Effective IAM for AWS

Secure AWS with IAM built for continuous delivery

A practitioner's guide by
Stephen Kuenzli

Effective IAM for AWS

Secure AWS with IAM built for continuous delivery

A practitioner's guide by
Stephen Kuenzli

November 2021

For Grace Kuenzli

Table of Contents

About this guide

Effective IAM with Amazon Web Services is for engineers with practical responsibility for securing AWS cloud deployments. These practitioners *actually* create and review security policies for applications and infrastructure. They often define security architecture, sometimes unknowingly. They usually have titles like Cloud, DevOps, Site Reliability, and Cloud Security Engineer or Architect. Whatever their titles, these people feel the weight of responsibility to secure AWS deployments and the pain of AWS IAM's complexity.

And it's heavy.

'Identity' is the system you use to authenticate users and authorize them access to the AWS services and data needed to do their job. Engineers manage access to Cloud resources using the AWS Identity and Access Management service, AWS IAM.

But effective identity management is hard and getting harder.

AWS identity security is complex, the number of IAM identities are increasing, and Cloud deployments change quickly.

If you struggle to deliver effective AWS security policies or you find yourself staring at an incoherent mess of security policies in many accounts, this book is for you.

This book will help you understand why AWS IAM is hard and how to leverage IAM's best features to secure apps & data continuously. It will help you design, develop, review, and *deliver* better AWS security policies, quickly and confidently.

What this guide is and what you will learn

This guide describes a practical strategy and tactics to simplify the incredibly powerful and complex AWS IAM service into something your whole organization can use safely.

You will learn how to:

- **solve** difficult security problems using the best parts of AWS IAM
- **simplify** AWS IAM into a set of secure infrastructure code building blocks to deliver changes quickly
- **verify** AWS IAM security policies protect resources as intended
- **secure** IAM continuously at any scale

What you need to know

Effective IAM is written for people who:

- use Amazon Web Services
- develop or review IAM security policies frequently
- must protect data and implement access controls, particularly 'least privilege'
- use infrastructure code tools and libraries to automate configuration

These activities are painful in many organizations and this book will help you get the job done. If you don't perform these activities directly, you will learn how to work with those who do or even decide to join them.

What this guide will not cover

This guide does not cover all of AWS Security or even all of IAM. That scope is entirely too large for a practitioner's guide and dulls the focus needed to build effective IAM implementations.

Effective IAM will not cover:

- The **complete set of AWS IAM features** (⚠️ +800 pages)
- Implementing a complete IAM infrastructure code library
- Network security
- Auditing logs
- Incident response

Those topics are all important, but complementary to a solid IAM foundation. This guide will help you design and implement IAM so that you can do each of those better.

We'll describe what the interfaces of infrastructure code libraries should look like so they are usable by non-experts and composable into delivery pipelines. We'll point to the best implementations of these ideas.

Additionally, we'll show how well-architected IAM integrates with and improves AWS network security, auditing, and incident response.

Why I'm a relevant source of advice

My name is **Stephen Kuenzli** and I have built, delivered, and operated applications and infrastructure for more than 20 years in high tech manufacturing, banking, and ecommerce. I have a B.S. in System Engineering and am always working on the bottleneck constraining delivery.

In 2014, I went all-in on delivering applications using infrastructure code, Docker containers, and Cloud - specifically AWS. I led several large migrations to AWS using those technologies as an architect and implementer. Those migrations succeeded, but the hardest part was always getting security right, particularly IAM.

No one understood how to create good IAM security policies for people and applications, verify they work as expected, and deliver changes at the speed of continuous delivery.

I interviewed more than 50 other practitioners (so far) and found they have similar problems.

In 2019, I launched k9 Security to solve those problems and make AWS IAM usable for Cloud teams. k9 helps many Cloud teams deliver apps securely with usable automation and insightful IAM access audits. But this guide *is not* a product pitch. Instead I'll share insights into the problems at the intersection of AWS IAM, infrastructure code, and continuous delivery that frustrate so many Cloud teams and proven solutions to those problems regardless of which tools you use.

Let's go fast, safely.

Acknowledgements

I would like to thank the many people who generously helped this book:

Jen Kuenzli, whose incredible support makes my work possible.

The customers and colleagues who inspired and motivated this work.

The reviewers whose feedback made this book much better, particularly:

- **Jeff Nickoloff**
- **Paul Swail**
- **Steve Sutton**
- **Matt Hickie**
- **Brent Ryan**
- **Lukas Ruebbelke**
- **Wes Novack**
- **Tomasz Stechly**

Sebastian Castro, who used his graphics skills to help me communicate clearly.

The Cloud Security and DevOps communities for working in the open and looking for better ways of working.

Amazon Web Services for building a wonderful, sometimes bewildering, machine.

Everyone who gave me the opportunities to get to this point.

Thank you.

Control access to any resource in AWS

Let's learn how to control access to any resource in AWS. That's a tall order, so we'll work through a 'simple' example that introduces the key IAM concepts you'll need to be effective.

We'll start with the basic IAM access control flow and elements of IAM policies. These security policy elements control a principal's ability to execute an API action that affects a resource.

Control access with IAM policies

IAM policies control whether a principal may act on a resource.

This diagram depicts a simplified IAM access control flow for an AWS API request:

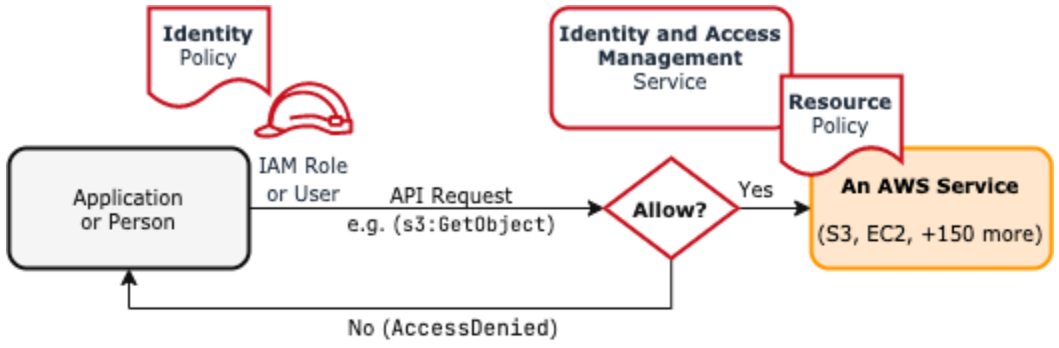


Figure 2.1: Simplified IAM Access Control Flow

First, an application or person authenticates as an IAM role or user principal. A principal is an entity authenticated by AWS and assigned privileges to use within AWS. Then that principal requests an AWS API action. The AWS Identity and

Access Management (IAM) system evaluates that request to determine if it is allowed. IAM does that by evaluating any:

- Identity policies attached to the principal
- Resource policies attached to the resource, e.g. S3 bucket
- Service Control policies attached to the AWS Account

Finally, IAM renders a decision either allowing the request to proceed to the target service API or responds with `AccessDenied`.

At its core, AWS IAM enables you to state whether a *principal* should be *allowed or denied* the ability to invoke an *API action* on a *resource*. Each of those emphasized terms are key elements of an IAM security policy *statement*. A policy statement describes which access control rules apply in a given situation.

Statements are collected into an IAM security policy document represented as **JSON** in this form:

general-form.json
```
{
 "Version": "2012-10-17",
 "Statement": [
   ... one or more Statement objects ...
   {
     "Sid": ... (Optional) Statement Identifier ...,
     "Effect": ...either "Allow" or "Deny" ...,
     "Action": [... array of Actions ...],
     "Resource": [ ... array of Resources ... ]
     ],
     "Principal": { ... one kind of principal ...
       "AWS": [... array of AWS accounts and IAM principals ...]
       "Service": [... array of AWS AWS services ...]
       ... others ...
     }
     "Condition": { ... (Optional) extra condition objects }
   }
 ]
}
```

Each policy document has a `Version` and `Statement` member element.

The `Version` member defines which version of the AWS Security Policy language the document is written in. Use the latest version, `2012-10-17`.

The `Statement` member contains a list of one or more statement objects. Each `Statement` object has the following form:

`Sid` (optional): a string identifying the statement's purpose

`Effect` (required): whether to `Allow` or `Deny` access if the statement applies; `Deny` always wins when multiple statements apply

`Principal` (required): which principals the statement applies to, most importantly and commonly an `AWS` principal. The AWS Security Policy language supports several kinds of principals:

- `AWS`: an AWS account or IAM user or role principals within an account; may be specified as a single string or list of strings. Usually specifies an IAM entity within your AWS account, your organization, or a partner's account.
- `Service`: the name of an AWS service such as `cloudtrail.amazonaws.com`
- `Federated`: a principal from a **federated identity provider** such as a corporate IdP integrated via SAML or a public IdP such as Cognito, Google, Facebook, or Amazon
- `Anonymous`: allow anyone access via `*`; this can be narrowed via conditions

`Action` (required): one or more AWS API actions the statement will `Allow` or `Deny` the `Principal` to invoke as a string or list of strings. Supports wildcards , `?` and `*`.

`Resource` (required): one or more AWS resources the statement applies to, specified as ARNs. Supports wildcards, `?` and `*`.

`Condition` (optional): one or more conditions that qualify when the statement applies using context from the request to verify a request was made in a certain way, such as to an encrypted endpoint. Some conditions support wildcards, `?` and `*`.

The `Principal`, `Action`, and `Resource` each have a negated form that can be used instead of the positive form: `NotPrincipal`, `NotAction`, and `NotResource`. The negated forms are primarily useful for advanced use cases such as granting limited access to an AWS service principal, but are difficult to use correctly. `NotPrincipal` is especially dangerous as it's easy to 'match' nearly every AWS account. So is `NotResource` combined with `Allow`, which makes it trivial to accidentally allow access to all buckets except one. Avoid `NotPrincipal` and `NotResource`.

While `Principal`, `Action`, and `Resource` (or their negative form), are required elements, AWS IAM infers the `Principal` for a policy when attached to an IAM user or role.

All AWS Security policy documents take the form described above. Consult the AWS IAM reference guide for the **full IAM policy language grammar**.

AWS supports **five types of security policy**, each applying to a different scope:

Policy Type	Scope / Attachment Point	Supported Effect(s)	Purpose
Service Control Policy	AWS Organizational Unit or Account	Limit Allows, Deny	*Limit* an entire AWS organizational unit or account's use of AWS service API actions
Identity Policy	IAM user, group, or role	Allow, Deny	*Grant* or *limit* a principal's use of AWS service API actions and resources within the account.
Permissions Boundary	IAM user or role	Limit Allows, Deny	*Limit* an IAM principal's use of AWS service API actions *granted via Identity policies*, particularly AWS Managed Policies. **Partially limits** permissions granted by Resource policies.

Session Policy	STS session	Limit Allows, Deny	*Limit* an IAM principal's use of AWS service API actions *granted via Identity policies* within a given Security Token Service (STS) session to those allowed by the Session policy. Does not limit permissions granted directly to the session by Resource policies.
Resource Policy	Resource	Allow, Deny	*Grant* or *limit* a principal's or session's use of AWS service API actions to a particular resource. Resource policies let you grant permissions to other AWS accounts or services on a per-resource basis, enabling cross-account and public access scenarios.

Take a moment to think about the level of control and flexibility this language provides. This makes the IAM security policy language very powerful, but also difficult to understand. We'll dive into **why IAM is hard** later.

Now let's see how these policy elements work in practice by controlling access to an S3 bucket.

Control Access to an S3 Bucket

Let’s start with a common deployment scenario and policy requirements.

Suppose we have a simple application deployed entirely in AWS:

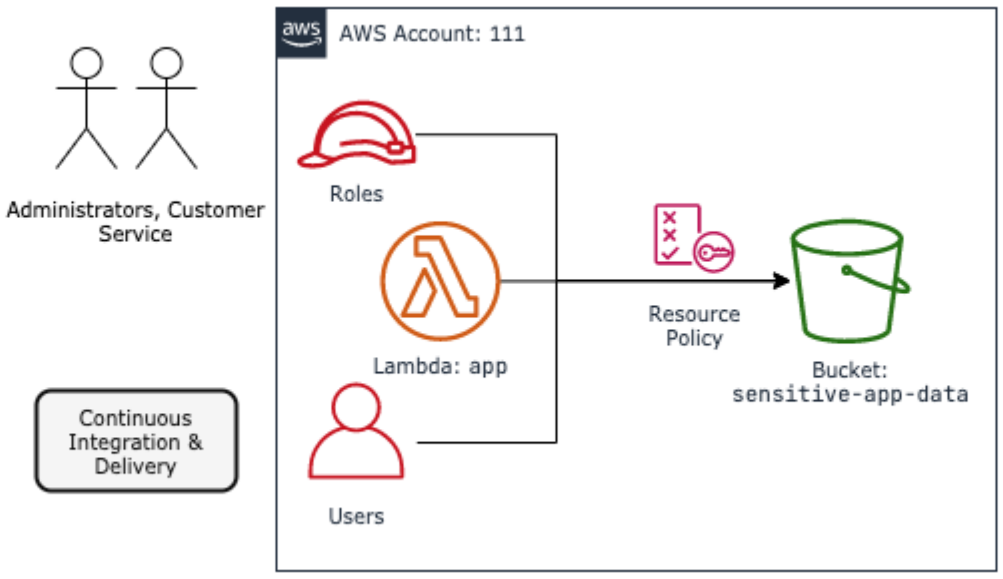

AWS Account: 111
Administrators, Customer Service
Continuous Integration & Delivery
Roles
Lambda: app
Users
Resource Policy
Bucket: sensitive-app-data

Figure 2.2 Simple App Using Lambda & S3

The application:

- deploys to AWS using an automated delivery pipeline
- runs on AWS Lambda using the `app` IAM role
- is supported by a customer service team
- stores data in the `sensitive-app-data` bucket

The deployment must implement the organization's high-level security policy requirements:

- implement least privilege, allowing only explicitly-specified principals the actions and access to data they need to perform their business function and denying access to all other principals
- require encryption at rest and in transport

Many people approach this by creating an Identity policy for the application's IAM role, so let's start there. We can create an Identity policy for the `app` role that grants read and write access to the `sensitive-app-data` bucket:

identity-policy.json

```
{
 "Version": "2012-10-17",
 "Statement": [
   {
     "Sid": "AllowRestrictedReadData",
     "Effect": "Allow",
     "Action": [
       "s3:ListBucket",
       "s3:GetObjectVersion",
       "s3:GetObject"
     ],
     "Resource": [
       "arn:aws:s3:::sensitive-app-data/*",  # for GetObject*
       "arn:aws:s3:::sensitive-app-data"     # for ListBucket
     ]
   },
   {
     "Sid": "AllowRestrictedWriteData",
     "Effect": "Allow",
     "Action": [
       "s3:PutObject",
       "s3:AbortMultipartUpload"
     ],
     "Resource": [
       "arn:aws:s3:::sensitive-app-data/*",
       "arn:aws:s3:::sensitive-app-data"
     ]
   }
 ]
}
```

Once attached to the `app` role, this Identity policy will allow the application to read and write data in the `sensitive-app-data` bucket.

But problems are lurking. This identity policy grants access to the `app` role in a minimal way, but it doesn't achieve the full mission of protecting the sensitive application data.

What if:

- The `app` role has another identity policy attached that allows it to delete objects in any bucket
- Other IAM users and roles have the ability to read objects from any S3 bucket in the account

These scenarios are very common. Many Identity policies include statements that do not limit the resources they apply to with wildcards, either:

- `"Resource": "*"` # all resources in the account
- `"Resource": "arn:aws:s3:::*"` # all S3 buckets

  This is the case when using AWS Managed Policies like `ReadOnlyAccess` or `AdministratorAccess`. **AWS Managed Policies** always use `"Resource": "*"` since they must be usable in any customer's account.

Depending on the use case, wildcards are useful, necessary, and dangerous. Carefully analyze the scope of resources the wildcard matches now, and what it could match in the future as resources change. Then decide if the matched scope is appropriate.

We've just seen that we can *allow* access to sensitive data with an Identity policy. But we can't *deny* unintended resource access with Identity policies in a scalable and maintainable way.

It's impractical for engineers to continually be aware of and evaluate every policy for every identity in an account. It's really impractical to update Identity policies every time a new resource needs to be protected.

So how will we implement the least privilege and encryption requirements for our sensitive data?

We need to block unauthorized and insecure access to sensitive data with a different approach, Resource policies.

Actually implement Least Privilege

Let's turn the problem around. Deny unauthorized and insecure access to the sensitive data using a resource policy attached to the bucket.

This flowchart shows how AWS IAM evaluates policies:

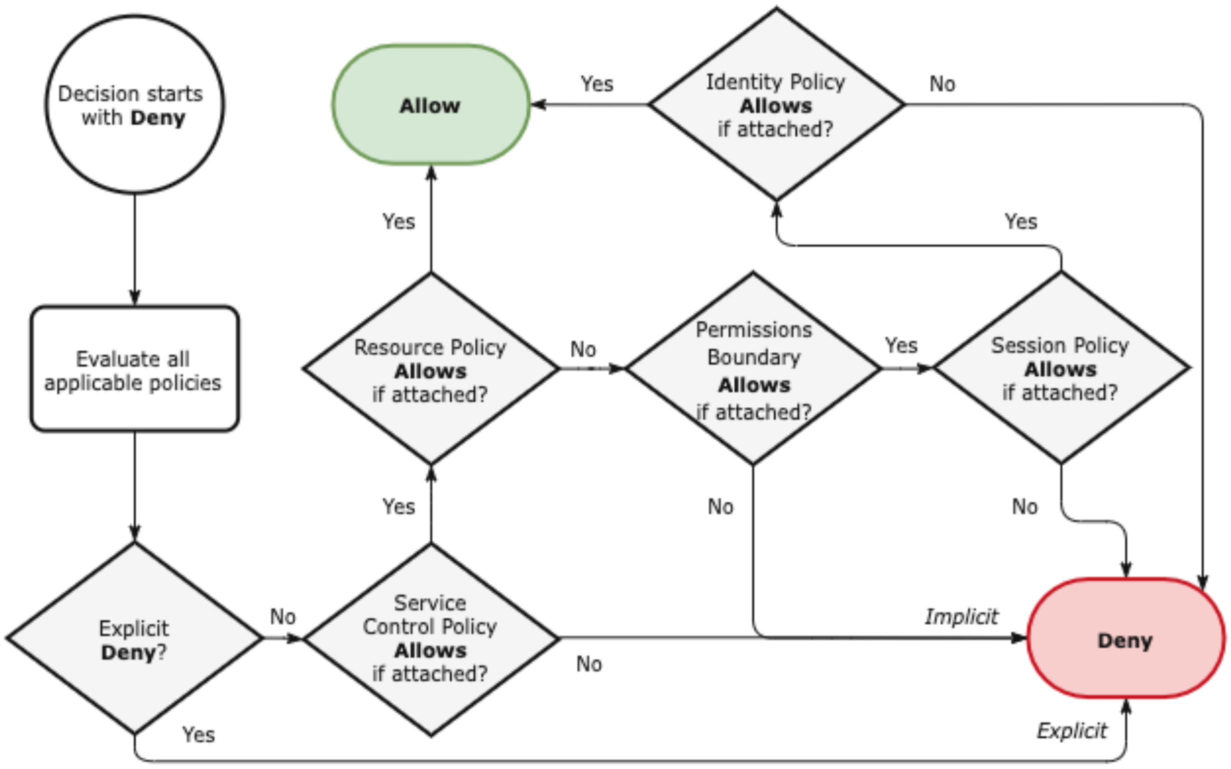

Decision starts with Deny
Evaluate all applicable policies
Explicit Deny?
Yes
No
Service Control Policy Allows if attached?
Yes
No
Resource Policy Allows if attached?
Yes
No
Allow
Permissions Boundary Allows if attached?
Yes
No
Session Policy Allows if attached?
Yes
No
Identity Policy Allows if attached?
Yes
No
Implicit
Explicit
Deny

Figure 2.3 AWS Policy Evaluation Logic

Whoa! That's a lot.

That's the high level process AWS uses to evaluate security policies and decide "can the caller execute this API action?"

Any `Allow` in the policy evaluation chain provides access to the resource unless there is an explicit `Deny`.

Notice that:

- Both Identity and Resource policies may `Allow` an action.
- Effects are calculated with this precedence:
    - Explicit `Deny` by matching statement, overriding `Allow`
    - Explicit `Allow` by matching statement
    - Implicit `Deny` when no statement matches

Resource policies apply security rules directly to a resource like an S3 bucket or KMS key. In practice, there are relatively few sensitive data resources in an account. So a more practical approach to protecting the sensitive data is to attach a resource policy that enforces least privilege and best practice.

Begin creating your resource policy by identifying precisely who needs access to the bucket and what kind of access they need.

Suppose interviews with the application delivery and customer support team reveal the following access requirements for IAM principals:

- `ci` user needs administration capabilities to deploy application updates

- `admin` role needs administration capabilities to fix urgent problems
- `app` role needs read and write data capabilities for the application to function
- `cust-service` role needs to read data capabilities to investigate problems

All of these identities and application resources exist within a single AWS account, `111`.

Now we have enough information to provision the *intended* access in the steps that follow.

Deny, Deny, Deny

Start by creating a policy with a `DenyEveryoneElse` statement that blocks access from all principals *who do not need access* to the bucket:

deny-everyone-else.resource-policy.json
```
{
     "Version": "2012-10-17",
     "Statement": [
       {
         "Sid": "DenyEveryoneElse",
         "Effect": "Deny",
         "Action": "s3:*",
         "Resource": [
           "arn:aws:s3:::sensitive-app-data",
           "arn:aws:s3:::sensitive-app-data/*"
         ],
         "Principal": {
           "AWS": "111"
         },
         "Condition": {
           "ArnNotEquals": {
             "aws:PrincipalArn": [
               "arn:aws:iam::111:role/admin",
               "arn:aws:iam::111:role/app",
                "arn:aws:iam::111:user/ci",
                "arn:aws:iam::111:role/cust-service"
              ]
            }
          }
        }
      ]
    }
```

The `DenyEveryoneElse` statement jumps right into the deep end of the IAM policy concepts. This statement denies all IAM principals in account `111` the ability to invoke any S3 api actions on this bucket, whenever the condition is met. That condition is when the requestor's IAM principal ARN *is not* one of the intended principals, `admin`, `app`, `ci`, or `cust-service`.

This statement narrows the scope of who can access the bucket to these four principals, *no matter what their Identity policies allow*. Remember explicit `Deny` takes precedence over explicit `Allow` in AWS IAM.

We also want to enforce the organization's encryption policy.

Require encryption in transport with a `DenyInsecureCommunications` statement:

```
{
  "Sid": "DenyInsecureCommunications",
  "Effect": "Deny",
  "Action": "s3:*",
  "Resource": [
    "arn:aws:s3:::sensitive-app-data/*",
    "arn:aws:s3:::sensitive-app-data"
  ],
  "Principal": {
    "AWS": "*"
  },
  "Condition": {
    "Bool": {
      "aws:SecureTransport": "false"
    }
  }
}
```

This denies all principals the ability to invoke an S3 api action using an insecure transport. This forces clients to use https. The AWS API will deny any request made with http when it reaches the AWS API endpoint. (Note: If requiring secure transport causes a problem, consult the relevant AWS SDK docs and reconfigure the application to use the `https` transport scheme.)

Enforce encryption at rest with these`DenyUnencryptedStorage` and `DenyStorageWithoutKMSEncryption` statements:

```
{
  "Sid": "DenyUnencryptedStorage",
  "Effect": "Deny",
  "Action": "s3:PutObject",
  "Resource": "arn:aws:s3:::sensitive-app-data/*",
  "Principal": {
    "AWS": "*"
  },
  "Condition": {
    "Null": {
      "s3:x-amz-server-side-encryption": "true"
    }
  }
},
{
  "Sid": "DenyStorageWithoutKMSEncryption",
  "Effect": "Deny",
  "Action": "s3:PutObject",
  "Resource": "arn:aws:s3:::sensitive-app-data/*",
  "Principal": {
    "AWS": "*"
  },
  "Condition": {
    "StringNotEquals": {
      "s3:x-amz-server-side-encryption": "aws:kms"
    }
  }
}
```

These statements force clients to do two things. First, all `S3:PutObject` operations must use AWS' server side encryption services on every put. Second, the encryption must use an encryption key managed by the AWS Key Management Service (KMS). We'll discuss these choices in more detail later. For now, know they enforce encryption at rest with a service you can also use to enforce fine-grained data access controls, KMS.

Allow, Allow, Allow

Now let's explicitly grant the access each of the principals need with an appropriate statement.

Organize the Allow statements by needed access capability, not principal. Organizing by access capability simplifies understanding what API actions principals *may* and *may not* perform on the resource. You will always know where to look to see if, e.g. a principal is allowed to write data.

Grant an access capability with a statement using this form:

```
{
  "Sid": "Allow<CapabilityName>",
  "Effect": "Allow",
  "Action": [
    "s3:<CapabilityAction1>",
    "s3:<CapabilityAction2>",
    "s3:<CapabilityAction...N>",
  ],
  "Resource": [
    "arn:aws:s3:::sensitive-app-data/*",
    "arn:aws:s3:::sensitive-app-data"
  ],
  "Principal": {
    "AWS": "*"
  },
  "Condition": {
    "ArnEquals": {
      "aws:PrincipalArn": [
        "arn:aws:iam::111:<Identity1>",
        "arn:aws:iam::111:<Identity2>",
        "arn:aws:iam::111:<Identity...N>"
      ]
    }
  }
}
```

The `Sid` describes that the statement allows a particular capability such as reading data or administering resources.

