**AWS CSA-Pro Linux Academy Notes Accounts:**

**Accounts Introduction:**

* This section is to focus on AWS accounts from a Solution Architect perspective
* Account Perspective, Networking Perspective, Global and Physical Perspective
* This is the Account Perspective videos
* AWS Accounts and orgs, Service Control Policies, AWS Account Limits, Config, Service Catalogue, Support Tiers, Billing Models, Resource Groups and Tags.
* Deep dive into IAM, Identity and Resource Policies, Long and Short term creds, Policy evaluation logic, Federation, Best practice for using IAM

**AWS Identity Basics Section 1:**

**IAM Overview:**

**IAM Essentials:**

**IAM:** (Identity and Access Management) allows you to control access to AWS services and resources. It manages identities and allows permissions to be allocated to those entities. As a global service the pool of identity is shared across all regions.

**Common use of IAM:**

* Users
* Groups – An organization or an admin construct. Not an identity, but rather contains users
* Roles
* IAM Policies
* Authentication attributes – Usernames, Passwords, Access Keys, MFA, and Password Policies
* By default, any new IAM user you create in an AWS account has no permissions
* Permission policies have an ‘implicit deny’ and Deny always overrides any allow
* For all users except root users, permissions must be given that grant access to AWS services which is done through IAM policies
* IAM provides identity services – but also coordinates with STS to allow Identity Federation to access AWS resources

**Best Practices for IAM:**

* Delete your root access keys and activate MFA on your Root Account
* Create and use an IAM user with Admin privileges instead of the Root Account
* Create individual IAM users and use groups to assign permissions
* Follow the “Principal of Least Privilege” and apply a password policy

**Authentication:**

* Verifying you are who you say you are and granting you the ability to login into the console as an identity
* 2 Types of authentication a user can do. Password and Secret key and Access key
* For console access use username and password
* For CLI access use Secret key and Access key
* These are both known as long term credentials
* Users and Roles are both known as real identities because they can be put into policies like resource policies

**Credential Report-** A report that contains details of all the IAM users in a single account

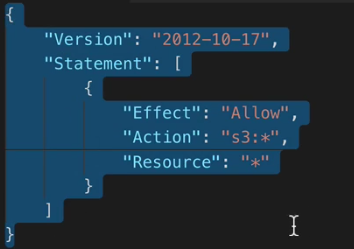
Hard to see, but an example of a credential report. Tells basic info about IAM users like last login, MFA enabled, etc…

C:\Users\Erika\Pictures\Screenpresso\2019-05-25_17h42_35.png

**Identity and Resource Policies: Part 1:**

**JSON Architecture of an IAM Policy:**

* Consists of 1 or more statements
* A statement consists of 3 key value pairs
  + Effect: Weather that statement allows or denies the user the action specified
  + Action: Defines what the principle is allowed or denied in terms of API calls
  + Resource: State which resource the statement effects

****

Policy above allows any user with this policy to do any S3 API action on any Resource (S3 resource technically because it only allows S3 API calls)

**Explicit Denies always override Explicit Allows. There is a default Implicit Deny**

More complex policy document:

* This policy document has a list of statements: 3 to be exact
* When determining what this policy allows you must evaluate all statements
  + 1st statement: Allows, S3 List all my buckets, Resource an S3 resource
  + 2nd statement: Allows, S3 List bucket and Get bucket location, only on the specific bucket of la-homefolders-bucket
  + 3rd statement: Allows, S3 Put, Get, Delete for objects and ACLs, again only for the la-homefolders-bucket



**Policy Types:**

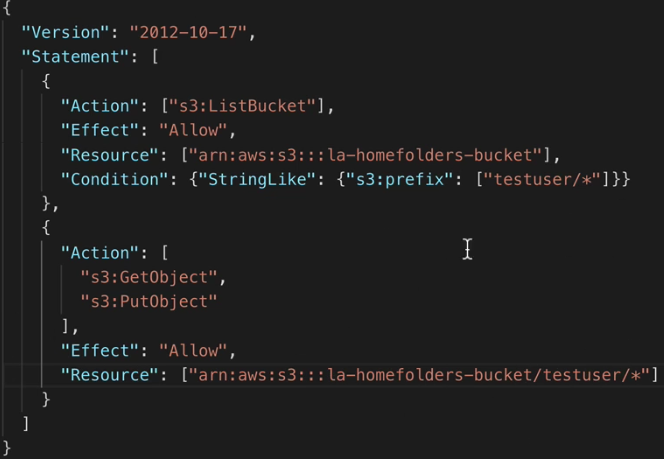
**Managed Policy-** AWS created policies that is separate from users or groups. This can be AWS made or custom policies you make

