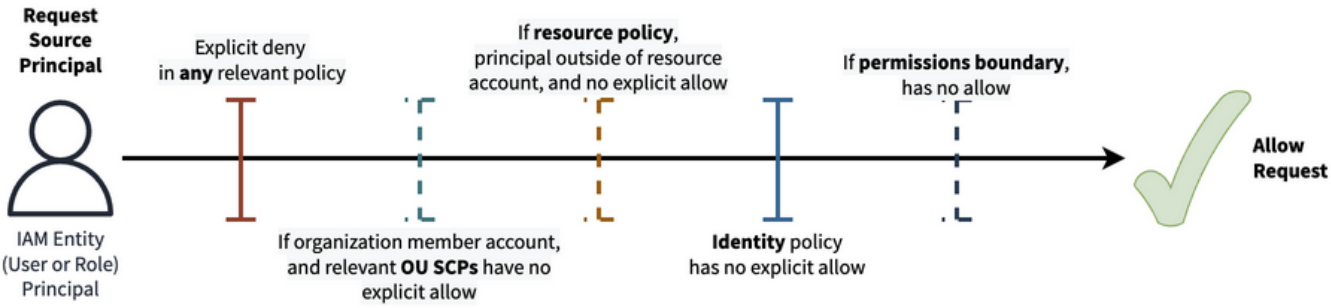


Anonymous principal request evaluation flow

1. If there is an explicit deny in any relevant policy, deny the request.
2. The relevant resources must support resource policies, and there must be a resource policy with an explicit allow for the action.
3. The request is allowed.

User or Role Principal

An IAM entity can be part of an organization, and can have identity policies and permissions boundaries associated with it. This makes the request evaluation process more involved than those made by an anonymous principal.



Entity principal request evaluation flow

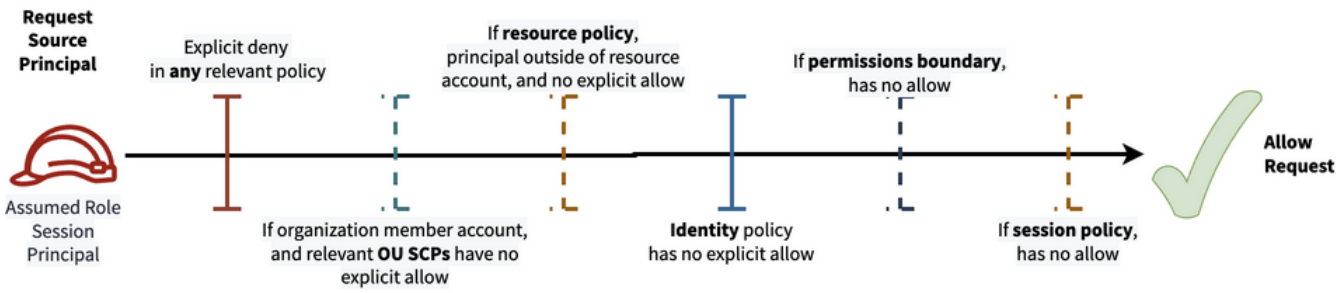
1. If there is an explicit deny in any relevant policy, deny the request.
2. If the principal is part of an organization with full features, there must be an explicit allow for action in an SCP associated with an OU the account is a member of.
3. If the principal is in a different AWS account to the relevant resources, and the resource supports resource policies, there must be a resource policy with an explicit allow for the relevant action.
4. There must be an identity policy with an explicit allow for the action.
5. If there is a permissions boundary associated with the entity, there must be an allow for the action.

6. The request is allowed.

For SCPs and OUs, remember that OUs can be nested in other OUs. This means that the required allow can be in any associated OU, not just the parent OU that contains the principal's AWS account.

Assumed Role Session Principal

Assumed roles sessions follow a similar process to IAM entities, with additional steps related to the presence of session policies. This flow applies to the common scenario of humans accessing AWS via a federated identity provider.



Session principal request evaluation flow

1. If there is an explicit deny in any relevant policy, deny the request.
2. If the principal is part of an organization with full features, there must be an explicit allow for action in an SCP associated with an OU the account is a member of.
3. If the principal is in a different AWS account to the relevant resources, and the resource supports resource policies, there must be a resource policy with an explicit allow for the relevant action.
4. There must be an identity policy with an explicit allow for the action.
5. If there is a permissions boundary associated with the entity, there must be an allow for the action.
6. If there is a session policy associated with the session, there must be an allow for the action.
7. The request is allowed.

This is the flow used for federated user principals, but with the added requirement that the principal **must** have an associated session policy with an explicit allow for the action an resource.

Policy Syntax

Though the IAM policy syntax is simple compared to spoken or programming languages, it can still result in complex policies. You write IAM policies in JavaScript Object Notation (JSON), which is a lightweight data-interchange format. JSON is both human and machine readable, but calling it "human friendly" might be a stretch.

Because of this, you should be familiar with JSON syntax before authoring policies. Fortunately, the JSON specification is not extensive, and you've probably seen and used it elsewhere already.

This example policy is about as simple as you can get and still be a valid policy.

AdministratorPolicy.json

```
{
  "Id": "b2600c49-c6cc-5003-b4bb-4de77abbb64c",
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "*",
      "Resource": "*"
    }
  ]
}
```

The following elements below are required for a valid policy, unless mentioned otherwise. Don't worry too much, as IAM will tell you quickly if you've left a required element out of a policy.

Id

The Id element is an optional identifier for the policy. Most services do not require an Id, but some older services such as Amazon SNS and Amazon SQS might give you an error if it is not specified. If a service requires an Id, you will get an specific error message telling you to include it.

The following is an example `Id` element.

```
"Id": "b2600c49-c6cc-5003-b4bb-4de77abbb64c"
```

If you include an Id element it should be unique. AWS recommends that you use a universally unique identifier (UUID) as the Id to ensure uniqueness. In practice, you can

leave the Id element out of your policy unless you have a need for it, because there's no way to use it to identify a policy via the API. For example, you might use it to store a name or label for your own use and reference. The API uses the policy ARN to identify it, and the Id is not part of the ARN.

Statement

The Statement element is the main element of a policy, and it deserves its own section.

Syntactically, the Statement is an array of JSON objects, and these are what IAM uses to evaluate requests. Any, all, or none of a policy's statements might match a request.

The following is an example Statement element.

```
"Statement": [
  {
    "Effect": "Allow",
    "Action": "*",
    "Resource": "*"
  }
]
```

This policy statement allows all actions on all resources, and is how administrator access is granted. It's also the simplest and smallest policy example there is.

Version

The Version element is optional in IAM policies. A policy's version determines which IAM policy language specification is used.

The following is an example Version element.

```
"Version": "2012-10-17"
```

Note that this element is different from the versions of AWS Managed Policies and Customer Managed Policies, which are the historical versions of a policy document and not the syntax of the policy itself.

You should always include a Version in your policies and it should always be "2012-10-17". This should be specified before the Statement element of the policy. Even though JSON object elements are not usually ordered, ordering them ensures the policy statements are evaluated with the correct syntax version.

The only other valid value for this element is "2008-10-17", which is the original policy syntax and does not support functionality such as Policy Variables. Unfortunately this old version is also the default value, which is why you should always specify then newer version in your policies.

Policy Statements

The Statement element of a policy is what makes each policy different and unique, and what makes them interesting. All but the simplest policies will have multiple statements in them. Thought the syntax of each statement is simple and clearly defined, they can become complicated. This means that it's often useful to think of each statement on its own, since all of them will apply to the principal or resource the policy is associated with.

To do this you must think clearly about *who* the policy applies to, *what* it allows or denies, *when* it applies, *why* you are writing it, and *where* it applies.

The *Who* (Principal)

The Principal element of your policy is the *who* of your policies. It is the person or service that has initiated the request to do something in AWS. The most common principal you should see in your environment is a role, as shown in the following example.

You should always aim to understand which principal is making a request. Being clear about who you're writing your policies for is crucial to write them well. Knowing your principal is critical when troubleshooting IAM policies. If you cannot articulate the principal involved, you might not be able to fix any issues related to it.

The following is an example `Principal` element.

```
"Principal": {
  "AWS": [
    "arn:aws:iam::AWS_ACCOUNT_ID:role/NAME"
  ]
}
```

This defines a single role principal for the policy statement. The official IAM

documentation defines an IAM principal as:

> A principal is a person or application that can make a request for an action or operation on an AWS resource.

Though this definition is correct, it does lack some nuance. You can define any of the following as principals in your policies:

- An AWS account or the account's root user
- IAM users
- IAM roles
- Assumed-role sessions
- Federated users (using web identity or SAML federation)
- AWS services, such as AWS Auto Scaling and AWS Lambda
- Canonical users
- Anonymous users

Though a principal is always required to evaluate a request, that doesn't mean it's always specified in a policy. Identity policies are attached to users, groups, and roles, and they do not need a Principal element. The principal for these policies is the resource they are attached to. Though you can attach identity policies to a group, a group is not a principal. For groups, the principal is the user who makes the request.

In resource policies, the Principal element is always specified. This is what makes cross-account access of resources possible because you must explicitly specify a Principal element that is outside the resource's zone of trust.

Changing your active principal is a normal activity when using AWS, and you should be doing it regularly. For example, you might start as an anonymous user, sign in to become a user, and assume a role to carry out a task. Changing principals is a good thing because it enables you to use the right level of access at the right time, but you can only ever be a single principal at a time. Think of principals like hats: You can have many hats, but you can only wear one at a time.[2] This means that even if you have access to different principals, you cannot combine their different levels of authorization simultaneously.

If you specify an AWS account principal, all of the principals inside it match the element, and they inherit the authoization. You cannot use partial wildcards such as `*` and `?` in the principal field. This is because IAM translates the ARN into an unique ID to link the principal and the policy, and these wildcards would make it unclear. The only time you

can use the `*` wildcard character is on its own to specify an anonymous principal.

Anonymous user principals are dangerous but sometimes necessary. They grant permission to unauthenticated entities. The most common scenario is when you want to use Amazon S3 to serve static web assets to the internet. Be very careful and deliberate when using it.

Types of Principals

The valid types of principals that you can use in policies are: `AWS`, `Service`, `Federated`, and `CanonicalUser`.

The `AWS` principal type is the most common because it covers AWS accounts, users, and roles. The `Service` type refers to AWS services, such as AWS Lambda, Amazon EC2, and Amazon S3. Inside the `Federated` principal property, you specify external identity providers, which are either web identity providers or SAML providers. The `CanonicalUser` principal establishes permissions between Amazon S3 and Amazon CloudFront to secure content behind a content delivery network (CDN).

You can specify a single principal per type as a string or multiple principals of the same type as an array of strings. Having more than one principal in a policy means that the policy applies to any and all principals specified.

AWS Accounts

Policies can specify AWS accounts and root users with their 12-digit account number or their full ARN. As such, the following two principals are the same.

```
"Principal": {
  "AWS": [
    "112233445566",
    "arn:aws:iam::112233445566:root"
  ]
}
```

You specify the AWS account and the root user the same way, because the root user did everything before the IAM service existed. Though you need to specify an account principal for cross-account access trust, you should never specify the root user account because you should not be using root account for day-to-day activities.

IAM User Principals

User principals correspond to authenticated IAM user resources, and are specified by their ARN as shown in the following example.

```
"Principal": {
  "AWS": [
    "arn:aws:iam::AWS_ACCOUNT_ID:user/NAME"
  ]
}
```

When specifying IAM users' ARNs in a `Principal` element, the `NAME` string value is case sensitive.

IAM Role Principals

IAM role principals are specified by ARN, as shown in the following example.

```
"Principal": {
  "AWS": [
    "arn:aws:iam::AWS_ACCOUNT_ID:role/NAME"
  ]
}
```

Unlike with IAM users, the `NAME` part of a role ARN is case insensitive.

Assumed Role Sessions

Assumed role sessions are a way to differentiate between different entities that are assuming a shared role. You start an assumed role session by supplying a session name when assuming a role, as showed in the following example.

```
assume-role --role-arn ROLE_ARN --role-session-name SESSION_NAME
```

Though it's similar to the role Principal element's ARN, an assumed-role session includes the `SESSION_NAME`, which is specified as a suffix and has its own prefix.

```
"Principal": {
  "AWS": [
    "arn:aws:sts::AWS_ACCOUNT_ID:assumed-role/NAME/SESSION_NAME"
  ]
}
```

An assumed-role session's name must be unique, and it is visible to the roles that are being assumed in other accounts. This means that the `SESSION_NAME` shows up in CloudTrail audit logs and can be used in conditions.

Web Identity

The following are the possible providers you can specify for web identities.

```
"Principal": {
  "Federated": [
    "www.amazon.com",
    "cognito-identity.amazonaws.com",
    "graph.facebook.com",
    "accounts.google.com",
  ]
}
```

Though the example above is a valid Principal element specification, in practice you should define them separately so that you can put conditions on them to limit access by provider.

SAML Provider

A SAML federation principal takes the ARN of the provider resource, as shown in the following example.

```
"Principal": {
  "Federated": "arn:aws:iam::AWS_ACCOUNT_ID:saml-provider/PROVIDER_NAME"
}
```

SAML Providers are an IAM resource that represent an external provider consistently in your environment.

AWS Services

AWS services have a long service name based on their `SERVICE_NAME.amazonaws.com`, and this is the name that represents the service principal. Slightly confusingly, AWS service principals don't use the `AWS` property, but they do go inside a `Service` property.

```
"Principal": {
  "Service": [
    "SERVICE_NAME.amazonaws.com",
    "SERVICE_NAME.ap-southeast-2.amazonaws.com"
  ]
}
```

The `SERVICE_NAME` is case sensitive. You can also specify a region-based service endpoint to limit the role to a specific region. By specifying a role that trusts a service principal, you create a service-linked role. To work out what services can be linked, look for a green check mark in the "Service-Linked Role Permissions" column of the official IAM documentation.

There doesn't seem to be an official and complete list of the `SERVICE_NAME` of every service principal, which are case sensitive. The closest I have found is this community-maintained list by the assiduous @ShortJared.

Canonical IDs

Canonical IDs are obfuscated AWS account IDs, and just like an AWS account they can be used as a principal.

```
"Principal": {
  "CanonicalUser":
"883c716af28a0e0b0dc3d6f2de8cea7177aa63e5a56d97cc0fc7e0cea3710083"
}
```

The official AWS documentation mentions that you can substitute it for an account ID under some circumstances, but this is not common.

Anonymous

The anonymous principal represents any entity, authenticated or not.

Be careful and deliberate when using the star in your Principal element. Granting unauthenticated, anonymous access to your resources is a security risk.

The anonymous principal is the only time you can use a wildcard in a principal property.

```
"Principal": "*"
```

You also can use the AWS property, as shown in the following example.

```
"Principal": {
  "AWS":"*"
}
```

Though this example principal above theoretically includes all AWS accounts, roles, and authenticated users, it would not include the AWS service principals. In practice, very few AWS services support the anonymous principal.

The main reason you would use this principal is when serving HTTP content out of an Amazon S3 bucket. Some other advanced scenarios involving AWS STS and Amazon SNS can use the anonymous principal, but they aren't common.

The *What* (Action)

The `Action` element of a policy statement details the API permissions that are allowed or denied to the principals, depending on the `Effect` in the statement. This element is where you will spend most of your time crafting and updating policies. Getting the list of actions right is key to ensuring you're following the best practice of granting *least privilege*, where only the minimum required actions are given to a principal to perform their current task. For more details, see the Granting Least Privilege Access scenario. The element name doesn't change if you specify one or many actions; it is always the singular "Action".

The following is an example `Action` element.

```
"Action": [
  "s3:GetObject",
  "s3:GetObjectAcl"
]
```

The Action element also is part of the policy where you need to have the most understanding of the service you're targeting, as each service defines its own actions. IAM checks that the principal making a request has the action allowed before the service will complete the request. Generally actions match the API actions of the service, but this is not always the case. For example, the `iam:PassRole` action is a commonly required permission, but IAM has no PassRole API method that you can call.

The authoritative reference for actions is Service Authorization Reference (SAR) page in the official AWS documentation. That page can be a little clunky to navigate, so a great community-maintained reference that collates multiple data sources is permissions.cloud by the excellent @iann0036.

The format of actions is the service's short name as a prefix, a colon (":") as a separator, and then the action. Most of the actions are in the format `VerbResource` (for example, `lambda:InvokeFunction` to invoke Lambda functions, and `iam:CreateRole` to create IAM roles). The most common verbs used are: `Create`, `Get`, `Describe`, `List`, `Update`, and `Delete`. The resource part of the action is specific to the service, such as `Instances` for EC2, `Functions` for Lambda.

Actions and their prefixes are case insensitive. Popular convention used in documentation and examples defines the service prefix in lowercase, and the action in CamelCase for readability.

Actions can be specified as either a single string or an array of strings. Very few *useful* policies will have only a single action, so the array format makes it easier to maintain and change the actions in your policies.

Wildcards

In some scenarios, having to specify many actions explicitly in a single policy becomes tedious and impractical. Due to this, the `Action` element supports two wildcard characters: the star/asterisk `*` and the question mark `?`. Both wildcards can be used after the service prefix to match multiple actions. The star matches any characters, and the question mark matches a single character. You cannot use the wildcards in the service name prefix, only for the action, after the service name separator `:`.

This matching makes it possible to group actions quickly to achieve common authorization scenarios. Using the wildcard with the service prefix lets you assign all actions in a service, effectively granting administration of that services and its resources (for example,`s3:*` to administer Amazon S3, and `ec2:*` for Amazon EC2). You also use the wildcard with a verb prefix to assign broad permissions that roughly correspond to create, read, update, and delete levels (CRUD) of access to a specific service e.g. `s3:Get*` for read access to S3, and `ec2:Create*` to create EC2 resources. Some service actions also specify the resource in their action names, so you can allow access to specific resources by combining wildcards e.g. `ec2:*Instance*` to specifically allow the administration of instances.

Be aware that when using the wildcards like this, you may end up assigning permissions unintentionally. AWS sometimes releases new actions or resources for services that might match an existing wildcard combination, so this means giving `…:Create*`

authorization will grant the ability to use all current and future actions that start with "Create" for that service.

The list of available actions is continually increasing. Occasionally, unused actions are removed, but removals these are outweighed by the addition of new actions as shown in this visualization from permissions.cloud 🔗.

https://permissions.cloud/: IAM Action summary for AWS

As an example of how wildcards can be used in practice, here are the number of actions matched by using wildcards in policies at the time of writing:

- Give administrator access to all services `"*"`: 10,100 actions matched
- Give administrator access to the S3 service `"s3:*"`: 117 actions matched
- Give read access to known S3 resources `"s3:Get*"`: 45 actions matched
- Give administrator access to S3 object resources `"s3:*Object*"`: 37 actions matched
- Give read access to specific S3 object resources `"s3:GetObject"`: 1 action matched

Since writing this paragraph, it's also likely the number of actions across all AWS services and for S3 has increased, which means that the numbers here are indicative but also out of date by the time you read this.