The `Resource` element applies to:

- all objects in the bucket, `arn:aws:s3:::sensitive-app-data/*`
- the bucket itself, `arn:aws:s3:::sensitive-app-data`

Buckets and Bucket Objects are distinct resource types. This is an important detail. Access to a bucket is identified and evaluated separately from the objects inside it. Most AWS API actions apply to a single resource type, though some API actions interact with multiple.

The `Principal` matches all AWS IAM principals in the `aws:principalArn` array.

A common policy mistake is to mismatch the API action and covered resource type. Allowing `S3:PutObject` to `arn:aws:s3:::sensitive-app-data` is an example of this mistake. AWS may save the policy depending on what else is in it. But when you actually invoke `S3:PutObject` the operation will fail. The action allowed puts to `arn:aws:s3:::sensitive-app-data`, a bucket type. But `S3:PutObject` operates against objects.

Allow Writes

The correct allow statement looks like this, which allows writes into the bucket:

```
{
 "Sid": "AllowRestrictedWriteData",
 "Effect": "Allow",
 "Action": [
   "s3:PutObject",
   "s3:AbortMultipartUpload"
 ],
 "Resource": [
   "arn:aws:s3:::sensitive-app-data/*"
 ],
 "Principal": {
   "AWS": "*"
 },
 "Condition": {
   "ArnEquals": {
     "aws:PrincipalArn": [
       "arn:aws:iam::111:role/app"
     ]
   }
 }
}
```

This statement allows the `app` role to put objects and also abort object uploads that are in progress. `AllowRestrictedWriteData`'s resource element was narrowed to cover only objects in the bucket because the API actions pertain only to objects. The AWS API docs document which resource types each action works with.

Allow Administration

Now allow the `ci` and `admin` principals to administer the bucket and its resources with a statement like:

```
{
 "Sid": "AllowRestrictedAdministerResource",
 "Effect": "Allow",
 "Action": [
   "s3:PutReplicationConfiguration",
   "s3:PutObjectVersionAcl",
   "s3:PutObjectRetention",
   "s3:PutObjectLegalHold",
   "s3:PutObjectAcl",
   "s3:PutMetricsConfiguration",
   "s3:PutLifecycleConfiguration",
   "s3:PutInventoryConfiguration",
   "s3:PutEncryptionConfiguration",
   "s3:PutBucketWebsite",
   "s3:PutBucketVersioning",
   "s3:PutBucketTagging",
   "s3:PutBucketRequestPayment",
   "s3:PutBucketPublicAccessBlock",
   "s3:PutBucketPolicy",
   "s3:PutBucketObjectLockConfiguration",
   "s3:PutBucketNotification",
   "s3:PutBucketLogging",
   "s3:PutBucketCORS",
   "s3:PutBucketAcl",
   "s3:PutAnalyticsConfiguration",
   "s3:PutAccelerateConfiguration",
   "s3:DeleteBucketWebsite",
   "s3:DeleteBucketPolicy",
   "s3:BypassGovernanceRetention"
 ],
 "Resource": [
   "arn:aws:s3:::sensitive-app-data/*",
   "arn:aws:s3:::sensitive-app-data"
 ],
 "Principal": {
   "AWS": "*"
 },
 "Condition": {
   "ArnEquals": {
     "aws:PrincipalArn": [
       "arn:aws:iam::111:user/ci",
       "arn:aws:iam::111:role/admin"
     ]
   }
 }
}
```

Notice the (long) list of actions includes some that operate on:

- buckets, e.g. `s3:PutBucket*`
- bucket objects, e.g. `s3:PutObject*`
- and some where it's not particularly clear what type of resource applies, e.g. `s3:PutMetricsConfiguration`.

We can create similar statements to allow reading configuration as well as reading or deleting data.

The full policy for this example appears in **Appendix - Least Privilege Bucket Policy**. We won't include it here because it's well over 200 lines.

To adopt this least privilege model for your own bucket, replace the principals and bucket name with your own. Then put the policy into effect by populating the policy contents into the bucket's policy attribute in CloudFormation or Terraform, the AWS cli, or console. These all invoke `s3:PutBucketPolicy`. Finally, verify principals can still access data as expected. Resource policies are stored within the scope of the resource and cannot be shared between resources directly. We'll scale implementing least privilege in AWS when we **'Simplify AWS IAM'**.

Let's wrap up our 'simple' example.

Summary

This 'simple' example demonstrated a few things.

First, the AWS IAM security policy language is flexible and powerful enough to implement fine-grained access controls to AWS API actions and data. The AWS identity and resource policies created in this chapter implement our high-level goals:

1. allow *only* authorized principals to access data in the way we intend
2. enforce encryption in transport and at rest

Second, it wasn't easy or straightforward to implement the policies we needed. The policy language is complex and some effects are difficult to anticipate. We had to close off unexpected access from other principals with excess privileges in the account. Enforcing basic encryption requirements is possible, but took three separate statements.

And the final policies constitute hundreds of lines instead of a few tens of lines you find in most 'Getting Started' blog posts and examples.

But keep in mind that the access control capabilities provided by the AWS IAM security service (and equivalent in other hyperscale Clouds) have no common analogue to on-premises data centers.

AWS IAM is over 10 years old but it's still 'early', especially when it comes to abstractions that help teams go fast safely.

Keep reading to learn about the problems in AWS IAM and the abstractions you need to solve them.

Why AWS IAM is so hard to use

The AWS Identity and Access Management system is a feat of modern engineering, correctly and quickly enforcing access controls at massive scale. The access evaluation engine is ubiquitously integrated, highly available, and quickly processes **hundreds of millions of access requests per second**. Engineers can use the flexible and powerful policy language to authorize access to all infrastructure and even applications.

But interviews with many Cloud, DevOps, and Security engineers revealed problems with AWS IAM.

Engineers say it's hard to:

- Write policies that do what they intend
- Understand what all the policies actually do
- Validate access controls without breaking things

Even experts find it difficult to create policies that do what they *intend*. My research found most practitioners feel like understanding who has access to their data is impossible.

The ***usability*** of AWS' powerful security policy engine leaves much to be desired. The problem is not that AWS IAM doesn't have enough features. The opposite is closer to the truth. There are many features, features interact in ways that are difficult to understand, and feedback on the correctness of a policy is slow and inadequate.

This chapter explains why AWS security policy engineering is so hard, and offers solutions to those usability problems. The 50+ engineers I interviewed really *want* to create good policies. They fear letting down customers when they don't. However, because policy development and testing is difficult, engineers run out of time, energy, and patience to do it well without blocking delivery.

We'll examine the design of AWS IAM and uncover problems that make it difficult for engineers and teams to use it correctly.

First, you'll learn IAM's complicated process for evaluating security policies to determine whether an API action should be allowed or denied.

Second, you'll learn how the flexibility of the AWS security policy language makes it difficult to identify what is in scope and how policies interact.

Then we'll examine IAM's usability through the lens of design to identify the problems that make it difficult for engineers and teams to use correctly.

Finally, we'll discuss how to address these design problems with components that wrap IAM's power in a usable package.

How AWS Security policies are evaluated

Let's uncover why evaluating the *effects* of AWS security policies is hard.

This flowchart depicts AWS IAM's security policy evaluation logic:

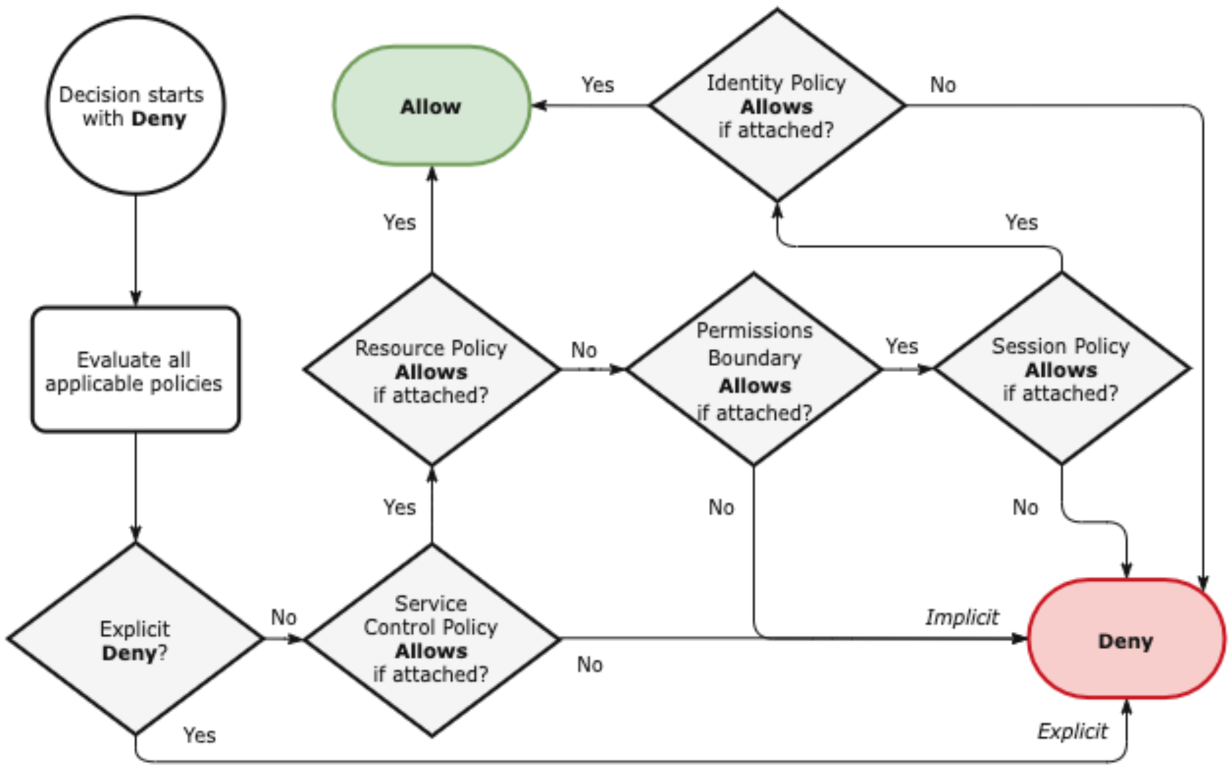

Decision starts with Deny
Evaluate all applicable policies
Explicit Deny?
No
Yes
Service Control Policy Allows if attached?
Yes
No
Resource Policy Allows if attached?
Yes
No
Allow
Yes
Identity Policy Allows if attached?
No
Permissions Boundary Allows if attached?
Yes
No
Session Policy Allows if attached?
Yes
No
Implicit
Explicit
Deny

Figure 3.1 AWS Security Policy Evaluation Logic

Each of the **five types of security policy** are integrated into the access decision making process. This is not simple to understand or evaluate.

Did you notice there are two paths for allowing access when a service supports resource policies? Look for paths to the green end state.

AWS IAM evaluates *all* the policies in scope for the account, principal, session, and resource. Any of those policies might `Deny` or limit access. Two kinds of policy may `Allow` access, Identity and Resource.

An Identity policy attached to an IAM principal can grant access to a specific resource or many resources, e.g. an S3 Bucket or all DynamoDB tables.

When an AWS service supports resource policies, a Resource policy can also grant access to the resource it is attached to, e.g. an S3 bucket or KMS key. More than 20 AWS services support resource policies, primarily those where it is useful to share a resource across accounts.

Resource policies are very powerful and have their own quirks. For example:

- Resource policies can grant permissions past a **Permissions Boundary** or **Session policy** in several scenarios, particularly when allowing an STS session directly or to an IAM role via Condition.
- KMS key policies are the primary way to control access to encryption keys and **introduce an exception to the standard policy evaluation flow**. *The account's Identity policies are only integrated into the policy evaluation flow when access is granted to the account's root user*. 😲 This makes implementing least privilege easier, but diverges from the standard model of allowing access from either Identity or Resource policies.

To understand effective access, engineers must evaluate these policies too.

Merely gathering a principal’s in-scope Identity policies is complicated.

Engineers need to account for *all* Identity policies managing an IAM principal's access. This might include:

- AWS Managed Policy attached directly to the IAM role, user, or a group the user belongs to
- Customer Managed Policy attached directly to the IAM role, user, or a group the user belongs to

- Inline policy attached directly to the IAM role, user, or a group the user belongs to

IAM supports attaching up to 10 policies to a role or group by default; AWS support can raise that limit to 20. An IAM user can be a member of up to 10 IAM groups. See **IAM Limits** for full details.

Depending on where you define policies, engineers may have to account for many policies, defined in many places.

And while convenient to grant a lot of access quickly, AWS Managed Policies:

- Do not limit resource scope. Actions in the policy apply to all resources in the account. So when the `ReadOnlyAccess` policy is applied to an IAM principal, it covers all resources and data in the account. Every S3 bucket, DynamoDB table, EBS volume, etc.
- Change over time. For example, the `ReadOnlyAccess` policy updates to grant read access to new services soon after launch. Even service and job-specific policies see updates. (💡 follow the MAMIP **twitter account** by Victor Grenu for updates)

⚠️ This policy evaluation logic also doesn't account for service-specific access control systems such as S3's Object ACLs, which are an extra layer applied in addition to IAM. Of course it is still the engineer's responsibility to understand how these work together.

Identify what's in scope

A security policy can bring a lot or a little into scope.

Control access to any resource described the form and operation of AWS security policy and statement elements. The `Principal`, `Action`, and `Resource` elements may include a very narrow or very wide scope, particularly when using wildcards.

Here are some examples :

Element	Narrow	Medium	Wide
Principal	Fully-qualified IAM role ARN: arn:aws:iam::111: ↲ role/app	N/A	Another AWS Account: 123456789012 or All AWS accounts *
Action	Specific actions: ["s3:GetObject", "s3:GetObjectAcl", "s3:PutObject"]	All S3 actions starting with GetObject: "s3:GetObject*"	Full access to S3: s3:* or full access to AWS APIs: *:*
Resource	Specific bucket and object: ["arn:aws:s3:::app-data", "arn:aws:s3:::app-data/file-1"]	All objects in a dedicated bucket: ["arn:aws:s3:::app-data/*"]	All objects in a shared bucket: ["arn:aws:s3:::shared-data/*" or all Buckets: * (this is what AWS Managed Policies do)

Condition - aws:PrincipalArn	arn:aws:iam::111: ↲ role/app (Same as Principal)	All IAM 'app' roles with a known variation: arn:aws:iam::111:role/app-* or arn:aws:iam::111:role/env-prod/app-????????	Another AWS Account: 123456789012 or All AWS accounts * (Same as Principal)

Table 3.1 How wildcards affect scope

So engineers now have a huge information gathering and evaluation task to perform. They must correctly:

- Retrieve from zero to *tens* of identity policies that may be directly attached as a managed or inline policy, or indirectly attached via a group, again managed or inline.
- Retrieve resource policies for relevant resources involved in the request path, e.g. S3 bucket or KMS key.
- Parse the policies and build a mental model of what each statement brings in and out of scope.
- Calculate the effects of the statement, properly accounting for `Effect` precedence and negation.

But engineers usually do this in their heads with less than perfect information about the policies in their system and how IAM works.

Mistakes will happen.

Engineers will get the `Allow` path to work. No team is going to give up until the application or person's principal's can execute requests successfully. But most teams will move on long before achieving the least privilege they intended.

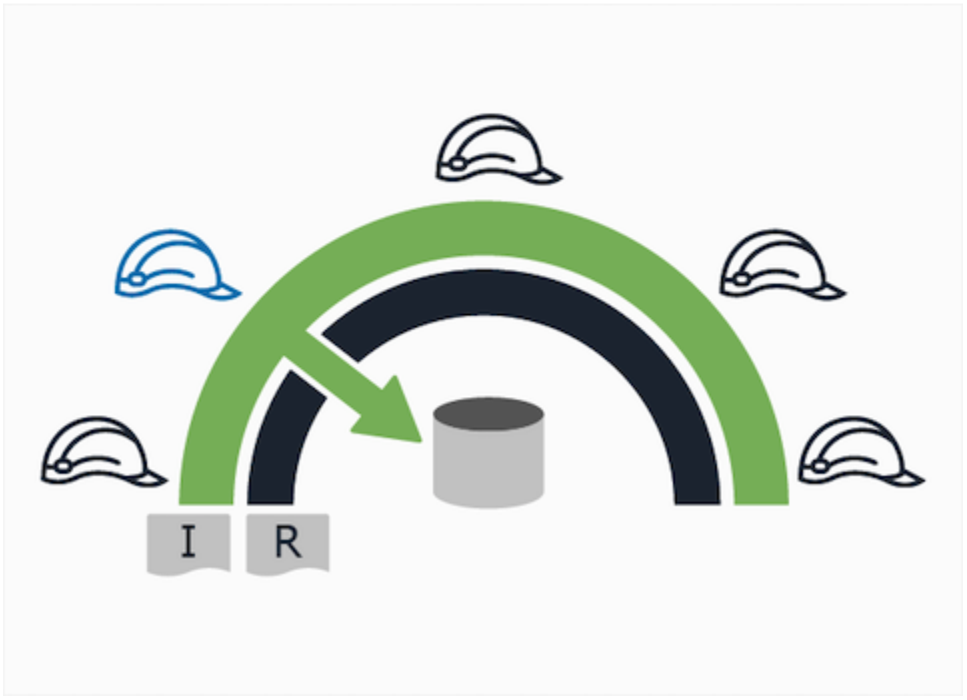


Figure 3.2 Implementing Least Privilege to Resources

Because for engineers to grant *only* the access they intended to principals and resources, two non-obvious things should be included in their security policies:

1. Identity policies attached to principals should scope resource access to implement least privilege for the *principal*
2. Resource policies should allow intended principals and deny everyone else to implement least privilege for the *resource*

Don't let anyone minimize this task.

Writing minimal access policies is impossible for humans working with real systems without serious help.

Problems with the usability of AWS IAM

The biggest problem with AWS security is that most people cannot confidently configure security policies using information they have in their head or at hand. They need to look things up constantly, and they're still unsure. While configuring security controls is an 'everyday' experience, engineers cannot configure policies quickly and correctly enough to achieve their goals.

This is *not* the engineers' fault.

Engineers are using a system which has not been designed for usability.

The AWS security policy system requires users to know and understand a tremendous amount in order to use it effectively. Engineers must understand a lot about how AWS security policies are evaluated, how the policy language works, the features of the policy language for a given service, and the state of the system in order to create a mental model and finally create policies.

This is a serious usability problem because there is a large difference between what most engineers have and what they need to secure resources effectively. Engineers must expend a lot of effort to configure AWS IAM correctly. Usability expert Don Norman describes this as two "Gulfs"[1] in **The Design of Everyday Things**:

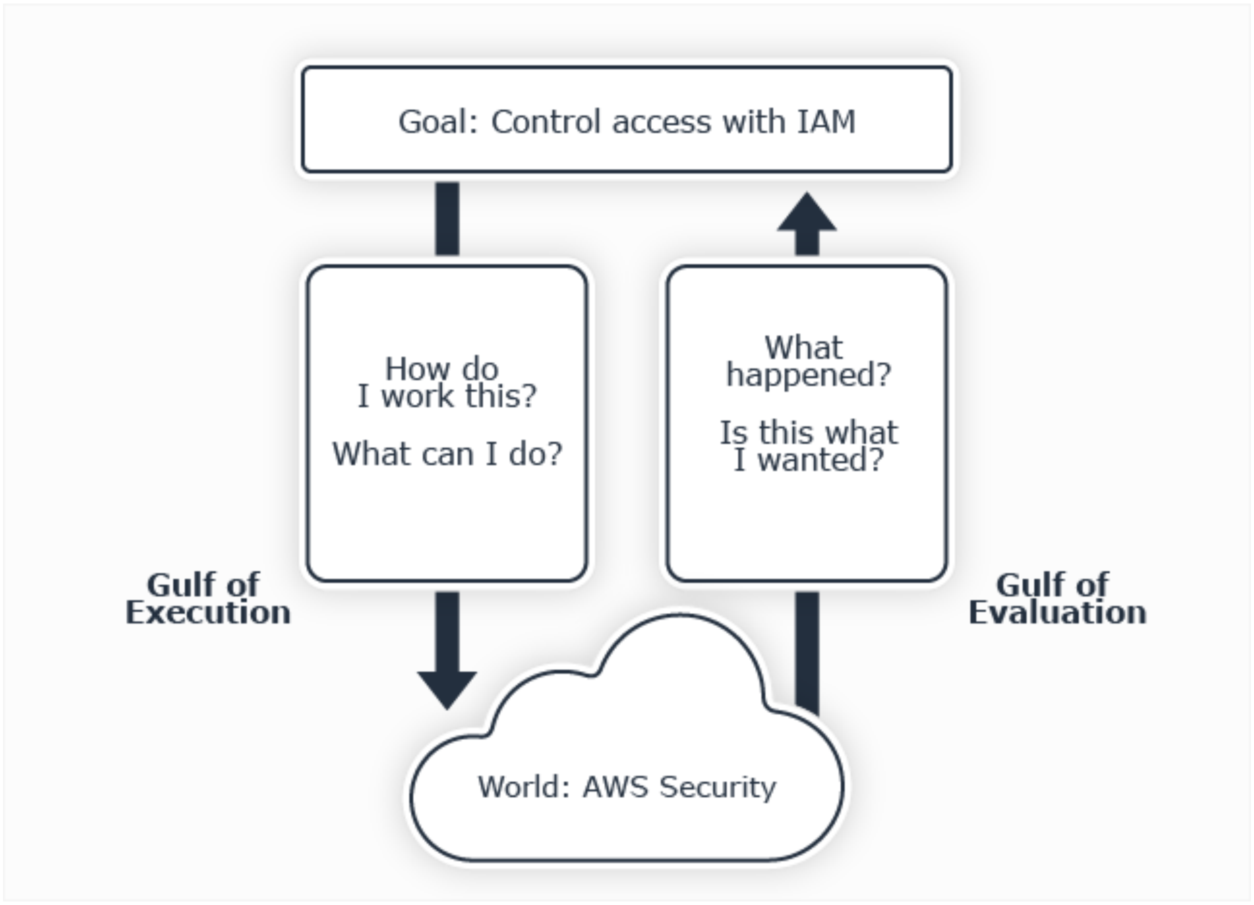


Figure 3.3 The Gulfs of Execution and Evaluation for AWS Security

To complete their goal of controlling access with IAM, Engineers must cross two gulfs:

1. The Gulf of Execution, where they try to figure out how to use IAM
2. The Gulf of Evaluation, where they try to figure out what state IAM is in and whether their actions got them to their goal

Engineers have difficulty answering Norman's critical usability questions for AWS Security:

How do I work this? 5 types of security policy and many attachment points interact within a complex control flow. 🤔

What can I do? The basic security policy language features are described on a single page, but engineers must understand volumes spread across specific AWS service docs. The **AWS IAM user guide** is over 800 pages long! 🤯

What happened? AWS provides no comprehensive way to understand the net effects of changes to principal and resource access. CloudTrail events provide delayed, sometimes incomplete feedback on why access was denied. 😕

Is this what I wanted? Difficult to tell if a given set of policies allows or denies access as intended. Critically, there is no comprehensive mechanism to identify excess access. 😖

AWS' flexibility requires engineers to gather and manage large amounts of security information in their heads. Norman calls this "knowledge in the head." AWS generally does not codify recommended security best practices directly in the service, which would be "knowledge in the world." Engineers must work hard to build this mental model and it takes time, so it will be different from colleagues' models.

In AWS' shared responsibility model, the responsibility for designing a cohesive set of access controls falls to the customer. AWS provides the security primitives, customers provide almost everything else.

AWS partially recognizes the burden of acquiring all this knowledge. They have tried to encode *some* security knowledge "in the world." Encoding knowledge in the world is "how designers can provide the critical information that allows people to know what to do, even when experiencing an unfamiliar device or situation." [2]

Some examples of AWS encoding knowledge in the world are:

- Managed policies for particular jobs: Data Scientist, Power User, etc
- Control Tower and Landing Zones' Service Control Policies
- Config Rules
- Beanstalk
- "Level 2" and "Level 3" constructs in AWS CDK

But AWS is famously flexible, backwards compatible, and built by hundreds of independent teams. So it's unlikely we'll see a cohesive set of usable security tools and libraries emerge from AWS which are an ideal fit for your specific applications. It's even more unlikely those security tools would work in other Clouds which have similar problems.

Third parties can help. Internal platform teams, communities, and vendors can create opinionated, usable solutions that simplify security for their niche. As Norman says[3]:

> Because behavior can be guided by the combination of internal and external knowledge and constraints, people can minimize the amount of material they must learn, as well as the completeness, precision, accuracy, or depth of the learning. They also can deliberately organize the environment to support behavior. This is how nonreaders can hide their inability, even in situations where their job requires reading skills.
>
> — Don Norman

Third-party toolmakers need to help AWS users create safer outcomes through a combination of approaches.

First, help users apply the knowledge they have to the unfamiliar AWS security system. Enable application engineers and others versed in the domain to describe what access *should* be.

Second, constrain configuration choices to a small set of safe options designed to work together.

Third, encapsulate expert knowledge into libraries and tools that meet engineers where they are and in the technology stack they already use.

These approaches have several benefits:

1. Minimizes the AWS security knowledge the majority of engineers must learn to a manageable corpus.
2. Improves communication with a higher level language, patterns, and reference architectures for designing and implementing safer systems.
3. Enables engineers to deliver secure configurations everyday.

We must help engineers use the information they *have* to create great security outcomes.

---

1. Norman, Donald A.. The Design of Everyday Things (p. 38). Basic Books. Kindle Edition.
2. Norman, Donald A.. The Design of Everyday Things (p. 123). Basic Books. Kindle Edition.
3. Norman, Donald A.. The Design of Everyday Things (p. 76). Basic Books. Kindle Edition.