**Inline Policy-** A policy with a 1:1 relationship between a policy and its entity

Policies can be applied to Users, Groups, and Roles

**Identity and Resource Policies: Part 2:**

* Allows a user to interact with a specific home folder
* Policies can have an optional condition statement which can come in many forms and can make policies very unique and very complex
  + 1st statement: List Bucket, Allow, on the resource la-homefolders-bucket, if the condition of string in the s3 prefix match testuser
  + 2nd statement: Allows, Get and Put objects on the resource la-homefolder-bucket/testuser/\*

****

IAM policies also allow the user of variables. The variables come built into AWS and from my knowledge you cannot create your own variables.

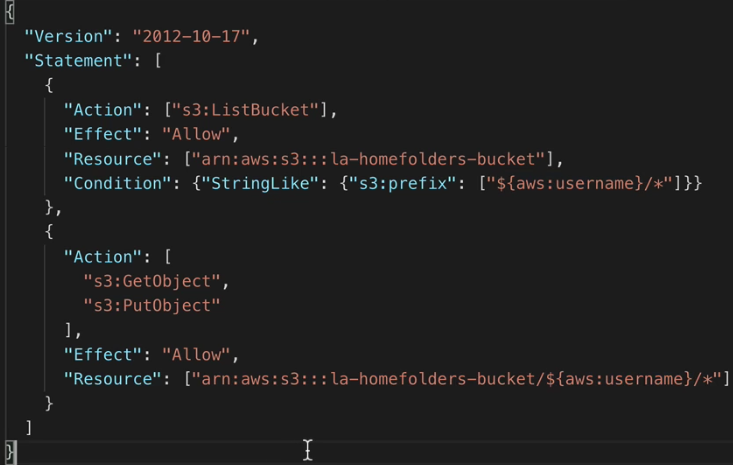
<https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_variables.html>

Link to possible variables you can use. Comes up about once or twice an exam for the SA-Pro

This is the same policy as above, with the exception of a variable. This allows you to have 1 policy attached to a group and only allow users to List, Put, and Get objects in the bucket with their specific username.

This decreases admin overhead when creating policies, because if you wanted to apply this document for each individual user you would have to make specific documents for everyone.

Apply 1 document with thousands of IAM users instead of making thousands of policies unique for every user.



You can use additional conditions for many things. Restricting bucket access from certain time periods. Maybe you have a blackout period where no one should do anything to your resources. Add conditions restricting that!

More in depth explaining the anatomy of policy documents here:

<https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_elements.html>

Conditions more in depth:

<https://docs.aws.amazon.com/IAM/latest/UserGuide/reference_policies_elements_condition.html>

**Resource Policy:**

* S3 have bucket policies: Certain services in AWS have resource based policies which are separate from certain users and instead are attached on certain resources.
* KMS, Buckets, Gateway Endpoints, and more
* When accessing resources all policies are taken into account
* Resource policies are flexible. Applies to any identity accessing that bucket (including non-authenticated users
* Make a bucket public to the world for serverless website hosting
* Principles are required in resource policies. Identity policies the principle is assumed to be the user, group, or role the policy is attached too
* Architecturally a resource policy is used when you want to attach a policy to many different identities even unauthenticated identities

**IAM Roles and Temporary Security Credentials: Part 1:**

You cannot log into a role

**IAM Roles:**

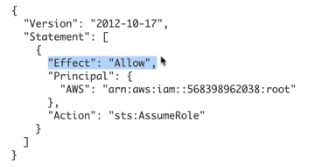
* An IAM role is an entity that can be granted permissions to interact with AWS resources, however, unlike IAM users, roles are designed to be assumed by anyone who needs to make user of their permissions
* A roles had 2 components: A trust policy, which defines the circumstances under which the role can be assumed, and a permissions policy, which defines the AWS access rights granted during that AssumeRole
* A role has no long-term access credentials of its own. Instead, when the role is assumed, STS generates temporary credentials, which can last anywhere from a few minutes to several hours
* Temporary credentials can be renewed, even before expiration, and will be granted for as long as the requesting identity has permissions to assume the role
* Roles are used extensively in AWS
* You grant permissions to service via roles: EC2,, Lambda, and many services that use CloudWatch for logging are provided those service via roles
* When an entity assumes a role, it becomes that role. If you assume a role in an external AWS account and then interact with that account, that accounts owns any resource you create
* You are an entity in that account for as long as you assume the role