These action numbers were generated by using the BigOrange.Cloud Actions, which uses the IAM Policy Simulator as its list of actions. Unfortunately, though this is an official AWS resource it doesn't always match the SAR.

The AWS Managed Policies in your AWS accounts provide a useful source of example policies that you might want to customize for your use. You can also browse them as JSON files via tools such as IAM Tracker repo by the inimitable Aidan Steele.

The *When* (Condition)

The Condition policy element is an optional way to control *when* the policy will apply. Though this is a advanced feature of IAM, it is a useful and common feature to use. Conditions do not change what (the Action) permissions the policy grants, but can change who (the Principal) and where (the Resource) the policy applies to. Like other elements the name doesn't change if you specify one or many conditions; it is always the singular "Condition".

The Request Context gives conditions access to information about the principal is making the request, what resources they are making requests of, and other environmental data.

Sometimes also referred to as a "condition block", each element in a policy Condition has three parts:

1. An **operator** that defines how keys will compare to each other.
2. A **key** that defines what part of the request context that the operator will check.
3. A **value** for the key that will determine if the condition passes or fails.

The following is an example `Condition` element.

```
"Condition": {
  "StringEquals": {
    "aws:PrincipalOrgID": [
      "o-e6ylpc5gjd"
    ]
  }
}
```

In this example of a condition on a resource policy there is a single condition block entry that checks the `aws:PrincipalOrgID` *key* to see that the request is being made by a principal in the organization with the ID *value* "o-e6ylpc5gjd" by using the `StringEquals` *operator*. If the principal making the request is in the specific organization, then the policy

will apply. This use of the ondition key is a common pattern to trust everyone/everything in a company's organization, even though there are likely too many principals to list in the policy explicitly.

Policies can have multiple condition block entries. If there are multiple condition block entries in the same policy, *all* of the operator entries must be true for the policy to apply (logical AND). If there are multiple keys being checked by the same operator, *all* of the key checks must be true for the policy to apply (logical AND). If there are multiple values for a key, then *any* of them can be matched according to the operator for the policy to apply (logical OR). If a value being checked is not present in the request, then it is likely the condition will not match and the policy will not apply.

Condition Operators

The operator depends on the key you are checking, and the type of its value. The high level categories of operators are:

String: Checks of string based values, which can be case-sensitive or not, and can use policy variables.

Numeric: Checks of numeric values, which can be comparatively larger or smaller.

Date and time: Checks for date/time values using either ISO 8601 date formates or Unix time W.

Boolean: Checks for values that are either "true" or "false".

Binary: Checks for binary values.

IP address: Checks the IP address of the request using the `aws:SourceIp` condition key.

ARN: Checks ARN values, in a case-sensitive way that is aware of the ":" between segments, and can use policy variables.

Null: Checks for if a key is present or not, regardless of the value.

...IfExists: A suffix that can be added to other operators to change the behavior when the key does not exist. The check will pass if the key does not exist in the request, which is the opposite of the standard operators. These operators are used for policies that will apply to multiple resources types that do not support the same keys. This suffix can be added to any operator except for the `Null` operator (for example `StringLikeIfExists`), and supports policy variables if the original operator does.

As detailed above, only a few of the operator types can use Policy Variables. The official

IAM documentaiton is the best reference for all the operator variations.

Condition Keys

The Keys of a condition represent the information in the request context that will be checked against. Key names are not case-sensitive, for example `aws:username` and `aws:UserName` are effectively the same, and will match the same thing. Keys can only be matched if they are present in the request. If a condition key is not present in the request being checked, then the condition will fail.

Condition keys fall in to two broad categories: **global** and **service-specific**.

Global condition keys start with the `aws` prefix, for example the `aws:CalledVia` key contains multiple values about what principals were involved in the request. Global keys contain information about the who is making the request and how they are making it, rather than what the request is doing. Unfortunately, global condition keys are not same for every request and vary depending on the type of principal. The official IAM documentaiton has a list of global condition keys, the types of values they contain, and the requests that include them.

Service-specific condition keys check parts of the request that are related to a specific service or resource. Like actions service keys start with the service's short name, for example `ec2:InstanceType` contains a single string value of an instance family type. Because they are so specific to the service, the SAR is the authoritative reference for all service-specific condition keys.

Condition Values

Where the value can have multiple values, most conditions will check all of the values for a match with the values you have specified. Most values will be a literal value that you specify in your policy, for example a number for checking number values, or a "true" of "false" for boolean value. String condition values are case-sensitive unless defined otherwise in the condition key, for example `StringEquals` is case-sensitive, and `StringEqualsIgnoreCase` is not.

The result of checking a condition value can be one of the four following outcomes:

- **True**: The value of the key in the request passes the operator check specified in the policy.
- **False**: The value of the key in the request does not pass the operator check specified in the policy.
- **Not present**: The key and value specified is not present in the request, so the check is

not passed.

- **Null**: The value of the key was a *null* value. A *null* value is when a value is present but empty, for example an empty string value "" for a string based condition operator. This can happen when the key is based on user input, for example when checking the tags on an AWS resource using the `aws:TagKeys` key.

The *Why* (Effect)

The reason *why* you write a policy is the most critical aspect of your policy, as it determines the effect it will have.

The Effect element accepts one of two values: "Allow" or "Deny".

The following is an example of an `Effect` *element.*

```
"Effect": "Allow"
```

IAM does not accept any other values or even alternate casing of these values, so you'll know quickly if you have it wrong.

By default, all principals that use policies have an *implicit* Deny applied to them. This inherent lack of permissions is part of why AWS can say that your environment is "secure by default". Without any action by you to add an Effect element with the value of "Allow", policies do not provide any permissions.

The root user for your AWS account *does not use* IAM policies. By default it has an implicit "Allow" for all resources in the same account, which is another reason you shouldn't be using your account's root user unless absolutely required.

An explicit "Deny" effect anywhere in the Policy Evaluation logic will override any explicit "Allow" that might be present. This approach means that allowing a principal an IAM action isn't enough to guarantee that it will receive the level of access you expect.

The *Where* (Resource)

The Resource element of your policies defines *where* your actions are applied. The Resource resource element can be a string or an array of strings that specify the ARNs of the resources that the policy will affect. The element name doesn't change if you specify one or many resources; it is always the singular "Resource".

The following is an example Resource element.

```
"Resource": [
  "arn:aws:s3:::ABucketName"
]
```

You can use both wildcard characters (* and ?) in the resource ARNs you include in your array of resources. You can use multiple wildcards in each ARN segment, but they cannot span segments. Wildcards are especially important when defining resources because they allow you to specify them in a way that applies to future resources of that type without having to update your policy.

An star wildcard character matches all resources of all types and services.

```
"Resource": "*"
```

An example of a partial wildcard is when you want to specify a bucket resource and all of the object resources in it.

```
"Resource": [
  "arn:aws:s3:::ABucketName",
  "arn:aws:s3:::ABucketName/*"
]
```

If you're doing anything other than a policy that applies to all resources with *, then you should use the array format. If an array of resources is specified, the policy statement applies to all of the resources that match any of the array members.

Actions and Resources

One thing to watch out for with the Resource element is that not all services and actions support specific resource ARNs. Sometimes, you will have actions that you want to use specific resource ARNs, but the service and action combination does not support such narrowing down. For these actions, you must use the star wildcard character * as the resource.

Be careful when mixing actions that support only the star wildcard character with more-specific resource ARNs in the same policy. In the following example, a policy statement grants actions to allow a principal the ability to put objects in a specific S3 bucket and describe EC2 instances.

```
"Action": [
  "s3:PutObject",
  "ec2:DescribeInstances"
],
"Resource": [
  "arn:aws:s3:::ABucketName/*",
  "*"
]
```

In the preceding example, the intention is for the `s3:PutObject` action to apply to all objects in the ABucketName bucket. Unfortunately, the `ec2:DescribeInstances` action cannot be limited to anything other than "*". Though giving the `ec2:DescribeInstances` access to the S3 resources has no unintended effects, giving the `s3:PubObject` access to all objects in the zone of trust. This combination means that the policy would be allowed to put objects in any bucket in its zone of trust. As a result, you should almost always separate services into separate policy statements so that actions apply to resources appropriately.

Check the official list of service actions that do and do not support specific Resource ARNs to learn which actions support specific ARNs.

Other Statement Elements

The following Statement elements are less frequently used or not required.

Sid

The statement ID or Sid element is usually an optional element to identify a specific statement in a policy.

The following is an example `Sid` element.

```
"Sid": "my-s3-read-policy"
```

If you specify a Sid element, it must be unique across all the statements in the same policy. Setting this element to something human-readable can help to explain the reason for the statement's purpose.

You cannot retrieve the Sid directly via the API, and you can view it only in the web console. If a service requires a Sid, you will get an error message telling you to include it.

Not Elements

The policy syntax also includes the following Not elements that you can use to invert matches:

- `NotAction`
- `NotPrincipal`
- `NotResource`

These inverted elements match all the actions, principals, and resources that are **not** specified in a policy. The implications of the Not elements are not always clear or easy to understand, and you should be careful when using them. These elements make sense to use when the list of things you want to explicitly specify is so long it would violate the policy length quota.

The Not elements are an advanced feature that can have unintended consequences if you are not careful with them. You will not need them for most AWS scenarios, and I would recommend avoiding them if possible.

A policy can have the basic version of the element **or** the Not… version, but not both.

NotAction

A policy that includes the NotAction element matches all actions that are not included in the policy. By combining the NotAction element with an Effect of "Deny", you can explicitly deny most actions but don't need to actually write them all in the statement.

This element doesn't make much sense with an Effect of "Allow" because it will then allow the actions you **haven't** specified. Given that AWS often releases new services and actions, this combination almost certainly will grant actions you didn't intend when writing the policy.

Effective actions when using `Action` *vs* `NotAction`

NotPrincipal

The NotPrincipal element matches all principals other than the principal explicitly specified in your policy. The danger of this element is that it grants privileges to principals you have not referenced directly in your policy and includes principals added after you write the policy. In most cases, this includes the anonymous principal, which means you may end up granting anonymous access to your resources, even though you don't see an effect of "Allow" and a principal of `*` in your policies.

To ensure you use this element to reduce permissions, you should only use the NotPrincipal element with an effect of "Deny".

NotResource

The NotResource element means the policy statement applies to resources that **do not** match those you specify. When combined with an effect of "Deny", this element allows you to define an explicit deny that overrides any other granted permissions. This combination is helpful for sensitive resources such as secret parameters in AWS Systems Manager Session Manager and AWS Systems Manager Parameter Store, objects in Amazon S3, and specific Amazon EBS volumes.

You should not combine this element with an Effect of "Allow" because it would permit the action list to all other resources, including those of other services and resources that haven't been created yet.

Policy Variables

Though the IAM policy language lets you specify what requests are allowed in your environment, you can't know everything up-front. Specifying all possible scenarios in a static language is also impractical because hard coding values directly in to your policies isn't maintainable beyond a single deployment. Because of this, the IAM policy language lets you use variables to define placeholders for important pieces of information that vary from request to request. Variables represent inputs that might not be defined when you write the policy, but will when the policy evaluation occurs.

Variables are only supported by the latest version of policy language. Make sure your `Version` is set to "2012-10-17" in all your policies, otherwise they will default to an older version of policy language that does not support variables.

Variables are only supported in the `Resource` and `Condition` elements of policies. Variables start with a dollar sign (`$`) followed by the variable name surrounded by curly brackets (`{ }`). The following is an example variable in a `Resource` element.

```
"Resource": [
  "arn:aws:s3:::ABucketName/${aws:userid}/*"
]
```

The above example specifies objects in the "ABucketName" S3 bucket that have a path prefix that matches the user ID of the principal making the request. This resource ARN could be used to grant access to write logs about a particular request, without allowing the principal to write data about other principals.

If you are still using IAM Groups in your AWS environment, policy variables are useful because they let you use the same policy but assign different levels of authorization to different users in the same group.

Like conditions, variables are resolved in the request context. Variables are substituted in to your requests as strings, so they only work with the string comparison operators. If a variable can't be resolved to a value, the variable will be empty. To include a variable that might not exist you can add the `IfExists` suffix to the variable name, which resolves to the variable value if present, or an empty string if not present.

The official IAM documentation has a list of the variables that are available, but keep in mind that not all of them will have a value for all requests. For example, the `${aws:username}` value used in many variable examples is only set for IAM users, compared to the `${aws:userid}` which has a value for all requests.

Default Values

The variable syntax supports setting a default value for when the variable is not set in the request. This is especially useful when using user-defined tags as variables, which may not be set on all resources.

The following is an example variable with a default value.

```
${aws:username, 'unknown-user'}
```

In the example above, the variable will be set to the username of the IAM user that made the request. For requests made by other principals (roles and federated users, for example) the variable will resolve to the value "unknown-user".

Conditions

If a condition key is a single-valued key, it can be used as a variable. For example, `aws:PrincipalOrgID` is a single-valued key that represents the organization ID of the principal making the request, and can be used as a variable by adding the variable syntax `${aws:PrincipalOrgID}`. Multivalued condition keys cannot be used as variables, because they can't be resolved to a single value in the request context.

To see a practical application of variables, see the Multi-Tenant Accounts example.

[2] This is probably why the AWS architecture icon for the most common principal, an IAM role, is a hat!

Scenarios

This section of the book contains example scenarios that you will likely come across when using AWS. These scenarios build on the details covered in the first part of this book and focus on the practical applications of IAM. Some scenarios involve AWS services other than IAM, but these are only covered in the context of managing them from IAM. Where possible, I have included links to documentation and other supporting materials so that you can explore further.

Using Multiple AWS Accounts

This scenario will explain how to use IAM in an AWS environment with multiple accounts. Using AWS beyond the most basic usage will very quickly require you to use multiple accounts. The AWS Well-Architected Framework and other official material[3] will mention a multi-account approach to AWS as being "best practice" but they don't always address some the of implications to authentication and authorization with these recommendations.

Other services that impact your use of multiple accounts are beyond the scope of this scenario, but some points to consider have been made in the related AWS services section below.

Why Use Multiple Accounts?

The main benefits of using multiple AWS accounts are security, control, and attribution. The AWS account boundary is the hardest and most secure boundary in AWS. It defines the *zone of trust* for AWS resources, and is what separates the resources of different customers from each other. Using multiple accounts means that there are more potential control points in your AWS environment. By using multiple accounts dedicated dedicated to specific teams, applications, or environments, you can reduce the potential scope of change to your resources. AWS accounts are also used for cost management, since not all resources support tagging for cost allocation.

Challenges With Multiple Accounts

Using multiple AWS accounts is not as easy as it could be, since security and convenience are *inversely proportional*, which means that as you increase one, the other decreases. Visibility can also be difficult in an environment with many accounts. Ultimately AWS has made the decision to keep your AWS environment secure at the cost of some convenience, and that's the right decision to make. Multiple accounts requires additional overhead, but the results and security can't be reliably achieved with other methods.

How many AWS accounts will vary depending on things like your multi-account strategy, your level of AWS usage, and how you manage your applications and the teams that own them. The number of accounts doesn't change how you use IAM to access your resources in a multi-account scenario.

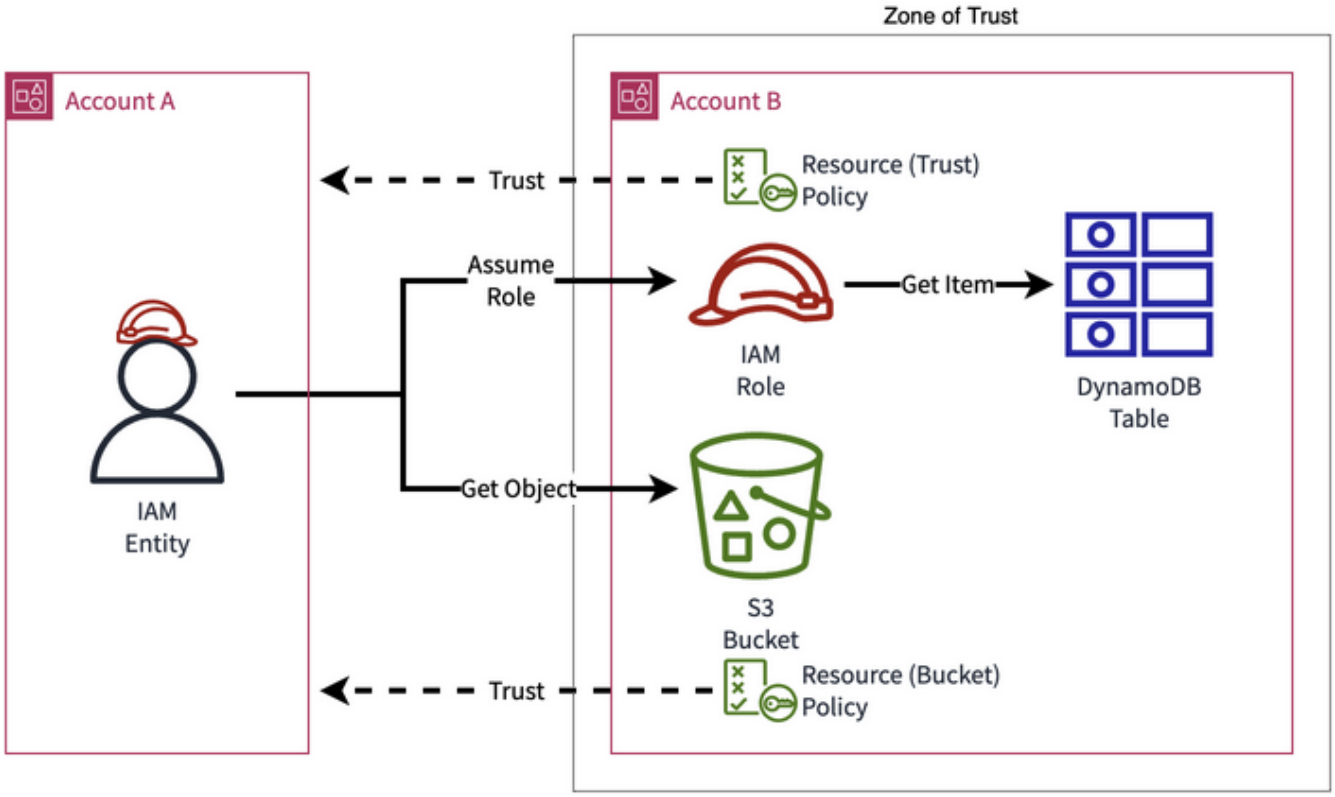
Using multiple accounts is not always intuitive, and part of this is probably due to historical reasons. Before IAM, fine grained authorization was not possible! It wasn't until AWS services and usage evolved that the multiple account approach became practical.