Architect AWS organization for scale

Do you have any dramatic stories of accidents triggered by important activities colliding in a single AWS account? Like:

An application engineer almost deleted a production database.

Load testing an application triggered a production outage by exceeding the AWS API rate limits.

A team new to AWS wants to deploy a prototype and they've asked for IAM admin privileges.

In this chapter you will learn how to solve those problems and architect your AWS organization for scale.

First, architect security domains using AWS accounts to limit the access control problems in any one environment. Then govern the activities that can occur in those accounts with Service Control Policies.

This builds the foundations of AWS Identity access control. And moves your organization towards a safe, sustainable path.

Create security domains with AWS accounts

The most fundamental tool to organize and protect AWS cloud resources is the AWS account. AWS accounts are architectural elements that create management, fault, and security domains with well-defined boundaries. Identities and cloud resources always reside within one and only one account. But many organizations do not use accounts properly, which puts the organization and its customers at risk.

The Single Use Case Rule: Operate each major use case in a dedicated AWS account.

Important resources and limits are shared within an account: IAM, resource limits, and API limits. Separate use cases so they do not interfere with each other.

Examples of use cases include:

- Developing and testing a customer-facing Ecommerce application so it can be deployed to production
- Operating the Ecommerce application so customers can place orders
- Continuous integration and delivery of changes to environments
- Collecting application and infrastructure telemetry for internal analysis

Use cases are supported by one to several workloads and are generally environment-specific.

The AWS Well-Architected program defines a **workload** as:

> A collection of resources and code that delivers business value, such as a customer-facing application or a backend process.
>
> A workload might consist of a subset of resources in a single AWS account or be a collection of multiple resources spanning multiple AWS accounts. A small business might have only a few workloads while a large enterprise might have thousands.
>
> — AWS Well-Architected

AWS recommends isolating workloads in accounts, but this may not be practical in your organization. Review **AWS Security's account reference architecture** for their view on how to organize accounts and deploy core Security services.

This section shows how to organize a medium or large organization's Cloud accounts to deliver changes quickly and operate safely. Tailor this to fit your needs.

Partition accounts by Use Case

Let's start with use cases shared across the organization, then examine those for running end-user applications.

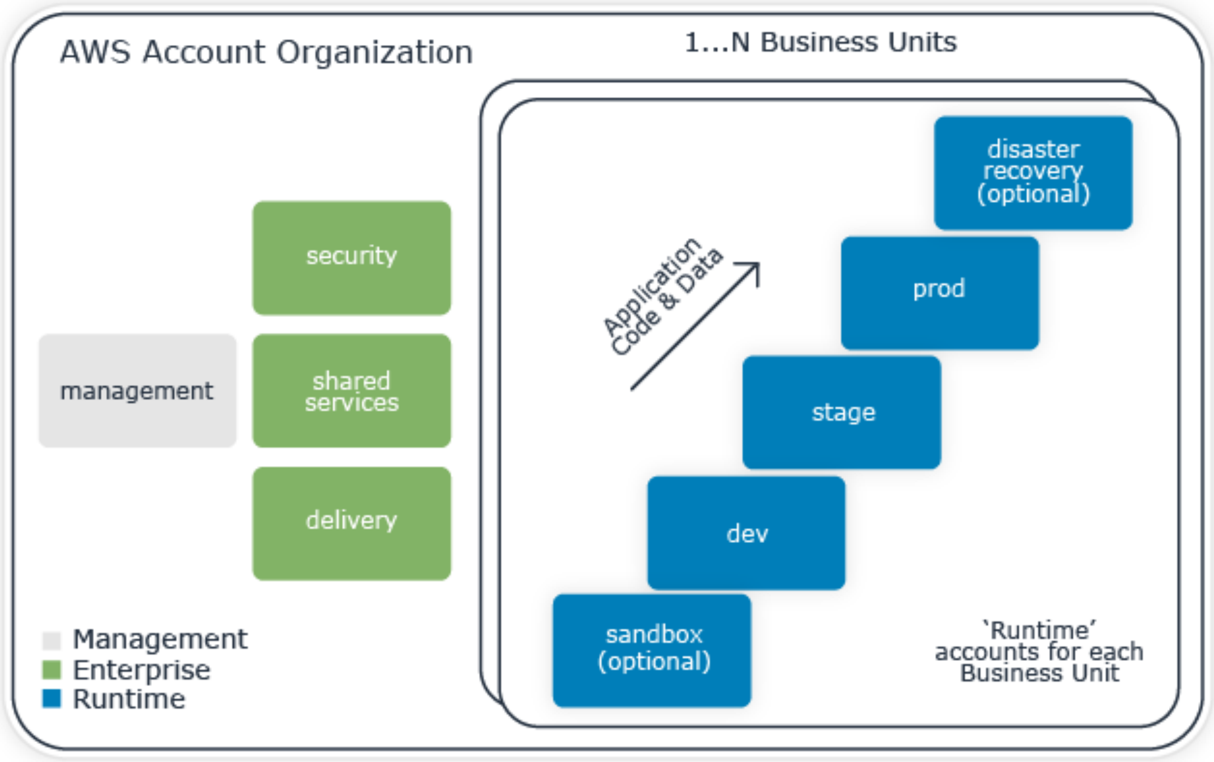


Figure 4.1 AWS Organization Reference Architecture

Enterprise use cases

There are several use cases every Enterprise must support in their Cloud deployment (green). Provision accounts for each use case:

Management

The Management account is the root of trust for the entire organization and its security is paramount. Use the Management account to manage accounts within the organization, consolidate billing, and to (optionally) provision people's access with **AWS Single Sign-on**. No other workloads should run in the Management account.

Security

The Security account contains the organization’s Cloud API activity logs (CloudTrail), access logs (S3, ELB, VPC, etc), and resource configuration inventory (Config). Ingest these logs into log search tools in the Shared Services account.

Shared Services

Operate monitoring, logging, DNS, directory, and security tools in a Shared Services account. Collect telemetry from AWS, third parties, your infrastructure, and your applications running in other accounts. People with high privileges in other accounts may use this data and services, but should not be able to modify operational telemetry.

Delivery

The Delivery account operates the powerful CI/CD systems that build applications and manage infrastructure. This account will have privileged access to most of the organization. Operating CI/CD in a dedicated account simplifies securing that function.

You may find more Enterprise accounts useful. Many organizations use dedicated accounts for centralized networking use cases. One common case is to route between cloud and traditional data center networks. Review the **AWS Secure Environment Accelerator architecture** for how to build highly secure networks across many accounts.

With Enterprise-wide accounts set, let's focus on running your applications.

Runtime use cases

Some organizations start with a single AWS account. Or they may have an account for each business unit. Those organizations often run into problems managing security and costs because use cases are not isolated.

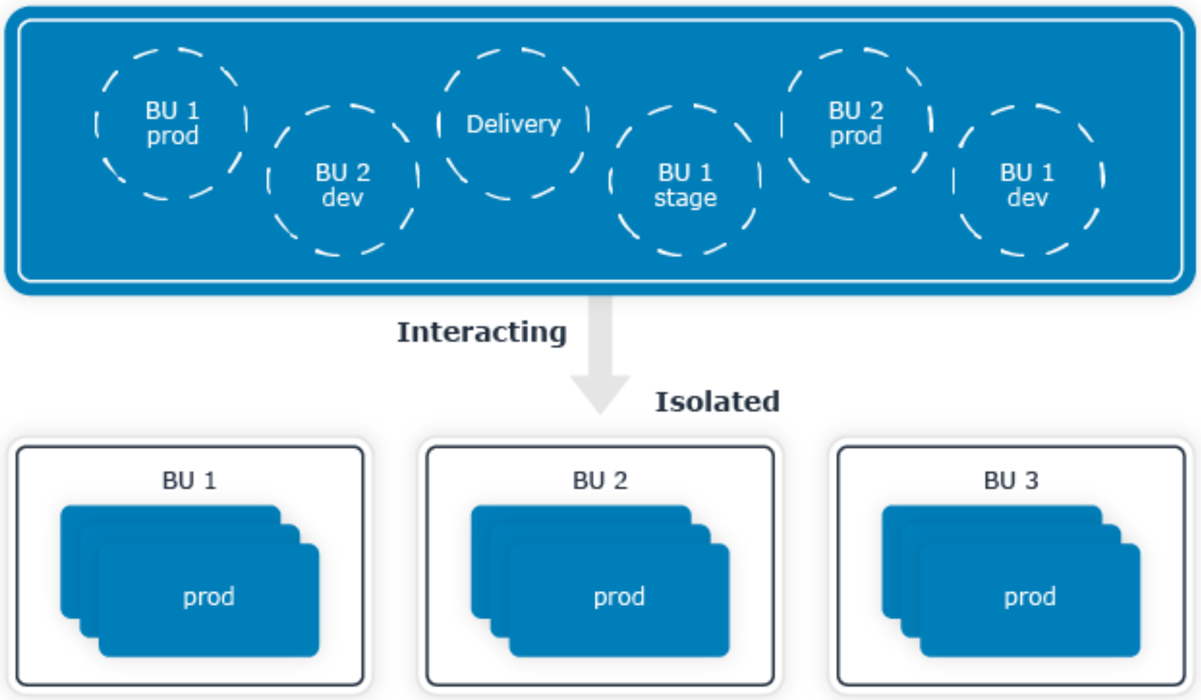


Figure 4.2 Isolate Business Unit use cases with dedicated Runtime accounts

Solve these problems with a set of ‘Runtime’ accounts for each business unit to develop, test, and operate their applications (blue).

Let’s see how partitioning Runtime accounts by business unit and then software delivery phase influences autonomy, security, and cost.

Partition by Business Unit

Most organizations have multiple business units (or departments). Decouple decision making and access management between business units by provisioning a set of Runtime accounts for each business unit. This provides the freedom necessary for business units to do their jobs with minimal coordination.

Then collect each business unit's accounts into an **AWS Organizational Unit (OU)**.

Recognize these choices will guide relationships between people and services within the enterprise going forward. Conway's Law is real and so is the effort to move workloads between accounts.

Autonomy

Business Units use different architectures, team structures, deployment technology, and operational practices. Recognize and accept these differences to help business units coexist and adopt the Cloud in harmony. Avoid battling over standards and shared implementations. Partitioning accounts by business unit creates a clear boundary for leaders to exercise authority within a business unit.

Don't force VPs to lobby their peers to use a technology or make a change within an AWS account. This frustrates everyone. In practice, an Ecommerce BU operates differently than a Data Warehouse. Enable those operational differences with dedicated Runtime accounts so each BU can get their work done.

Security and Safety

IAM users, roles, and policies are scoped to an account. Consequently, an engineer or application in one business unit can use resources without affecting another business unit. This limits risk of security compromises, too. An attacker with a foothold in one business unit cannot automatically access another. Cross-account access can be enabled, but must be done so explicitly.

Cost Management

Tracking and managing AWS operational costs at the business unit level is much easier with both AWS and third-party Cloud cost management tooling.

Partition by Delivery Phase

Most business units deliver applications through multiple phases. Create accounts that match each business unit's delivery phases so they can satisfy the requirements of each phase safely and efficiently.

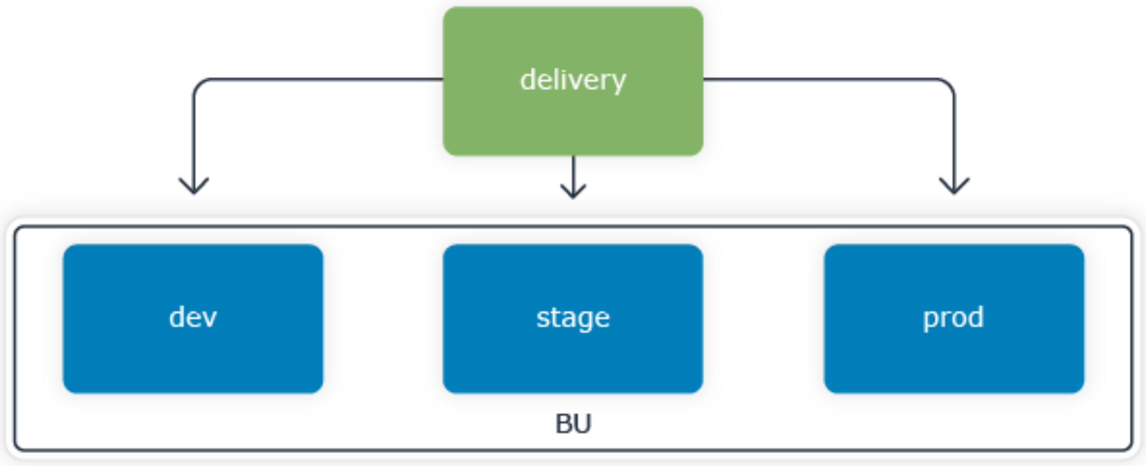


Figure 4.3 Isolate environments by delivery phase within each Business Unit

For example, the Ecommerce BU with delivery phases dev, stage, and prod would have accounts: `ecommerce-dev`, `ecommerce-stage`, and `ecommerce-prod`.

Generally these phases map to what the organization calls 'environments.' Some organizations deploy multiple environments into a single account when their purposes and security requirements are similar. For example, the Ecommerce team might deploy both the development and user acceptance testing environments into the `ecommerce-dev` account.

Deploy changes to environments using automation running in a delivery account. When the organization uses centralized CI/CD services, the delivery account will probably be shared too. If the business unit runs their own CI/CD services, they should use their own delivery account as well.

Autonomy

Application development teams can deploy changes and get feedback rapidly without fear of breaking downstream environments, particularly production.

Security and Safety

Varying a person’s permissions by delivery phase is straightforward when each phase has its own account. An IAM user or role in the dev account won’t automatically get the same permissions in stage or prod. This simplifies giving the right level of access to data and operations at each phase of delivery. Deleting databases may be ok in dev, but almost never in prod.

Partitioning accounts by delivery phase also demarcates audit boundaries, keeping non-production accounts out of scope.

Cost Management

Partitioning by delivery phase helps you understand how money is spent on each environment and set resource usage limits appropriately. You can configure modest resource limits in dev and high limits in prod. These limits often vary by an order of magnitude, so can only be useful with account-level separation.

Let's pull all of this together in an example AWS Organization.

Full example of an AWS Organization

When you apply these principles to a business operating Ecommerce and Data Warehouse business units in AWS, you'll get an AWS organization that looks like:

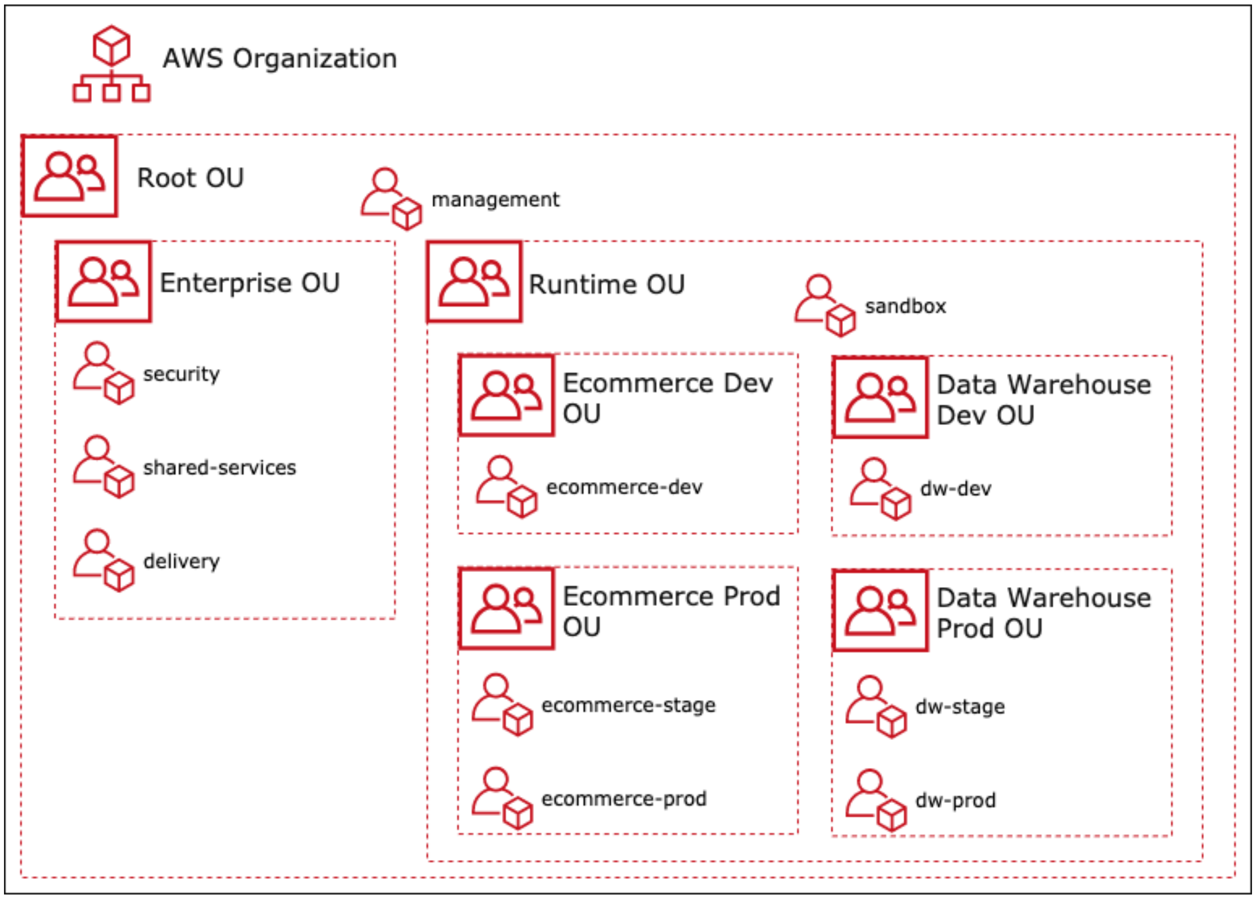


Figure 4.5 Example: AWS Organization architecture

Organize AWS accounts into Organizational Units (OUs) within your AWS Organization. OUs help you administer accounts as a single unit. OUs may be nested into a hierarchy up to five levels deep. You can attach Service Control Policies to an OU, and all accounts within the OU will inherit those security policies.

Model your organization's management and operational requirements with OUs. Then collect accounts into the appropriate OU. Each account belongs to a single OU, at any level of the hierarchy.

The example in Figure 4.5 collects accounts for Enterprise use cases into the Enterprise OU, except for the `management` account. The `management` account is left alone within the `Root` OU as a reminder that Service Control Policies do not apply to an AWS Organization's Management account.

The Runtime use cases are organized into a `Runtime` OU, which contains dedicated OUs for the Ecommerce and Data Warehouse delivery phases. The `Runtime` OU also contains a shared `sandbox` account which supports exploratory development work.

Now let's enable just the capabilities needed to support these accounts' workloads.

Govern capabilities with Service Control Policies

Govern which AWS service capabilities are available in your organization with Service Control Policies. AWS has more than 150 services and 20 regions. You're not going to use them all. This excess capability creates latent risk of accidental use or abuse.

A Service Control Policy (SCP) is a security policy that limits an entire AWS organizational unit or account's use of an AWS service's API actions. SCPs establish the *maximum* set of allowed actions that IAM can allow within a given account. SCPs have properties that are very useful for governance and security:

- SCPs can be applied at any point in an AWS Organizational hierarchy, providing the only policy inheritance capability in the AWS IAM ecosystem.

- SCPs are defined within the Management account, and cannot be overridden in a managed account, even by an IAM administrator.

The **SCP inheritance model** is sequential and reductive.

SCPs *filter* which permissions flow through the organizational hierarchy to the accounts below.

IAM evaluates each AWS API request against the SCPs at each level of the hierarchy. This determines whether the request proceeds to the account's IAM policies:

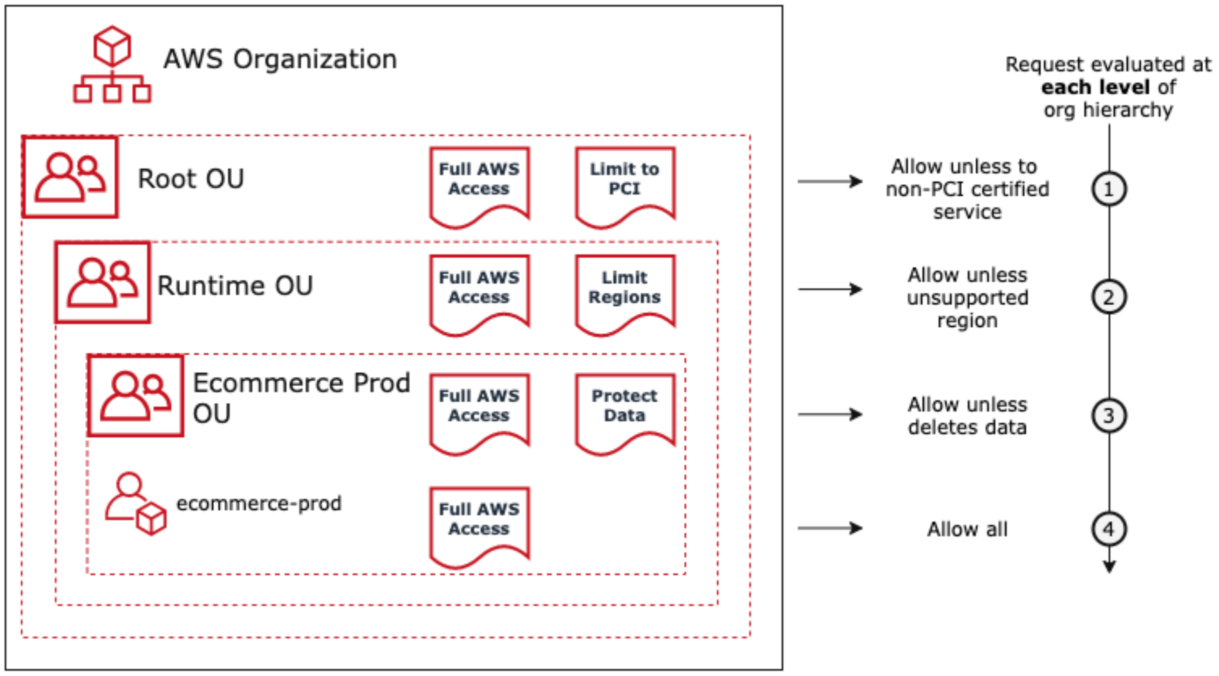


Figure 4.6 Example: AWS Organization with Deny list SCPs

SCPs are *not* gathered from the path to the account then evaluated as a single set.

Each level of the hierarchy filters permissions: root OU, intermediate OUs, then account.

There are two strategies for implementing SCPs.

The allow list strategy explicitly allows only the desired actions at each level of the hierarchy. Implementing this can be difficult. To allow a new capability in an account, you must allow those actions at each level on the path from the root OU to the account. You can attach up to 5 policies at each level of the hierarchy.

The deny list strategy allows everything, then denies unwanted actions where you choose. This approach is generally more maintainable and flexible. Also, SCPs only support conditions in `Deny` statements. So some rules like region restrictions are only possible with denies.

To use a deny list strategy, first attach a policy that allows actions at that level. This is usually the `FullAWSAccess` managed policy. Without a policy that allows, the level will allow no access because no permissions will flow through (implicit deny). Then attach specific policies that explicitly deny unwanted actions at that level.

Let's illustrate the utility of SCPs with a few examples that satisfy common operational and regulatory requirements. We'll start by enforcing the use of approved services.

Limit use to certified or approved services

Many organizations want to limit the AWS services used by their organization. Limiting available services eases compliance with PCI, HIPAA, FedRAMP, ISO27001, and other standards. These standards commonly require use of encryption at rest, access audit, and more. Preventing teams from adopting a non-compliant service dependency can save a lot of trouble down the road. SCPs are a great way to do that.

This service control policy only allows services that AWS certifies as PCI-compliant:

```
{
     "Version": "2012-10-17",
     "Statement": {
       "Sid": "AllowPCICompliantServices",
       "Effect": "Deny",
       "Resource": "*",
       "NotAction": [
         "access-analyzer:*",
         "account:*",
         "acm:*",
         "amplify:*",
         "amplifybackend:*",
         "apigateway:*",
         "application-autoscaling:*",
         "appstream:*",
         "appsync:*",
         "artifact:*",
         "athena:*",
         "autoscaling:*",
         "autoscaling-plans:*",
         "aws-portal:*",
         "backup:*",
         "... snip ~140 services..."
       ]
     }
   }
```

The **full policy** allows 157 services certified by AWS as PCI compliant or are security services that are out of scope. The list of AWS services compliant with one scheme or another grows constantly. So you'll need a strategy to get started and keep up. Fortunately, there are open source tools that generate an SCP that only allows access to services supporting a particular compliance scheme (examples: **aws-allowlister** and **terraform_aws_scp**).

You can stay out of trouble with regulators by only using AWS services certified for your compliance requirements. But you should go further.

Limit AWS services used within your organization to a strategic and sustainable set. The greatest adoption cost for an AWS service is often not found in the monthly AWS bill. Rather it's the effort spent understanding and making things work every day.

You'll need to find a balance between "using the best tool for the job" and "doing the job with the tools *we* know best".

AWS service sprawl is a real problem in many organizations. What portion of AWS services do you think delivery teams actually need to support the business mission? One-third? One half? Surely not all of them.

You don't need to be heavy-handed, but you do need to be deliberate about adopting new services. At least a few people will need to become skilled in using each of them. Define a process for reviewing and adopting new services. Let teams champion a new service by proposing how to use and support that service.