**Assuming a Role:**

**Trust Policy:**

* When you assume a role this trust policy is checked to verify you are allowed to assume a role

Sample trust policy for cross account access:

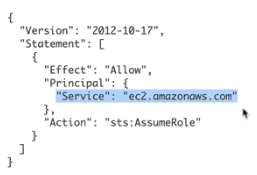


**Permissions Policy:**

* What access is granted to the entity that assumes a role

Roles are often used for AWS services to grant access to other AWS services.

Sample trust policy for a role that is used by EC2 instances:



**IAM Roles and Temporary Security Credentials: Part 2:**

**Revoke Sessions on a Role:**

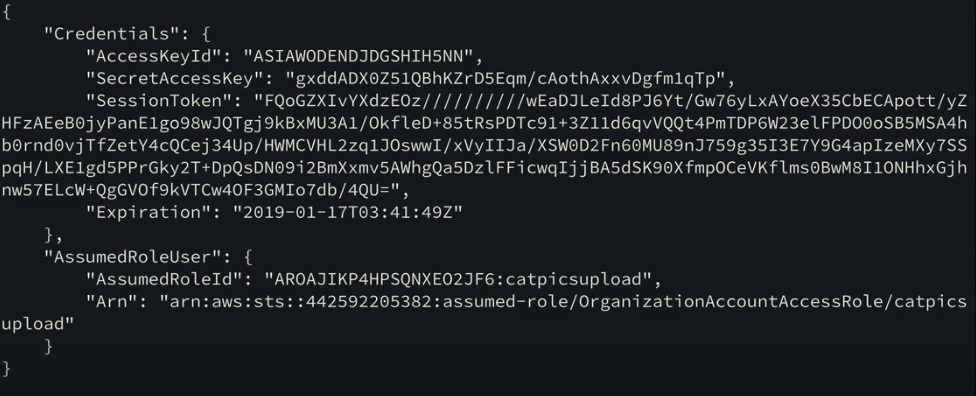
* Make any existing assumptions of a role invalid
* Adds a new inline permissions policy with a Deny statement and a condition stating that is the Token Issue Times data is less than when you revoked it then Deny all access

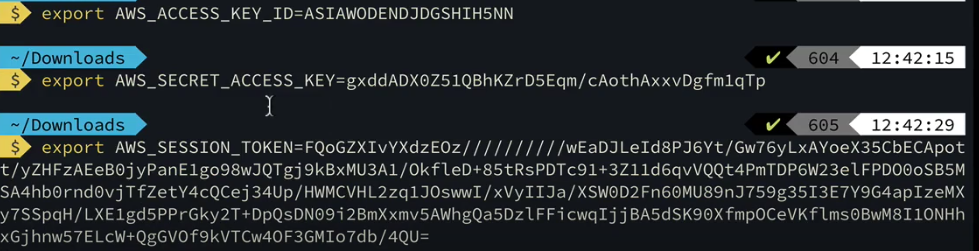
****

**Cross Account Bucket Access:**

* 2 Option: 1 add a bucket policy allowing master account to list the bucket in separate account. 2 create a cross account role and assume it with your user

**C:\Users\Erika\Pictures\Screenpresso\2019-05-26_14h17_51.png**

Command to assume a role.

Output from the above command. After generating your temp security creds you need to store the Access Key, Secret Key, and Session Token in environment variables.

Use above command to store the temp credentials in environment variables.