Zone of Trust

All resources exist in the context of a zone of trust, which impacts which principals can access them, and how they are accessed. The zone of trust for a resource is its AWS account, which is why multiple accounts are so important to security. Principals in the same zone of trust as a resource do not require a resource policy to access the resource. Outside of the zone of trust there is no implicit access allowed, and this approach is what enables AWS to be "secure by default". A principal outside of resource's zone of trust must be allowed to perform an action by an explicit allow in the resource's resource policy. Not all resources support resource policies, and those that don't cannot be directly accessed by principals outside of their own zone of trust. The only way to access a resource that doesn't support resource policies is by becoming a principal in their zone of trust. This is why cross-account role assumption becomes critical, since it gives you the ability to change your principal, and therefore change the zone of trust that you operate in.

Cross-Account Access

At a high level, there are two ways to access resources in another account or zone of trust: Directly accessing resources that support resource policies, and assuming a role in the resources' zone of trust, and accessing the resources directly in that new context. At a lower level, the only kind access that can cross the zone of trust is via resource policies, because this is how cross-account role assumption itself is implemented. This scenario will focus on cross-account role assumption, which can work for all resources, rather than cross-account access via service-specific resources policies.



Accessing resources in multiple accounts

In this scenario, there are two AWS accounts: The IAM entity in account A is the *source* of our requests, and the resources in account B that is the *destination* account. You will set up permissions so that a principal in account A can access items in a DynamoDB table in account B.

Though this scenario mentions getting an item from an Amazon DynamoDB table, it is just an example of a resource which doesn't support resource policies, it this approach would work for any resource that doesn't support IAM directly. The diagram above also shows an example of the role directly accessing a S3 bucket across accounts, because S3 buckets support IAM via bucket policies. For an example of this pattern, see the sharing S3 buckets scenario for steps that will work with resources that support IAM directly.

Cross-Account Role Assumption

A big part of understanding cross-account role assumption is that **roles are just resources too**. As you'd hope, IAM resources support IAM! IAM roles have a resource policy called a *trust policy* that allows principals outside of their zone of trust to make an AWS STS assume role API call on them. Once assumed, they return credentials that can be used for future requests in the context of the role's own zone of trust. This is what enables cross-account access of other AWS resources that don't support resource policies. To facilitate cross-account role assumption you need to create a destination role in account B that trusts account A.

Before you can create the destination role, you must prepare a trust policy for it to use.

Create the following JSON document, updating the `SOURCE_ACCOUNT_ID` placeholder with the 12-digit account number of the source AWS account to trust.

DestinationRoleTrustPolicy.json

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::SOURCE_ACCOUNT_ID:root"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

This trust policy grants the principals in the specified account permission to call the `sts:AssumeRole` action on the role it is attached to.

It is possible to narrow this `Principal` element down to a specific principal in the source account such as user or role, but this doesn't actually provide a meaningful improvement to security, because the destination account does not control the configuration of the source account and its principals. In some cases, this may lead to a false sense of security and additional management overhead, such as when the source principal is recreated, which would require the destination role to be reconfigured. For an impressively detailed deconstruction, the unsurpassed Ben Kehoe covers this in an in-depth blog post.

Now that you have a trust policy, you can use it to create a role via the web console or the CLI.

```
aws iam create-role \
  --role-name DEST_ROLE_NAME \
  --assume-role-policy-document file://DestinationRoleTrustPolicy.json
```

The value you give `DEST_ROLE_NAME` is not critical, but it is needed to be known for the next steps. Keep in mind role names are case-insensitive, and must be unique in the account. When successful this command will return the created role, where you will be able to see the role definition, including the trust policy.

At this stage the role can be assumed, but without any identity policies associated it won't have any authorization to access our DynamoDB table. To complete the scenario and for simplicity, you can associate an AWS managed policy with our destination role to take advantage of common access patterns for the service In production environments you should substitute your own policy with relevant and limited permissions.

```
aws iam attach-role-policy \
  --role-name DEST_ROLE_NAME \
  --policy-arn arn:aws:iam::aws:policy/AmazonDynamoDBReadOnlyAccess
```

In this case we're using the `AmazonDynamoDBReadOnlyAccess` managed policy which will give the role the `dynamodb:GetItem` permission, as well as some related permissions.

You can identify an AWS managed policy because it has "aws" in the segment that would usually hold the AWS account ID. Using AWS managed policies is suitable for humans, as they will often be accessing resources to explore and troubleshoot issues. If you are configuring access for programmatic or machine-based access, you should create a specific customer managed policy for the role to use. You can review the full list of permissions granted to this managed policy in the AWS console.

Assuming the Role

With your destination role ready, you must ensure that your source IAM entity has permissions to assume the role. This policy requires you to know the ARN of the destination role.

SourceRoleIdentityPolicy.json

```
{
  "Version": "2012-10-17",
  "Statement": {
    "Effect": "Allow",
    "Action": "sts:AssumeRole",
    "Resource": "arn:aws:iam::DEST_ACCOUNT_ID:role/ROLE_NAME"
  }
}
```

Attach this policy to your entity, using the appropriate attach group/role/user command for your environment.

Once the role has permission to use the `sts:AssumeRole` action, you can make the API call as the source entity to start the assumed role session with the AWS CLI.

```
aws sts assume-role \
  --role-arn arn:aws:iam::DEST_ACCOUNT_ID:role/DEST_ROLE_NAME \
  --role-session-name SESSION_NAME
```

The `SESSION_NAME` you choose must be a unique string with no spaces, and it becomes part of the ARN of the assumed role session that started by the command. The returned response contains the credentials for the assumed role session. The critical values are

`AccessKey`, `SecretAccessKey`, and `SessionToken`, all of which must be included in futures requests to take advantage of the new role.

This BASH script gives an example of parsing the response and setting environment variables so that future AWS CLI commands will be executed in the context of the destination role.

assume-role.sh

```
export $(printf "AWS_ACCESS_KEY_ID=%s AWS_SECRET_ACCESS_KEY=%s
AWS_SESSION_TOKEN=%s" \
  $(aws sts assume-role \
    --role-arn arn:aws:iam::DEST_ACCOUNT_ID:role/DEST_ROLE_NAME \
    --role-session-name SESSION_NAME \
    --query "Credentials.[AccessKeyId,SecretAccessKey,SessionToken]" \
    --output text
  )
)
```

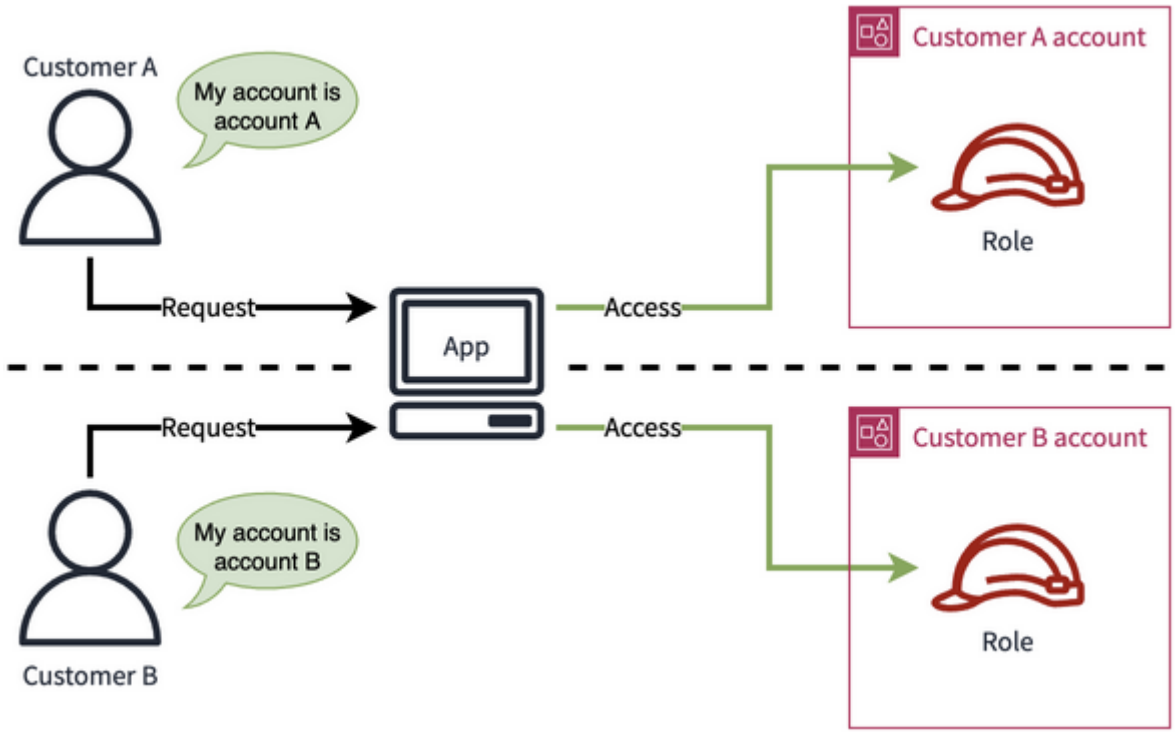
This small script[4] takes care of querying the JSON response and returning the required fields, then exporting them. Remember that AWS permissions are not cumulative, so by using the destination role you lose all of the permissions that the source role had.

At this stage you now have an active session as the destination role, and can make requests of the DynamoDB table in the new zone of trust.

Trusting Third Parties

So far this scenario has focused on scenario within a single organization. Things become subtly different when providing an external or third party access into your environment. For example, a supplier or vendor such as a SaaS provider will likely have access to many different AWS environments, so they can monitor their infrastructure and costs. This delegation of access can make the entity open to suffering from the confused deputy problem described in the IAM PassRole, which can lead to unintended access. To address this, IAM provides the `sts:ExternalId` condition.

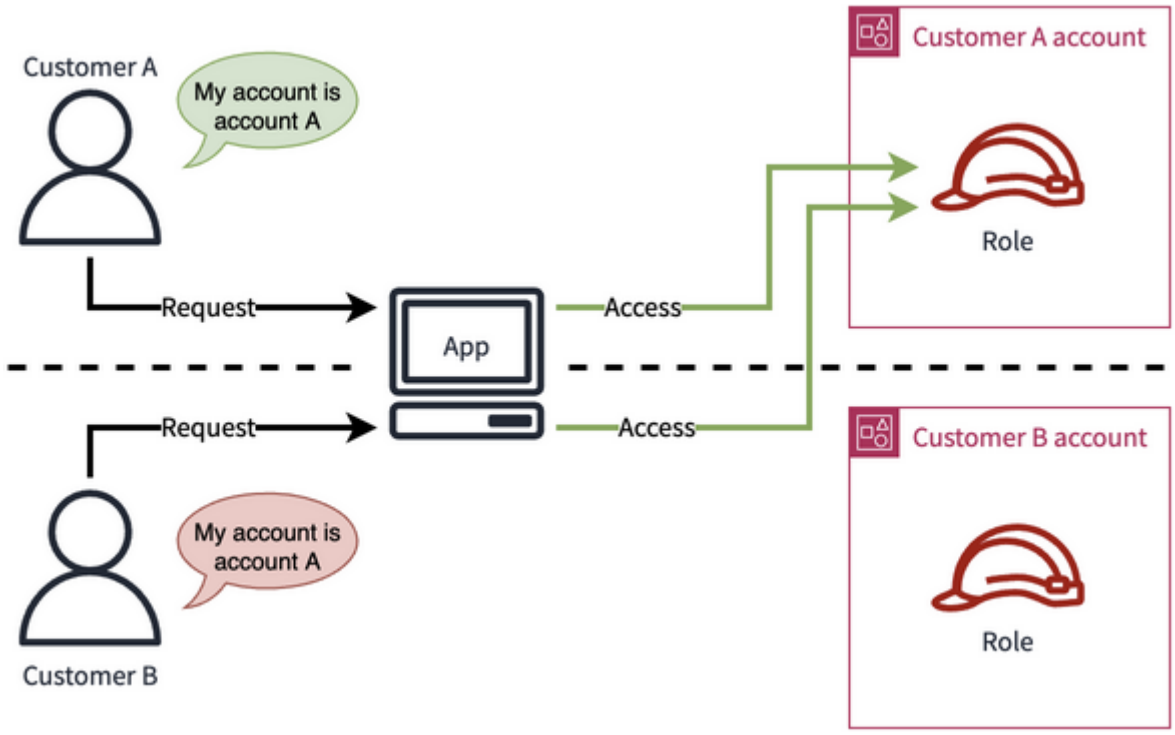
This example shows the "happy path" of delegating AWS access to a third party application, when everyone does the right thing.



Delegated access to AWS roles

In this diagram customers have delegated access to the application so that it can access their AWS environment and perform activities on their behalf. In the above diagram, each customer makes requests of their own environments, everything works as expected. However, Without any additional checks this delegation can be abused intentionally or accidentally if one of the customers gives the other customer's account or role details.

Continuing the example, though the application has legitimate access to both customer A and B's AWS accounts customer B misleads the application by supplying another customer's information.

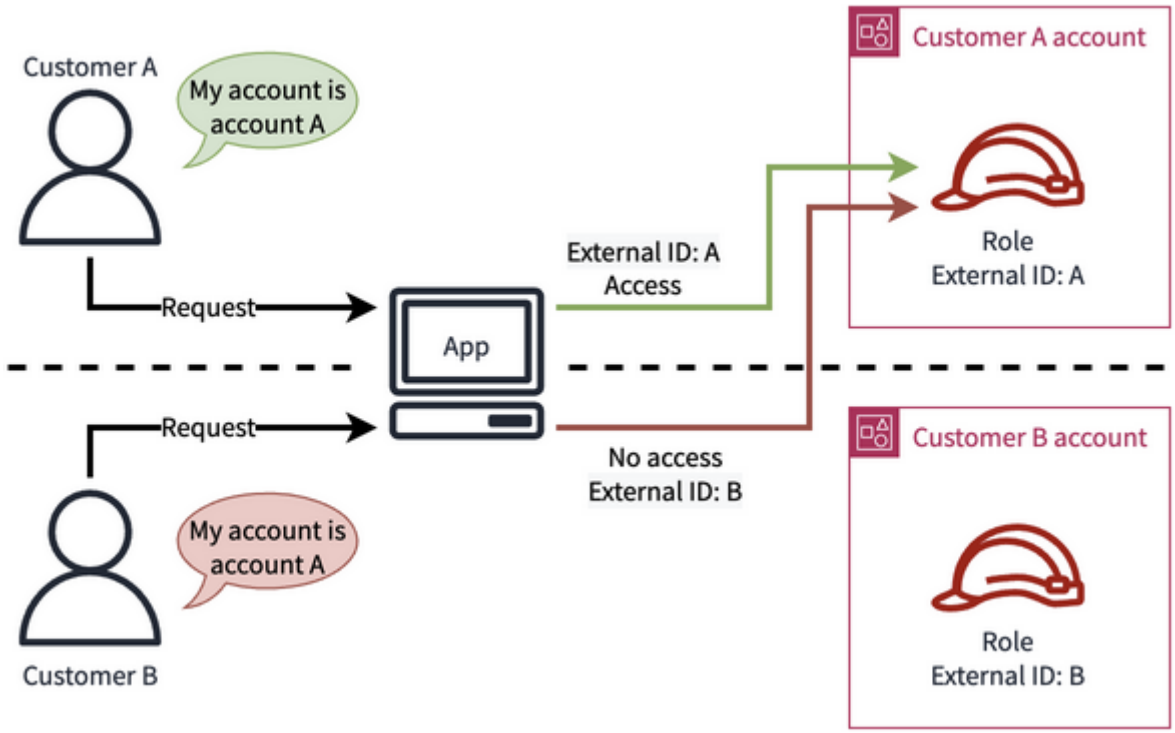


Abusing delegated access to AWS roles

Customer B has request access to information or functionality in account A, by knowing or guessing the AWS account ID or role ARN, neither of which are secrets. Without any additional checks, the application will dutifully retrieve the information from account A and serve it. The role in account A will trust that the application is doing the right thing, even if it is not!

External ID

To address this, IAM supports setting a external ID when assuming a role. This attribute is set when assuming the role, and can be checked as a condition on the destination role, which ensures that the right role is used at the right time.



Delegated access to AWS roles with external ID

The difference between this diagram and the previous one is twofold: The application is including an external ID in it's request to assume the role, and the role's trust policy includes a condition to check the external ID is the expected value. The key trait of the external ID is that is unique for each user/customer, and cannot be set or modified by the customers of the application, it is set for them. By deploying the destination role with the `sts:ExternalId` condition check, the customer is trusting the application, but also verifying its use is deliberate and correct.

An external ID is not a secret value, and can be checked with a simple `StringEquals` condition.

DestinationRoleTrustPolicyExternalId.json

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::SOURCE_ACCOUNT_ID:root"
      },
      "Action": "sts:AssumeRole",
      "Condition": {
        "StringEquals": {
          "sts:ExternalId": "EXTERNAL_ID"
        }
      }
    }
  ]
}
```

Assuming the role is as simple as the original assume role API call, with the external ID included.