Next, we'll restrict operations to certain regions by leveraging request context keys.

Limit use to particular regions

Suppose you want to ensure your organization uses only two AWS regions: `us-east-1` and `us-west-2`. You can't say that directly in the AWS console, but you can prevent use of AWS services outside of those regions using a Service Control Policy like:

```
{
    "Version": "2012-10-17",
    "Statement": [
      {
        "Sid": "DenyUnsupportedRegions",
        "Effect": "Deny",
        "Resource": "*",
        "NotAction": [
          "... list of services used by your organization ..."
        ],
        "Condition": {
          "StringNotEquals": {
            "aws:RequestedRegion": [
              "us-east-1",
              "us-west-2"
            ]
          }
        }
      }
    ]
   }
```

This policy works by denying requests to global services unless the request is executed in an approved region. AWS identifies the region every request is executed in with the `aws:RequestedRegion` context key so you can write logic against it.

Without the ability to use services like `iam`, it's practically impossible to use a region. Allow the services used in your organization by adding them to the `NotAction` list. When you need to accept requests from across the world, you may need to enable services like CloudFront in more regions.

If the entire organization operates in the same regions, you can attach the policy to the root OU so that it applies to all accounts. When business units operate in different regions of the world, apply an appropriate SCP to each business unit's OU.

Finally, let's use SCPs to prevent known-dangerous actions.

Prevent Dangerous Actions from Being Executed

You can prevent specific dangerous actions from being executed with SCPs. For example, once you've setup an AWS account there's a good chance you don't want anyone to be able to:

- Remove or disassociate the AWS account from the AWS Organization
- Stop CloudTrail logs
- Stop or delete AWS Config recording

This **protect account** policy enforces those rules. Consider attaching a similar policy to the organizational root so it applies to all accounts.

You may want to **protect data** by denying the ability to delete RDS database clusters or DynamoDB tables. This policy might be attached only to production accounts. Or if deleting databases is only normal activity in dev, attach it to all non-development accounts.

For more SCP ideas, review the service's **examples** and the Secure Environment Accelerator's **reference policies**.

You can reduce a lot of big risks to the organization with Service Control Policy. Think through what your risks are, and what your target delivery process and architecture look like. This enables you to write and attach service control policies to the right accounts in the organization.

Summary

Scale AWS security with a strong account architecture that supports your desired delivery process and team autonomy.

Organize AWS accounts to support your organization's use cases, structure, and delivery processes. Partition major use cases into dedicated AWS accounts to create safe environments for data and workloads. Then control risk by limiting the AWS services and actions in use with Service Control Policies.

Next, **create IAM principals for people and applications**, then provision only the access they need.

Create IAM principals and provision access

With your **AWS account foundation established**, it is time to create principals that will use those accounts and provision access to necessary resources.

If you've set up network access controls for a traditional datacenter, this job will probably look significantly different. IAM principals and policies are first-class resources. They are tightly integrated into the compute platform and AWS service authorization. IAM principals are recorded in CloudTrail audit logs, and are even usable in many AWS network access controls.

In this chapter you'll learn how to model principals for people and applications, then provision access policies that match their use cases. Finally, we will establish a control loop that secures access over time.

Getting started

An AWS account's root user is too powerful for everyday use. Logging in with the root user should be done rarely, usually for 'break glass' scenarios to deal with major problems. After securing the root user, you should control access to it tightly, monitor its activity using CloudTrail, and consider restricting root's capabilities using Service Control Policies. Each account's root user is automatically identified in IAM with an ARN that looks like: `arn:aws:iam::12345678910:root`.

You will need the root user for one important task. Create an IAM user with administrative privileges to bootstrap the account, unless you already created a privileged IAM principal. Now you can create IAM roles for both people and applications, then grant access to the AWS services and data resources they need.

All AWS compute services support running workloads as an IAM role. And you can connect people's existing corporate identities to IAM roles.

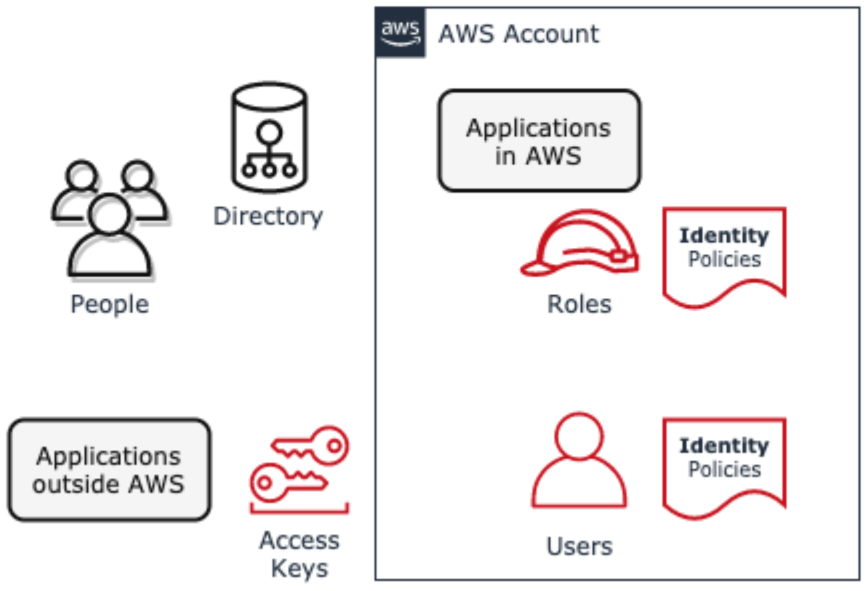


Figure 5.1 Create IAM Principals for People and Applications

EC2 compute instances, container tasks, and Lambda functions can all retrieve an identity document with short term credentials from the compute service's built-in **metadata endpoint**. Applications use these credentials to authenticate as the assigned workload's IAM role to AWS services such as S3. The **AWS SDKs**, AWS cli, and applications built for AWS use these credentials automatically as part of the standard AWS credential chain.

Generally, IAM users should only be used by applications running outside of AWS. Avoid granting permissions to IAM users directly. Instead grant access to roles and enable users to assume roles they need.

Using roles this way enables least privilege access control and only grants temporary credentials. Additionally, federated access only works with roles.

But what roles should we create?

Roles for people

First we'll need roles for people. These roles should map to a job function or job to be done.

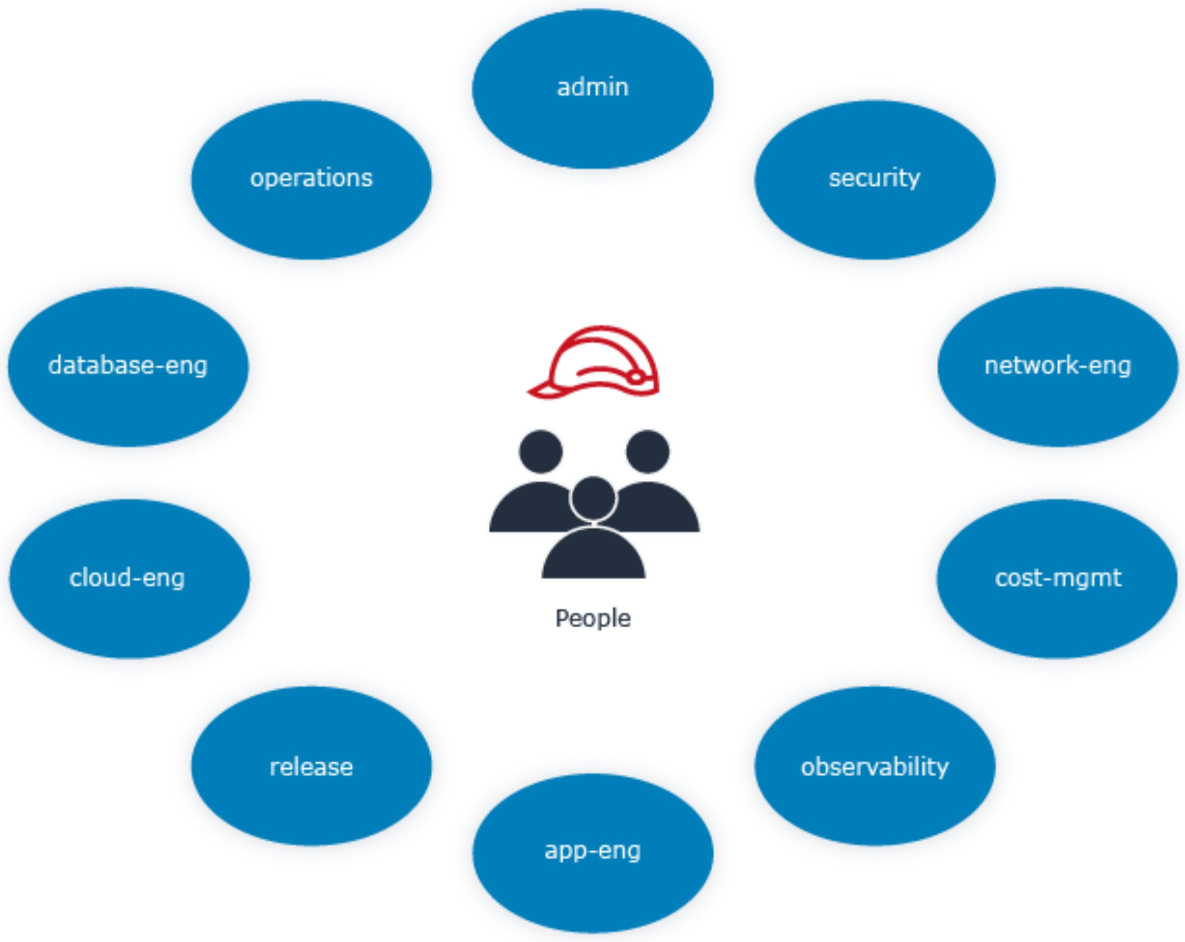


Figure 5.2 IAM roles for people

Define a common set of logical roles for people and deploy them into your AWS accounts. You will probably need a set of roles like:

- admin
- security
- operations
- network-eng
- database-eng
- cloud-eng

- release
- observability
- app-eng
- cost-mgmt

See **Appendix - IAM Role Quickstart** for these roles' responsibilities and which **Enterprise and Runtime accounts** they have access to.

When a person or group does multiple jobs, give them access to multiple roles. This helps you implement least privilege and avoid every role's permissions converging to administrator.

There are a few important things to note about these standard people roles.

First, these roles will generally exist in each account, but often have different privileges appropriate for the phase of delivery. Many roles will have more abilities in development than production. For example, application engineers may have the ability to create and delete databases in dev, but not in prod.

Second, people in different departments or business units will use the same logical role, but only in the AWS accounts for their business unit. For example, a Cloud Engineer in the Personal Banking department will only have access to the `cloud-eng` role in the Personal Banking department's AWS accounts, not the Data Warehouse's accounts. The converse is true of the Data Warehouse's Cloud engineering team; they do not have access to Personal Banking, even though they use the same logical `cloud-eng` role.

How do we link people to the right roles in our accounts? By federating corporate identities.

Federate identity for human users

Organizations must enable people to access AWS securely. You could do this by creating IAM users and access keys for everyone. But that would create a huge secret management problem and access to the web console is inconvenient. Besides, people already have trusted identities managed by their organization's corporate directory (Microsoft ActiveDirectory, Google Workspace). Instead of creating IAM users, organizations should federate access from their existing directory to AWS roles.

Organizations federate identity by linking a person's identity in the central authoritative corporate directory to other identity management systems. Federation allows people to move quickly and securely between systems — without managing another password.

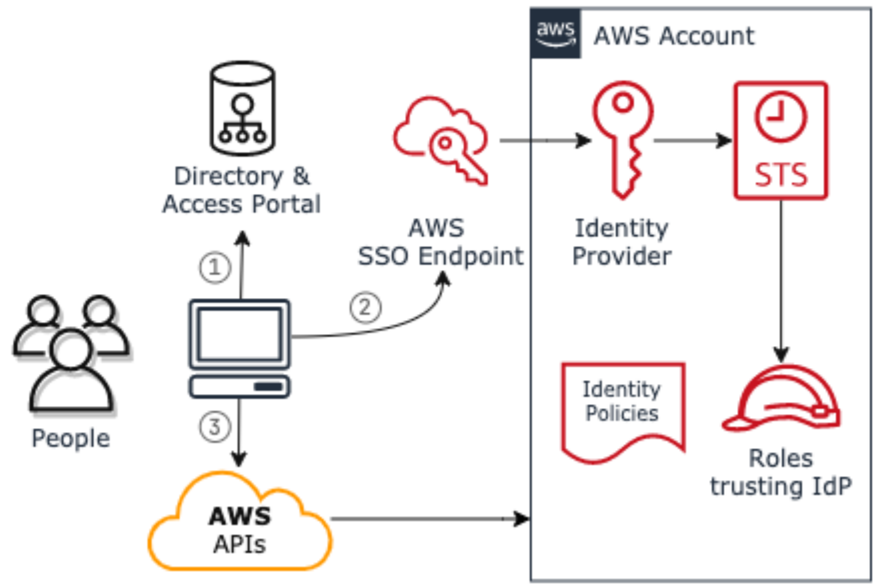


Figure 5.3 Integrate corporate identities to IAM roles using an Identity Provider

Connect your organization's existing people identities to AWS IAM roles in an account with an **identity provider (IdP)** using SAML or SCIM. The AWS account will now trust the IdP to authenticate users. Access administrators map users or groups in the corporate directory to IAM roles in AWS using the IdP. People access AWS by signing into the identity provider's access portal with their username or email address, then selecting the AWS account and IAM role they want to use.

People now have access to AWS without managing a new password or AWS access key, which would be required when using IAM users to access AWS accounts.

When integrating AWS with your identity provider, design high-integrity change workflows.

First, define a repeatable process for provisioning access for a new IAM role in the IdP. Consider defining:

- the allowed access as data in the corporate directory
- a mapping of authoritative directory groups to IAM roles in a table or other easily reviewable data structure

Second, you need a robust change process for adding and removing people from the groups that allow access to IAM roles, particularly privileged ones. Review and approve requests for access to IAM roles per organization security policy. People might initiate their request for privileged IAM roles in production by contacting the

IT helpdesk. But adding a person to a privileged group should require approval from managers of that AWS account.

Roles for applications

IAM roles for applications differ significantly from people. The specific set of application roles an organization needs for its application workloads are highly dependent on those workloads. But those workloads will fall into a few categories:

- End-user Applications
- Orchestration
- Security and Governance
- Observability

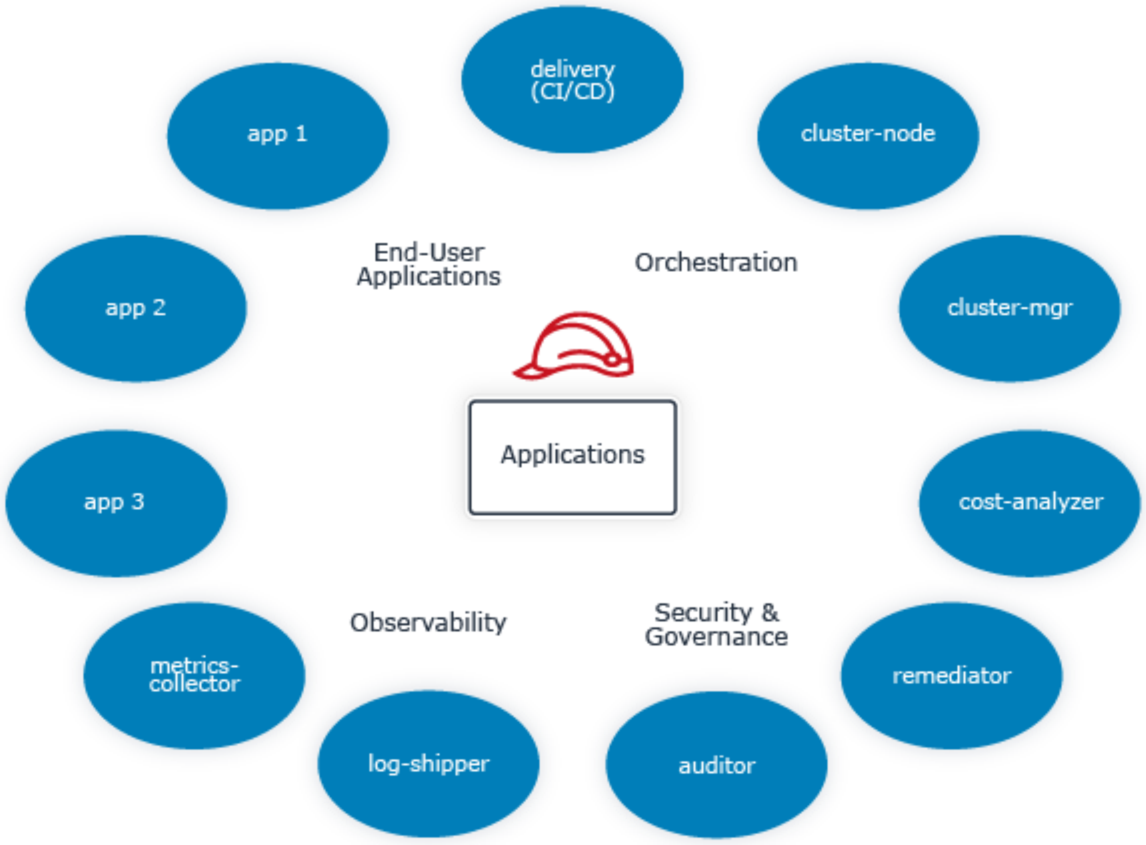


Figure 5.4 IAM roles for applications

Think of the compute instance running an application as a generic process container. In order to execute the process inside that compute container successfully, one or more programs may need to access an AWS service. The programs may need to do any of:

- load configuration from SSM Parameter Store
- read or write objects in S3
- store items in DynamoDB
- read or change AWS compute, data, or security resource configurations
- read CloudWatch metrics or logs

Almost every interaction with an AWS service requires permission to be granted using IAM. There are very few application deployments that don't need *any* IAM permissions. Even a web server deployment on EC2 serving a static website will likely need to ship its logs to S3.

Applications are programmed to accomplish specific tasks and so will require specific IAM permissions to operate correctly. These permissions generally do not vary by software delivery phase because the application's programming does not change. Only the environment the application runs in and the data it processes changes by delivery phase. And while many applications are similar to each other, it's uncommon for two applications to need exactly the same permissions, and almost never to the same data sources. This is true even for applications following the organization's standard architecture(s).

So create an IAM role for each application and phase of delivery. This includes both your applications and third-party applications running in AWS. Develop the application's Identity policy and any relevant Resource Policies *with* the application. IAM policies control whether the application will function correctly, so it's critical to develop, test and promote the application's IAM policies along with the rest of the application code. Ideally, IAM policies should be managed within the application's source repository and delivered with the application's delivery pipeline.

This approach automatically synchronizes security policy changes with the application, and avoids coordinating security policy updates via email or chat.

The applications serving end customers are not the only applications running in AWS. There are also automation applications that deploy, orchestrate, and support operations. These automation applications should also have their own distinct roles and appropriate identity policies.

Automation services often need access to powerful AWS control plane APIs or many data sources. Regular applications should never run with an automation application's principal.

Many people accidentally run containerized application services as the container cluster's EC2 instance orchestrator role. This mistake is easy to make with Elastic Container Service (ECS) because of that service's task defaults. However, running applications as the orchestrator:

1. gives all applications the ability to operate with the elevated privileges of the orchestrator, which usually has the ability to provision compute and access secrets
2. eliminates isolation *between* applications running on the cluster, enabling easy horizontal pivots between applications by an attacker

This example illustrates why it's so important to run applications with dedicated IAM roles. You'll need many IAM roles to model your applications, often more than 100 in each account. Fortunately, there is no direct cost for IAM resources. The real costs come from managing their access.

Now that we've provisioned AWS access to people and apps, we need to ensure those principals have *only* the access they need. Forever.

Secure access over time

Many organizations spend a lot of energy, time, and money implementing and monitoring access controls with little to show for it. Those organizations may have deployed several security tools and stored a lot of data in a SIEM. But they still have weak security policies and wait a long time for them.

Why is that?

Previously, we established that AWS security policies are hard to get right and difficult to validate. Those are important factors, but not the whole story.

Suppose an organization has information confidentiality requirements such as:

- Must preserve privacy of users' data
- Must preserve confidentiality of organization's intellectual property

These requirements might be collected into a 'least privilege access' goal for the organization's applications or at least applications with 'critical' data.

But no tool can directly tell you if you've implemented "least privilege" access correctly, let alone do it for you. *That tool won't have all the necessary information to make and implement a decision.*

No tool will.

Implementing and verifying least privilege access requires information spread across many people and tools. When security policy engineering processes are not well-defined or optimized, security specialists spend more time collecting and synthesizing data than analyzing and improving security policies. And people who are not AWS security experts cannot help.

Let’s fix that!

⚠️ Warning: We are not going to invent anything new here.

We *will* create a general process control loop to define, implement, and verify access controls function as intended using established best practices from Process and Safety Engineering.

So you can secure access now, and in the future, as the world changes.

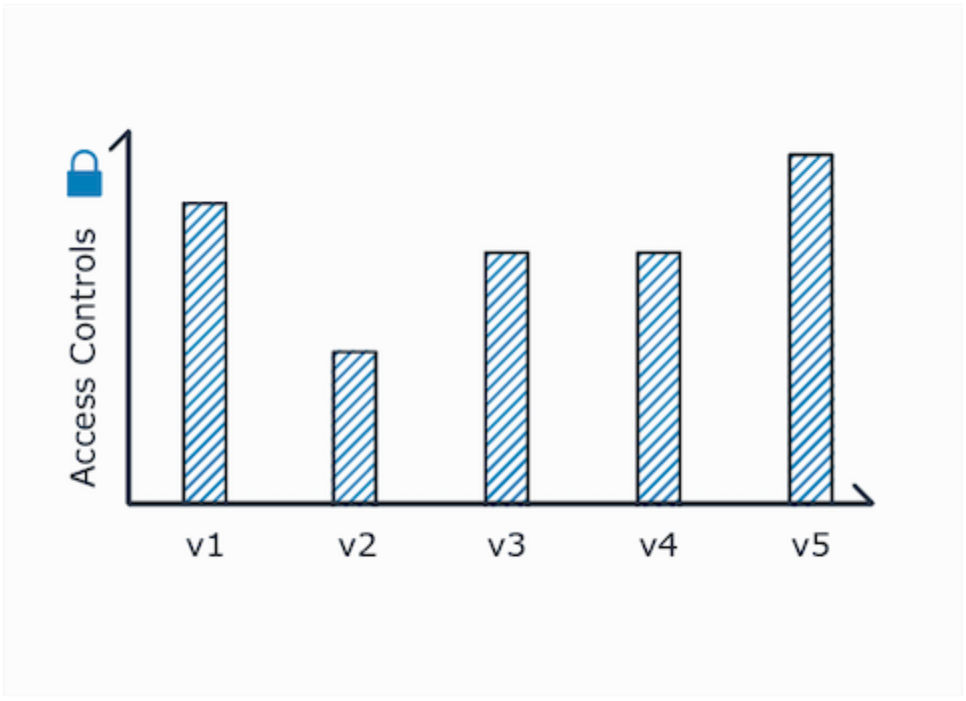


Figure 5.5 Secure access over time

Start with the high level needs of the information security control process. You need to:

1. implement initial access controls
2. update access controls as the application and business changes over time

You can do this with a process control loop that continuously converges security policies to intended access.

A real-world AWS application access control loop

Consider a `credit-processor` application that stores its data in S3. The application is the process and we want to control that process' behavior.

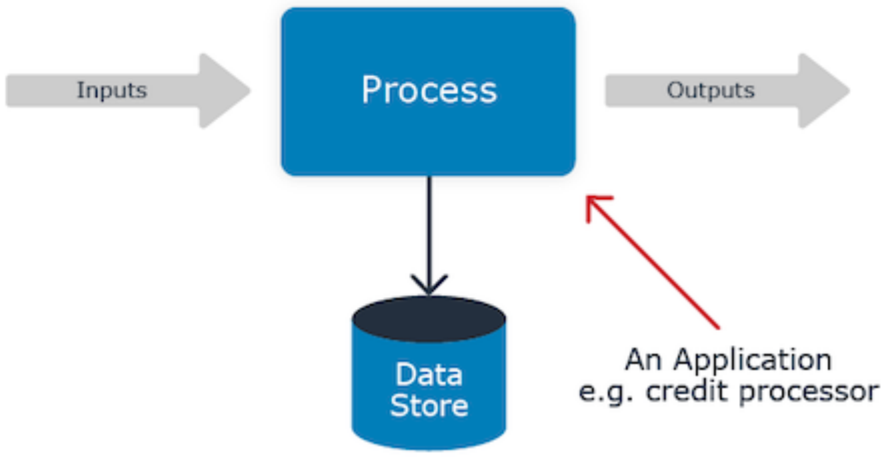


Figure 5.6 Process behavior to control

Process controllers maintain process output or operating conditions by modeling process operation, measuring process inputs and outputs, and adjusting inputs to ensure the process operates within the desired range.

Process control is pervasive:

- Thermostats control climate in buildings

- Electrical utilities monitor load and generate electricity to match
- Agile teams gather product feedback and prioritize the most important work

You can also use feedback process control to ensure information security requirements are met.