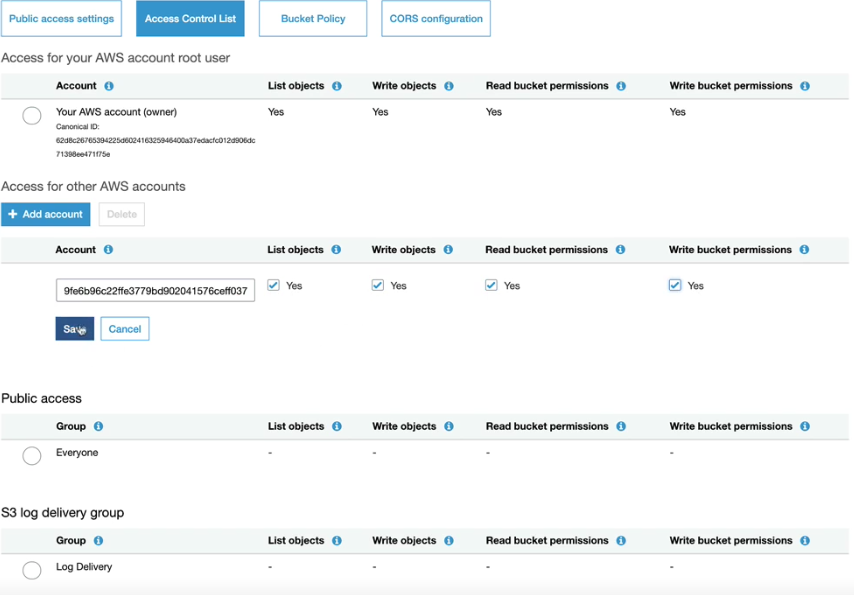
**Exam Tips Summary:**

* Many AWS services utilize IAM roles to interact with other AWS service. EC2, Lambda, VPC Flow Logs, and more
* IAM roles are also useful for cross account functionality with organizations
* You can use cross account roles in the command line along with the console
* You do not long into them, but rather you assume them
* Trust policy and permissions policy

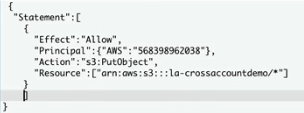
**Cross-Account Access: Resource Permissions vs. Cross-Account Roles:**

**Scenario:**

* 2 Different AWS accounts: Within the same organization or 2 accounts, 1 is yours, 1 is a different companies account/3rd party vendor. Auditing company
* There are 3 main ways to provide access to your S3 buckets from external AWS accounts: IAM Roles, Bucket Policies, and Bucket ACLs
* **ACLs:** ACLs are a legacy access control method for S3, Access control only. Objects are owned by the identity who PUTS them. ACLs can apply to individual objects too.
  + Usually do not use ACLs. Use when you want to grant objects permissions at an object level.
  + Use when you want to configure S3 Access Logging. Grant appropriate permissions via ACLs. Example of using a Bucket Level ACL for granting an account access to a specific bucket below.
  + When using ACLs the account uploading the object is the owner which may cause problems for the account the bucket is in

****

* **Bucket Policy:** Permission control is handled within S3. There is no IAM involvement. Bucket Policies can required Account A be the owner for objects as they are put in the bucket.
  + Example Bucket Policy below. Grants access to Put Objects in crossaccountdemo bucket from the principle which is a specific AWS account.
  + Same owner issue as ACLs. You can add to bucket policies conditions that only allow users to upload if owner access is granted to main account.
  + You can see the updated Bucket Policy that add that condition below

****

****

* **IAM Role:** Account A users assume IAM role. Objects are owned by that role so Account A. Permissions are managed by IAM, not S3.
  + Defaults with the correct owner because when you assume a role in account A from Account B you become that role.
  + When you upload using the role you are an identity in the account with the bucket
  + Lowers admin overhead and guarantees objects uploaded are owned by that account. Preferred way to do this architecture

**When choosing between these 3:**

* Avoid ACLs where possible. Good for object level permissions
* Use Bucket policies when you want to control a specific Buckets Access
* Use Cross-Account roles when you are dealing with User access or you want to extend permissions beyond Buckets
* Keep in mind object ownership

**Account Management Section 2:**

**AWS Accounts and AWS Organizations:**

The account and services layer represents how we create, access, and manage an AWS account and its services, from interacting with an AWS account and managing user rights, to accessing and using various AWS services and features.