```
aws sts assume-role \
  --role-arn arn:aws:iam::DEST_ACCOUNT_ID:role/DEST_ROLE_NAME \
  --external-id EXTERNAL_ID \
  --role-session-name SESSION_NAME
```

Related AWS Services

The following services are not strictly required to use IAM across multiple accounts, but it's extremely likely you will come across them using AWS in a multi-account environment. Knowing where these services fit with IAM will help make your life easier.

AWS Organizations

To mange an AWS environment beyond the smallest scale, you will want to use AWS Organizations. Organizations is the service that allows you to pro grammatically create, manage, and delete AWS accounts. It is also important because enabling the full feature set of AWS Organizations allows you to use SCPs in your environment.

Requests made in organizations have conditions which you can check for to membership of an organization, which can simplify checks when you have many principals and accounts needing to access shared resources. Principals in an organization will have the `aws:PrincipalOrgID` attribute in their request context, and resources have `aws:ResourceOrgID`, which provides another level of control for your `Condition` checks.

Though this can make managing your identity perimeter easier, but doesn't actually make the zone of trust any larger by default.

AWS CloudTrail

As the audit logging service of AWS, CloudTrail is critical not only for securing your environment, it is useful for troubleshooting IAM when things don't behave as you expect. By finding the related events in CloudTrail you can see why API calls have been denied. This is especially useful in a multi-account scenario, where you have multiple layers of policies to troubleshoot.

Like many services that are required to operate at the ridiculous scale of AWS, CloudTrail is *eventually consistent*, so API calls may take a little while to appear in your trail. This generally doesn't take more than 15 minutes, and is much often faster, it just doesn't feel like it when you're waiting for something to appear.

AWS IAM Access Analyzer

A sub-service of IAM, Access Analyzer has grown to include a collection of features that help you manage and secure your identity and resource policies. It has three main areas of functionality: identity policy generation, identity policy review, and resource policy review.

- Access Analyzer can look at your CloudTrail logs and generate an identity policy that includes the actions it has seen that principal take. Be aware that policies generated this way will likely need some additional review and enhancement, because not all actions are logged to CloudTrail, and Access Analyzer does not support all action types.
- When you are creating policies in the IAM console, Access Analyzer is automatically used to review your policies according to best practices, giving you recommendations and warnings. These recommendations can be accessed via the CLI and API as well.
- In a multi-account scenario, Access Analyzer can review your resource policies and generate findings when external identities that have some level of access to resources in your accounts. To reduce some noise, it does not report findings that relate to AWS services or internal AWS service accounts that have access to your resources. Findings do not take in to account any log sources, so they only show potential trust issues, not if the access has been taken advantage of. Access Analyzer supports specifying your entire organization as a zone of trust, in addition operating at the account level.

AWS IAM Identity Center

Previously know as AWS Single Sign-On (SSO), Identity Center gives you the ability to avoid having any users in your AWS environment, which improves security by removing long-term credentials. Via its integration with AWS Organizations it can deploy the destination roles that you will use in a multi-account environment, automating many of the steps in this scenario.

Identity Center has the functionality to operate as an identity provider, but most organizations will bring their own external identity provider such as Azure AD, Okta, Ping Identity, and others.

AWS Resource Access Manager

AWS Resource Access Manager (RAM) is a service that allows you to share selected resources with other principals RAM has been created to allow certain resources to be shared and used more easily in the context of a large organization, and must configured at the organization level to be available to its members. For example, a large enterprise might have a central team that manages the networking resources in their company's AWS environment, and want to share those with other teams that manage the servers that run in those networks.

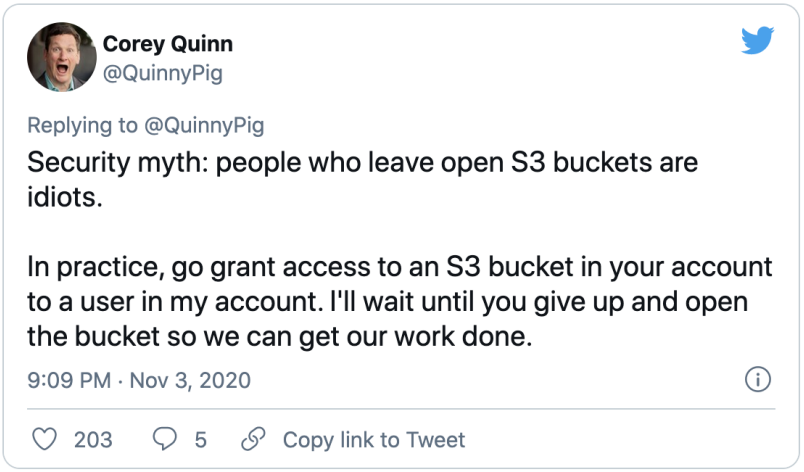
Using RAM requires some additional complexity, but unlocks certain scenarios that aren't possible in IAM on its own. For example it allows cross-account sharing specific resources that don't support resource policies, without requiring you to assume a role. RAM resource shares can also make resources visible in the trusted principal's AWS web console, and can share resources with OUs, neither of which is possible with resource policies. Resource support is limited top a specific subset of resources, and you cannot share a resource that's already been shared with you.

Sharing a S3 Bucket Securely

Sharing an Amazon Simple Storage Service (Amazon S3) bucket is a common task that most builders on AWS have to perform at some stage. For better or worse, it's where most developers resource policies. It's also a pattern often used by many AWS services, which use Amazon S3 as a staging point for data.

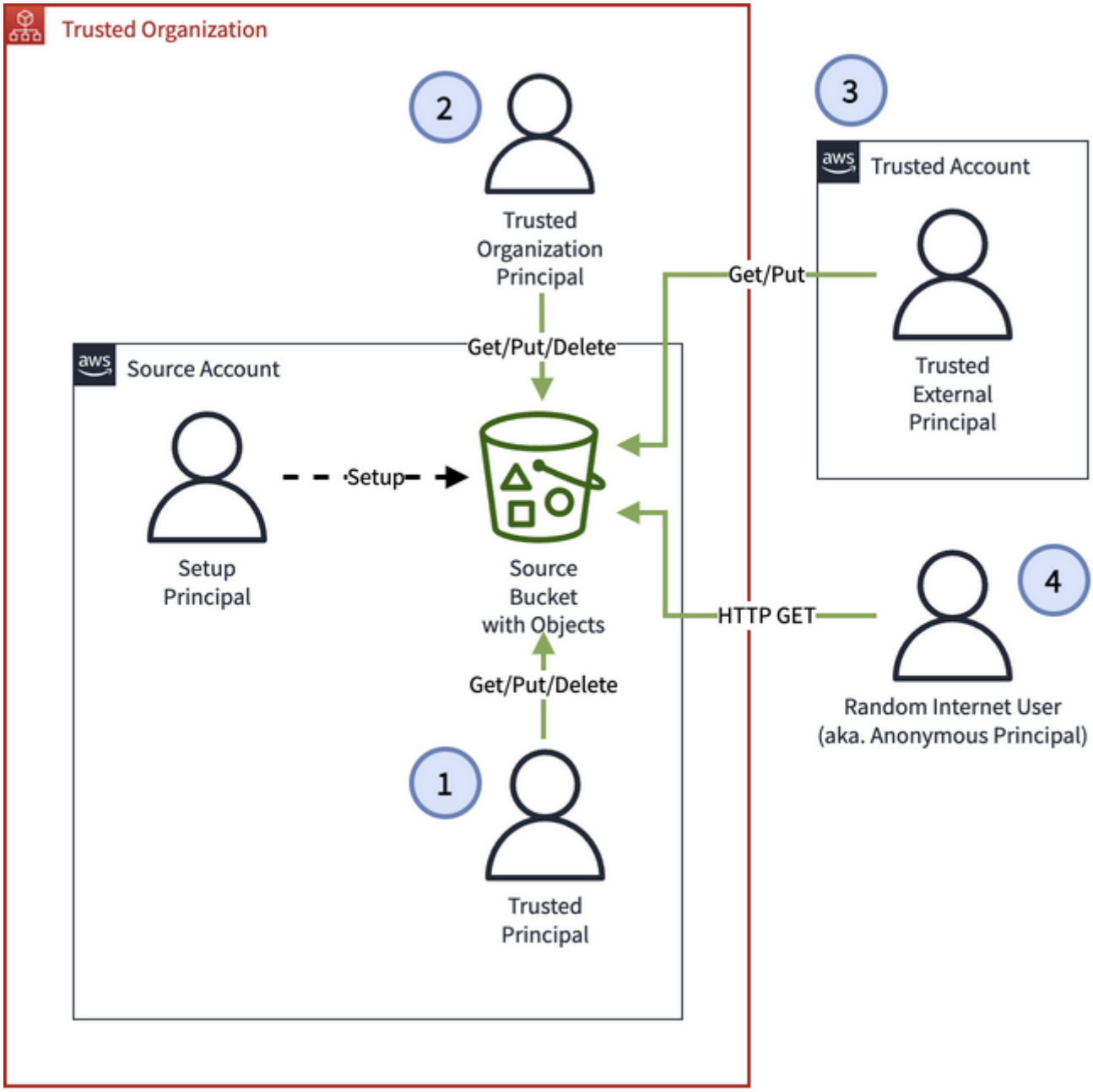
In 2006, Amazon S3 was the first AWS service offered to the public.[5] Because of its age, Amazon S3 has some features that sometimes cause confusion and overlap with other functionality found in IAM. Keep that in mind as you work with Amazon S3.

A common scenario such as this one can come with a few possible gotchas. As well as being a common pattern, it's also a common scenario that people get wrong on AWS. Most of the security breaches you hear about on AWS are Amazon S3 buckets that are unintentionally left open to the public, as summarized in the following popular tweet.



Because of the important and sometimes sensitive data stored in Amazon S3 buckets, getting this simple scenario wrong has caused a lot of issues. This scenario will have you understanding the reason why sharing an Amazon S3 bucket is done how it's done, and you won't just be copying and pasting some random code from Stack Overflow.

The following image shows the resources and principals involved in this scenario.

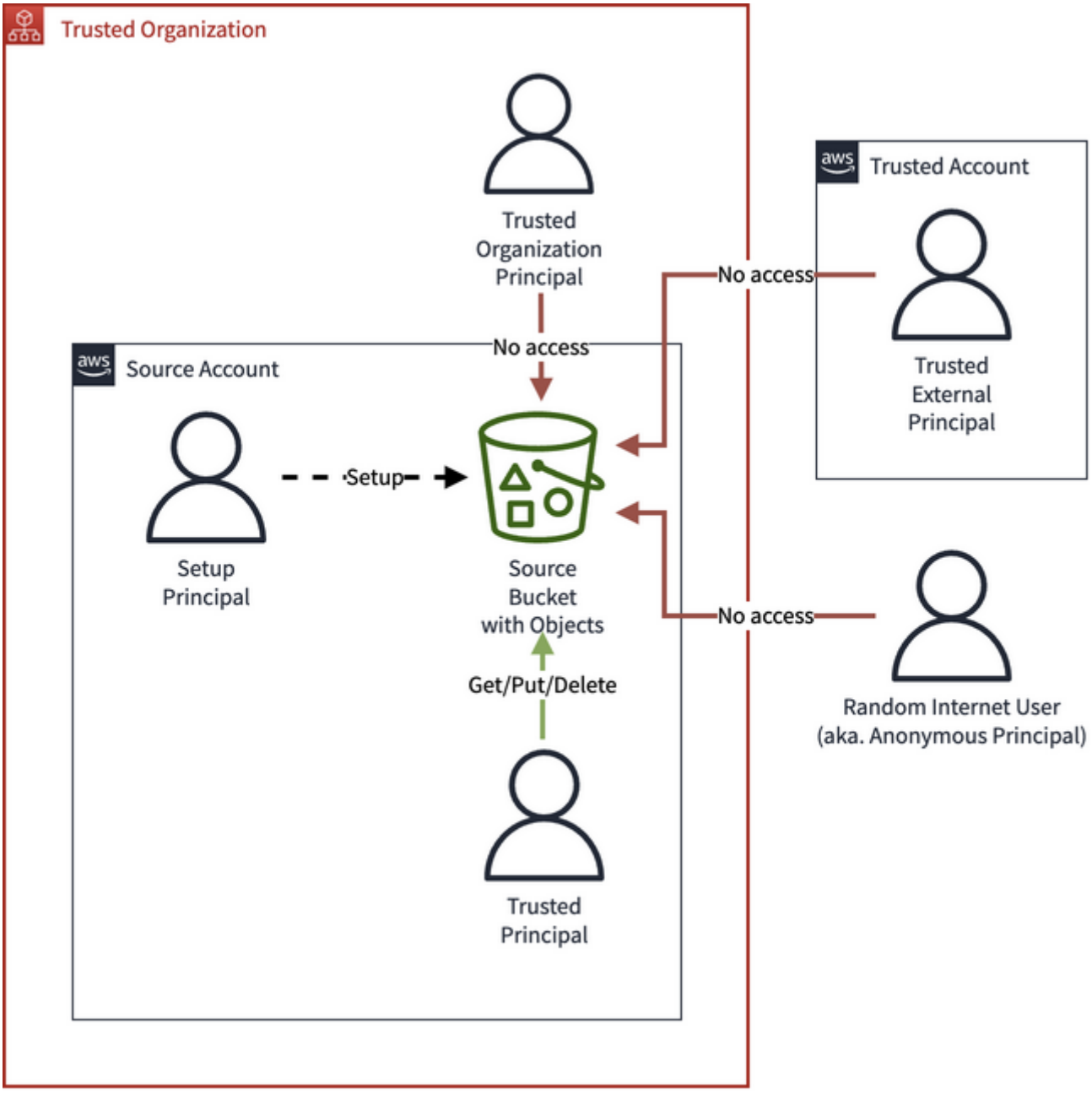


This scenario covers the following outcomes:

1. Sharing a bucket in the same account
2. Sharing a bucket with an organization
3. Sharing a bucket with an external account
4. Sharing a bucket with the internet

Sharing an Amazon S3 bucket in the same account

Though sharing an Amazon S3 bucket is a basic activity that you've probably already done, starting with the setup of a bucket in a single account is a good starting point for the more complicated scenarios that follow. By building from a simple scenario, you can see how the parts of Amazon S3 and IAM fit together to achieve a secure solution.



As shown in the preceding diagram this version of the scenario has two active principals, both in the source account:

- The **Setup Principal** that configures the bucket initially, but does not necessarily interact with the bucket on a regular basis.
- The **Trusted Principal** that accesses objects in the bucket.

IAM user icons have been used for simplicity, but ideally the principals are IAM roles.

A role also might be linked to an Amazon EC2 instance profile, which means the access granted is available to processes running on instances. The other principals in the diagram do not have access to the bucket in this scenario, as shown by their connectors.

The resources involved in this scenario are:

- The principals
- A Bucket administration policy that is associated with the Setup Principal
- An object access policy that is associated with the Trusted Principal
- The source S3 bucket

Note that the bucket has no S3 bucket policy because the Trusted Principal is in the zone of trust (the same AWS account) of the source bucket. Resource policies are not required in this scenario to grant access to principals in the same zone of trust.

This pattern is also used by Service-Linked Roles so that an AWS service can access your bucket.

Setup Principal

Setting up an Amazon S3 bucket is usually a one-off task that involves creating a new resource, so locking it down to a specific resource is hard. Because the bucket doesn't yet exist, you can't yet be certain about the resource's name and its ARN. Without the ARN, you cannot limit the policy of the Setup Principal to a specific resource.

In practice I would usually use the `AmazonS3FullAccess` AWS managed policy, and attach it to the Setup Principal directly.

Even though AWS Managed Policies are useful, don't forget about their drawbacks. Because AWS managed policies are generic, they work on all resources in an account.

The following is this scenario's policy.

AmazonS3FullAccess.json

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "s3:*",
      "Resource": "*"
    }
  ]
}
```

Though this policy doesn't follow the best practice of least privilege, it is appropriate for infrequent, human-driven administration tasks. Do not use this policy for day-to-day

bucket access—only the initial setup.

With this policy, the Setup Principal can create an Amazon S3 bucket using the AWS CLI with the following command.

```
aws s3 mb s3://MY_BUCKET_NAME
```

See the Conventions in This Book for links to configure the AWS CLI if needed.

If your bucket is created and set up by an automated process, you should lock down the permissions to the least number of actions that can achieve the task. See the Granting Least Privilege Access scenario to see how to craft an appropriate policy in this case.

Trusted Principal

This is the principal that uses the bucket on a regular basis, and it is most likely an application or automated process. Some common examples of this principal are a role associated with an AWS Lambda function or with an Amazon EC2 instance through an instance profile, so that applications running on the instance can communicate with AWS services.

TrustedPrincipalS3Access.json

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "s3:DeleteObject",
        "s3:GetObject",
        "s3:GetObjectVersion",
        "s3:PutObject"
      ],
      "Resource": [
        "arn:aws:s3:::MY_BUCKET_NAME/*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "s3:ListBucket"
      ],
      "Resource": [
        "arn:aws:s3:::MY_BUCKET_NAME"
      ]
    }
  ]
}
```

This policy is another identity policy, so it needs to be attached to the principal. The first statement allows the principal to get, put, and delete objects in the bucket, which is why the resource ARN is scoped to objects in the bucket. This is done with a wildcard `*` after the slash `/` that separates the bucket and object names because object names are not usually known in advance.

Amazon S3 Buckets vs Amazon S3 Objects

A common sticking point when working with Amazon S3 actions for IAM is the fact that buckets and objects are two distinct resources in S3, even though they are closely related and share an ARN format. An object without a bucket is not possible. A bucket without an object is possible, but useless as a bucket.

In a simple scenario like this where the sharing happens within an account, sometimes the S3 actions are combined in the same statement as in the following policy.

TrustedPrincipalCombinedS3Access.json

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "s3:DeleteObject",
        "s3:GetObject",
        "s3:GetObjectVersion",
        "s3:ListBucket",
        "s3:PutObject"
      ],
      "Resource": [
        "arn:aws:s3:::MY_BUCKET_NAME",
        "arn:aws:s3:::MY_BUCKET_NAME/*"
      ]
    }
  ]
}
```

Combining Amazon S3 actions in the same statement works because the object actions apply to the object ARN, and the bucket action applies to the bucket ARN in the Resource field.

Though writing the policy this way saves a few lines of JSON, I encourage you to keep the actions and resources that they apply to in their own statements to make your policies easier to maintain, update, and troubleshoot. The most common reason to combine actions and resources is if you are up against the policy size limit.

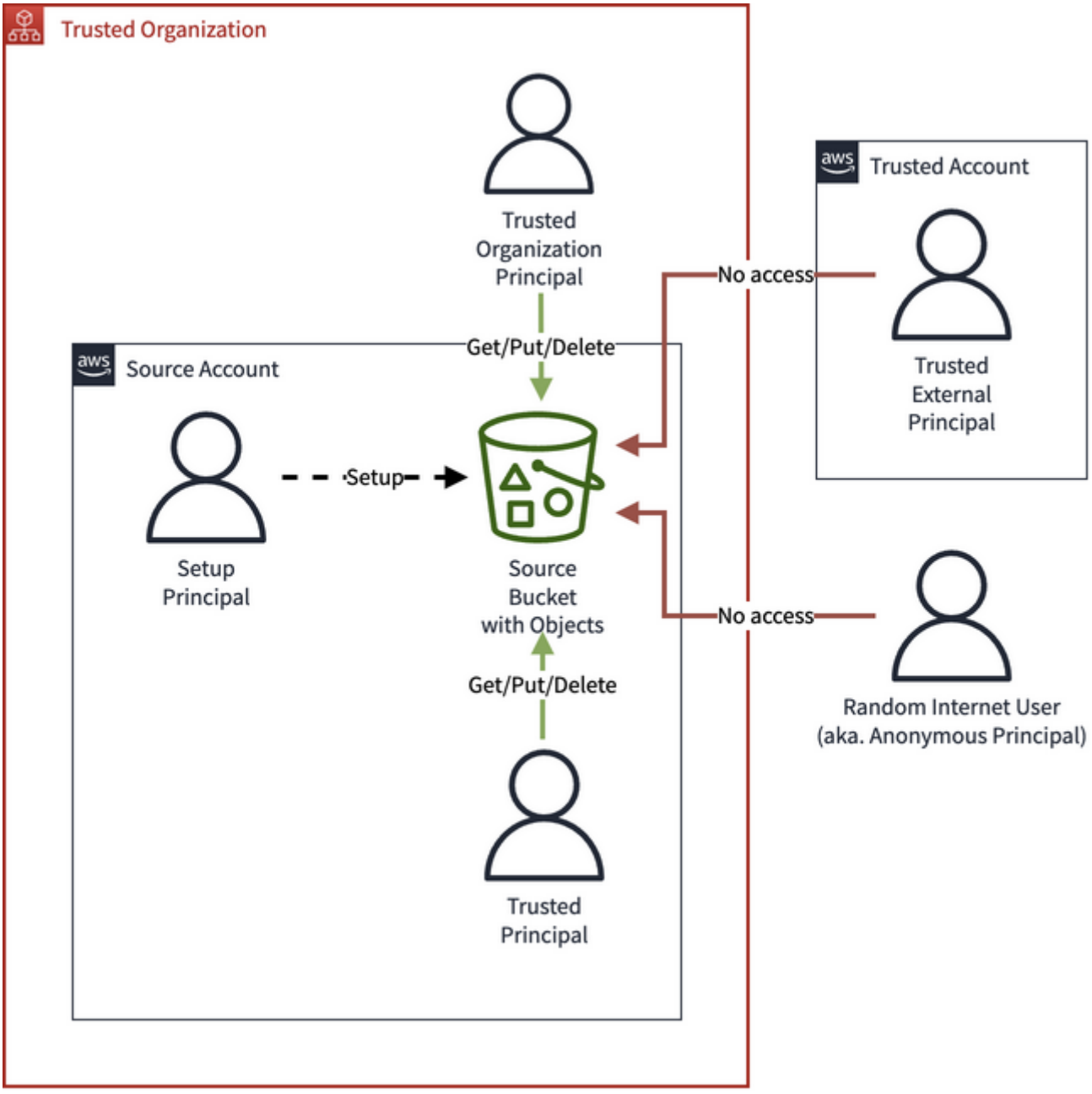
Though the most common actions that target buckets have the word `Bucket` in them and most actions that target objects have the word `Object` in them, not all do. See Actions to learn which actions apply to which resource.