This diagram illustrates securing access to an application’s data with a simple control process that integrates feedback:

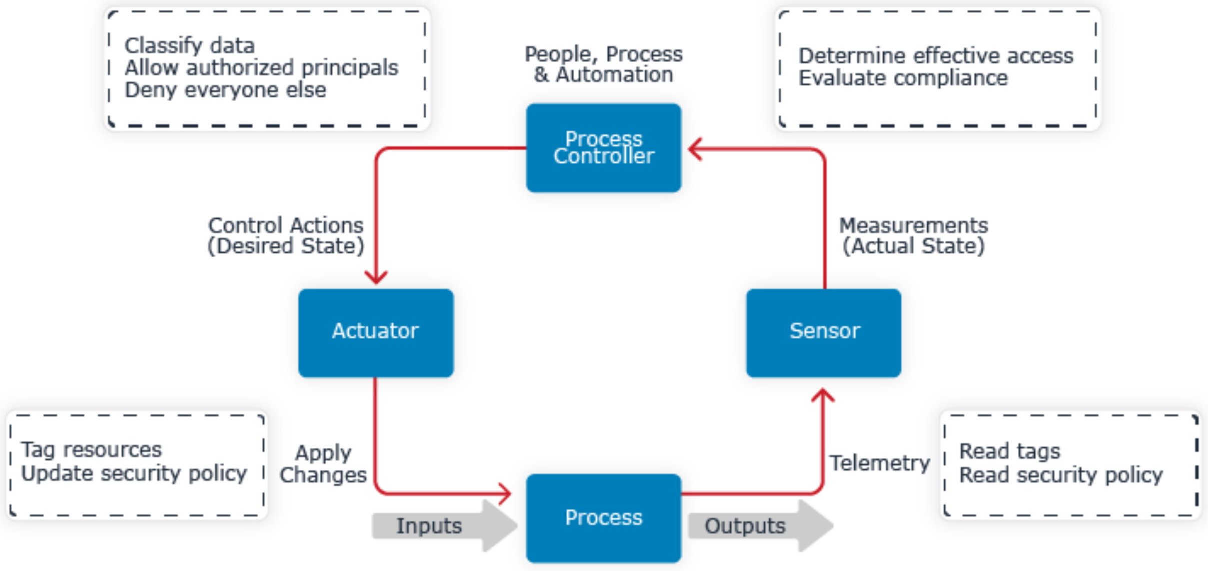


Figure 5.7 Secure data using process control with feedback

Every control process needs a process controller, actuator, and sensor. When a component is missing, faulty, or overloaded, the process will not be controlled effectively. The process *may* function properly and it *may* pass an audit (heh), but it won't be because the controls had anything to do with it.

Organizations often have elements of a control process, particularly ‘sensor’ tools that gather raw telemetry. But maybe:

- Measurements are not converted into understandable, actionable information
- People reviewing measurements don't have sufficient context to decide whether the configuration is correct
- People are overloaded by the volume of measurements
- Security controls are not updated based on the collected information

Let’s examine each component’s responsibilities and trace information through the control loop. As we step through each component in the process, think about whether:

- Your access control process has an implementation of each logical component
- Components are automated or manual
- Information flows between each component
- The process control loop completes in your team or organization

How long does it take to go around the loop in practice? A day, a week, a month, or more?

Process Controller

The first process control component is the controller. Suppose the organization implements the “least privilege access for confidential data” constraint by ensuring between 1 and 5 authorized principals have access to Confidential application data. That general constraint provides room for the application, backup processes, and Tier-3 customer service to have access to that data. That constraint would be violated if all applications had access to the sensitive application data, a common problem.

Guided by clear requirements for implementing least privilege, we can automate the policy implementation. Use infrastructure management tools like Terraform or CloudFormation to automatically:

- Classify data and compute resources
- Specify who is allowed access to the data
- Deny everyone else access

Enable engineers to make good access management decisions by adopting libraries with usable interfaces that encapsulate expert AWS security knowledge. Provide engineers with a way to specify who should have access with the information they already have: IAM principal ARNs and high level access capabilities. Use a standardized set of high level access capabilities to keep your team out of the weeds of AWS IAM and API actions. This greatly simplifies both implementing and reviewing security policy changes.

If the application's data is stored in an S3 bucket you could implement that with **k9 Security Terraform module for S3**:

```
module "s3_bucket" {
 source = "k9securityio/s3-bucket/aws"

 logical_name = "credit-applications"
 logging_target_bucket = "secureorg-logs-bucket"

 org   = "secureorg"
 owner = "credit-team"
 env   = "prod"
 app   = "credit-processor"

 confidentiality = "Confidential"

 allow_administer_resource_arns = [
     "arn:aws:iam::111:user/ci",
     "arn:aws:iam::111:role/admin"
 ]
 allow_read_data_arns = [
     "arn:aws:iam::111:role/credit-processor",
     "arn:aws:iam::111:role/cust-service"
 ]
 allow_write_data_arns = ["arn:aws:iam::111:role/credit-processor"]
}
```

This code describes the *desired* state of the credit processor application's data resources and security policies.

Engineers declare access to data in the bucket with words like `administer_resource`, `read_data`, and `write_data`. Changes to these simplified access capabilities are easy to implement and review. Reviewing the ~200 line bucket policy generated by the library much less so. In a code review, engineers and tools can see the data is tagged `Confidential` and 4 principals are allowed to access it. So the least privilege access constraint is met and our business objective achieved.

Codifying security best practice into libraries makes those practices accessible to every team and engineer.

Once the team has decided on the desired access to grant, the actuator component implements any needed changes.

Actuator

The process controller communicates the desired state of the system to an actuator. The actuator is an infrastructure management tool that knows how to examine the running system and converge it to our desired state by:

- Computing changes required to converge reality to desired state
- Applying changes to security policies and resource tags

Infrastructure management tools split computing and applying changes into two steps. This enables engineers to verify the changes are safe and will do what is desired.

Engineers can review the change plan to double-check the least privilege access constraint is met prior to applying.

Once reviewed and approved, engineers apply changes using the infrastructure management tool.

At this point several things can and do happen.

The infrastructure management tool may apply the desired changes successfully. Or it may fail, sometimes partially. Even if a change application succeeds, another actor may reconfigure the system manually in response to a production incident or to test something. A competing control plane may overwrite changes. The world is a complex place.

So while we have described how access should be configured and used a tool to implement that in the running system, we need to verify reality matches our desired state.

Sensor

Sensors gather data from the running system so that engineers can analyze its actual state. Actual systems change for many reasons, both planned and unplanned, modeled and unmodeled. We need sensors to collect data from the running system continuously so the control loop can verify constraints are met.

To analyze access control, sensors may read low level telemetry like:

- The actual security policies
- The actual tags on the resource
- A log of what access has been used recently
- A system probe's report of whether access is allowed
- Differences between actual and desired state defined by infrastructure code

The sensor can transmit this raw telemetry directly to the process controller for evaluation. For the `credit-processor` application the sensor could report the actual policies and tags. But then the process controller needs to compute access itself. As of mid-2021, there are no native AWS tools that sense and report what access IAM principals in an account have to an S3 bucket. The closest options are:

- AWS Access Analyzer which will generate findings for buckets that are externally accessible
- AWS Access Analyzer last used service will tell you which S3 actions an IAM principal last used
- CloudTrail logs when a principal accessed a bucket via a control API or data API (optional)

Alternatively, a sensor may compute higher level measurements that are easier to understand and send that to the process controller.

An example of a higher level measurement is the effective access IAM principals have to data and API actions. Here's how k9 Security reports the effective access to the credit applications bucket:

Service Name	Resource ARN	Access Capability	Principal Name	Principal Type
S3	arn:aws:s3:::secureorg-credit-applications	administer-resource	admin	IAMRole
S3	arn:aws:s3:::secureorg-credit-applications	administer-resource	ci	IAMUser
S3	arn:aws:s3:::secureorg-credit-applications	read-data	credit-processor	IAMRole
S3	arn:aws:s3:::secureorg-credit-applications	read-data	cust-service	IAMRole
S3	arn:aws:s3:::secureorg-credit-applications	write-data	credit-processor	IAMRole

Table 5.1: Example of high level measurement of effective access to an S3 bucket

These measurements are intended for direct consumption by a process controller. The report clearly shows:

1. the access capabilities that the `admin`, `ci`, `cust-service`, and `credit-processor` principals have
2. that no unexpected users or roles have access

People acting as process controllers find this information much easier to interpret than analyzing policies in their heads, and the result is far more accurate.

Closing the loop

Now let's close the control loop. The people and tools that implement the process controller receive the sensor's information and evaluate compliance with the "least privilege access" constraint. Then the controller decides if additional control actions are needed.

If the constraint is violated it could be that too many principals were given access, the data source supports multiple use cases, or another issue.

To secure access over time, you'll need to execute the control loop frequently enough to keep pace with changes in the environment and security objectives.

Organizations should adopt standardized models, processes, and self-service tools that help all teams implement common constraints like least privilege access. This includes:

- Classifying data sources according to a standard so everyone has the context they need with, e.g. a tagging guide and infrastructure code support for that tagging standard
- Simplifying access capability models for declaration and analysis

- Generating secure policies using infrastructure code libraries within delivery pipelines
- Analyzing effective access principals have to API actions and data
- Notifying teams when the actual state of access does not match the desired state

These tools help teams implement and execute the access control loop easily and autonomously. Teams can declare the access they intend, detect when reality drifts, and address issues themselves using their local context. This improves and scales information security while unburdening security specialists from repetitive work.

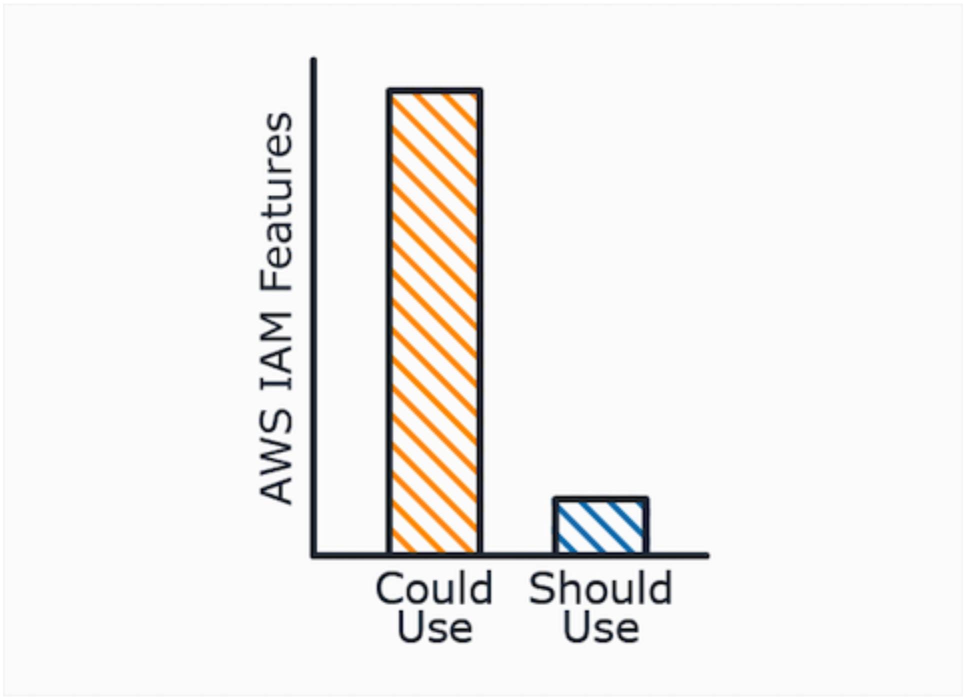
Summary

Create IAM roles for people and applications. Avoid IAM users and credential management where possible. Run applications using IAM roles with least privilege Identity & Resource policies, ideally defined and delivered by the application's automated delivery process. Connect people to IAM roles for their job function in AWS using their existing corporate identity.

Then secure access over time with a process control loop that repeatably defines, applies, and reviews access policies. Focus on the most important controls. Support people's decision making with automation so that you can make the most of people's context, attention, and skills. When you provide process controllers digestible and actionable information, the control loop can execute frequently, and your organization can detect and resolve hazard conditions quickly.

Next you'll learn how to simplify access control implementation by packaging the **best, highest leverage parts of AWS IAM** into secure building blocks.

Simplify AWS IAM by using the best parts



Less is more.

Simplify AWS security by using the best parts of IAM consistently. That's how you realize AWS IAM best practices at scale. You'll never be able to understand all of IAM. But you don't need to. AWS IAM's core capabilities can satisfy nearly all information security requirements. Focus your effort. Put AWS IAM's core capabilities into usable packages and patterns for developers. Then leverage those capabilities across your organization.

Your approach must work in the messy world of existing AWS deployments. The real world is full of AWS accounts that:

- grew organically, perhaps starting with an experiment and now running production
- operate software produced by internal and external teams with varying AWS skills
- use AWS managed policies
- use AWS managed data services

Many organizations' most critical AWS accounts look like a kitchen sink full of dirty dishes. You can't just start over. And the cleanup procedure isn't in the AWS IAM documentation.

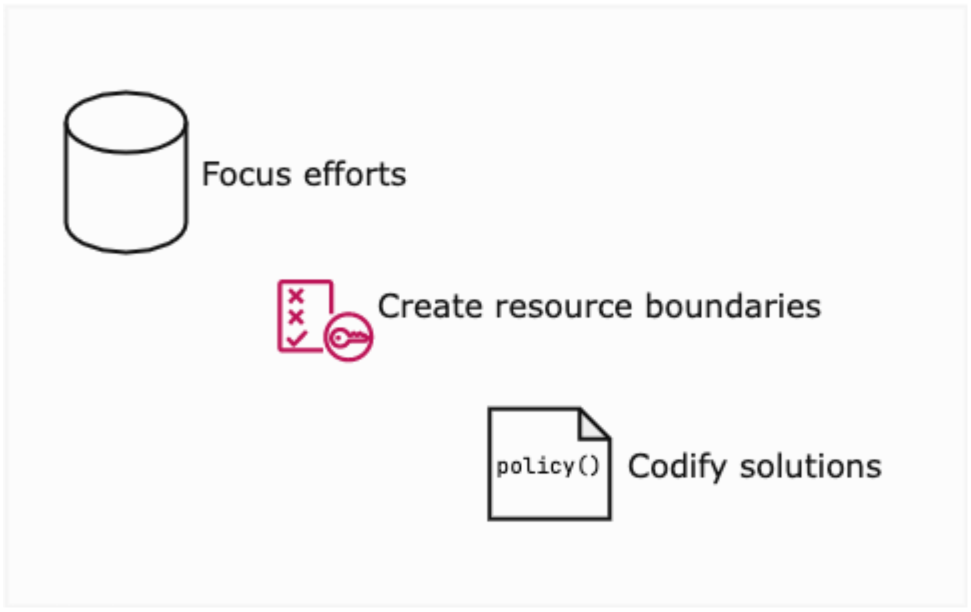
How will you get this under control?

Adopt a security strategy that helps you clean up your accounts incrementally. Start by securing your most critical data with resource boundaries — a simple, scalable pattern.

Points of Leverage

Consistent abstractions provide leverage. When a feature works the same way across the environment, you can use that feature as a foundational building block in your security architecture. However, AWS is famously built by many independently operating teams that ship minimal services early. So most services are designed and behave a little bit differently than other services.

So what levers do you have to simplify access control?



First, focus access control efforts by organizing each use case's data into AWS accounts, then select the IAM context you will leverage in policies.

Second, use resource policies to control access to critical data within an account.

Third, use infrastructure code libraries and tools to simplify and scale IAM configuration.

Focus access control efforts

Organize and separate major use cases from each other using the approach described in **'Architect AWS organization for scale'**.

If you have a single AWS account running multiple use cases that you need to separate (dev, stage, and prod for an app or multiple business units), then start planning a migration.

It's usually easiest to migrate non-production workloads to new accounts. Yes, migrating workloads out of an AWS account is often a major endeavor. But it regularly succeeds and simplifies security over the long run. Don't worry. After isolating use cases into dedicated AWS accounts, you can explicitly grant access between workloads using resource policies.

Tune access control focus by selecting a handful of policy condition context keys you will use to authorize requests consistently. AWS populates each API request with relevant context. The request context and policy condition keys enable more flexible policies. They also allow fine-grained control over requests originating both inside and outside your account or organization.

For example, instead of requiring an exact match of an AWS account id, IAM user arn, or role arn, you can use wildcards to match identities according to a convention, even when they don't exist yet.

```
{
 "Condition": {
   "ArnLike": {
     "aws:PrincipalArn": [
       "arn:aws:iam::111:user/admin",
       "arn:aws:iam::222:role/delivery",
       "arn:aws:iam::333:role/auditor-*"
     ]
   },
   "StringEquals": {
     "aws:PrincipalOrgID": [
       "o-abc1234"
     ]
   }
 }
}
```

Each of the condition operations (`ArnLike`, `StringEquals`) must be satisfied for the request to proceed. When a context key like `aws:PrincipalArn` specifies multiple values, then a request with any of those values satisfies the condition operation (**details**).

This `Condition` block explicitly identifies the admin user in account `111` and delivery role in account `222`. It also identifies any roles whose name begins with auditor in account `333`. Those principals' requests are *only* allowed if they belong to the AWS organization id `o-abc1234`.

These are the most useful **global condition context keys**:

1. `aws:PrincipalArn` (string): Compare the principal ARN that made the request with the ARN or pattern that you specify in the policy.
2. `aws:PrincipalIsAWSService` (boolean): Check whether the call to your resource is being made directly by an AWS service principal.
3. `aws:PrincipalOrgId` (string): Compare the requesting principal's AWS Organization id to an expected identifier in the policy.
4. `aws:PrincipalOrgPaths`(string): Compare the requesting principal's AWS Organizations path to an expected path in the policy.
5. `aws:SourceVPC` (string): Check whether the request came from a VPC specified in the policy.

These context keys enable you to identify AWS IAM and Service principals, what organization or OU they belong to, and if the request is coming from a particular network. These context keys solve most of the authentication problems policy authors face.

Let's discuss an approach that simplifies access control to data in a scalable way: Resource Policies.

Control access to data with resource policies

Use resource policies to control access to critical data within an account. Resource policies are ideal for establishing a **resource boundary (pdf)**. A resource boundary controls how principals may use a specific resource and where those principals access data from. Because resource policies integrate directly into data access paths you can create resource boundaries easily.

There are few truly critical data sources in most organizations. Create resource policies that enforce least privilege access to those resources. This approach improves security posture quickly.

Most resources commonly shared between AWS accounts support resource policies: S3 buckets, SQS queues, Lambda functions, SSM parameters, and more. Critically, KMS encryption keys support resource policies.

You can leverage resource policies in two ways:

1. Control access to data by applying resource policies directly to the resource
2. Control access to data by encrypting it with a KMS key and controlling access to (just) the key

We used the first approach in **'Control access to any resource'**. We protected a bucket containing critical data with a least privilege bucket policy. That S3 bucket resource policy:

- Allowed authorized principals to administer the bucket and access data
- Denied unauthorized principals access to the bucket
- Denied insecure access and storage by enforcing encryption in-transit and at-rest

But the second approach of encrypting data with KMS and controlling access to the key is more general, and provides you more leverage.

Secure data in AWS with Key Management Service (KMS)

This section explains core KMS concepts then shows how to encrypt your data to scale access management uniformly across disparate data services.

AWS Key Management Service (KMS) is a fully managed encryption API and key vault. KMS provides encryption operations like encrypt, decrypt, sign, and verify as APIs for use by your applications and by other AWS services. When you encrypt data in AWS data services like S3 or SQS, KMS is the service that safely performs those operations on your behalf. **KMS integrates with more than 65 AWS services**. Most engineers' first encounter with KMS is to satisfy a requirement to encrypt data at rest, but you can take it much further.

Core KMS Concepts & Resources

Encryption keys are the foundational resource of KMS, and are called customer master keys (CMKs). A customer master key resource contains secret material used by encryption and decryption operations. A CMK also has metadata that identifies the key and a lifecycle to help you manage it.

KMS supports two types of customer master keys:

- **AWS managed CMK**: a key created automatically by AWS when you first create an encrypted resource in a service such as S3 or SQS. You can track the usage of an AWS managed CMK, but the lifecycle and permissions of the key are managed on your behalf.
- **Customer managed CMK**: a key created by an AWS customer for a specific use case. Customer managed CMKs give you full control over the lifecycle and permissions that determine who can use the key.

Some people are satisfied by encrypting data with an AWS managed CMK. An AWS managed CMK is used when you tell S3 or another data service to encrypt your data, but you don't specify a key. These keys have aliases with friendly names like `aws/s3`. Indeed this does encrypt your data at rest. This encryption protects your data from such threats as an AWS datacenter employee ripping a drive out of a server and walking out the door. But this is probably not the information security threat you should worry about.

Encrypting data with AWS managed CMKs *does not* prevent a person or application with access to your account from reading that data using AWS APIs. You cannot control which IAM principals within the account use these default keys. **AWS-managed key resource permissions are managed on your behalf**. You cannot change those permissions. All the account's IAM principals can encrypt and decrypt data with AWS managed CMKs as long as the IAM principal can access the datastore. AWS accomplishes this with an uneditable key resource policy statement like:

```
{
 "Sid": "Allow access through S3 for all principals in the account that
are authorized to use S3",
 "Effect": "Allow",
 "Principal": {
   "AWS": "*"
 },
 "Action": [
   "kms:Encrypt",
   "kms:Decrypt",
   "kms:ReEncrypt*",
   "kms:GenerateDataKey*",
   "kms:DescribeKey"
 ],
 "Resource": "*",
 "Condition": {
   "StringEquals": {
     "kms:ViaService": "s3.us-east-1.amazonaws.com",
     "kms:CallerAccount": "111"
   }
 }
}
```

This statement from an `aws/s3` key policy allows anyone in account `111` to decrypt or encrypt data using the key as long as the request came via S3.

A customer managed CMK is different. You **create customer managed CMKs and control who may use them**. Now you can control access to data for any specific use case. Create a key for the use case. Then configure the key's resource access policy for that use case's requirements. Finally encrypt data with the key.

Let's dive into the uniquely powerful world of key resource policy.

KMS Key Resource Policy

KMS key policies behave very differently than other resource policies even though they look the same.

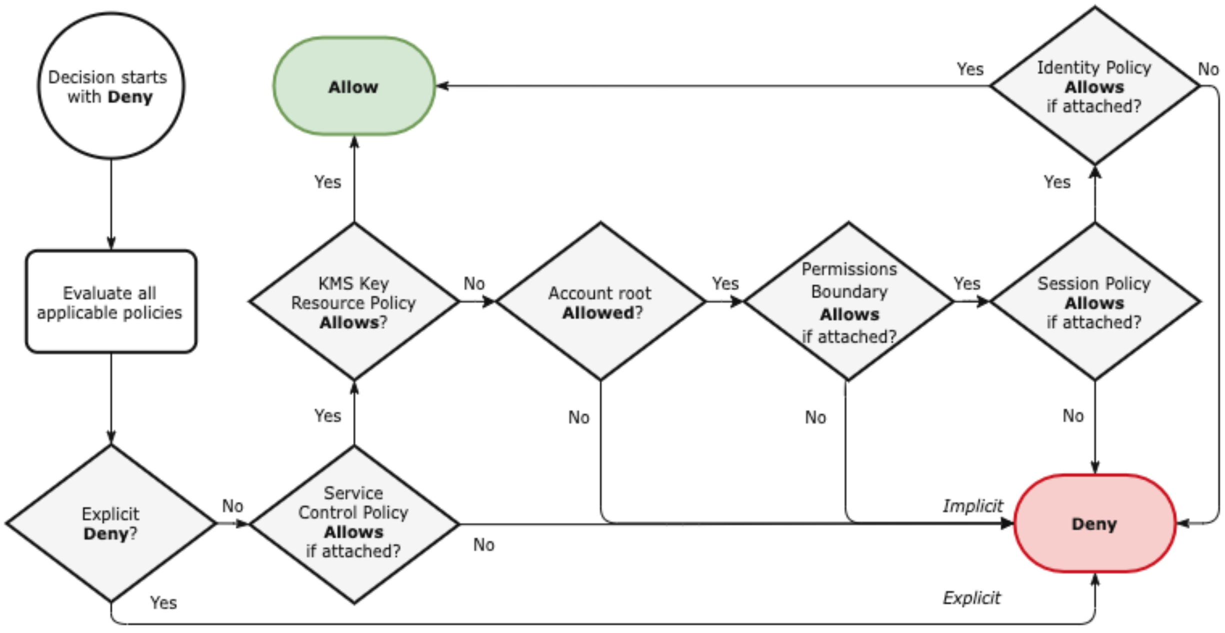

Decision starts with **Deny**
Evaluate all applicable policies
Explicit **Deny**?
No
Yes
Service Control Policy **Allows** if attached?
Yes
No
KMS Key Resource Policy **Allows**?
Yes
No
Allow
Account root **Allowed**?
Yes
No
Permissions Boundary **Allows** if attached?
Yes
No
Session Policy **Allows** if attached?
Yes
No
Identity Policy **Allows** if attached?
Yes
No
Implicit
Explicit
Deny

Figure 6.1 KMS Key Access Evaluation Logic

Every KMS key must have a key policy. Critically, that single key policy is the root of control for all access to the key. Key policy is the primary way to control access to a key and **introduces an exception to the standard security policy evaluation flow**.

⚠️ *The account's identity policies are only integrated into the policy evaluation flow when the key policy allows access to the account's root user.*

An account's root user cannot access a key unless explicitly allowed by the key policy. Consequently, the account's IAM principals can only access the key when explicitly granted by the key policy or access is granted to the root user and the account's identity policies grant access to the key.

This access evaluation behavior diverges from IAM's standard approach that allows access via *either* identity or resource policies. But it arguably makes implementing least privilege easier.