**AWS Organizations:**

* AWS Organizations is a service that lets you consolidate multiple AWS accounts into a single organization. This allows you to manage all accounts within the organization in one place.
* AWS Organizations makes billing and permissions easier and allows for the creation of managed accounts. Accounts are organized hierarchically, which provides better security and compliance controls.
* Service is free

**Manage Multiple Accounts:**

**IAM policy management:**

* Account groups
* Create accounts programmatically

**Consolidated Billing:**

* Manage payment methods

Organizations have 2 modes you can set it as. **Consolidated Billing** or **All Features** mode and these modes can be modified later.

**Organization Terms:**

**Master Account-** The top account of the hierarchical structure of all your accounts. Main account. Only account without the ability to restrict the Root User.

**Member Accounts-** The other accounts under the Master.

**Root Container-** Created when your AWS Organization is created and is attached to the Master Account. Applying policies at the Root level propagates them to all OUs and accounts below.

**Organizational Units-** Just nested containers. They can contain AWS accounts or other Organizational Units. Attaching policies at an OU level automatically propagates to all accounts that are a part of it.

By combining bills you benefit from **Economies of Scale**. The more you use a service the more discount you get.

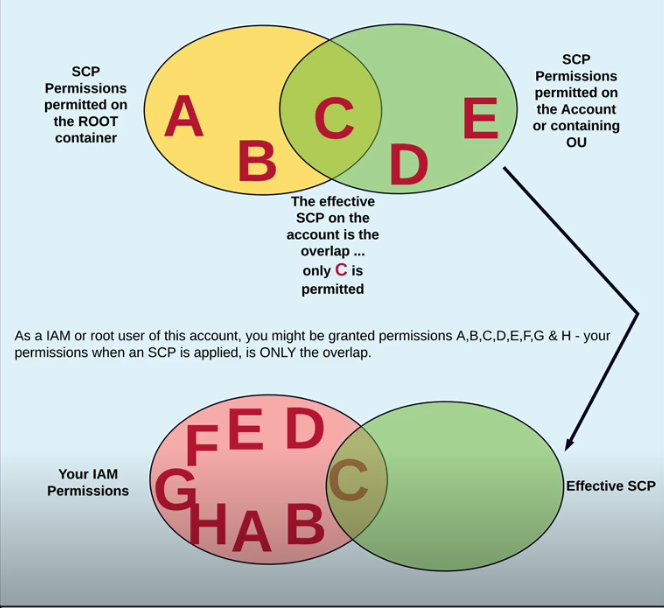
**Service Control Policies-** Policies that can restrict what member accounts can do/what permissions they have. This even effects the Root user of the members account and the only way to restrict Root user access.

**Service Control Policies:**

* JSON policy documents that can be directly applied to an Account, Organizational Units, or even the Root container
* They apply to accounts they are directly too or under it in the hierarchical structure
* One exception is the Service Control Policies applied to the Master account do nothing
* Apply to all principles in the account they are attached too including the Root user of member accounts. Exception would be the Master account

**Service Control Policies:**

* Service Control Policies when applied directly or indirectly to AWS accounts define what actions can be performed on what services within that account.
* The ACTIONS and SERVICES can NEVER exceed those specified by any applicable SCPs – so they act as a way of limiting permissions in member accounts
* SCPs contain explicit ALLOW and DENY statements, but these don’t grant permissions, they only say those permissions are permitted. Anything not explicitly allowed, is implicitly denied
* If multiple SCPs apply to an account … only the OVERLAP of those SCPs is permitted

****

By default there is a Root Container SCP that applies to all new accounts allowing full access of services in that new account. So new accounts get no restrictions until you apply a custom made policy.

Your effective permissions are what your IAM permissions allow, and what the Service Control Policy permits. (This can go even further when you throw in Resource Policies)

You can have multiple Service Control Policies attached. Your effective Service Control Policies is the overlap of all the Service Control Policies.

Service Control Policies do not affect the Master account at all. Including user accounts.