Some API actions only accept a wildcard star `*` in the `Resource` field, even though they only apply to specific resources.

Sharing a bucket with an organization

This scenario builds on the previous scenario Sharing an Amazon S3 bucket in the same account to provide access to another principal in AWS Organizations.

This scenario builds on the previous to provide access to other principals the AWS Organization.



The Trusted Principal continues to have access to the source bucket because it is in the same organization. The identity policy for the Trusted Organization Principal is the same as the Trusted Principal in the previous scenario.

TrustedOrganizationPrincipal.json

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "s3:DeleteObject",
        "s3:GetObject",
        "s3:GetObjectVersion",
        "s3:PutObject"
      ],
      "Resource": [
        "arn:aws:s3:::MY_BUCKET_NAME/*"
      ]
    },
    {
      "Effect": "Allow",
      "Action": [
        "s3:ListBucket"
      ],
      "Resource": [
        "arn:aws:s3:::MY_BUCKET_NAME"
      ]
    }
  ]
}
```

Bucket Policies

Since this requires sharing the source bucket outside of its *zone of trust* it requires a resource policy. Unsurprisingly S3 resource policies are called *bucket policies*, and the terms are used interchangeably.

Sometimes you might not know the exact principal ARN that is going to access the resource in advance. You might also have a legitimate case where you want many principals to be able to access a certain resource, and you would run in to policy size limit Quotas. An example in most AWS environments is a centralized S3 bucket for CloudTrail audit logs, where want all of your AWS accounts to send their logs for safe keeping. These constraints require you to set a more permissive `Principal` value than you would usually like, and then limit the policy's use via a `Condition`. In this case we use the `aws:PrincipalOrgID` condition to limit access to principals in the same AWS Organization, which is a string condition that takes the AWS Organization ID:

TrustedOrganizationBucket.json

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "s3:DeleteObject",
        "s3:GetObject",
        "s3:GetObjectVersion",
        "s3:ListBucket",
        "s3:PutObject"
      ],
      "Resource": [
        "arn:aws:s3:::MY_BUCKET_NAME",
        "arn:aws:s3:::MY_BUCKET_NAME/*"
      ],
      "Principal": "*",
      "Condition": {
        "StringEquals": {
          "aws:PrincipalOrgID": [
            "MY_TRUSTED_ORG_ID"
          ]
        }
      }
    }
  ]
}
```

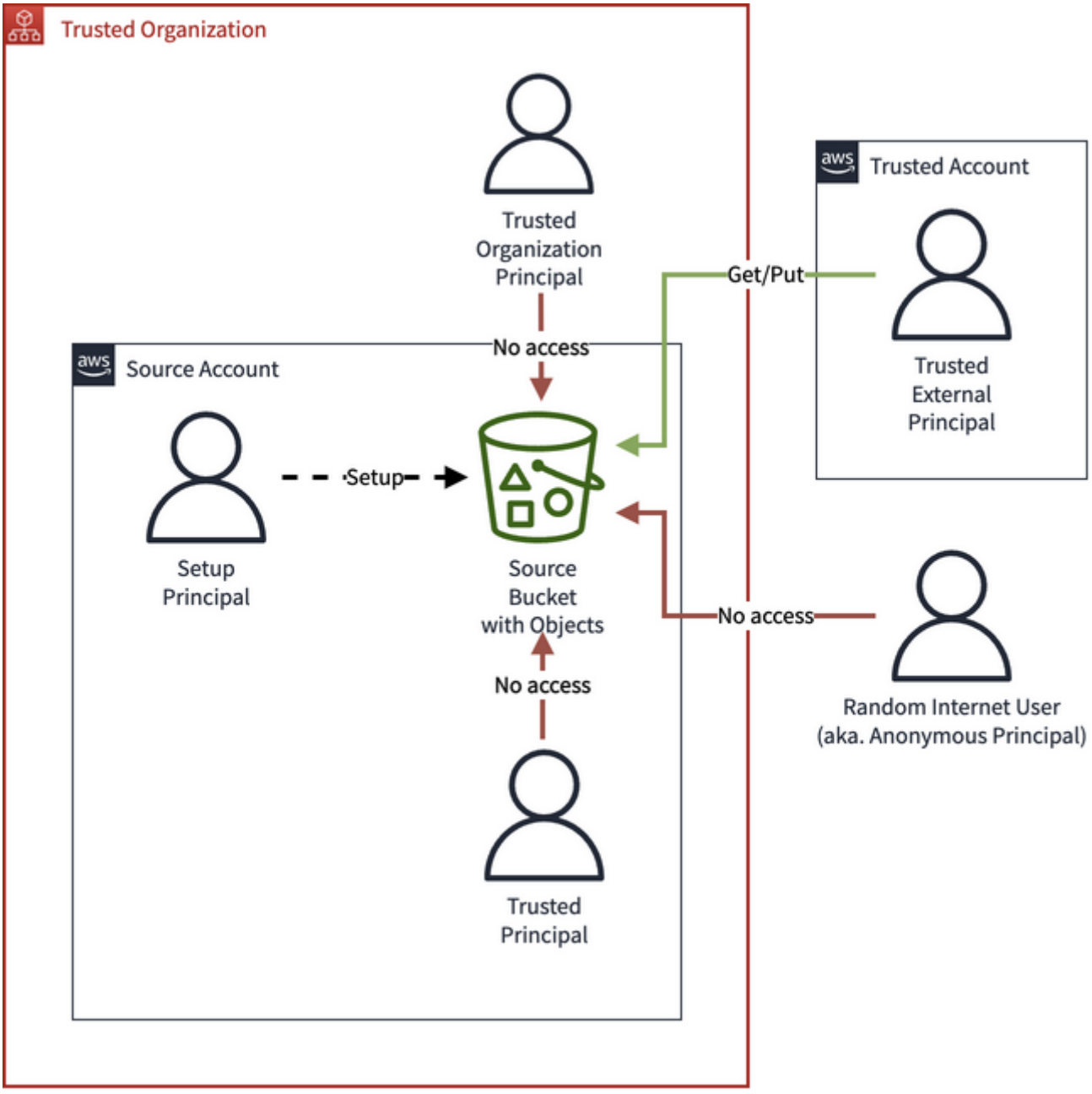
Once you have updated the value of `MY_TRUSTED_ORG_ID` to your organization, the resource policy is *put* on the bucket via a call to the S3 API:

```
aws s3api put-bucket-policy --bucket MY_BUCKET_NAME --policy
file://TrustedOrganizationBucket.json
```

You could follow the same process as these to share your bucket with a different organization. The only difference is the value you set for `MY_TRUSTED_ORG_ID` the `aws:PrincipalOrgID` condition. Make sure you use the right one!

With an External Account

Probably the most common S3 sharing scenario is making your data available to an external entity via their AWS account.



As with the previous example this involves sharing outside of the zone of trust, so resource policies must be involved. It is unlikely that you will want to trust a large number of principals in this scenario, so using the `Principal` field is the most appropriate way to control access. This policy includes a few example `Principal` types:

TrustedExternalBucket.json

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "AWS": [
          "TRUSTED_ACCOUNT_ID",
          "arn:aws:iam::TRUSTED_ACCOUNT_ID:root",
          "arn:aws:iam::TRUSTED_ACCOUNT_ID:role/ROLE_NAME",
          "arn:aws:iam::TRUSTED_ACCOUNT_ID:user/USER_NAME"
        ]
      },
      "Action": [
        "s3:GetObject",
        "s3:GetObjectVersion",
        "s3:PutObject"
      ],
      "Resource": [
        "arn:aws:s3:::MY_BUCKET_NAME/*"
      ]
    }
  ]
}
```

To apply this policy once you've updated the variables with your own values, you *put* it on the bucket:

```
aws s3api put-bucket-policy --bucket MY_BUCKET_NAME --policy
file://TrustedExternalBucket.json
```

This example principals in this policy are:

- `TRUSTED_ACCOUNT_ID` The trusted AWS account's 12 digit ID e.g. 112233445566
- `arn:aws:iam::TRUSTED_ACCOUNT_ID:root` The trusted AWS account's ARN
- `arn:aws:iam::TRUSTED_ACCOUNT_ID:role/ROLE_NAME` A specific role in the trusted account
- `arn:aws:iam::TRUSTED_ACCOUNT_ID:user/USER_NAME` A specific user in the trusted account

While the diagrams show user icons, you shouldn't be using IAM Users if at all possible. You **definitely** should not be creating IAM users and giving those credentials to an external party.

When you specify an external account you are implicitly trusting *any* principals inside that account. They will need to have the corresponding identity policy, but that's up to the external account anyway. If you want to limit access to a specific principal, the specify it explicitly.

Specifying the trusted account's ID and ARN are the same from a functionality point of view.

If the trusted AWS account is inside your AWS organization but you don't want to trust your whole organization you will need to use this method to share your S3 bucket, even if it's not technically *external*.

The principal will need appropriate identity policies to access S3, just as they would if the bucket was in their own zone of trust. In this scenario you often wouldn't want the external principal to do things like delete data so the policy is slightly different to the previous identity policies:

TrustedExternalPrincipal.json

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "s3:GetObject",
        "s3:GetObjectVersion",
        "s3:PutObject"
      ],
      "Resource": [
        "arn:aws:s3:::MY_BUCKET_NAME/*"
      ]
    }
  ]
}
```

With this policy, the organization-based principal has lost access. You could include the Trusted Organization Principal by using the `Condition` and `Princapal` fields from both scenarios in the same bucket policy.

Got something in your company, but outside of AWS that needs to talk to an S3 bucket securely? This might be a legitimate use for an IAM user, but **only if** the system is ignorant or unable to integrate with IAM natively.

Create a user with access keys, and give those to your external/on-premises system so

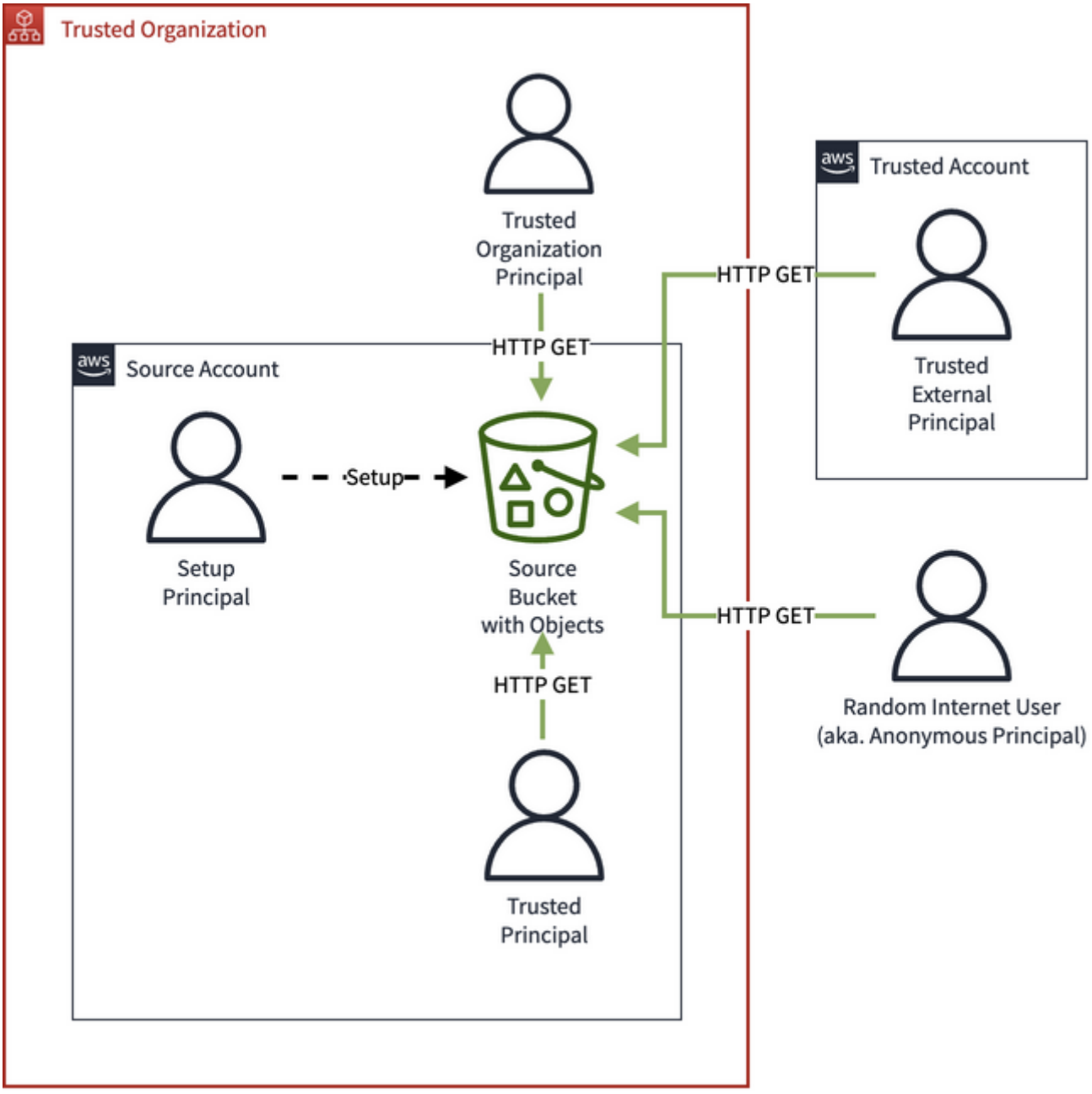
that they can authenticate to IAM. Just make sure you limit the user to the least privilege required, and make sure you create users/keys for each and every activity so you can revoke access as needed.

Sharing a bucket with the internet

S3 has the ability to serve a bucket's objects over HTTP. If you want to serve static content to the Internet, this is a good option so that you don't have to worry about managing servers to do the job. Unfortunately, this scenario is a common source of "accidents" on AWS.

While your environment and resources are *secure by default*, this is probably the most common misconfiguration of AWS out there. So much so that AWS added additional layers of protection and notification to the feature in 2018.

In this scenario I will detail the required steps to serve bucket objects over HTTP, but this is by no means an exhaustive look at S3 website hosting. Beyond setting up access, see the S3 documentation for the full details on how to host your website on S3.



Unlike all the earlier scenarios, no identity policies are required here. A few AWS services allow requests from Anonymous Principals, and S3 is one of those services. While the Anonymous Principal does not have an associated identity policy, there is no explicit `Deny` in this scenario. This, when combined with the corresponding resource policy means that access is allowed.

Very few services allow requests from the anonymous user principal. SQS, SNS, and STS are the only services other than S3 I could find direct references about permitting anonymous requests.

As shown in the diagram, all the other principals in the scenario will also gain HTTP access to the bucket, because they default to the anonymous principal's level of access. Note that the access type has changed to a HTTP GET, rather than the usual API-based access.