Let's examine the default key policy since there are many keys with this policy. The AWS console, CLI, SDKs, and CDK all generate a default key policy for customer managed CMKs. There are several variations of the default key policy, but they are effectively the same. The default key policies allow access to the root user and enable identity policies. They look like:

Example: Default Key Policy for a Customer Managed Key

```
{
 "Version": "2012-10-17",
 "Id": "DefaultKeyPolicy",
 "Statement": [
   {
     "Sid": "Allow Root User to Administer Key And Identity Policies",
     "Effect": "Allow",
     "Principal": {
       "AWS": "arn:aws:iam::123456789012:root"
     },
     "Action": "kms:*",
     "Resource": "*"
   }
 ]
}
```

The default key resource policy:

- explicitly allows the AWS account full access to administer and use the key
- implicitly allows use of the key by any IAM principals permitted by identity policies via the root user
- does not `Deny` any IAM principals the ability to use the key

This is effectively the same level of access control as an AWS managed key. All the account's IAM principals with broad KMS permissions granted by identity policies can use the key.

Suppose you wanted to narrow this key's usage to only the `appA` IAM role and `ci` IAM user. You can do that in two ways:

1. Grant access to *only* those principals in the key policy, but not root
2. Grant access to root, but restrict the IAM principals that can use the key

Let's examine both approaches.

Grant access solely via key resource policy

First, you can grant access to only the `ci` and `appA` IAM principals in the key policy, omitting account root user access:

Example: Control access solely via key policy

```
{
 "Version": "2012-10-17",
 "Id": "AllowAccessOnlyToSpecificIAMPrincipals",
 "Statement": [
   {
     "Sid": "AllowFullAccess",
     "Effect": "Allow",
     "Action": "kms:*",
     "Resource": "*",
     "Principal": {
       "AWS": [
         "arn:aws:iam::123456789012:user/ci"
       ]
     }
   },
   {
     "Sid": "AllowReadAndWrite",
     "Effect": "Allow",
     "Action": [
       "kms:Decrypt",
       "kms:Encrypt",
     ],
     "Resource": "*",
     "Principal": {
       "AWS": [
         "arn:aws:iam::123456789012:role/appA"
       ]
     }
   }
 ]
}
```

Only the `ci` and `appA` principals will be able to use this key or delegate access to it. And they can only use the permissions allowed by the key policy. So only the `ci` user can manage this key and its access policy.

The account's root user *does not have access*. Consequently:

- the Identity policy system is not enabled, so Identity policies cannot allow additional access
- the root user cannot access the key, not even to change the key policy or manage the key's lifecycle

Finally, least privilege!

But perhaps this is too little privilege in practice. If the `ci` and `appA` principals are removed from IAM, the key will become unmanageable and data encrypted with this key inaccessible. You can file a ticket with AWS Support to restore access to the key. However, you will not have access to data encrypted with this key until access is restored. AWS' default key policy makes sense with this context.

But there is another path to least privilege that is more flexible.

Grant access to account and narrow via key resource policy

You can enable account root access and identity policies then use key resource policy to narrow access to specific IAM principals. Consider this `NarrowerKeyPolicy`:

Example: Narrow Key Policy

```
{
 "Version": "2012-10-17",
 "Id": "NarrowerKeyPolicy",
 "Statement": [
   {
     "Sid": "AllowFullAccess",
     "Effect": "Allow",
     "Action": "kms:*",
     "Resource": "*",
     "Principal": {
       "AWS": "*"
     },
     "Condition": {
       "ArnEquals": {
         "aws:PrincipalArn": [
           "arn:aws:iam::123456789012:role/appA",
           "arn:aws:iam::123456789012:user/ci"
         ]
       }
     }
   },
   {
     "Sid": "DenyEveryoneElse",
     "Effect": "Deny",
     "Action": "kms:*",
     "Resource": "*",
     "Principal": {
       "AWS": "*"
     },
     "Condition": {
       "ArnNotEquals": {
         "aws:PrincipalArn": [
           "arn:aws:iam::123456789012:root",
           "arn:aws:iam::123456789012:user/ci",
           "arn:aws:iam::123456789012:role/appA"
         ]
       },
       "Bool": {
         "aws:PrincipalIsAWSService": "false",
         "kms:GrantIsForAWSResource": "false"
       }
     }
   }
 ]
}
```

The `NarrowerKeyPolicy`'s `AllowFullAccess` statement permits full access by the `appA` and `ci` principals. This policy also denies access to all other principals. The `DenyEveryoneElse` statement closes off unintended access granted by overly-permissive Identity policies attached to principals in the account. The **`aws:PrincipalIsAWSService`** condition ensures the deny does not apply to AWS services like DynamoDB and S3 so they can use the key on your behalf. Similarly, the **`kms:GrantIsForAWSResource`** condition permits AWS services like RDS to maintain KMS key grants, a form of delegated access.

This policy takes a step towards least privilege. It is also more flexible, identifying principals with the `Arn` condition operators.

You may want to refine the policy further. The application's `appA` role probably does not need to administer the key. You could create a statement that only gives `appA` privileges to encrypt and decrypt data. You might also only allow `ci` to administer the key. That key policy could look like the one generated to protect the bucket in **'Control access to any resource'**.

Consider the least privilege key policy that the k9 Security libraries for **Terraform** and **CDK** generate:

Example: Least Privilege Key Policy

```
{
 "Version": "2012-10-17",
 "Id": "LeastPrivilegeKeyPolicy",
 "Statement": [
   {
     "Sid": "AllowRestrictedAdministerResource",
     "Effect": "Allow",
     "Action": [
       "kms:CreateAlias",
       "kms:CreateKey",
       "kms:ScheduleKeyDeletion",

       # omit 17 Administration actions

     ],
     "Resource": "*",
     "Principal": {
       "AWS": "*"
     },
     "Condition": {
       "ArnEquals": {
         "aws:PrincipalArn": [
           "arn:aws:iam::123456789012:user/person1",
           "arn:aws:iam::123456789012:user/ci"
         ]
       }
     }
   },

   # omit Read Config statement

   {
     "Sid": "AllowRestrictedReadData",
     "Effect": "Allow",
     "Action": [
       "kms:Decrypt",
       "kms:Verify",
     ],
     "Resource": "*",
     "Principal": {
       "AWS": "*"
     },
     "Condition": {
       "ArnEquals": {
         "aws:PrincipalArn": [
           "arn:aws:iam::123456789012:user/person1",
           "arn:aws:iam::123456789012:role/appA"
         ]
       }
     }
   },
   {
     "Sid": "AllowRestrictedWriteData",
     "Effect": "Allow",
     "Action": [
       "kms:Encrypt",
       "kms:GenerateDataKey",
       "kms:Sign"

      # omit 7 Write actions
     ],
     "Resource": "*",
     "Principal": {
      "AWS": "*"
     },
     "Condition": {
       "ArnEquals": {
         "aws:PrincipalArn": [
           "arn:aws:iam::123456789012:user/person1",
           "arn:aws:iam::123456789012:role/appA"
         ]
       }
     }
   },

  # omit unused Delete Data and Custom capability statements

  {
     "Sid": "DenyEveryoneElse",
     "Effect": "Deny",
     "Action": "kms:*",
     "Resource": "*",
     "Principal": {
       "AWS": "*"
     },
     "Condition": {
       "ArnNotEquals": {
         "aws:PrincipalArn": [
           "arn:aws:iam::123456789012:root",
           "arn:aws:iam::123456789012:user/person1",
           "arn:aws:iam::123456789012:user/ci",
           "arn:aws:iam::123456789012:role/appA"
         ]
       },
       "Bool": {
         "aws:PrincipalIsAWSService": "false",
         "kms:GrantIsForAWSResource": "false"
       }
     }
   }
 ]
}
```

The `LeastPrivilegeKeyPolicy` allows `ci` to administer the key, `appA` to read and write data with the key, and `person1` to administer, read, and write. The `DenyEveryoneElse` statement denies access to everyone but those principals and the root user. **Appendix Least Privilege Key Policy** contains the full 175-line policy, including actions and unused statements omitted for brevity.

Use the general form of the `LeastPrivilegeKeyPolicy`'s Allow and Deny statements across all KMS keys and other resources that support policies.

- Allow access capabilities to explicitly-named IAM principals
- Deny everyone else, except for AWS Service principals and service-linked roles

Designing and implementing resource policies from scratch takes a lot of time, energy, and expertise. Reuse known-good solutions to ship quickly with confidence.

Now let’s leverage key policy to control access to multiple data stores.

Pattern: Partition Data with Encryption

Here’s where access control gets exciting (no, really). You can think of KMS resource policy as a composable access control that is usable from any other AWS data service.

> KMS is a composable, scalable, uniform mechanism for controlling access to data across data stores.

Once you learn how to use KMS resource policies effectively, **you can use encryption keys and key resource policies as a uniform core access control mechanism for all AWS data services**.

By managing access to a handful of critical encryption keys, you can worry less about the tens of data stores or hundreds (thousands?) of principals in your AWS account.

Leverage.

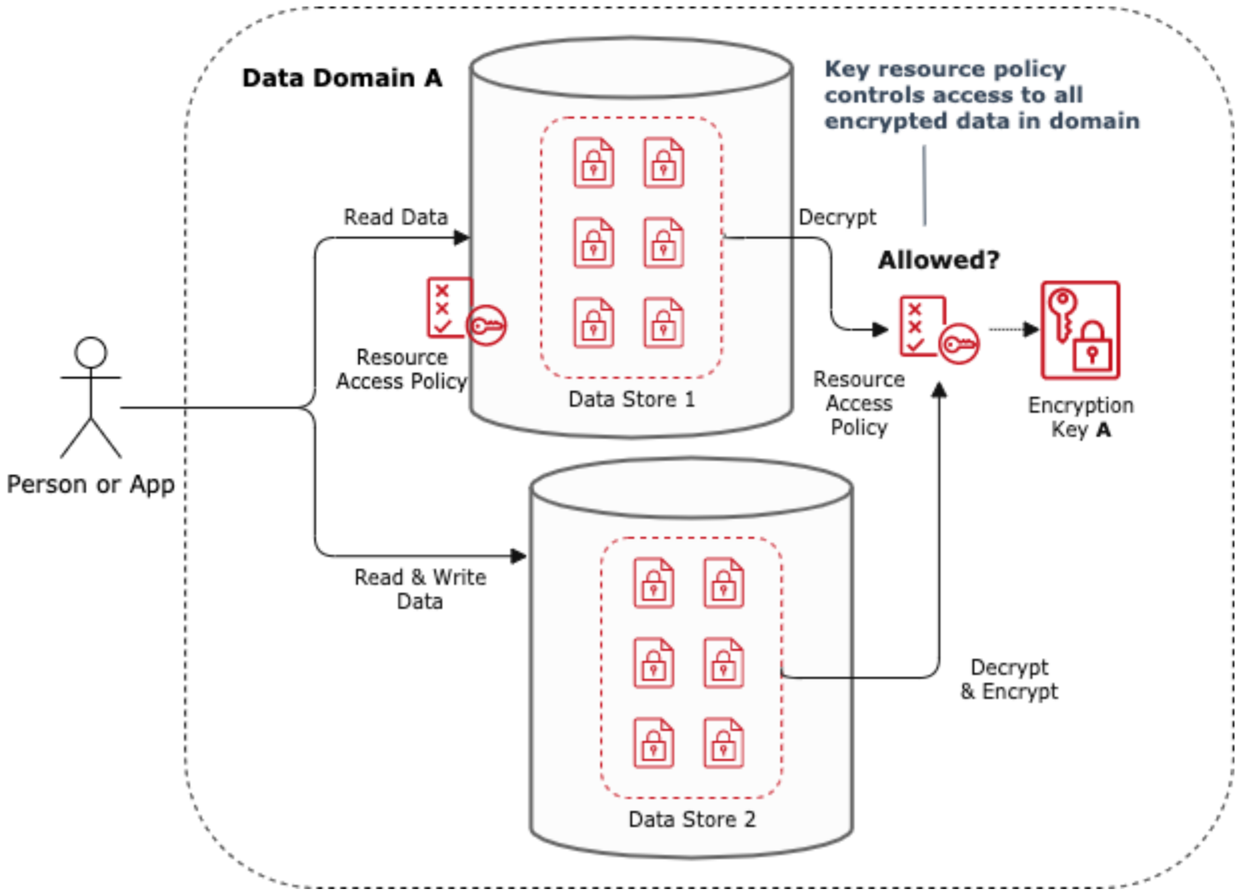


Figure 6.2 Control access to entire data domain with KMS

The Partition Data with Encryption pattern partitions and controls access to an entire data domain. This security architecture pattern encrypts that domain's data with a dedicated customer managed KMS encryption key and applies a key resource access policy that is appropriate for the people and applications in that domain. A data domain encompasses the data managed by a single service or collaborating set of microservices such as 'credit-applications' or 'orders' (c.f. **Bounded Context**). This pattern adds resource access policy capabilities into data services without native support for resource policies such as Elastic Block Store and Relational Database Service.

This pattern helps you:

- Manage data access uniformly across disparate data services
- Simplify architecture, development, and audit processes with a consistent access control strategy
- Reduce access policy language knowledge and implementation requirements
- Explicitly control access to data even when the AWS data service does not support resource policies

How it works

In order for a person or application to read and write data to an AWS data store, their IAM principal must also have access to use the KMS key needed for those operations. This makes customer managed encryption keys a convenient point of access control.

For example, when an IAM user reads encrypted data from S3 using the `s3:GetObject` API, the user also needs permission to call `kms:Decrypt` with the key used to encrypt the data. In this scenario, the S3 service invokes `kms:Decrypt` on the IAM user's behalf. If the user has permission to call `kms:Decrypt` with the required key, then the `s3:GetObject` operation can proceed. Without decrypt permissions for the key, `s3:GetObject` will fail.

The KMS API has a small set of actions relevant to reading and writing data via another service:

- Read data: `Decrypt`

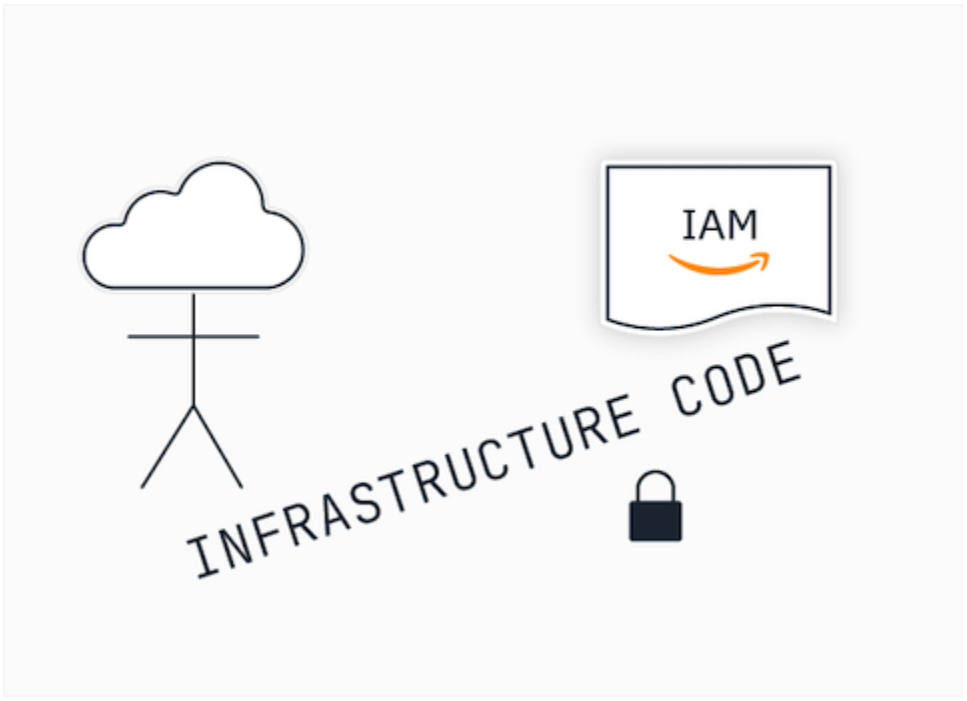
- Write data: `Encrypt`, `GenerateDataKey`, `GenerateDataKeyPair`, `GenerateDataKeyPairWithoutPlaintext`, `GenerateDataKeyWithoutPlaintext`, `ReEncryptFrom`, `ReEncryptTo`

You can control access to an entire data domain by encrypting that data with a dedicated CMK and applying a key policy that allows intended access to the key's read, write, and administration operations.

The Partition Data with Encryption pattern helps you scalably control and audit access to each domain's data with a resource boundary. The key resource policy clearly identifies who and what kind of access is allowed to the domain's data. This key policy is better positioned to prevent unauthorized access than getting every related data service's resource policy and every identity policy correct. Identity policies attached to IAM principals can be treated as controls that enable authorized access to services rather than prevent unauthorized access to data.

Hopefully you're convinced resource policies provide a warm, weighted security blanket for your data. But those policies are complicated and are not going to write themselves. Let's deal with that.

Simplify and scale IAM configuration with usable automation



Leverage infrastructure code to elevate your AWS security posture.

Security policies are often complex and nuanced, but perhaps no more so than other Cloud resources such as databases and compute clusters. We don't want to think through the details of any of these on every change.

That's why it's useful to package expert knowledge into reusable libraries and tools that:

1. provide a simple interface that non-experts understand
2. generate great security policies
3. integrate well with your delivery process
4. unload Cloud security specialists and accelerate delivery

Writing secure IAM policies is an exacting, tedious practice. Everything needs to be *just* so. But we've already established a general form for secure policies, even least privilege ones. A lot of them look the same.

We can generate sophisticated security policies with automation. But only if that automation is usable by the engineers that need to use it.

Usability is necessary to scale security.

So what does a usable policy generator interface look like? (examples in a moment)

First, specify who the user of this generator is.

The target user is every engineer in your org who will likely need to secure a resource within their control. Yes, explicitly target non-experts in AWS cloud security.

These generators need to work for regular application, cloud, site reliability, and security engineers, not only cloud security specialists.

Enable these engineers to create great security policies, so cloud security specialists can focus on other things like architecting and creating secure building blocks like secure policy generators. Empowering non-experts with safe tools and effective training is the most effective way to scale security when using automated delivery.

Second, simplify the interface to AWS IAM.

Application engineers and collaborating colleagues like SREs usually have the best knowledge of who needs access to data and which service API capabilities their application needs. But people are easily overwhelmed by the tens or hundreds of API actions in each relevant AWS service.

8 hours of research to create a policy or 2 hours to review isn't going to work. Access management needs to make sense and be pretty darn secure right away.

Engineers need to be able to read and write the intended access capability in plain language, quickly and safely.

Simplify defining intended access so engineers can choose from a small set of consistent categories non-experts understand such as:

- administer resource
- read config
- read data
- write data
- delete data

Use no more than seven categories. This enables users to choose the appropriate categories using their imprecise knowledge[1].

Some examples of simplified IAM access models are:

- **AWS Access Levels**: List, Read, Write, Permissions Management, Tagging
- **k9 Security Access Capabilities**: administer-resource, read-config, read-data, write-data, delete-data, use-resource

These are the types of categories non-experts should use to declare their intended access to AWS APIs and specific resources.

With a simplified access model, engineers can declare intended access accurately and quickly.

Let's see what this looks like as a standalone tool and a composable infrastructure code library.

Standalone tool for usable policy generation: Policy Sentry

Policy Sentry (by Salesforce) is a great example of a standalone tool that simplifies least privilege identity policy generation. Engineers specify intended access in a configuration file using the AWS Access Level model. This configuration declares an identity policy granting access to secrets in AWS Systems Manager and Secrets Manager:

```yaml
mode: crud
read:
- 'arn:aws:ssm:us-east-1:123456789012:parameter/myparameter'
write:
- 'arn:aws:ssm:us-east-1:123456789012:parameter/myparameter'
list:
- 'arn:aws:ssm:us-east-1:123456789012:parameter/myparameter'
tagging:
- 'arn:aws:secretsmanager:us-east-1:123456789012:secret:mysecret'
permissions-management:
- 'arn:aws:secretsmanager:us-east-1:123456789012:secret:mysecret'
```

When you run `policy_sentry`, it generates a policy like:

```
{
   "Version": "2012-10-17",
   "Statement": [
       {
           "Sid": "SsmReadParameter",
           "Effect": "Allow",
           "Action": [
               "ssm:GetParameter",
               "ssm:GetParameterHistory",
               "ssm:GetParameters",
               "ssm:GetParametersByPath",
               "ssm:ListTagsForResource"
           ],
           "Resource": [

"arn:aws:ssm:us-east-1:123456789012:parameter/myparameter"
           ]
       },
       {
           "Sid": "SsmWriteParameter",
           "Effect": "Allow",
           "Action": [
               "ssm:DeleteParameter",
               "ssm:DeleteParameters",
               "ssm:LabelParameterVersion",
               "ssm:PutParameter"
           ],
           "Resource": [

"arn:aws:ssm:us-east-1:123456789012:parameter/myparameter"
           ]
       },
       {
           "Sid": "SecretsmanagerPermissionsmanagementSecret",
           "Effect": "Allow",
           "Action": [
               "secretsmanager:DeleteResourcePolicy",
               "secretsmanager:PutResourcePolicy"
           ],
           "Resource": [

"arn:aws:secretsmanager:us-east-1:123456789012:secret:mysecret"
           ]
       },
       {
           "Sid": "SecretsmanagerTaggingSecret",
           "Effect": "Allow",
           "Action": [
               "secretsmanager:TagResource",
               "secretsmanager:UntagResource"
           ],
           "Resource": [

"arn:aws:secretsmanager:us-east-1:123456789012:secret:mysecret"
           ]
       }
   ]
}
```

Then you can take that policy and apply it to the right identity via the console or some automation.

`policy_sentry` is a very useful tool, and organizations can adopt it regardless of which delivery technologies they use. However its capabilities are not natively accessible from common automated infrastructure delivery processes. So adopters need to integrate the tool and generated policies into their delivery workflows.

Alternatively, infrastructure code libraries can provide a simplified interface.

Integrated infra code library for usable policy generation: k9 Terraform for KMS

Infrastructure code libraries can provide simplified, declarative access control interfaces. When you integrate libraries that generate secure policies into automated delivery processes:

1. engineers work within tools they already know
2. pipelines deploy secure policies continuously

You'll need libraries that integrate with your supported infrastructure automation technologies. But the simplification can be profound.

Consider this Terraform configuration which generates a KMS Key resource policy:

```
# Define which principals may access the key and what capabilities they
should have
locals {
 administrator_arns = [
   "arn:aws:iam::123456789012:user/ci"
   , "arn:aws:iam::123456789012:user/person1"
 ]


 read_config_arns = concat(local.administrator_arns, [
   arn:aws:iam::123456789012:role/auditor"
 ])


 read_data_arns = [
   "arn:aws:iam::123456789012:user/person1",
   "arn:aws:iam::123456789012:role/appA",
 ]
 write_data_arns = local.read_data_arns
}


module "least_privilege_key_resource_policy" {

 source = "k9securityio/kms-key/aws//k9policy"

 allow_administer_resource_arns = local.administrator_arns
 allow_read_config_arns         = local.read_config_arns
 allow_read_data_arns           = local.read_data_arns
 allow_write_data_arns          = local.write_data_arns
}
```

Engineers already comfortable with Terraform can write that easily with information they already have. This is the interface to the **k9 Security Terraform library for KMS keys**. The library generates a policy stronger than most Cloud security specialists would write, but everyone can actually read the definition and learn from the result (**full policy**).

Integrate with your delivery process

Before selecting a tool, think about the delivery process.

Who *should be* writing and reviewing policies? Those are your users.

What are those people's existing delivery tools? How are those tools evolving? These are the technologies the 'simplifying tool' needs to work with. Consider whether an access simplifying tool written in Python is going to be well-adopted by a Java application team. How will you manage upgrades? Distribution matters.

When does policy authoring and review occur? These are the moments within the development and delivery process that the tool really needs to work. Effort should be spent on managing access, not shuffling files around.

Congratulations

Once you simplify *how* you're going to use AWS IAM, you've completed a major step. Choosing a few core security patterns and components lets you:

- enable engineers to use the knowledge they already have in their head
- focus and build expertise in a few techniques and tools instead of trying to learn IAM minutia
- productize your selected techniques so your whole organization can use them

- train your team on those techniques so your organization uses them effectively
- gain a lot of experience across your organization using a few techniques so that people can help each other instead of relying on security specialists

These simplifications help engineers cross Norman's **Gulf of Execution**. Now engineers can now easily figure out how to use IAM. We'll cross the Gulf of Evaluation using a similar approach in the next chapter.

With execution of AWS security controls simplified, you can sleep much better tonight.

---

1. Norman, Donald A.. The Design of Everyday Things (pp. 75-76). Basic Books. Kindle Edition.↩

Understand what your IAM policies actually do

> Prove it.
>
> — Vito Paine, Engineer

Now you know how to **create distinct security domains, govern capabilities with SCPs**, and **protect data with least privilege security policies**.

But how can you tell your policies do what you think they do?

It's one thing for you to know how to configure IAM. It's another thing for your organization to actually do it as intended every day.

AWS deployments are complex systems with independent and interacting agents: you, your teams, AWS. Best intentions and efforts do not secure APIs & data. And it's easy to make mistakes, as described in **'Why AWS IAM is so hard to use'**.

You need to know your policies *actually* work.

And you need to know where the problems are *today*.

In this chapter, you will learn how to assess access quickly so you can audit frequently.

You should operate several IAM access control processes that identify excess privileges continuously. Identify in each AWS account:

- unused principals

- access granted to external and internal principals by resource policies for IAM roles and data resources
- access capabilities granted to principals to use AWS services and resources in your account

Identifying excess access capabilities within an account is arguably the most important and most difficult control to operationalize. So we will show you how to do that using several approaches.