**AWS Account Limits:**

* Some limits can be change and some cannot be changed
* Might impact how you design and impact architectures

**AWS Support Tiers:**

**Developer:** Recommended if you are experimenting or testing in AWS

**Business:** Recommended if you have production workloads in AWS

**Enterprise:** Recommended if you have business and/or mission critical workload in AWS

Programmatic Case Management- Only comes with the Business and Enterprise plans

Proactive Programs- Enterprise mostly, Business some

TAM- Enterprise only

**AWS Config:**

A detailed view of the configurations of AWS resources (EC2, EBS, Security Groups, VPC)

**With AWS Config, you can:**

* Evaluate resource configurations for desired settings
* Get a snapshot of the current configurations associated with your accounts
* Retrieve current or historical configurations of resources in your account
* Receive a notification for creations, deletions, and modifications
* View relationships between resource (Member of a security group)

**Uses of AWS Config:**

* Identity usage/change in config over time – current and historical
* Helps with architecture planning – shows product/service relationships
* Can generate events on changes
* Can help keep an implementation in a compliant state
* Assists security teams – changes to SGs and NACLs over time
* Fault finding and investigation – last known good config

**Config Terms:**

**Configuration Recorder-** Records all of the information about resources and all of their configurations. Enabled on a per region basis

You can use 1 bucket for all regions by giving a bucket prefixes. Central config records = good

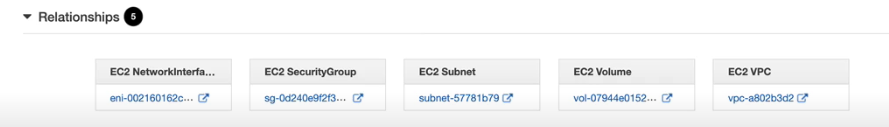
**AWS Resource-** Any resource in an AWS account that is supported by AWS config. S3 bucket, EC2 instance, etc…

**Configuration Item-** A record of a state of a particular resource at a particular point in time. Exist from the point you enable the recorder forward and they store attributes for particular resources for a point in time.

**Configuration History-** Specific AWS resource with a collection of all its configuration items over time. Allows resource configuration over time. Good for compliance!

**Configuration Stream-** A way config can communicate changes. SNS to stream changes to email or lambda for event driven security/reactions

AWS Config is not real time.

You can get a list of all relationships for a specific resource. Below is a relationships from an EC2 Instance

**AWS Configuration Rules:**

* Allows AWS Config to evaluate the state of resources that it monitors
* Define rules to monitor if resources are compliant or non-compliant
* Additional ports have been added to security groups? Exam question. Use Config pre-made rule
* You can also make custom rules
* CloudTrail can feed extra information into Config

**AWS Service Catalog:**

* Implement an IT service catalog within AWS
* Service catalog is a product which allows admins to define products and portfolios (groups of products and configurations). Users can be allowed to self-service deploy these products without the usual IAM permissions required to do so directly with AWS services

**Service Catalog Components:**

**Product-** The thing you want to implement. A CloudFormation template that defines the infrastructure you want someone to deploy

**Portfolio-** Define permissions and constraints for your Products. Can have many products in your portfolio. End users use to launch products.

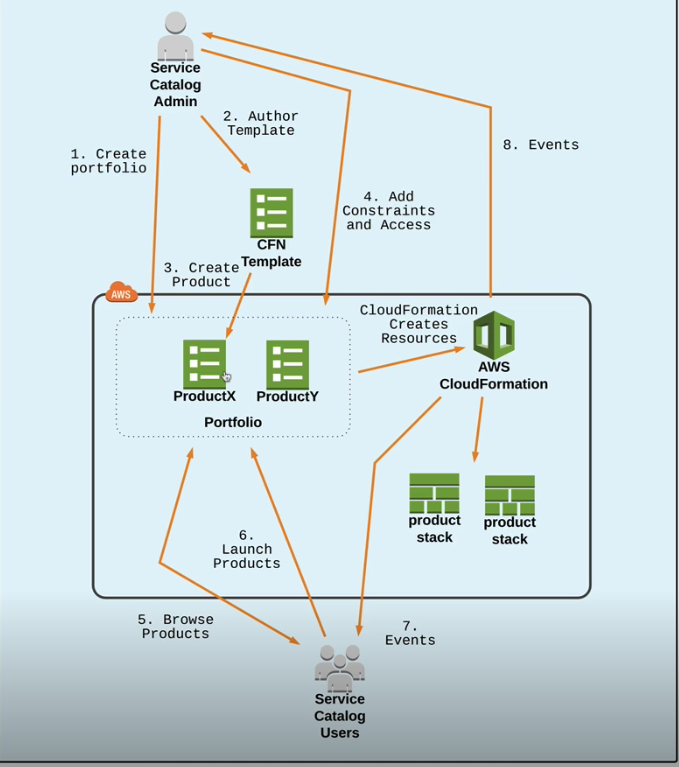
**Constraint-** Allows you to assign an IAM role to the product that is used to provision the AWS resources. End users do not need permissions in AWS to launch it