The resource policy provides access to the bucket and its objects to the anonymous principal:

InternetHttpBucket.json

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": "*",
      "Action": [
        "s3:GetObject",
        "s3:GetObjectVersion"
      ],
      "Resource": [
        "arn:aws:s3:::MY_BUCKET_NAME/*"
      ]
    }
  ]
}
```

Apply this the policy to the bucket:

```
aws s3api put-bucket-policy --bucket MY_BUCKET_NAME --policy
file://InternetHttpBucket.json
```

This policy is again more restrictive than the previous external scenario. You generally won't want users over the Internet to list all the contents in your bucket, or update your objects.

Enable S3 Website Hosting

Enabling the bucket for website hosting is what enables the Anonymous principal to make requests of the resource. This is in addition to the resource policy, which grants the actual permissions to the anonymous principal.

You can enable website hosting via the web console, or via a short AWS CLI command:

```
aws s3 website s3://MY_BUCKET_NAME/ --index-document index.html --error
-document error.html
```

This will make your bucket available at the domain `http://MY_BUCKET_NAME.s3-website-AWS_REGION_SHORTNAME.amazonaws.com`. If you're browsing the web console you will see special "Public" badges on buckets that are public.

If you navigate to your bucket, you will see a "404 Not Found" error, with the code "NoSuchKey". This is a good thing! S3 is attempting to serve your objects via HTTP, but you will have to have put some HTML in there to see anything interesting.

S3 website buckets **only** support HTTP (note no "S") access. For HTTPS support you must put something in front of the bucket, like CloudFront aws.

S3 Block Public Access

If at this stage you're seeing a "403 Forbidden" error, with the code "AccessDenied" you might have **Block Public Access** enabled on your account or bucket. This is an additional layer of protection that AWS added to prevent accidental misconfiguration of S3.

When it is enabled the resource policy applied to a bucket will be overridden, and have no effect. If public buckets are blocked at an account level, the block applies to all regions.

For the full variety of options to block S3 public access, see the documentation aws.

Other Ways to Access S3

The following methods of accessing S3 don't involve IAM, so haven't been covered in depth here. In some cases they predate IAM being available as service.

CloudFront

S3 buckets are frequently served behind Amazon CloudFront aws for caching. As part of this, access to the buckets are often restricted so that the assets in the buckets can't be accessed outside of CloudFront. The CloudFront distribution needs to be configured with an Origin Access Identiy (OAI) aws, and the OAI is used in the `Principal` field of your bucket resource policies.

Amazon S3 access control lists

While it is possible to share data between accounts using Amazon S3 object access control lists (ACLs), this is the pre-IAM way of sharing S3 and shouldn't be used unless there's something that can't be done with resource policies.

The only recommended reason is to grant write permission to the Amazon S3 Log Delivery group to write access log objects to your bucket aws. One hint that ACLs are an *older* feature of S3 is that they're in XML, unlike IAM policies which are all JSON.

ACLs also rely on Canonical User IDs, which are obfuscated AWS account IDs and look like `883c716af28a0e0b0dc3d6f2de8cea7177aa63e5a56d97cc0fc7e0cea3710083`. They can only trust entire AWS accounts, and cannot be limited to specific principals like bucket policies can.

An advantage ACLs have over bucket policies is that can apply at an object level. This allows you be extremely fine-grained approach, but this is excessive in most cases. It's usually simpler to create a dedicated bucket for the security posture you require, since there is not cost for additional buckets.

The main use for S3 ACLs you may see is if objects in a bucket are not owned by the bucket owner.[6]

S3 Presigned URLs

Generating a S3 presigned URL is another way to access objects in a S3 bucket, and is also a feature unique to S3. The permissions granted by a presigned URL are for a specific action, and cannot exceed that of the user that created the URL. Presigned URLs have an expiry time, and are effectively a "bearer token" that grants limited permissions to the holder.

Cross-Account Role Assumption

As with any other AWS resource, you might also choose to assume a role in the source bucket's AWS account and access it directly. Like in the Trusted Principal scenario, this means that no resource policy would be required. See the multi-account AWS scenario for this pattern.

Granting Least Privilege Access

In security in general, and in identity and access management in particular, it is important to plan for the worse-case scenario. Though you don't want the worse-case to happen, you acknowledge that it *might* happen and prepare for it. This is the motivation behind the principal of *least privilege* in computer security. By giving entities the least functional level of access to your resources in your environment, you limit what can possibly go wrong.

For IAM this means that your policies should be granting principals in your environment the least amount of actions, to the most specific resources. In practice, this is a lot harder than it might sound. The IAM policy language is JSON, which at a glance looks pretty simple, right? This sometimes hides the fact that simple-looking statements and policies

combined together can result in something complicated. With thousands of IAM actions to choose from, narrowing them down to a specific list is not a small or easy task. To be able to give the minimum number of actions, you need to know everything that a specific principal might need to do to perform their job. Their job might change over time, or as AWS releases new services and features. Some tasks on AWS require permissions across more than one service, which is not always obvious when planning.

The reaction of some teams and security engineers is to try to lock down policies as soon as possible, hoping to limit risk and avoid security issues. Unfortunately this approach doesn't take in to account the impact this has on developers and engineers trying to do work on AWS, and solve business problems. Taken to an extreme, I have seen teams given principals no access to start with, and have developers request every permission they require. This makes the development process long, inefficient, painful, and developers hate it. In this case, too much security too soon is almost as bad as doing too little too late. Preventing people from doing their job, or making it slow and painful to do it, is a sure way to have them ask for more permissions than they need, or look for ways to get around the limitations.

The reality is that there is no single solution to this problem on AWS. You will probably need to use multiple approaches to achieve your outcome, and as with many areas of security you will have to make trade-offs. The most secure policy is one which grants no permissions, but that isn't useful; the most useful policy is one that grants all permissions, but that isn't secure.

Wildcards in Policies

The main challenge when working towards least privilege IAM policies is managing wildcards. To craft policies with the minimum privileges required you will need to remove wildcards as much as possible, while still allowing principals to perform their job function.

To use an extreme but simple example, consider the only statement in the Administrator AWS Managed Policy that almost everyone on AWS has used at some point.

```
{
  "Statement": [
    {
      "Action": "*",
      "Effect": "Allow",
      "Resource": "*"
    }
  ]
}
```

The reality is that the simplest policy is also the most insecure, since this policy grants all actions to all resources for all services, present and future.

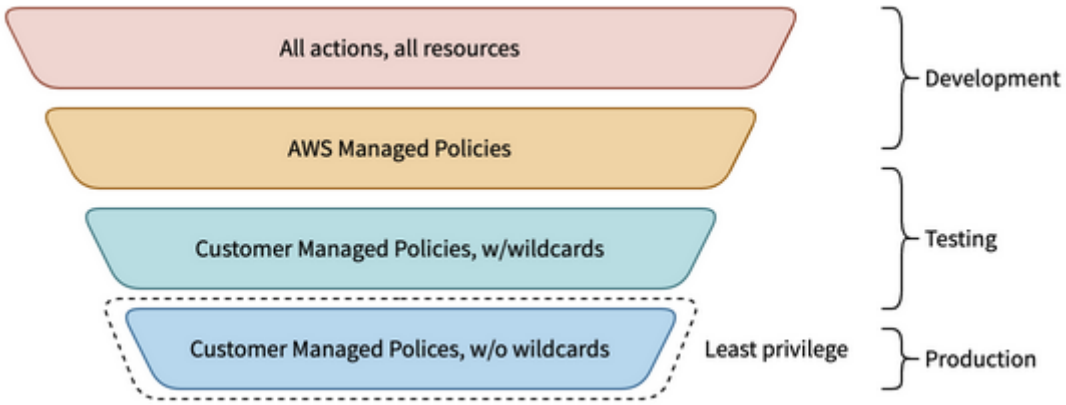
"Painting" Policies Progressively

An alternative approach to locking down all your policies is to deliberately and progressively refine your policies over time. Don't try to meet least privilege from the beginning of the development process, but instead acknowledge that the process will take time. Think about painting a picture; while you might know the subject and what you want it to be, you don't know all the details when you start with a blank canvas. By starting with a painting-like approach, you can unblock your engineers, and still achieve your security outcomes.

By starting at a high or coarse level of detail and progressing to your least privilege outcome, you can work towards the most secure outcome while demonstrating progress. Adding detail to your policies gradually means that they get increasingly clear and explicit, and reduce the amount of privileges granted. If something needs a touch-up because desires or requirements change, you can add it at a later stage without needing to start over from scratch. At the same time, developers and engineers have a level of access they need to do their jobs, without hitting being limited by overly restrictive policies.

This approach necessarily takes some time and iteration, but in reality it's unlikely you

can avoid this. Though this may feel slower to start with, in practice I have seen this result in better outcomes in the long run. In IAM terms, this means using the right policy at the right time. By iterating on your policies and leveraging the different IAM policy types available, you get the flexibility and security.



This refinement means that you do the work at the right time, as your understanding of requirements improves, while also having the least impact on developer velocity.

This approach works well when used in the context of a multi-account environment. As shown by the labels, the different types of policies cross environments so that you can test both functionality and security before releasing to the next level of security. By using more than one AWS account, you have another dimension to restrict your policies: the zone of trust provided by an account. This means that even though you are giving excessive privileges in the early stages of your policy development, the blast radius is still limited by the AWS account boundary.

An exception to the "no wildcards in production" rule might be for human access. If your environment is following least privilege for programmatic access, then humans should only be accessing the environment to troubleshoot. In these cases, having additional access to actions and resources might be a good thing, so they can fix them.

Without the account-level separation between environments you are relying on developers and engineers to do the right thing, and not make any mistakes.

Use AWS Managed Policies, Carefully

To start with, your access could be a "blank canvas", which in policy terms means administrator access. From that point you should quickly move to AWS managed policies as you decide at a high level what the principal should do. AWS manged policies are more convenient than they are secure, but they're still better than administrator access. This is like filling in the background of a picture, giving the policy its general shape and definition.

Since these managed policies are available in all AWS accounts, they are quick and easy to get started with. Because they always apply to all resources, they can't be considered least privilege, so they should be used only in the early stage of your policy development.

The naming convention of AWS managed services makes it easy to start with, since they generally follow the convention of service name + role or level of access. This makes it easy to search and filter for managed policies relevant to the services involved. Most services have a "...ReadOnly" policy which means you can apply a limit on your principal's access to services that you might remove. These managed policies are also aware of most of the cross-service interactions that require IAM actions from other services.

Since these polices are managed by AWS, you cannot edit them to restrict their access. AWS can and will update these policies when new service features require access to new actions and other services. Though these limitations prevent AWS managed policies from being a least privilege outcome, they can still be part of the policy development process.

Craft Your Own Policies

Moving beyond AWS managed policies, customer managed policies represent customized policies that can meet your least privilege requirements. This is the stage of painting your policies where you start to add specific details and features, making them unique to your scenario and requirements. Customer managed policies give you complete control over your policies, which is what you need if you are going to achieve your security outcomes. Since your write the policies you can scope them specific actions and resources, but this flexibility comes at a cost since it is now your responsibility to give the least access possible.

At the beginning the development of your custom policies, I recommend using wildcards. For example, you can use `…:Get*`, `…:List*`, and `…:Describe*` to grant most read actions for specific services.

At this stage of painting your policy it can be helpful to go back and review related AWS managed policies, especially when you need to identify what other services your principal needs access to. Just as with painting a picture, it's ok to take inspiration from other sources! As an example, some AWS managed policies include actions that a principal only needs if they want to use the web console.

Going through the effort of creating and managing your policies is extra work that you will need to take on when using customer managed policies. If your requirements or the relevant service permissions change over time, it's up to you to update your policies. This is why it makes sense to use them once you've already identified the limited services and access required, and not earlier in the development process when requirements often

change.

Lock Down Your Policies

As confidence in your policies increases through repeated deployment and use, and the number of changes required decreases, you should lock down your customer managed policies. In the context of policies, this means removing all wildcards from your action list. Even read-only wildcards like `…:Get*` can still grant unexpected privileges if AWS releases a new feature for the service that match, and this would violate the principal of least privilege.

Note that for the resources in your policies, you might not be able to get rid of **all** wildcards in your resource ARNs. This means that you should spend the time to review and restrict them as much as possible, but that you may still use some wildcards. For example, CloudFormation appends a random string to the end of resource names to ensure there's no conflict, but the beginning of resource name will follow a pattern you can use with a wildcard to increase the security.

AWS Services to Help

The challenges achieving least privilege are common to all AWS customers. Different customers with different risk postures and security budgets will spend different levels of time and effort on solving it, but the problem is not unique to any customer on AWS. Because of this, AWS provides some tools and services to help, write, generate, and test your policies.

AWS IAM Access Analyzer

The Access Analyzer service has three main areas of functionality: policy generation, policy validation, and resource policy review.

The policy generation feature looks at principal activity in Amazon CloudTrail for a specific date range, and generates a policy based on the actions seen. This means you must already have a CloudTrail trail in your account for Access Analyzer to use. Being able to access this kind of functionality without having to setup your own infrastructure is a valuable feature, even if there are a few limitations with the generated policies For example, you can only generate one policy at a time, and the policy will not include data action-level permissions, even if those data events are present in your trail. Regardless, this is a free offering from AWS and it provides a good starting point for your least privilege policies that you can improve on.

You can use the policy validation functionality can via the web console or the APIs. It gives you feedback on your policies to ensure that they use correct policy grammar, as

well as follow common best practices. The official documentation has a full list of the checks used. From an IAM perspective, this functionality supersedes the older AWS Policy Generator, which gave you a web-based interface to generate policies for services such as IAM, S3, and VPC Endpoints.

The resource policy reporting checks the resource policies in your account, and returns findings on the policies that allow access from outside your zone of trust, which may be the account or the organization. This functionality is useful for reviewing your resource-level access, and identifying those with policies that can restricted or removed.

Don't confuse this service with AWS IAM Access Advisor, which appears as a tab for users and roles. Access Advisor shows the days since the principal used an AWS service, but doesn't show any of the specific actions called. This lack of detail makes it of limited use for working out least privilege access, and this combined with lack of updates to Access Advisor mean that you should almost always use Access Analyzer instead.

AWS Policy Simulator

The IAM Policy Simulator is another tool provided by AWS to review and test your policies. It has the benefit that it does not make any service calls, which means you can test policies without impacting your real resources. There are some limitations around what you can and can't simulate, such as SCPs not supporting global condition keys, but it's still a useful tool test your policies before you deploy them. Though not a part of the web console, you must still have a valid AWS session to use it.

Open Source Resources

Permissions.cloud

As mentioned elsewhere in the book, permissions.cloud is a community generated and maintained reference that is the easiest way to navigate the IAM actions, conditions, and resources namespace. Based on a variety of official sources, it takes care of cross-references between different services, so that you can find relationships between things like conditions and resources, and actions and conditions, quicker than if you had to use only the official documentation.

iamlive

The iamlive command line tool leverages the little-known Client Side Monitoring (CSM) functionality in the AWS CLI to generate policies based on your API calls to AWS. This is yet another great open source work by the genial Ian Mckay @iann0036.

When running locally during your development phase, all API calls made by the SDK get

recorded. `iamlive` uses these to generate and output a policy that includes all the actions and resources that were used. You can then edit this generated policy to work in different environments and accounts, with much less effort than writing the policy from scratch.

Cloudsplaining

Cloudsplaining specifically reviews your policies for least privilege violations, and privilege escalation attack vectors. It generates a nice-looking HTML report that also includes directions for remediation. Cloudsplaining is by the thoughtful Kinnaird McQuade @kmcquade3.

Cloudsplaining is easy to integrate in your development workflow by scanning specific policy files, or as a remediation tool by scanning all policies in an AWS account. This can be useful from a compliance and improvement perspective, as it provides documented proof of your review and refinement process.

Parliament

Parliament is a command line policy linter. Like linters for other languages, it checks that a policy is syntactically correct, as well as checking for common misconfigurations or security issues. Parliament is a tool by the prolific Scott Piper @0xdabbad00.

Though some of the functionality is now available in AWS IAM Access Analyzer, it was not when the tool was launched. Parliament can run locally, without an active AWS session. This makes it simple to integrate with your local development workflow, and reduce the time it takes to get feedback when writing policies.

A DynamoDB Example

In this example you will see how an policy evolves from the development, through testing, and to production. This policy was for a AWS Lambda function that at a high level needed to talk to an Amazon DynamoDB table to store and retrieve data as part of a serverless application.

Though this is a common pattern, the required functionality wasn't known in advance early development effort was focused on writing application code, not on locking down all the infrastructure. This means that while the resulting policy is simple, the policy still need to evolve over time.

In Development

Since it was known that the function would need to talk to Amazon DynamoDB, it was initially assigned the `AmazonDynamoDBFullAccess` AWS managed policy. This one-line

addition to the function's configuration made it easy to get started on the application, and the blast radius from the decision was limited to just the development environment, where many developers had administrator access already.

As development progressed and the function was only required to talk to a DynamoDB table, and not all the other services included in the AWS managed policy, it was a simple change to a customer managed policy that only granted access to DynamoDB.

DynamoDBAdmin.json

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "DynamoDBAdministrator",
            "Effect": "Allow",
            "Action": [
                "dynamodb:*"
            ],
            "Resource": "*"
        }
    ]
}
```

Note also that this was not the only policies used by the function's role, just the one that related to DynamoDB access.

In Testing

As development progresses, the function was deployed to a testing environment in a separate account. Since the function was part of a broader application, it didn't need to administer the DynamoDB resources, only access the data stored in them, so `Create*` permissions could be removed, as well as many of the permissions related to other services.

DynamoDBAccess.json

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "DynamoDBAdministerTable",
            "Effect": "Allow",
            "Action": [
                "dynamodb:BatchGet*",
                "dynamodb:DescribeStream",
                "dynamodb:DescribeTable",
                "dynamodb:Get*",
                "dynamodb:Query",
                "dynamodb:Scan",
                "dynamodb:BatchWrite*",
                "dynamodb:Delete*",
                "dynamodb:Update*",
                "dynamodb:PutItem"
            ],
            "Resource": "arn:aws:dynamodb:*:*:table/TABLE_NAME"
        }
    ]
}
```

This policy allows the function to access, change, and delete data from a specific. The wildcards in the account and region segments of the ARN make it easy to deploy the function to many accounts/locations, but the access granted is still scoped to a single table, specified by the value of `TABLE_NAME`. This greatly limits the potential blast radius of the function, so that it cannot impact other DynamoDB tables.

Once the application has been extensively tested, it's required actions and access patterns should be well understood. If possible, the final and most restrictive policy changes should be deployed independently of any code changes, making them easy to test and rollback in the event of any issues. In the function's final form, it only needed to put, get, and delete items in the DynamoDB table.

DynamoDBLeastPrivilege.json

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "dynamodb:DeleteItem",
        "dynamodb:GetItem",
        "dynamodb:PutItem"
      ],
      "Resource": [
        "arn:aws:dynamodb:REGION:ACCOUNT_ID:table/TABLE_NAME"
      ]
    }
  ]
}
```

Though a simple result, the iterative process by iterating and performing policy changes independently of code changes as the application's requirements mature and stabilize, least privilege policy development doesn't have to interrupt or slow down development.

In Production

Since the resource scope of the final policy includes a specific DynamoDB table ARN, the deployment of the application to production will need to set this via variables in the infrastructure as code. A case could be made to use wildcards in the account and region segments of the DynamoDB table ARN, which would make the policy reusable between account and regions without change.

Multi-Tenant Accounts

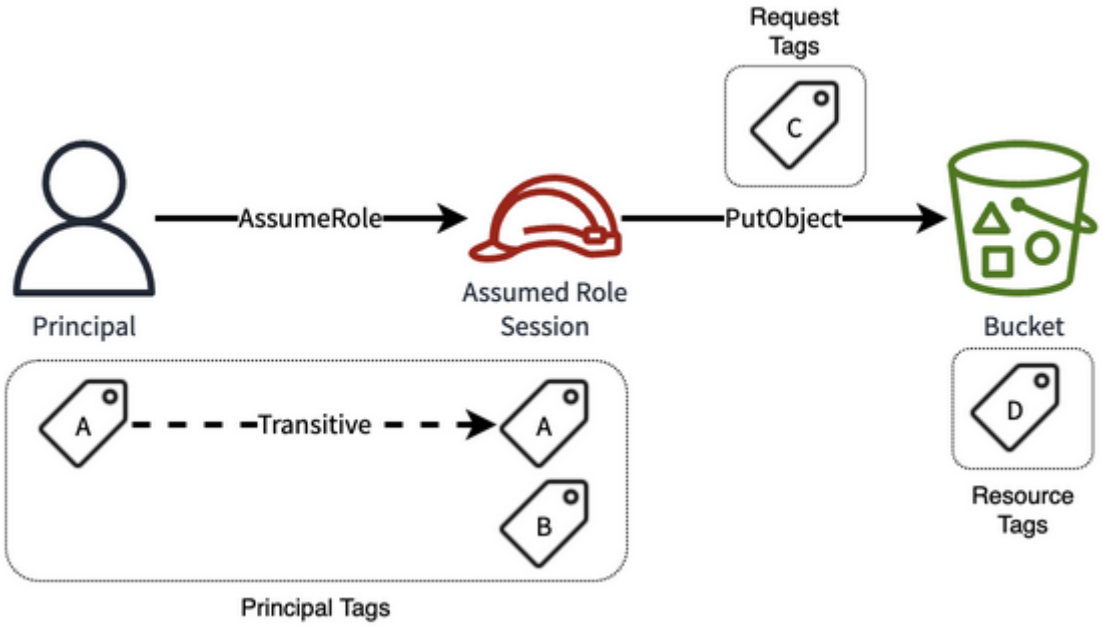
In scenarios where you have different people, teams, or projects operating in the same AWS account, using a role-based approach may not give you the level of granularity you need to implement your authorization. Roles that reference specific resources can become difficult to maintain, and be too large for policy quotas at scale. For this situation, IAM provides you the ability to use attributes to give your fine-grained control over the permissions and resources your principals can access. This approach is referred to as attribute-based access control (ABAC), and is an advanced approach to authorization in IAM.