Onward...

Close the loop

We anticipated adjusting and monitoring access when we **initially provisioned IAM principals and policies**:

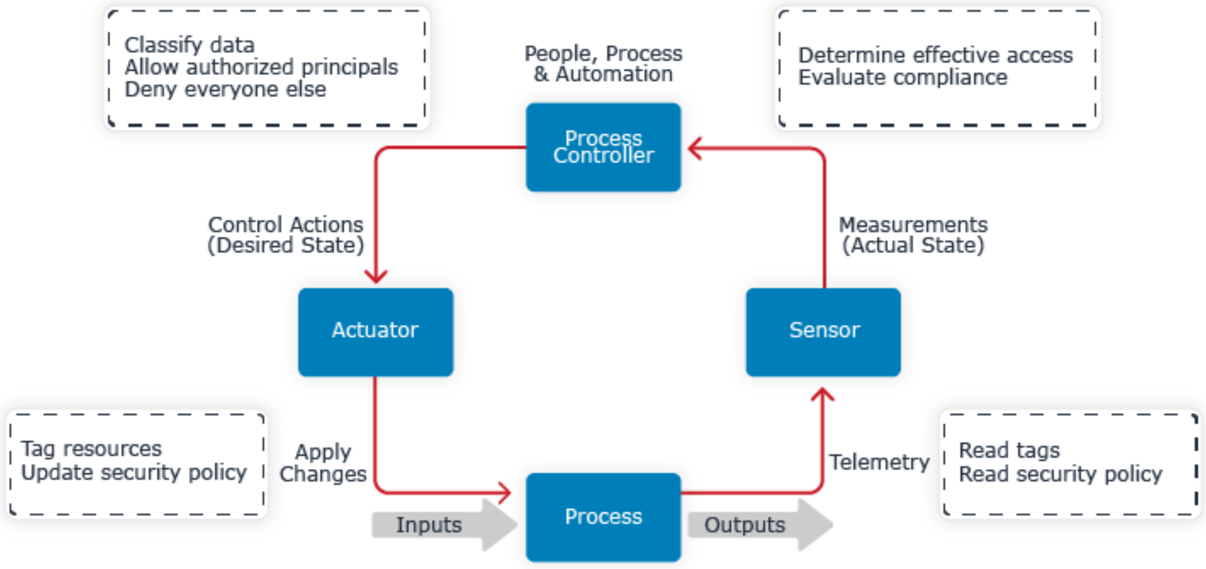


Figure 7.1 Secure data using process control with feedback

To close our control loop, we need sensors to gather data from the running system so that we can analyze the system's actual state.

The primary problems we face are scale and accuracy. We want to verify conditions like:

- only authorized principals have access to your data
- buckets, keys, and roles are not externally accessible
- only the expected principals can administer IAM and the account

The control loop needs sensors to analyze the entire AWS account and report effective access. The sensors should report access in a form that is easy for the process controller to understand and act upon. The process controller is the people, processes, and automation controlling access.

Do not use people to implement the sensing portion of the control loop. If you ask people to be your access evaluation engine, you'll quickly exceed their capacity, and they'll be inaccurate to boot.

Instead, automate reporting who has access to what in AWS so people can consume that reporting. Many AWS accounts have hundreds of principals and resources, or more. Even reporting access in a way people can quickly understand is a challenge. These reports need to accurately summarize a lot of data to help engineers across the **Gulf of Evaluation**.

Automated access analysis methods

Let's examine several approaches for analyzing IAM access using native AWS services, static policy analyzers, and dynamic access analyzers. This section will describe each class of tool and its capabilities. We'll illustrate each tool's capabilities for two common scenarios, understanding:

1. Who is an IAM administrator?
2. Who can access data in an S3 bucket?

Native AWS

The AWS IAM service provides several helpful capabilities via the console and APIs:

- Policy validator
- Access Analyzer
- IAM simulator

Policy Validator

The AWS IAM Access Analyzer Policy Validator tool is an AWS security policy linter. Policy Validator identifies when a policy is invalid, overly permissive, or deviates from best practice. Validator provides feedback on policies as you write them in the AWS console or via API (**full explainer**). The Validator supports more than 100 policy checks. The +10 Security Warnings identify likely problems arising from misuse of advanced security language features like `NotPrincipal`, `NotResource`, and `iam:PassRole`.

The Validator analyzes a single policy in isolation, so its power is limited. The Validator does not know the principals or resources the policy is attached to or what other policies are in effect. Consequently, the Validator and similar linters cannot say whether a principal has the ability to administer IAM or access specific data sources.

Access Analyzer

The AWS IAM Access Analyzer tool suite sounds exactly like what we want, at least at first. Access Analyzer provides two capabilities.

First, Access Analyzer tells you when principals *outside* of your account can access the account's resources via a resource policy. This is extremely useful. Access Analyzer helps you identify S3 buckets, KMS Keys, SQS queues, and more that are accessible publicly or from another account. Access Analyzer reports the resource that is accessible, the principals that can access it, and which API actions they can invoke, e.g. `s3:GetObject`.

Access Analyzer *does not* analyze principals' access *within* the account.

So we have a very helpful, but incomplete answer to our second question about who can access data in an S3 bucket.

Second, Access Analyzer's 'last used' feature tells you when a principal last used an AWS service. For some services like IAM, it also reports when a principal last used each service API action. This is helpful for determining if a principal is actually using permissions it was granted. But it does not tell you if a principal *could* administer IAM.

But there is another tool.

IAM simulator

The AWS IAM simulator lets you simulate what would happen if a principal executed an AWS API request. The simulator supports specifying request resources, relevant policies, and policy context keys. Then the simulator reports whether the API request would be allowed or denied, and why.

This gives you the low-level data you need to assess access capabilities.

However:

1. There are *many* possible combinations to test. Interactive use of the simulator is best for tasks like **verifying that your S3 bucket policy does what you think it does** during development. Fortunately, you can automate verification of certain properties via the IAM simulator APIs.
2. The simulator reports access at a very fine-grained technical level. e.g. Principal A is allowed to call `iam:CreatePolicy`, `iam:AttachPolicy`, and ten other actions. Not "Principal A is allowed to administer IAM."

So you can use the simulator to check if a principal can administer IAM or access data in S3, at least at small scale.

Now let's move from specific native AWS tools to the general class of policy analysis tools.

Policy analyzers

Policy analyzers report whether a (proposed) policy: grants a certain capability, aligns to best practice, or violates some rule.

Strengths: Most policy analyzers are available as command-line tools for easy integration into development and delivery processes. They usually accept policies as both local files and policy ARNs. So they can run frequently and provide fast feedback to change authors and other process controllers.

Policy analyzers are generally good at identifying whether a specific policy grants IAM administration capabilities or makes a bucket public.

Weaknesses: Policy analyzers typically analyze a single policy at a time or the resources within a particular change plan. So they probably won't account for interactions between policies and may miss context that controllers care about.

Here's how policy analyzers perform on our audit questions:

Q1. Who is an IAM administrator?

Static policy analyzers usually report which *policies* grant IAM administration or full access capabilities. However, they probably cannot identify IAM administrators directly.

Q2. Who can access data in an S3 bucket?

Static policy analyzers usually cannot report which IAM principals have access to data in an S3 bucket. Answering this question requires analyzing all the account's principals and identity policies according to the AWS policy evaluation logic, not just a single resource policy.

Access analyzers

Access analyzers examine the policies, principals, and resources in an account then report principals' effective access. They operate with much more context and can answer many more questions than a policy analyzer. Access analyzers report whether a principal may:

- change security policies
- delete databases or a particular database
- decrypt data with a particular key

Access analyzers may report access to AWS APIs or high level access capabilities to AWS services. Some analyzers report access to specific resources.

The primary challenges for access analyzers are to:

1. Evaluate access accurately, accounting for at least Service Control, Identity, and Resource policies
2. Scale analysis to large numbers of principals and resources
3. Report access in a way that can be understood

Each access analyzer is built on an IAM access evaluation engine. Some access analyzers use the AWS IAM simulation APIs. Some analyzers implement their own IAM access evaluation engine. An analyzer built on the simulation APIs uses the actual AWS IAM evaluation engine and the account's service control policies, but is difficult to scale. A 3rd-party evaluation engine is easier to scale, but must implement IAM security policy semantics, which is difficult to do correctly.

Capable access analyzers answer both of our questions.

Access is usually organized into a table of allowed operations. Consider this hypothetical access report excerpt:

Principal	Action	Resource
appA	kms:Decrypt	key-1234
appA	s3:GetObject	*
appA	s3:GetObjectVersion	*
appA	s3:GetObject	sensitive-app-data/*
appA	s3:GetObjectVersion	sensitive-app-data/*
appA	s3:GetObject	bucket-1/*
appA	s3:GetObjectVersion	bucket-1/*
	... snip ...	

This example illustrates the third challenge: reporting access so it can be understood by the people and automation controlling access.

Let's return to our questions.

Q1. Who is an IAM administrator?

We can conclude that `appA` principal is not an IAM administrator because no access to `iam` is reported. We would need to see a full report for all principals to understand who is and is not an IAM administrator.

Q2. Who can access data in an S3 bucket?

`appA` can invoke a couple `s3` APIs, `GetObject` and `GetObjectVersion`, which allows retrieving data. The table says`appA` can use these APIs with resources: `*`, `sensitive-app-data`, and `bucket-1`. Suppose this notation means the principal can get objects:

- from buckets generally (`*`), meaning the principal can invoke those S3 APIs generally against all bucket objects in the account unless denied by a specific bucket policy
- from two specific buckets: `sensitive-app-data` and `bucket-1` after accounting for their resource policies

This information may help the people managing `appA` or this AWS account evaluate access. We can see `appA` has access to `sensitive-app-data`, which we granted in **'Control access to any resource'**. We can also guess that `appA` has *access to all buckets in the account*, unless denied by a specific bucket policy. `bucket-1`'s bucket resource policy certainly does not deny access to `appA`.

Should `appA` have access to `bucket-1` or unrestricted access to `S3:GetObject` operations? Probably not, but we don't know. There's very little context available to support a decision.

Context is crucial when evaluating access to resources, particularly:

- Owner
- Application
- DataClassification or expected Confidentiality

(See this **Cloud deployment tagging guide** for a full treatment of this topic.)

This context helps us go from the initial question of "who has access to data in S3?" to "*should* a principal have access to data in S3?"

Evaluating each principal's access to each of the thousands of AWS API actions and resources overloads people quickly. First with sheer volume. Second, is it meaningful to distinguish between `s3:GetObject` and `s3:GetObjectVersion`? Those are both operations for reading data. Most analysts benefit from aggregating the read capabilities into a single row.

Aggregating actions by capability:

1. simplifies analysis to a handful of categories
2. offloads the effort of categorizing API actions to a consistent automated process
3. complements **declaring intended access with high-level capabilities**

Consider this report designed to help analysts evaluate access quickly:

Principal	Service	Capability	Resource	Owner	Data Classification
appA	KMS	read-data	key-1234	Team A	Internal
appA	S3	read-data	*	N/A	N/A
appA	S3	read-data	sensitive-app-data/*	Team A	Confidential
appA	S3	read-data	bucket-1/*	Team B	Internal
		... snip ...			

Analysts and automation can evaluate reports using plain, high-level language with more context. They can consume this information easily and focus on important questions.

For example, an engineer on `appA`'s delivery team can easily confirm `appA` needs read capabilities to `sensitive-app-data`. They can also easily determine that `appA` has unneeded access to other buckets. An automated analysis program can easily identify principals with access to many S3 buckets. It can also determine which resources are classified as `Confidential` and are accessible by more than 10 principals.

Having showed this style of report to many security practitioners, I have observed:

1. everyone understands this form right away and starts to dig into their data
2. people rarely have questions about what API actions are grouped into a capability[1]

People go straight to analyzing their access and identifying excess permissions, particularly to:

1. administer IAM
2. read, write, and delete critical data sources and keys
3. use IAM role resources

This is exactly what we want. Process controllers analyze the report quickly and decide what to do next.

Find IAM administrators quickly

Understanding IAM administrators is so important that further optimization is warranted.

IAM administration capabilities could be represented in the Principal-Service-Capability form:

Principal	Service	Capability
admin	IAM	administer-resource
admin	IAM	read-config
support	IAM	read-config
temp-incident-response-q4	IAM	administer-resource
... snip ...		

The table clearly shows the `admin` and `temp-incident-response-q4` principals can administer IAM resources. But it could be clearer. Analyzers can report who the IAM administrators are directly:

Principal	Is IAM Admin?
admin	TRUE
support	FALSE
temp-incident-response-q4	TRUE
... snip ...	

This form is very easy to filter for IAM admins and diff for changes. Because the burden for consuming this information by the process controller is so low, this information is suitable for a frequently-executed access review loop: weekly, daily, continuously.

Summary

We started with the goal of creating an effective access management control loop. The most difficult part of implementing AWS access controls is implementing the sensor component because understanding the effects of policies is so difficult.

AWS security policy and access analyzers differ in the kinds of questions they try to answer. Policy analyzers focus on policies, access analyzers on effective access to APIs and resources. They also differ in power, accuracy, and speed.

The sensor component must provide actionable information to the process controller so it can accurately and efficiently decide if a change is needed. The definitions of efficiently and accurately will vary by organization. But if you want to execute a control loop once a month or more frequently, process controllers will need to make hundreds of access review decisions quickly. Select sensor components that meet your efficiency and accuracy requirements. Support process controllers with information that helps them decide if a change is needed in less than 30 seconds per decision.

In the next chapter, we'll describe how to **secure AWS continuously** and describe how to operationalize the process in your organization.

---

1. the access capabilities are well-defined, and capability-action mapping reference provided

Secure AWS IAM continuously

It is not enough to do your best; you must know what to do, and THEN do your best.

— W. Edwards Deming

Secure AWS IAM continuously.

That might be the ultimate challenge for AWS customers.

We've covered the knowledge, architecture, and tools to meet this challenge.

But how does it all fit together? Who should do what?

We need a process to operationalize AWS IAM security at scale. That process needs to continuously secure resources and verify they're actually secure. And real engineers need to execute that process with reasonable effort on a useful frequency.

A few clear-cut hours each week, not arbitrary fire drills.

In this chapter, we'll secure AWS continuously by synthesizing the concepts we've learned into an effective and scalable control loop. We'll highlight the key integration and performance requirements of each component so that you can audit and improve IAM access controls frequently. Then we'll discuss how to deploy this process successfully within your organization.

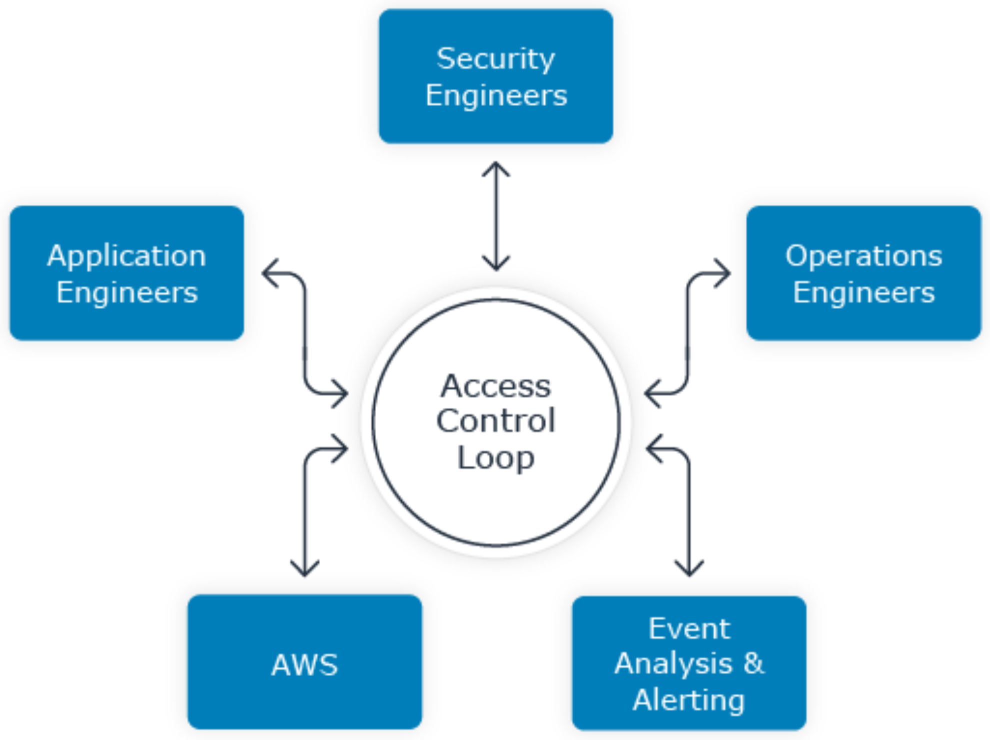


Figure 8.1 Access Control Loop in Context

Skilled people are the most valuable part of any control loop. But there is conflict:

1. ultimately, people must decide if access is correct and what changes to make
2. they have many other decisions to make and work to do

We need to use people's time and attention wisely. This is particularly true of cloud security engineers, those seemingly mythical unicorns spanning Development, Security, and Operations. So we'll simplify and productize the access control process. Then we'll train and motivate the whole organization to execute it. We'll get to that in a moment.

Start by determining the process capability needed to be effective.

Capability

We want to design our control process so it executes frequently enough that we detect and resolve problems quickly. But we don't want to waste time and money executing a process more frequently than needed.

First, establish a target objective for the control. For example, an organization may want to ensure access to confidential data sources is correct monthly.

Second, develop a realistic understanding of how much energy is required to execute the control loop once. If it takes 4 person-days to execute the loop, you're unlikely to execute it monthly.

Setting frequency requirements and effort budgets allows you to eliminate design alternatives with quick math rather than painful experience.

Let's step through an example. Suppose a Fintech company wants to verify its confidential data access controls work as intended. This company has 10 related applications. The applications are delivered together through three AWS accounts: dev, stage, and prod. Three of the applications manage and should have access to confidential data.

The organization has a 'least privilege' access policy, so the org must:

- verify each application has the access it needs
- verify each application does not have unneeded, excess permissions, particularly to data

- verify internal processes and people do not have excess permissions, particularly to Confidential data

The organization must analyze access for 10 applications plus internal processes, then converge to least privilege each month. By policy this needs to happen in production. Stage is probably in scope since it has an unscrubbed copy of production data. Engineers manage infrastructure with code. They promote security changes with the application from dev to stage then prod. Analyzing access in dev will help application teams get the policies correct.

These applications interact and each application changes independently. It's a complex system. To verify access, the organization should check 30+ configuration sets each month. Or at least the 10+ production configurations.

The key observation is that the organization must be able to execute the access control loop **tens of times per month**.

The organization is growing and adding a couple new applications per year. So the Security Architect planned for **50** executions per month. That gives the process room to work for the next 2-3 years.

Who actually knows what access should be?

Now let's figure out who knows what the access *should be*.

Application architects and engineers are generally the people who know who *should* have access to data. They build the software that interacts with data stores. They know whether collaborating applications use their application APIs or data stores. Of course, sometimes nobody knows and you have to go figure it out.

Use tags to communicate which application owns each data set and its expected confidentiality. Sharing that context enables people and automation to understand the purpose of a particular data store. Now Security and Cloud teams can create governance processes that identify gross irregularities. And they can empower application engineers to validate access.

If an application engineer reads a report describing:

1. the effective access each of application IAM role or user
2. who can access their application's data sources

Then the engineer might be able to verify access is correct in 5 minutes or less. If they have a list of access changes since the last review, then they might be able to check access in 30 seconds.

But if that same engineer has to go figure out what the expected access is on their own, they will likely spend a couple days (or weeks) working out the effective access, only to produce an inaccurate answer. Wise engineers will recognize the futility of the task and respond "I don't know" `¯\_(ツ)_/¯`. Hopefully they say that instead of, "looks good to me."

Application engineers should be able to say what access should be and whether the current access is correct.

Who can change access?

Access changes have a similar story. When an application needs additional access to AWS services or data, application engineers work with cloud and site reliability engineers to provision access. Without usable libraries to generate policies, teams often spend days figuring out what should go into a policy and implementing it.

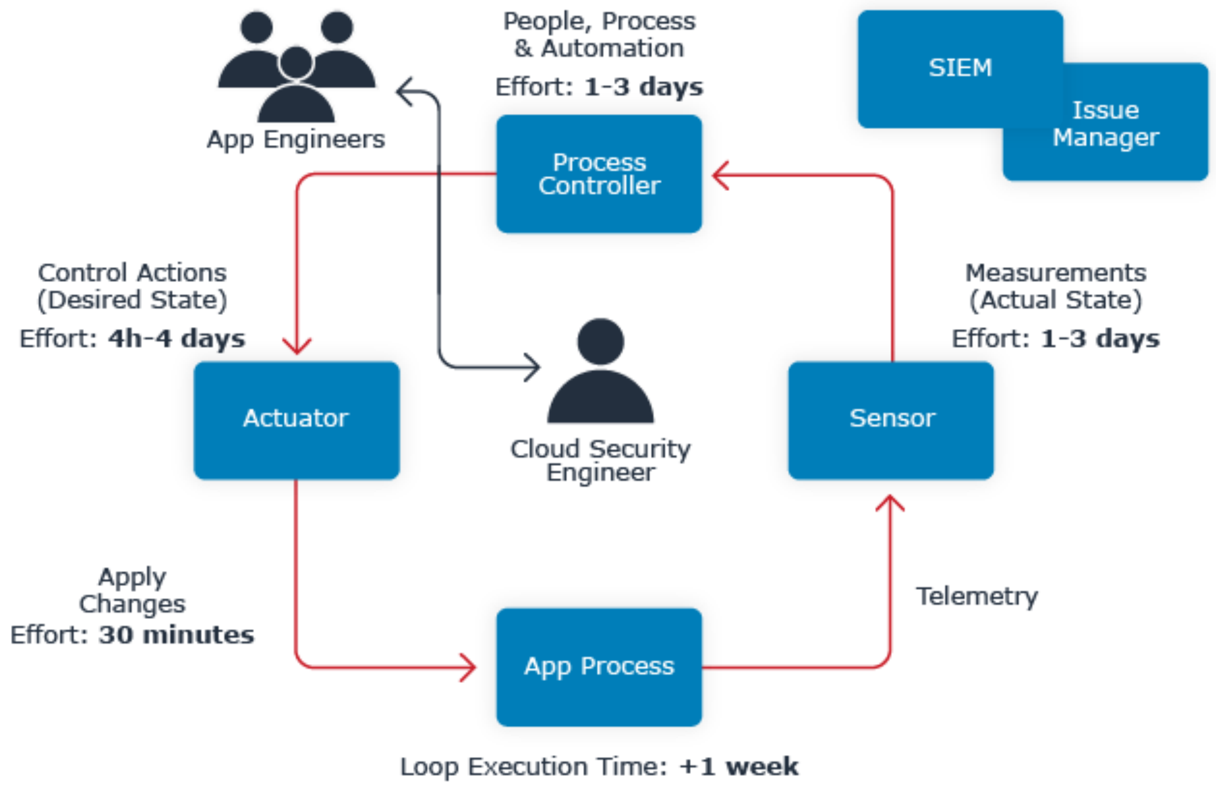


Figure 8.2: Unscalable Access Control Loop

A couple of days here, a meeting there, pretty soon you're talking about *real* waste.

In the **Fintech cloud migration example**, custom policy development overloaded the lone cloud security expert. The app team only knew what access the app needed at a high level. The expert implemented custom policies for each app based on the app team's understanding and test deployments. The organization delivered about one newly-secured application per week.

This approach consumed the Cloud team's security expert for a full quarter. Policy development delayed delivery until they decided to ship apps with excess permissions, then fix it after launch. Further, changes are still running through the Cloud security expert. Only the expert really understands the policies.

The organization averages 1-2 changes per week. The Cloud team now reserves 50% of the Cloud security expert's capacity for those events.

Unfortunately, the organization cannot implement periodic access reviews without overloading the expert. They've been trying to hire another expert but that position has been open for 5 months.

Key Result: The organization cannot fulfill its confidential data access review requirement.

Further, there's no capacity buffer to absorb unforeseen events and vacations are a problem. They currently scale cloud security by adding experts, a difficult and expensive approach.

But consider an organization that manages security policies with usable infrastructure code libraries. Application engineers can safely configure access on their own. Cloud and Security Engineers can review access changes during delivery or when requested. Now delivering an access change might consume a few hours of application engineering time and an hour or two of review time. This unblocks delivery and relieves pressure on experts.

Notice a difference in kind emerging:

- Efficient and scalable access control processes have the potential to succeed.
- Inefficient and unscalable processes will not succeed.

Application engineers must be able to change access safely with minimal demands on cloud security experts.

We know what we need to do: enable application engineers to review and change access easily.

That's a big change in many organizations. So let's discuss how to change how you do security.

Change the way you do Security, successfully

Changing the way an organization works is hard. But you can influence the outcome towards success. Use the insights and playbook from Influencer: The New Science of Leading Change[1].

Influencers focus and measure the key behaviors that create change. In this case, the behaviors are *actually* reviewing and improving access control policies so that information security improves. Influencers use multiple sources of influence to reshape behavior in the right direction and increase probability of success by 10x. These influence sources span ability and motivation:

	Ability	Motivation
Personal	Help them do what they can't	Help them love what they hate
Social	Provide assistance	Provide encouragement
Structure	Change their space	Change their economy

Ability

Increase the organization's ability to execute the control process successfully. Focus effort on improving frontline workers' capacity to manage IAM safely. Provide engineers with robust tools, practice, and support.

Focus on app engineers because: They have the application domain knowledge (context) necessary to secure applications. Enabling app engineers unloads Cloud security specialists. Security changes should integrate with the application's regular delivery process.

Enabling application engineers to manage access may feel unnatural at first. And application engineers may not want new responsibilities

But you must increase the set of people who can manage access safely. And access changes must get on a fast, well-trodden path to production. Only then can you meet the organization's requirements and better security outcomes.

Improve engineers' ability to control access at three levels: Personal, Social, Structural.

Personal: Help them do what they can't

Application and cloud engineers won't write great policies or assess access correctly on their own. You have to help individuals create the outcomes you want in the environment they work in.

Don't just point engineers to the 800+ page AWS IAM user guide and say, "use the best practices" or send them to a half-day workshop on IAM.

Instead, provide engineers with:

- Simplified interfaces to AWS security that help them make good decisions with the knowledge in their head
- Productized components that generate secure policies and report access in language they understand
- Training on how to use the security components and execute the control process
- Safe ways to practice using what they've learned
- Support configuring components and security advice

Increase the ability of individual engineers to build more secure systems. It's the most important factor for improvement program success. If the engineers closest to the problem can't understand and improve access safely, it's likely no one will.

Observe application engineers using the security tools and components you adopt. Notice what people don't understand and where they make mistakes. It's not enough for an engineer to be able to get it right eventually. Improve those tools so they "can't get it wrong" — like professionals.

Provide a safe way for engineers to practice using these tools in a training or test environment. Start by integrating the tools into your reference and training applications. Engineers build knowledge of how to use the tools safely and what "good" looks like through practice. This knowledge and experience is essential for them to operate independently. Then they can try with their own application.

Unload experts by supporting engineers with good documentation, examples, and how-to guides.

Of course there will always be questions in complex domains like security.

Social: Provide assistance

Engineers will have questions, need help, and "permission" to change security controls. Create a place where adopters can ask questions and get good answers, delivered with a smile. 🙂

The environment should encourage engineers to say "I don't know" and ask for help. Otherwise your security issues will remain "undiscovered" and unaddressed.

Consider creating a Security Guild that develops, supports and encourages security practices. The guild can congregate in an open group chat channel. Engineers can get help from people with both formal and informal security responsibilities.

Guilds are a great way to:

- scale the support load
- surface problems with the process, training, and components in an informal way

- identify and document frequently asked questions
- identify topics that need a deeper discussion
- develop & share best practices within your organization
- let people demonstrate the knowledge they've gained and lessons they've learned

Once you've established a guild, lift relevant private conversations into the guild's view. This shows everyone the normal day-to-day experience with the process and opens it to improvement.

Structure: Change their space

If security is going to be every engineer's job, then security should show up where engineers work.

Integrate security into engineers' existing workflows and information sources. Make security capabilities available to engineers on-demand.

Some ways to do this are:

- Let engineers pull routine security work to them through normal product delivery processes. It's less overhead than pushing security work through special projects.
- Provide libraries that generate secure policies for their infrastructure code and delivery process.
- Display access control information in their existing monitoring dashboard.

Don't force each team's engineers to figure out how to integrate security. They either won't integrate it or each team will do it differently.

Scaling Security

Information Security teams have screamed about a 'personnel shortage' for years. And it's only getting worse, "5M+ open positions!" 🙄.

It's time to change the way we secure information.

Don't put security specialists within the control loop. That directs application changes right into one of the organization's tightest bottlenecks. We're blocked! 👿

Security specialists should set up a scalable control loop. Then they can govern its execution with periodic inspections.

If your Enterprise has a Security Operations Center (SOC), the SOC can triage access alerts. Start with Enterprise-wide duties like monitoring IAM admins and access to known critical data stores. To integrate the SOC further into the loop, you must tag data resources with context needed to analyze access and owner contact info (c.f. **Cloud Deployment Tagging Guide**). The SOC can't look at a list of buckets and 'just know' which ones are important, nor who to contact when they're overly accessible.

These recommendations will *enable* your development organization to secure access continuously. Now let's motivate them to do it.

Motivation

Enabling people to do the right thing is necessary. But you also need to motivate people to *use* those abilities. Motivate people at three levels:

- Personal
- Social
- Structural

Personal: Help them love what they hate

> Hard work pays off in the future. Laziness pays off now.
>
> — Steven Wright

Application and Cloud engineers may not like reviewing or improving access right now, particularly repeatedly. You'll need to change that. Interviews with practitioners identify the primary reasons: security policies are difficult to write and nearly impossible to validate without breaking something. That experience is time consuming and painful. So normal people won't do it. Security is deferred until 'later' (like after a breach).

But it's essential to actually complete the loop.

There are several ways to help people love what they currently hate. Let's examine three tactics to motivate individuals to secure access.

First, allow for choice. Recognize that completing this security loop is likely a medium-priority task. Agree upon and clearly communicate the priority of this task throughout your organization. If a team isn't able to complete the task, ask them what led to that and listen. Were they overloaded? Was there a problem with a component? What led to them trading off this security task for something else? Then check if this is happening elsewhere.

Second, create direct experiences that show the risk of excess permissions in a "game day" exercise. Configure a test environment with a copy of a team's application. Give participants a set of actions they can execute to exfiltrate or destroy their application's data. Then let them do it.

Three, make it a game. Implementing least privilege can be extremely challenging and you can use this to your advantage. People *like* challenges, especially engineers. Use a tool to analyze, then score each principal's access to APIs and data. More points for more access and a multiplier for critical APIs like IAM and data sources. Lowest access scores win, just like golf. Engineers will be shooting for par in no time.

Social: Provide encouragement

No influence is more powerful and accessible than persuasion from those we work with. Ridicule and praise from our peers and organizational leaders can do more to assist or hinder change efforts than any other source. Rolled eyes from a team lead can negate the good work of a cloud security team and the CTO's call to action.

Praise people's effort to adopt your improved security processes. Sympathize with their struggles. Encourage their peers to help. This can be as simple as admitting a component is not working well and capturing that feedback for improvement.

Conversely, sanction negative behaviors that affect the program's adoption. Call out nonconstructive feedback as toxic to the organization and its customers.

This is tough work.

So ensure the right people lead with encouragement, coaching, and accountability.

Sometimes all you need is a respected individual or team to adopt your change and show securing AWS is possible. It's likely their peers will model that same change if they can. Pick your early adopters carefully and help them succeed. Once an early adopter succeeds, promote that trailblazer's success to their peers.

Structural: Change their economy

Integrate security into your organization's economy of measurements and incentives. Security effort should pay off.

Ensure that positive and negative incentives aren’t undermining security. Both generally and in the operation of your critical control loops.

First *remove disincentives* for integrating security into daily work.

After finding a security issue, be careful not to blame or penalize well-intentioned engineers. These are complex tasks. Show them a better way to complete that task when resolving the issue. Reduce friction for getting help and using tools properly.

Then incentivize sparingly.

Reward teams for doing the right thing and operating safely in their daily work. Don't over-justify security work with special bonuses. Treat security improvements as product improvements and recognize the importance of that work. One way to do this is to create an issue for each team to review their access controls each month. Then publish the total count of completed reviews and improvements in an organization wide newsletter. If you gamified least privilege, publish a leader board. Report security metrics in your organization's 'Ops Reviews' alongside other key metrics.

Hold teams accountable to organization-wide standards for delivering work according to its priority.

Hold security process and component providers accountable. Gather feedback on the process and remove friction. Improve components with weak capabilities or poor usability. This makes the process easier to use over time and shows component providers have skin in the game.

Put it all together

Optimize the access control loop so it produces good results and operates affordably. Set targets for the loop's total execution time and effort for each step. Then find component implementations that enable engineers to hit the targets.

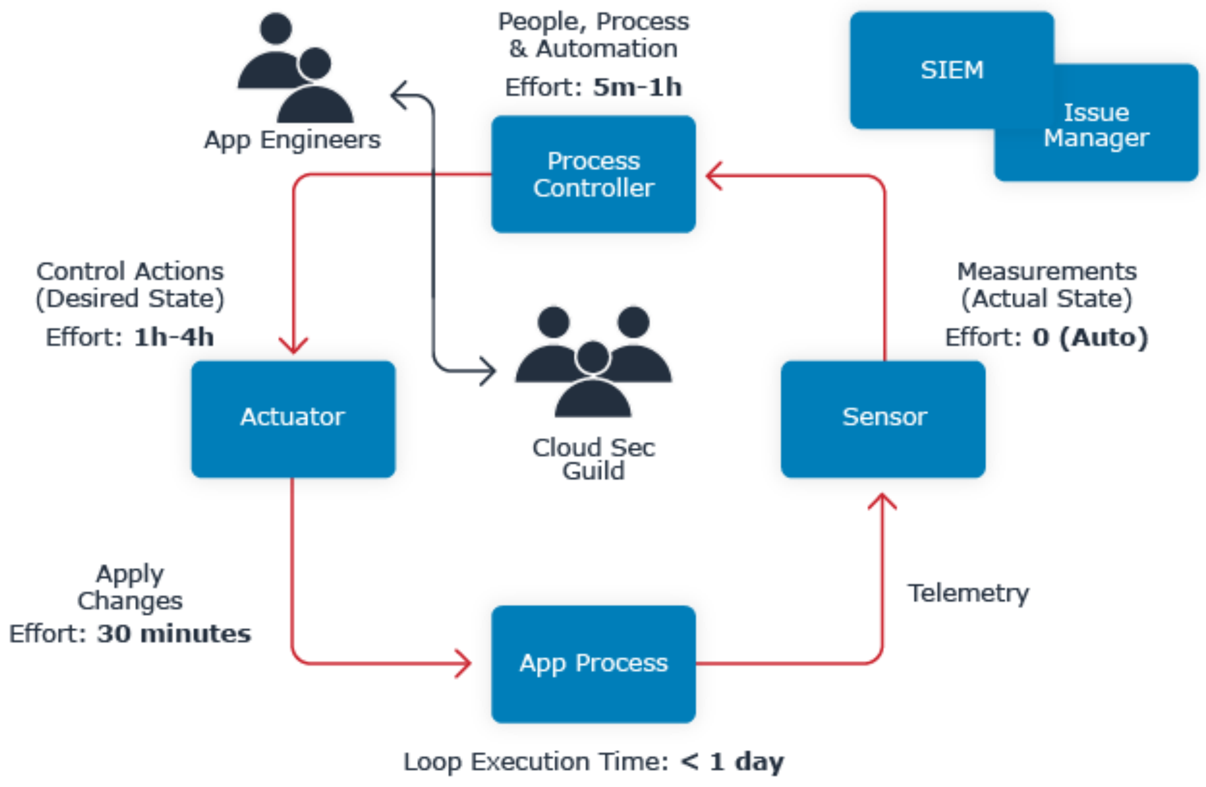


Figure 8.3: Scalable Access Control Loop

Configuring access controls should take less than an hour with usable infrastructure code libraries. Reviewing access for a single application's resources should take 15 minutes or less.

Executing the entire loop should normally take less than one business day. You should be able to deliver a change in a couple hours when there's an incident.

Now you have the information you need to design, implement, and deploy a scalable access control loop in your organization. 🎉

I hope this book has helped you understand the challenges and solutions for building effective access control systems with AWS IAM. You have made a significant and long-lasting investment in your professional development. AWS IAM's features and complexity will grow over time. But AWS is famously committed to backwards compatibility and stability. The concepts and approach described in this book leverage timeless parts of AWS IAM. You can depend on them for many years. Integrate what works for your organization so you can change quickly, confidently, and securely.

Go Fast, *Safely*.

---

1. Influencer, 2ed, Grenny et al, **https://www.vitalsmarts.com/resource/influencer-book/**↩

Discuss AWS Security

I'd love to talk with you about AWS Security. Particularly:

- A general and open discussion about AWS security and why it's hard
- A specific aspect of AWS security IAM architecture, strategy, patterns, or implementation
- Clarifications and extended discussion about a topic in Effective IAM

Share your question or feedback by either:

- Emailing **effectiveiam@k9security.io**
- Scheduling a **30 minute video discussion**

I'm looking forward to chatting about AWS Security with you!

Appendix - Least Privilege S3 Bucket Policy

This is the complete least privilege S3 bucket policy for the 'simple' application described in **Control Access to Any Resource**.

```
{
 "Version": "2012-10-17",
 "Statement": [
   {
     "Sid": "AllowRestrictedAdministerResource",
     "Effect": "Allow",
     "Action": [
       "s3:PutReplicationConfiguration",
       "s3:PutObjectVersionAcl",
       "s3:PutObjectRetention",
       "s3:PutObjectLegalHold",
       "s3:PutObjectAcl",
       "s3:PutMetricsConfiguration",
       "s3:PutLifecycleConfiguration",
       "s3:PutInventoryConfiguration",
       "s3:PutEncryptionConfiguration",
       "s3:PutBucketWebsite",
       "s3:PutBucketVersioning",
       "s3:PutBucketTagging",
       "s3:PutBucketRequestPayment",
       "s3:PutBucketPublicAccessBlock",
       "s3:PutBucketPolicy",
       "s3:PutBucketObjectLockConfiguration",
       "s3:PutBucketNotification",
       "s3:PutBucketLogging",
       "s3:PutBucketCORS",
       "s3:PutBucketAcl",
       "s3:PutAnalyticsConfiguration",
       "s3:PutAccelerateConfiguration",
       "s3:DeleteBucketWebsite",
       "s3:DeleteBucketPolicy",
       "s3:BypassGovernanceRetention"
     ],
     "Resource": [
        "arn:aws:s3:::sensitive-app-data/*",
        "arn:aws:s3:::sensitive-app-data"
      ],
      "Principal": {
        "AWS": "*"
      },
      "Condition": {
        "ArnEquals": {
          "aws:PrincipalArn": [
            "arn:aws:iam::111:user/ci",
            "arn:aws:iam::111:role/admin"
          ]
        }
      }
    },
    {
      "Sid": "AllowRestrictedReadData",
      "Effect": "Allow",
      "Action": [
        "s3:ListMultipartUploadParts",
        "s3:ListBucketVersions",
        "s3:ListBucketMultipartUploads",
        "s3:ListBucket",
        "s3:GetObjectVersionTorrent",
        "s3:GetObjectVersionTagging",
        "s3:GetObjectVersionForReplication",
        "s3:GetObjectVersionAcl",
        "s3:GetObjectVersion",
        "s3:GetObjectTorrent",
        "s3:GetObjectTagging",
        "s3:GetObjectRetention",
        "s3:GetObjectLegalHold",
        "s3:GetObjectAcl",
        "s3:GetObject",
        "s3:GetMetricsConfiguration",
        "s3:GetLifecycleConfiguration",
        "s3:GetInventoryConfiguration",
        "s3:GetEncryptionConfiguration",
        "s3:GetBucketWebsite",
        "s3:GetBucketVersioning",
        "s3:GetBucketTagging",
        "s3:GetBucketRequestPayment",
        "s3:GetBucketPublicAccessBlock",
        "s3:GetBucketPolicyStatus",
        "s3:GetBucketPolicy",
        "s3:GetBucketObjectLockConfiguration",
        "s3:GetBucketNotification",
        "s3:GetBucketLogging",
        "s3:GetBucketLocation",
        "s3:GetBucketCORS",
        "s3:GetBucketAcl"
      ],
      "Resource": [
        "arn:aws:s3:::sensitive-app-data/*",
        "arn:aws:s3:::sensitive-app-data"
      ],
      "Principal": {
        "AWS": "*"
      },
      "Condition": {
        "ArnEquals": {
          "aws:PrincipalArn": [
            "arn:aws:iam::111:role/cust-service",
            "arn:aws:iam::111:role/app"
          ]
        }
      }
    },
    {
      "Sid": "AllowRestrictedWriteData",
      "Effect": "Allow",
      "Action": [
        "s3:RestoreObject",
        "s3:ReplicateTags",
        "s3:ReplicateObject",
        "s3:ReplicateDelete",
        "s3:PutObjectVersionTagging",
        "s3:PutObjectTagging",
        "s3:PutObject",
        "s3:PutBucketTagging",
        "s3:AbortMultipartUpload"
      ],
      "Resource": [
        "arn:aws:s3:::sensitive-app-data/*",
        "arn:aws:s3:::sensitive-app-data"
      ],
      "Principal": {
        "AWS": "*"
      },
      "Condition": {
        "ArnEquals": {
          "aws:PrincipalArn": "arn:aws:iam::111:role/app"
        }
      }
    },
    {
      "Sid": "AllowRestrictedDeleteData",
      "Effect": "Allow",
      "Action": [
        "s3:DeleteObjectVersionTagging",
        "s3:DeleteObjectVersion",
        "s3:DeleteObjectTagging",
        "s3:DeleteObject"
      ],
      "Resource": "arn:aws:s3:::sensitive-app-data/*",
      "Principal": {
        "AWS": "*"
      },
      "Condition": {
        "ArnEquals": {
          "aws:PrincipalArn": []
        }
      }
    },
    {
      "Sid": "AllowRestrictedCustomActions",
      "Effect": "Allow",
      "Action": "s3:GetAnalyticsConfiguration",
      "Resource": [
        "arn:aws:s3:::sensitive-app-data/*",
        "arn:aws:s3:::sensitive-app-data"
      ],
      "Principal": {
        "AWS": "*"
      },
      "Condition": {
        "ArnEquals": {
          "aws:PrincipalArn": []
        }
      }
    },
    {
      "Sid": "DenyEveryoneElse",
      "Effect": "Deny",
      "Action": "s3:*",
    "Resource": [
      "arn:aws:s3:::sensitive-app-data/*",
      "arn:aws:s3:::sensitive-app-data"
    ],
    "Principal": {
      "AWS": "111"
    },
    "Condition": {
      "ArnNotEquals": {
        "aws:PrincipalArn": [
          "arn:aws:iam::111:user/ci",
          "arn:aws:iam::111:role/cust-service",
          "arn:aws:iam::111:role/app",
          "arn:aws:iam::111:role/admin"
        ]
      }
    }
  },
  {
    "Sid": "DenyInsecureCommunications",
    "Effect": "Deny",
    "Action": "s3:*",
    "Resource": [
      "arn:aws:s3:::sensitive-app-data/*",
      "arn:aws:s3:::sensitive-app-data"
    ],
    "Principal": {
      "AWS": "*"
    },
    "Condition": {
      "Bool": {
        "aws:SecureTransport": "false"
      }
    }
  },
  {
    "Sid": "DenyUnencryptedStorage",
    "Effect": "Deny",
    "Action": "s3:PutObject",
    "Resource": "arn:aws:s3:::sensitive-app-data/*",
    "Principal": {
      "AWS": "*"
    },
    "Condition": {
      "Null": {
          "s3:x-amz-server-side-encryption": "true"
        }
      }
    },
    {
      "Sid": "DenyStorageWithoutKMSEncryption",
      "Effect": "Deny",
      "Action": "s3:PutObject",
      "Resource": "arn:aws:s3:::sensitive-app-data/*",
      "Principal": {
        "AWS": "*"
      },
      "Condition": {
        "StringNotEquals": {
          "s3:x-amz-server-side-encryption": "aws:kms"
        }
      }
    }
  ]
}
```

Appendix - Least Privilege KMS Key Policy

This is the complete least privilege key policy for the Customer Managed CMK used to secure a data domain in **Simplify IAM by using the best parts**.

Notice that it follows the recommended general form of:

- An `Allow` statement per capability, e.g Administer Resource
- `Deny` all principals not explicitly allowed

```
{
 "Version": "2012-10-17",
 "Id": "LeastPrivilegeKeyPolicy",
 "Statement": [
   {
     "Sid": "AllowRestrictedAdministerResource",
     "Effect": "Allow",
     "Action": [
       "kms:CancelKeyDeletion",
       "kms:ConnectCustomKeyStore",
       "kms:CreateAlias",
       "kms:CreateCustomKeyStore",
       "kms:CreateGrant",
       "kms:CreateKey",
       "kms:DeleteAlias",
       "kms:DisableKey",
       "kms:DisableKeyRotation",
       "kms:DisconnectCustomKeyStore",
       "kms:EnableKey",
       "kms:EnableKeyRotation",
       "kms:PutKeyPolicy",
       "kms:RetireGrant",
       "kms:RevokeGrant",
       "kms:ScheduleKeyDeletion",
       "kms:TagResource",
       "kms:UntagResource",
        "kms:UpdateAlias",
        "kms:UpdateCustomKeyStore",
        "kms:UpdateKeyDescription"
      ],
      "Resource": "*",
      "Principal": {
        "AWS": "*"
      },
      "Condition": {
        "ArnEquals": {
          "aws:PrincipalArn": [
            "arn:aws:iam::123456789012:user/person1",
            "arn:aws:iam::123456789012:user/ci"
          ]
        }
      }
    },
    {
      "Sid": "AllowRestrictedReadConfig",
      "Effect": "Allow",
      "Action": [
        "kms:DescribeCustomKeyStores",
        "kms:DescribeKey",
        "kms:GetKeyPolicy",
        "kms:GetKeyRotationStatus",
        "kms:GetParametersForImport",
        "kms:GetPublicKey",
        "kms:ListAliases",
        "kms:ListGrants",
        "kms:ListKeyPolicies",
        "kms:ListKeys",
        "kms:ListResourceTags",
        "kms:ListRetirableGrants"
      ],
      "Resource": "*",
      "Principal": {
        "AWS": "*"
      },
      "Condition": {
        "ArnEquals": {
          "aws:PrincipalArn": [
            "arn:aws:iam::123456789012:user/person1",
            "arn:aws:iam::123456789012:user/ci"
          ]
        }
    }
  },
  {
    "Sid": "AllowRestrictedReadData",
    "Effect": "Allow",
    "Action": [
      "kms:Decrypt",
      "kms:Verify"
    ],
    "Resource": "*",
    "Principal": {
      "AWS": "*"
    },
    "Condition": {
      "ArnEquals": {
        "aws:PrincipalArn": [
          "arn:aws:iam::123456789012:user/person1",
          "arn:aws:iam::123456789012:role/appA"
        ]
      }
    }
  },
  {
    "Sid": "AllowRestrictedWriteData",
    "Effect": "Allow",
    "Action": [
      "kms:Encrypt",
      "kms:GenerateDataKey",
      "kms:GenerateDataKeyPair",
      "kms:GenerateDataKeyPairWithoutPlaintext",
      "kms:GenerateDataKeyWithoutPlaintext",
      "kms:GenerateRandom",
      "kms:ImportKeyMaterial",
      "kms:ReEncryptFrom",
      "kms:ReEncryptTo",
      "kms:Sign"
    ],
    "Resource": "*",
    "Principal": {
      "AWS": "*"
    },
    "Condition": {
      "ArnEquals": {
        "aws:PrincipalArn": [
          "arn:aws:iam::123456789012:user/person1",
            "arn:aws:iam::123456789012:role/appA"
          ]
        }
      }
    },
    {
      "Sid": "AllowRestrictedDeleteData",
      "Effect": "Allow",
      "Action": [
        "kms:DeleteCustomKeyStore",
        "kms:DeleteImportedKeyMaterial"
      ],
      "Resource": "*",
      "Principal": {
        "AWS": "*"
      },
      "Condition": {
        "ArnEquals": {
          "aws:PrincipalArn": []
        }
      }
    },
    {
      "Sid": "AllowRestrictedCustomActions",
      "Effect": "Allow",
      "Action": "kms:DescribeKey",
      "Resource": "*",
      "Principal": {
        "AWS": "*"
      },
      "Condition": {
        "ArnEquals": {
          "aws:PrincipalArn": []
        }
      }
    },
    {
      "Sid": "DenyEveryoneElse",
      "Effect": "Deny",
      "Action": "kms:*",
      "Resource": "*",
      "Principal": {
        "AWS": "*"
      },
      "Condition": {
        "ArnNotEquals": {
          "aws:PrincipalArn": [
            "arn:aws:iam::123456789012:root",
            "arn:aws:iam::123456789012:user/person1",
            "arn:aws:iam::123456789012:user/ci",
            "arn:aws:iam::123456789012:role/appA"
          ]
        },
        "Bool": {
          "aws:PrincipalIsAWSService": "false",
          "kms:GrantIsForAWSResource": "false"
        }
      }
    }
  ]
}
```

Appendix - IAM roles quickstart

This table details the common people IAM roles defined in **Create IAM principals and provision access**. Consider creating these roles in your AWS accounts to support common needs:

Role Name	Type	Description	Present in Account(s)
admin	Person	Used to perform emergency operational tasks and initial account configuration activities manually: ● create initial or fix broken IAM configurations ● configure resources needed by automated configuration management tools ● file support cases	All accounts
security	Person	Used by security engineers to inspect, build, and manage security policies and security-related infrastructure.	All accounts
operations	Person	Used to perform common operational tasks in an account manually, such as: ● adjusting autoscaling rules (urgent) ● reconfiguring a load balancer (urgent) ● deploying certificates ● adding a DNS record ● filing support cases	Runtime, Shared Services, Delivery
network-eng	Person	Used by network engineers to inspect, build, and manage network infrastructure, network security policies, and logs.	Runtime, Shared Services, Delivery
database-eng	Person	Used by database engineers and administrators to inspect, build, and manage datastores.	Runtime, Shared Services, Delivery

cloud-eng	Person	Used by platform engineers to build and manage the common infrastructure that applications deploy onto or into.	Runtime, Shared Services, Delivery
release	Person	Used by release (build, cm) engineers to build and manage delivery pipelines.	Runtime, Delivery
observability	Person	Used by observability engineers to build and manage monitoring and logging systems that collect telemetry from AWS accounts, systems, and applications.	All accounts
cost-mgmt	Person	Used by accounting or finance teams to investigate AWS expenditure	Management account
app-eng	Person	Used by application engineers (developers) to build and manage applications used by either external or internal customers.	Runtime