Unfortunately, not all AWS services support ABAC for authorization. For each service you are interested in, check the official documentation for AWS services that work with IAM and look at the "Authorization based on tags" column for the services you are interested in.

Though ABAC focuses on attributes to determine access, it still uses roles to implement access. Given some of the limitations with ABAC on AWS, it's most likely that you will use it to compliment your role-based access control (RBAC) and multi-account strategy, rather than replace it. By combining both approaches to authorization, you can tighten your control, improve your security, and reduce your operational and administrative overhead at the same time.

Attributes and Tags

As the name suggests, ABAC relies on the *attributes* of the principal, resource, and request, to determine if the request should be allowed or denied. Attributes in AWS are implemented via *tags*, which are key-value pair that can be attached to most AWS resources. Both the key and value of a tag must be set, but the value can be set to a null value, represented by an empty string.



Though IAM relies on tags as attributes, they are not part of IAM. Historically, tags were only used for cost allocation and reporting. This made the management of tags useful and important in your environment, but not a security risk. Even the services that do have the appropriate tagging API methods use inconsistent names, for example deleting a tag might use `Untag` or `RemoveTag` method names. All of this makes managing tags at scale a challenge.

What tags keys do you need? Is it `team-name`, `cost-center`, `project-name`, `division`, `owner`, or something else specific to your company? Though there are some common examples like these, the specific keys and values can vary greatly between AWS environments.

Since ABAC relies completely on tags, anyone with the ability to tag a resource has the ability to change the effective permissions associated. This means that the ability to set tags is now a security concern. Users in your environment can escalate their privileges if they can set their tag values on themselves or resources, even if they don't have access to IAM actions.

Challenges Using Tags

Unfortunately, tag management is not consistent between AWS services. This also means that each AWS service team must implement their own functionality to set, update, and delete tags, and unfortunately not all have, or have done so consistently. Not all resources support tags, and though the list of resources that can't be tagged is getting shrinking, it could be smaller still.

For the resources that can be tagged, not all of them support being used for ABAC. These tags are only good for cost allocation. For some services, you can tag a resource when you create or update it, but others have specific API methods to tag resources. And others only let you set tags on creation of resources, and cannot be updated. For example, IAM lets you set tags on a role when you create it with the `CreateRole`, but not when updating the role with `UpdateRole`; to update the existing role you will need to call the `TagRole` method. The tag keys on some resources are case-sensitive, and for other resources they are not. This means that while one resource will allow you to set two distinct tags with the keys `Environment` and `environment`, but on other resources you will only have one tag.

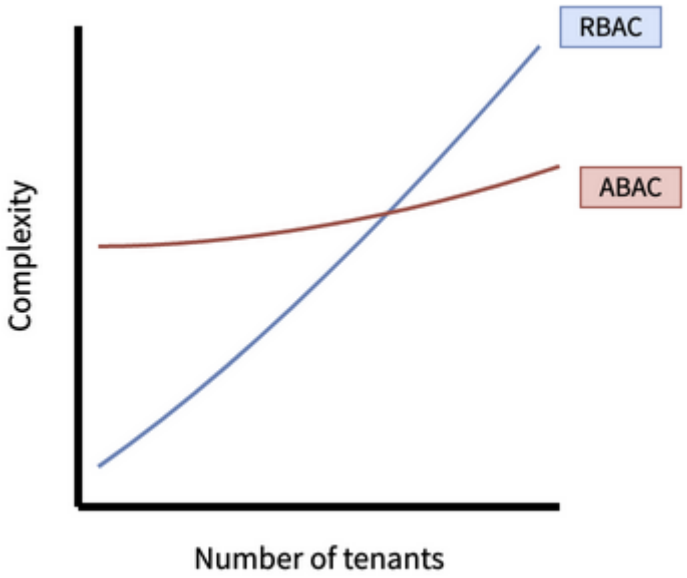
If all this seems confusing, it's because it is! Managing tags is complicated, and becomes an important task when using ABAC. As with many security-related decisions, the ideal situation is to avoid complexity for as long as possible. In practical terms, this means starting with RBAC, and only considering ABAC when you have reached the limits of RBAC.

ABAC and RBAC

One of the advantages of ABAC over RBAC is that changes only impact the principals, resources, and requests related to the change. This means that something as common as deploying a new resource doesn't require any principals to be modified. Compare this to RBAC, where new resources and ARNs can require a permissions change for an existing

principal to access them.

ABAC should also result in more granular, smaller policies in your environment. With ABAC, often just a single role is used, but with access customized based on their attributes. The trade-off is that those policies can be combined in many different ways, which can make reasoning about the resulting access harder.



This simple graph shows the utility of both approaches for more complex environments. ABAC is a valid approach in more complicated environments, not because it removes complexity, but because it makes complexity easier to manage. The complexity in an ABAC approach to authorization is brought forward to the planning setup phase, which makes the up-front planning of ABAC much more important. When you have combinations of access that can vary from principal to principal, or change frequently, then you should consider an ABAC approach.

ABAC supports also supports cumulative permissions by adding extra attributes to principals. This features means that ABAC supports scenarios where the required access pattern doesn't fit neatly in to the abstract idea of a role, compared to RBAC where roles are changeable but a can only be used one at a time. To control access to specific resources using RBAC, you will need to craft specific ARNs or a suitably specific pattern. At a certain scale this becomes impractical as changes to your resources require frequent updates to your policies, and policies approach the character limits of IAM.

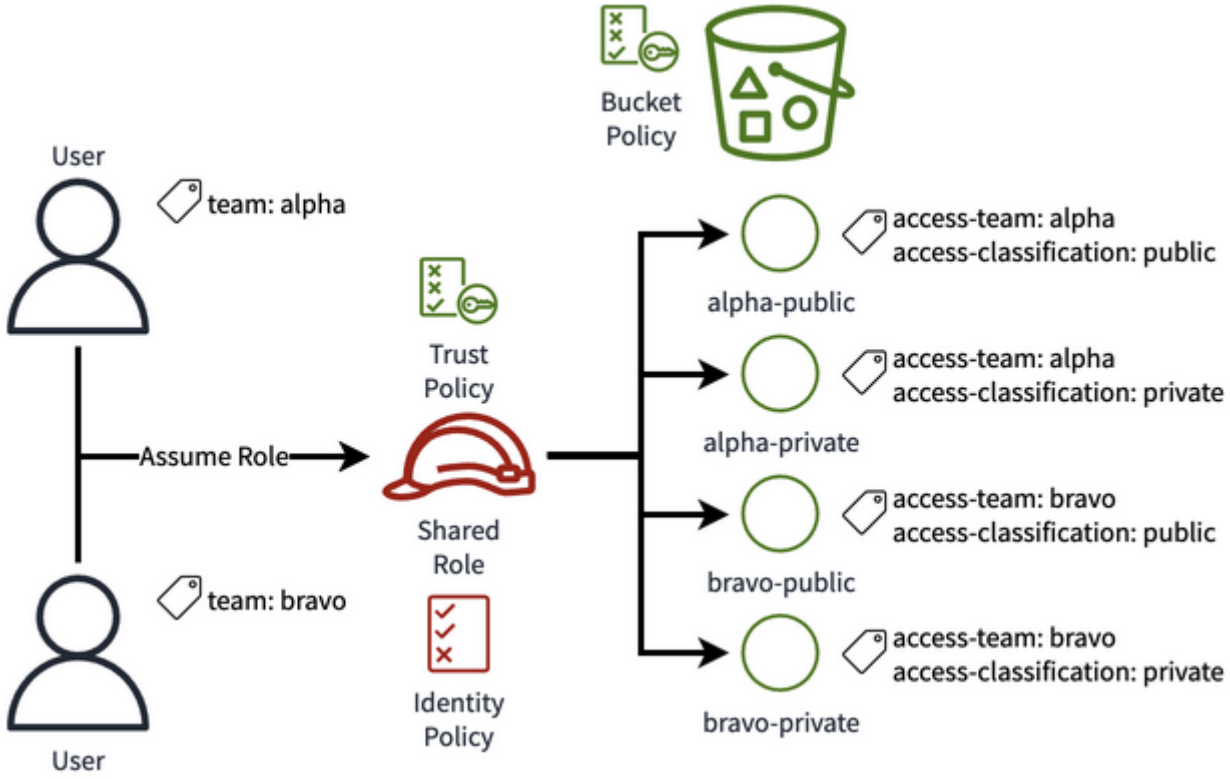
Data Security in S3

In this scenario you will see how to implement ABAC policies that allow principals to collaborate, without negatively impacting each other.

Managing and securing data at scale is an increasing challenge for AWS customers.

Amazon S3 in particular provides nearly limitless object storage, with an high level of durability. Using S3 means you probably don't need to worry about losing your data, and its price means you rarely have to worry about the cost of it either. For environments with sensitive data, the next challenge is how you secure that data.

This scenario fits with the strengths of ABAC, due to the volume of object resources involved. Changing your principals to restrict their access because some new objects get uploaded to your buckets is almost immediately impractical in the real world.



Authorization based on tags is only supported by S3 objects, and not on S3 buckets. By tagging objects with the team that owns them and their level of classification, you can provide principals access to the data they need to do their job, without providing too much access.

To model this, the following requirements need to be met.

- Developers must have a `team` tag
- Objects must have an `access-team` tag
- Objects must have an `access-classification` tag, with valid values "public" or "private"
- Developers can read and write objects that their team owns
- Developers can read objects owned by other teams, but not objects with a classification of "private"

- Developers cannot write objects other teams' objects

Though this scenario only includes the teams "alpha" and "bravo" other team values are valid. With these tags in mind, we can model the access patterns we want to see in a table for easy reference.

Table 1. Object Access Table

Principal Tags	Object Tags	Access Granted
team: alpha	team: alpha	Read, Write
access-team: alpha	access-team: bravo, access-classification: public	Read
access-team: alpha	access-team: bravo, access-classification: private	None

In this example, the tags on the objects include the prefix `access-` which makes it clear that the tags are for access management, and shouldn't be changed without reason. This also gives the potential later on for control on who can modify tags related to access management, by checking the prefix of the tag keys.

Resource Policy

S3 objects don't support resource policies, so the policy is attached to the bucket, though the policy statements apply to the objects in the bucket. All of the conditions use string operators, since tag values are always considered strings by IAM.

The S3 service uses service-specific tags to enforce ABAC. These tag keys start with the `s3:` prefix, for example `s3:ExistingObjectTag`. More recent services will use the AWS global condition condition keys that start with `aws:` such as `aws:ResourceTag`.

You can create and attach this bucket policy document to the bucket either via the console, or using the CLI as described in the cross-account S3 bucket sharing scenario. Due to the length and complexity of the bucket policy, we will review it as individual policy statements, even though they are all part of the same policy document. The full policy is available in the appendix.

Allow Team Read

```
{
  "Sid": "AllowTeamRead",
  "Effect": "Allow",
  "Principal": {
    "AWS": "arn:aws:iam::ACCOUNT_ID:role/ROLE_NAME"
  },
  "Action": [
    "s3:GetObject",
    "s3:GetObjectTagging"
  ],
  "Resource": "arn:aws:s3:::BUCKET_NAME/*",
  "Condition": {
    "StringEquals": {
      "s3:ExistingObjectTag/access-team": "${aws:PrincipalTag/team}"
    }
  }
}
```

This checks that the current value of the `access-team` tag of the object matches the value of the `team` tag of the principal that is making the requesting the object. If these values don't match then the principal will not be granted the permission to get objects and their tags. This statement meets the requirement to allow teams to read their own objects, regardless of classification.

Allow Team Write

```
{
  "Sid": "AllowTeamWrite",
  "Effect": "Allow",
  "Principal": {
    "AWS": "arn:aws:iam::ACCOUNT_ID:role/ROLE_NAME"
  },
  "Action": [
    "s3:PutObject",
    "s3:PutObjectTagging"
  ],
  "Resource": "arn:aws:s3:::BUCKET_NAME/*",
  "Condition": {
    "StringEquals": {
      "s3:RequestObjectTag/access-team": "${aws:PrincipalTag/team}"
    }
  }
}
```

This allows the shared role used by principals the ability to put objects and their tags, as long as the object in the put object request has the same `access-team` tag value as the principal's own `team` tag. Any other `access-team` value, including a blank value, will fail the condition and the principal will not have permission to put objects in the bucket.

For brevity and focus, the statements in this policy have focused on the `PutObject` and `PutObjectTagging` permissions. In many scenarios, you would want to give principals additional write permissions such as `DeleteObject`.

This statement meets the requirement to allow teams to write their own objects, regardless of classification.

Allow Public Read

```
{
  "Sid": "AllowPublicRead",
  "Effect": "Allow",
  "Principal": {
    "AWS": "arn:aws:iam::ACCOUNT_ID:role/ROLE_NAME"
  },
  "Action": [
    "s3:GetObject",
    "s3:GetObjectTagging"
  ],
  "Resource": "arn:aws:s3:::BUCKET_NAME/*",
  "Condition": {
    "StringEquals": {
      "s3:ExistingObjectTag/access-classification": "public"
    }
  }
}
```

This condition grants permissions to objects with the `access-classification` value of "public". This statement meets the requirement to allow public objects to be read, regardless of their team.

Deny Team Write

```
{
  "Sid": "DenyTeamWrite",
  "Effect": "Deny",
  "Principal": "*",
  "Action": [
    "s3:PutObject",
    "s3:PutObjectTagging"
  ],
  "Resource": "arn:aws:s3:::BUCKET_NAME/*",
  "Condition": {
    "StringNotEquals": {
      "s3:RequestObjectTag/access-team": "${aws:PrincipalTag/team}"
    }
  }
}
```

The first deny statement in the bucket policy prevents principals from putting objects or object tags if they don't also include an `access-team` with a value that matches their own `team` value. Since explicit deny statements override any allow statements, this ensures that the consistency and integrity of object tags, even by principals that already have put object permissions.

For added security, this and the other deny statements apply to all principals, not just the role that previous statements have been granting permissions to. This statement also controls access to object tagging actions, which means that principals can't change an object's tags to get around the controls. This statement meets the requirement to prevent objects from being written by other teams.

Deny Private Read

```
{
  "Sid": "DenyPrivateRead",
  "Effect": "Deny",
  "Principal": "*",
  "Action": [
    "s3:GetObject",
    "s3:GetObjectTagging"
  ],
  "Resource": "arn:aws:s3:::BUCKET_NAME/*",
  "Condition": {
    "StringEquals": {
      "s3:ExistingObjectTag/access-classification": "private"
    },
    "StringNotEquals": {
      "s3:ExistingObjectTag/access-team": "${aws:PrincipalTag/team}"
    }
  }
}
```

The statement above checks multiple conditions, which means that all the conditions operators must be true for the statement to affect the request, in this case granting the permission to get read private objects. Since multiple operators in a condition, and also keys in an operator block, get evaluated using a logical AND operand, the ordering of the condition operators does not matter. This statement meets the requirement to prevent private classified objects from being read by other teams.

Deny Classification Invalid

```
    {
      "Sid": "DenyClassificationInvalid",
      "Effect": "Deny",
      "Principal": "*",
      "Action": [
        "s3:PutObject",
        "s3:PutObjectTagging"
      ],
      "Resource": "arn:aws:s3:::BUCKET_NAME/*",
      "Condition": {
        "StringNotEquals": {
          "s3:RequestObjectTag/access-classification": [
            "public",
            "private"
          ]
        }
      }
    }
```

For ABAC to work as intended, the right tag keys must be present and have valid values. This denies the ability put object or tags to all principals, which includes the role we have granted access to in earlier statements.

Inside a condition key, multiple values are matched using the logical `OR` operand, which means the condition will be met when one of the values specified. If the value of the `access-classification` tag does not equal one of the values, or is blank, then the request will be denied. This statement ensures that you don't end up with objects that have invalid values for the `access-classification` tag key, so you can be confident when checking the values in earlier statements.

Identity Policy

In this scenario, and identity policy for the shared role used to access objects in the bucket is optional. Since the role is in the same zone of trust as the bucket, the bucket's own resource policy is able to grant all of the relevant permissions to meet the requirements. This is ideal because it means that the bucket is the source of truth for its own access. Even if a principal has a more permissive identity policy, for example the AWS managed

policy `AmazonS3FullAccess`, the deny statements in the policy will restrict the access to the appropriate level.

If the principal was outside of the bucket's zone of trust, then it would need an identity policy in addition to the bucket policy listed here.

Summary

These policy statements, when combined and applied as a bucket policy, grant least privilege, multi-tenant access to objects in the bucket based on their attributes. The allow permissions statements apply to a specific principal, limiting the scope of permissions granted. The deny permissions statements apply to all principals, ensuring the security and integrity of the objects in the bucket.

All of the deny conditions are deliberately separate from other deny statements in the policy. Since multiple conditions are evaluated using a logical `AND` operand, if you combine conditions in the same deny statement, all of them must be true for the request to be denied.

ABAC for Other Services

More recent services use standard conditions that don't reference the resource types specifically in their name. This makes implementing ABAC simpler, because your conditions are applicable to more than one service, further simplifying and reducing the amount of policies you need.

Compared to the policies in this scenario, the following condition keys are used in newer services:

- Use `aws:ResourceTag` in place of `s3:ExistingObjectTag` to check what tags a resource has
- Use `aws:RequestTag` in place of `s3:RequestObjectTag` to check the in-flight tag values of a request
- Use `aws:TagKeys` in place of `s3:RequestObjectTagKeys` to check the in-flight tag keys of a request

As always, the SAR aws page in the official AWS documentation is the canonical reference for which services and actions support the standardized conditions, or their own service-specific conditions.

When to use ABAC

Given the strengths and weaknesses of ABAC, you should consider the following checklist when you are thinking about using it in your scenarios.

- ☐ You have complex, fine-grained authorization requirements that are well-defined
- ☐ Different levels of access are required for resources, for example reading and writing
- ☐ A principal's level of access can change, for example people moving between project teams, or people using different levels of access for different tasks
- ☐ All the resources that you need to control access to support tagging and authorization based on tags
- ☐ Your principals have attributes that determine their level of authorization
- ☐ Resources have tags that are consistent and reliable
- ☐ Your policies are reaching the character limits aws
- ☐ You cannot achieve your requirements with RBAC

If most of these points apply to your situation, or you can do the work to achieve them, then you should use ABAC. If not, then you should use RBAC for your authorization requirements.

Cleaning an Existing Environment

This scenario details the parts of IAM and related services that you use to secure an existing AWS environment. Over time, changes and growth in an AWS environment can lead to complexity and blind spots that reduce your security. A new and clean AWS environment is not only pleasing to the eye and budget, it is also more likely to be secure.

Unfortunately, throwing out your existing environment and starting again from scratch is impractical, and unlikely to be a legitimate option. This problem is not helped by AWS growing and improving their service offering and features over time, so some things that used to be considered good practice are now the opposite. Because of this, it is important to be able to take control of an existing AWS environment and secure it.

This scenario covers more AWS services than just IAM, but it still focuses on the features and aspects of those services that relate directly to IAM. For other aspects to securing your data and applications on AWS, see the Next Steps section at the end of this scenario for considerations that are beyond the scope of this book, such as networking and monitoring. A step-by-step checklist is included at the end of the scenario for you to follow.

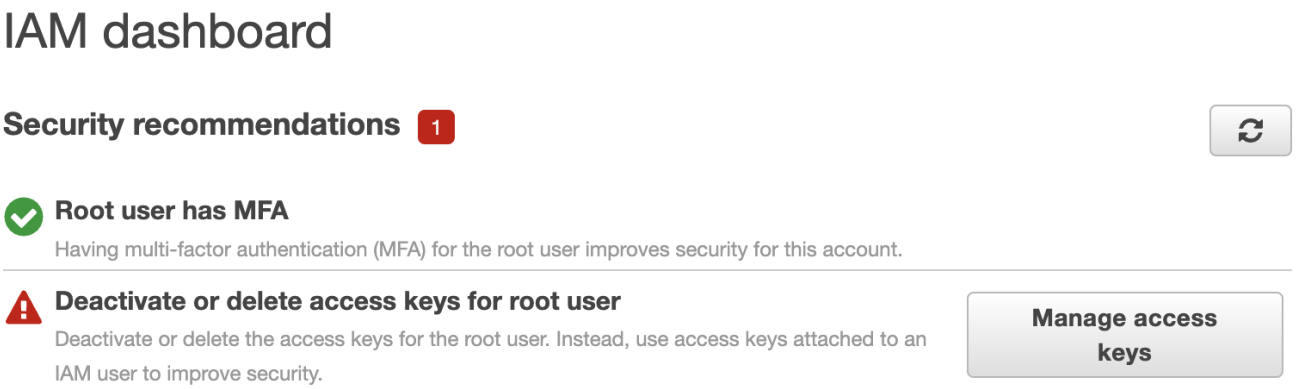
Emails, Emails, Everywhere

For better or worse an email address is still the starting point for all your AWS accounts, even those created via AWS Organizations. The first step in gaining control of your environment is to know and control the email addresses used to create your AWS accounts.

If you don't control the email address of an account, then you don't really control the AWS account. Don't laugh! I have seen personal email addresses used for accounts hosting company resources more than once.

All email addresses used for AWS accounts must be globally unique across all of AWS. Ideally the email domain you use will be the same for all of your accounts, to ensure control and consistency. You can change the email address for an AWS account by following this knowledge center article. Having addresses that are human readable is fine, but should not be a priority when deciding what email addresses to use as humans should not be using login details often. It is better to have your addresses follow a pattern that is consistent and unique, rather than human-friendly.

Once you have identified all of the email addresses, you should reset the root user password for all your accounts to ensure they are not compromised. At the same time, you should deactivate and delete any root user access keys that exist.



The IAM web console when the root user has access keys

If you are worried the access key might be in use, you can deactivate the keys for a period of time before deleting them. This means you can quickly reenable them if needed, without needing to change them them. You should aim to delete any root user access keys as soon as possible.

Store root account passwords in a password manager if you have plans to use them soon. If you don't need to use the root account anytime soon, then you should consider following AWS Organizations' own practice: the root account password is set to a

random, complex, 64 character password, and then thrown away. If you need to access the root user account you follow the standard reset password process, which relies on the email address associated with the account.

Accounts and organizations

When cleaning up existing AWS environments, there are two main scenarios you will come across: not enough accounts, or too many accounts.

Not having enough accounts usually means that resources are provisioned in accounts with other unrelated environments or applications. This makes it harder to manage from both an IAM and cost perspective, resulting in complex environments that can be difficult to troubleshoot. Having unrelated resources in the same account means that the blast radius of any changes is larger than needed.

Having one AWS account is an unfortunately common scenario. This happens in environments that predate the AWS Organizations service, or where the owners have not kept up to date with multi-account recommendations from AWS, such as in the multiple accounts whitepaper.

In the too many accounts scenario, an environment can suffer from *account sprawl*. This is where the number and variety of AWS accounts makes them hard to manage and keep updated. Account sprawl can happen because businesses and their AWS accounts were acquired in to an organization, or because there was no oversight or ownership for the account creation process.

Regardless of you scenario or reasons behind it, you should use AWS Organizations. If you've got too few accounts, you'll need to create more. If you've got lots of accounts, you'll need to organize them. Once you've done that, you will need to clean up and close some existing accounts. If AWS Organizations is already enabled, make sure it has all features enabled so that you can use SCPs.

Organizations have one *management account* (previously known as the "root account"), and multiple *member accounts* (previously known as "child accounts"). If your environment needs to enable AWS Organizations for the first time, you should do it in a new AWS account. This will ensure that the management account is clean and secure with no unnecessary resources or access provisioned to it.

Invite any accounts that are not part of the organization to be members, using the root account email and password that you recovered in previous steps. AWS has some simple but effective best practices for member accounts in the official AWS Organizations documentation, like using complex passwords and setting the appropriate contact details.

At this stage the important thing is to bring your accounts into your organization, and so you can use SCPs in the future. For more details on managing multiple AWS accounts, see the Using Multiple AWS Accounts scenario.

Secure the Management Root Account

The root user account of your organization management account is the "keys to the castle" of your AWS environment, and if compromised then all your other security efforts are in vain. The root account for the Organizations management account should have **hardware** multi-factor authentication (MFA) configured and secured.

The AWS Organizations user guide spells-out a number of steps to take when securing your organization management account.

Taking the time to do this properly is time well spent, given the blast radius for this account is your entire AWS environment.

Disable Member Root Accounts

For member accounts in your organization, setting up hardware or software MFA can become impractical with many accounts. Even AWS Organizations does not configure MFA for root accounts in the AWS accounts it creates.

Instead, you can deploy *mitigating controls* to remove the risk of your member accounts' root accounts being compromised. In this case the control is to disable the root user account for your member accounts, so that even if the email address and password are compromised they cannot be used.

The following SCP, applied to the root of your organization, will disable all actions for the root account principal.

DisableRootAccount.json

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Action": "*",
      "Resource": "*",
      "Effect": "Deny",
      "Condition": {
        "StringLike": {
          "aws:PrincipalArn": [
            "arn:aws:iam::*:root"
          ]
        }
      }
    }
  ]
}
```

SCPs do not apply to the organization's management account or the root account for that user, even if you attach the SCP to the root of your organization. This is why you need to configure a MFA device to protect the management account's root user specifically.

This policy will need to be disabled for an account if you need to do any tasks that require root user credentials, but those tasks are uncommon and should only be done by exception.

No More Users

You should aim to have no IAM users in your environment. While they were required in the past, the AWS service offering has expanded so that this is no longer the case. The long-term credentials associated with users (passwords and access keys) are a security risk that is no longer required to use AWS. Instead, you should use identity federation to give your human users short-term credentials to access AWS.

As a priority, you should run the IAM credential report in any accounts that have IAM users. Disable any credentials that have not been used in more than 30 days. Disabled credentials can be quickly re-enabled if required, while limiting the risk associated with long-term credentials. The credential report also reports on root user credentials, which was hopefully addressed in previous steps.

While you might not be able to get rid of all IAM users, you should question each and every user in your environment. There are some situations where programmatic access is required, and IAM users are a way to achieve that. You should periodically review your

users, and challenge their continued use. In the case of users that are required for programmatic access, you should follow the principal of granting least privilege. See the least privilege scenario for an example of this.

Identity Federation

To fully remove IAM users from your environment you will need to setup *identity federation*. This requires you to create a trust relationship between your accounts, and a SAML-compliant IdP.

By configuring a SAML provider resource in your AWS accounts, you allow IAM roles to be assumed by users from the trusted IdP. These resources can be configured directly in your accounts, or you can use the AWS Identity Center to do this for your entire organization. Identity Center can also be used as an IdP, and though this is better than IAM users in your environment, you probably will still want a third-party/dedicated IdP for larger environments.

Configuring your IdP of choice is beyond the scope of this book. The most common third-party IdP options are Azure Active Directory, Google Workspace, Okta, OneLogin, and Ping Identity. All of these IdPs have documentation for how to configure them to work with AWS. See the Identity Federation in AWS page for more information.

Bunker Down

Once you have secured your access to your environment, you should be able to audit it. For IAM this means ensuring that AWS CloudTrail is enabled in all of your accounts. CloudTrail is the auditing service from AWS, and it reports on the actions your principals are performing in your environment. You should enable CloudTrail for your entire organization. Being able to use organization-wide services like this is another reason to use AWS Organizations with all features enabled. Be aware that there is a cost associated with delivering and storing your CloudTrail logs.

CloudTrail audit logs should be stored in a centralized, shared, S3 bucket. You can follow the steps in Sharing a bucket with an organization to share the bucket so that accounts in your organization can write to the bucket, but not read, update, or delete its contents. This means you can use it for the CloudTrail trails in all of your accounts to store their logs securely. Ideally this bucket should be in a separate and dedicated account, so that the audit trail for an account is outside of its own blast radius. This means that if a single account is compromised you will still have an audit of actions in that account.

Enabling CloudTrail should be the absolute minimum that you use in your AWS environment. There are other security services from AWS that you should strongly consider using, such as Amazon GuardDuty for threat detection on your resources,

and AWS Security Hub for cloud security posture management.

Close Old Accounts

At this stage you have control of your access in to your AWS environment, and are able to audit actions via CloudTrail. At this stage you should look for opportunities to close any AWS accounts that are not required.

You may still be charged for some AWS resource costs after you have closed an account, but before the account is destroyed. This means you should stop, terminate, or delete as many resources as you can *before* closing an account.

For all accounts, you can follow the same steps to close an AWS account. For accounts in an organization, there is now an API to close accounts.

Check Your Bill

The AWS billing statement reflects what AWS knows has been run in your accounts. Even resources that you only run for a few milliseconds will show up on your AWS bill at the end of the month. Reviewing your bill is quickest way to find charges for unexpected services or region. The billing statement is usually updated a few times per day, and the exact numbers and details can change until the end of the month. This means you should not rely on the bill as a real-time resource, but review it periodically to confirm your expectations.

Account Security Cleaning Checklist

Activities that might interrupt applications or humans in your environment have marked "disruptive".

- [ ] Review all root account emails, and ensure you control them
- [ ] Reset root user credentials, and disable and delete all root user access keys (disruptive)
- [ ] Setup AWS Organizations with all features
- [ ] Integrate you identity provider, using AWS Identity Center across the organization
- [ ] Invite all accounts to the organization
- [ ] Disable or rotate access keys older than 30 days (disruptive)
- [ ] Review all IAM Users
- [ ] Delete as many IAM Users as possible (disruptive)
- [ ] Review and update Support levels in each account
- [ ] Disable the root user account in member accounts using SCPs
- [ ] Review trust relationships with IAM Access Analyzer
- [ ] Create a centralized logging bucket
- [ ] Setup multi-region CloudTrail trails to log to a centralized bucket
- [ ] Close old accounts (disruptive)
- [ ] Review your bill for unexpected services or regions

Next Steps

Once you've followed these steps, you control the identity perimeter of your AWS environment. From this point you should consider your other infrastructure. All of these activities are aimed at simplifying your environment, and reducing the attack surface of your AWS environment.

- **Network perimeter** What is externally accessible?
- **DNS domains** Can any of my domains be transferred away?
- **Externally accessible data** What can be accessed from the public internet? For example S3 buckets, EBS snapshots, and shared AMIs.
- **Monitoring tools** What has visibility in to my environment, and how much visibility do they have?

Fixing these aspects of your environment things will take longer, and be more disruptive. In some cases, you will have to change how your applications run. Unfortunately changing applications positively from a security point of view can have negative impacts from an availability, cost, or performance perspective, so must consider them carefully.

[3] https://docs.aws.amazon.com/whitepapers/latest/organizing-your-aws-environment/organizing-your-aws-environment.html

[4] https://stackoverflow.com/a/67636523

[5] https://en.wikipedia.org/wiki/Timeline_of_Amazon_Web_Services

[6] If that sentence doesn't make a lot of sense, great! You almost certainly don't need ACLs.

Acknowledgments

Even though they call it "self publishing" a book, it's a shocking amount of work, and definitely not something I could ever do on my own.

Thanks to Ian Mckay for writing the foreword, and the other AWS Ambassadors for their support and good-natured (and not undeserved) ribbing.

Thank you Craig Liebendorfer for your unsolicited edits on the early version of the book. You made it a much more professional book, and I learnt a lot about writing from your feedback.

A very incomplete thank you to all the draft readers who took the time to provide feedback, such as Matthew Daniel, Kells Kearney, Chris Scott, Vidit Shah, and Brian Snyder. Thanks for making the book better, and my apologies to those who provided feedback who I missed.

Thanks to Alex De Brie and his DynamoDB Book, for setting a great example (and writing such an amazing resource!), and also his personal encouragement and help wrangling Asciidoctor along the way.

Thanks to my partner M.P. and my kids for supporting me during the long (much longer than expected) writing process, even after I said there was no way I was going to write another book about AWS - You're the reason I do any of this.

Rowan Udell, February 2023

Glossary

ABAC

Attribute-based access control.

API

Application Programming Interface

Array

A data structure for a group of elements. Sometimes also referred to as a **List**.

ARN

Amazon Resource Names.

BAU

Business As Usual. Something standard or expected, which likely happens most or all days.

CDN

Content Delivery Network. A service comprised of geographically distributed servers that make it quick to distribute data to end users.

Cloud Native

An approach to running applications that takes advantage of cloud-based services *as much as possible*.

Console

The AWS web console.

Delegate

Authorize another entity to perform a function.

Entity

The original source of a request for an action or operation on AWS.

Escalation of Privileges

A type of attack where an entity with access to your environment gains more privileges than originally intended.

Eventually Consistent

A model of distributed computing that prioritizes **availability over correctness**. It means that request made aren't guaranteed to reflect the most recent change, but will over time.

Explicit

Something that is stated clearly and leaves no room for confusion or doubt. Used in contrast to **Implicit**.

IdP

Identity Provider.

Implicit

Suggested but not directly expressed. Used in contrast to **Explicit**. List: See **Array**.

OAI

Origin Access Identity. A special Amazon CloudFront aws user that can be used to restrict access to a S3 origin.

Partition

A group of AWS regions and services. Used in Amazon Resource Names to identify how they are managed. The most common partition is `aws` which represents the standard, commercial regions. Other partitions are `aws-cn` for China regions, and `aws-us-gov`, `aws-iso`, and `aws-iso-b` for US government regions.

Principal

A person or application that makes requests for an action or oration on a resource.

SAR

The Service Authentication Reference is the official list of the actions, resources, and condition keys that are supported by each AWS service. Not to be confused with the AWS Serverless Application Repository icon:aws@fab

SaaS

Software-as-a-Service.

Segment

Amazon Resource Names are made of up segments, which are separated by `:`.

TAM

Technical Account Manager, which are made available to customers that have AWS

Enterprise Support aws.

UUID

Universal Unique Identifier. A 128-bit label that are for all practically purposes unique, but are not centrally managed or verified.

Wildcards

The * and ? characters in the that represent zero or more characters, or a single character respectively.

Zone of Trust

An **implicit** trust relationship between resources and principals.

Appendix A: ABAC Bucket Policy

BucketPolicy.json

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AllowTeamRead",
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::ACCOUNT_ID:role/ROLE_NAME"
      },
      "Action": [
        "s3:GetObject",
        "s3:GetObjectTagging"
      ],
      "Resource": "arn:aws:s3:::BUCKET_NAME/*",
      "Condition": {
        "StringEquals": {
          "s3:ExistingObjectTag/access-team": "${aws:PrincipalTag/team}"
        }
      }
    },
    {
      "Sid": "AllowTeamWrite",
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::ACCOUNT_ID:role/ROLE_NAME"
      },
      "Action": [
        "s3:PutObject",
        "s3:PutObjectTagging"
      ],
      "Resource": "arn:aws:s3:::BUCKET_NAME/*",
      "Condition": {
        "StringEquals": {
          "s3:RequestObjectTag/access-team": "${aws:PrincipalTag/team}"
        }
      }
    },
    {
      "Sid": "AllowReadPublicRead",
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::ACCOUNT_ID:role/ROLE_NAME"
      },
      "Action": [
        "s3:GetObject",
        "s3:GetObjectTagging"
      ],
      "Resource": "arn:aws:s3:::BUCKET_NAME/*",
      "Condition": {
        "StringEquals": {
          "s3:ExistingObjectTag/access-classification": "public"
```

```
        }
      }
    },
    {
      "Sid": "DenyTeamWrite",
      "Effect": "Deny",
      "Principal": "*",
      "Action": [
        "s3:PutObject",
        "s3:PutObjectTagging"
      ],
      "Resource": "arn:aws:s3:::BUCKET_NAME/*",
      "Condition": {
        "StringNotEquals": {
          "s3:RequestObjectTag/access-team": "${aws:PrincipalTag/team}"
        }
      }
    },
    {
      "Sid": "DenyPrivateRead",
      "Effect": "Deny",
      "Principal": "*",
      "Action": [
        "s3:GetObject",
        "s3:GetObjectTagging"
      ],
      "Resource": "arn:aws:s3:::BUCKET_NAME/*",
      "Condition": {
        "StringEquals": {
          "s3:ExistingObjectTag/access-classification": "private"
        },
        "StringNotEquals": {
          "s3:ExistingObjectTag/access-team": "${aws:PrincipalTag/team}"
        }
      }
    },
    {
      "Sid": "DenyClassificationInvalid",
      "Effect": "Deny",
      "Principal": "*",
      "Action": [
        "s3:PutObject",
        "s3:PutObjectTagging"
      ],
      "Resource": "arn:aws:s3:::BUCKET_NAME/*",
      "Condition": {
        "StringNotEquals": {
          "s3:RequestObjectTag/access-classification": [
            "public",
            "private"
          ]
        }
      }
    }
  ]
}
```