**Example:**

* Create portfolio
* Author a template (CloudFormation Template)
* Create a Product (For the example a Word Press site)
* Usually permissions to launch a CloudFormation Template for a wordpress site might be a lot. (EC2, RDS, VPC)
* Along with the fact that the technically skills to deploy it might be too much for this end user
* Service Catalog allows you as the Service Catalog Admin to do all the technically permissions work in the background and then offer a portal for end users to use as self service
* You can define permissions on the Portfolios
* End users can deploy portfolios without actual AWS permissions to services
* Service Catalog has a Role needed for the CloudFormation templates to launch, end user just needs service catalog permissions
* Per region service

**Constraint:**

* **Launch Constraint:**
* **Template Constraint:** Restrict the options the user has the ability to deploy. Example would be restricting the Instance size the user can provision
* **Notification Constraint:**



Use Service Catalog when you want to grant End Users self-service deployments of applications without allowing them access to AWS resources.

Allows for Role Separation

**Cost and Cost Optimization Part 3:**

**Resource Billing Modes: On-Demand, Reserved, and Spot:**

**Billing Models:**

**On-Demand:**

On-Demand is the default AWS billing model. You pay for what you consume. $ Per Hour, $ per GB transferred, $ per GB month stored. On demand is ideally suited for ad-hoc usage, where you can’t calculate future usage patterns. It provides small usage discounts for high volume – but generally the most expensive AWS billing model – it has ‘standard’ startup priority.

**Reserved:**

The reserved billing model provides significant reductions in cost & (optional) capacity reservations in exchange for committing to a 12 or 36 month term. You can pay ‘All Upfront’, ‘Partial Upfront’ or ‘No Upfront’ – with full upfront providing the best cost advantages.

Reservations can be used for EC2 instances, RDS instances, DynamoDB performance and many other services in AWS.

Reserved terms are useful when your usage is known and steady-state. Reserved billing model can also reserve capacity …providing high priority startup.

You can pick a region for reserved or a specific availability zone. Picking specific availability zones reserves capacity in advance.

**Spot:**

Spot pricing is ideally suited to sporadic workloads, where you can tolerate interruptions and want the lowest price.

Spot pricing can be significantly cheaper than on demand – but is 100% dependent on spare capacity – price can be higher if capacity is constrained.

Spot pricing is ideally suited to sporadic workloads, where you can tolerate interruptions and always want the best price.

Resources you choose to allocate using a spot pricing plan can be terminated with very little notice … any workloads need to be tolerant of interruptions.

Spot resource have the lowest startup priority vs On-Demand and Reserved.

Can’t hold session data.

Per availability zone and per instance type.

**Advanced Identity in AWS: Part 4:**

**Identity Federation:**

Identity Federation: The ability to bridge to isolated domains.

**IAM Security Token Service (STS):**

* STS allows you to create temporary security credentials that grant trusted users access to your AWS resources
* These temporary credentials are for short-term use, with a configurable session duration between 15 minutes and 12 or 36 hours
* Once expires, they can no longer be used to access your AWS resources
* When requested through an STS API call, a credentials object is returned containing
  + Session Token
  + An Access Key ID
  + A Secret Access Key
  + Expiration Timestamp

**When to use STS:**

* Identity Federation:
  + Enterprise identity federation (authenticate through your companies network)
    - STS supports SAML, which allows for use of Microsoft AD
  + Web identity federation (3rd party identity providers, Facebook, Google)
* Roles for Cross-Account Access
  + Use for organizations that have more than one AWS account
* Roles for Amazon EC2 (and other AWS services)
  + Grant access to an application running no an EC2 instance to access other AWS services without having to imbed credentials

For mobile applications AWS recommend using Cognito rather than STS directly. Cognito provides additional mobile specific functionality which makes the flow easier to manage.

**AssumeRole:**

* Using the AssumeRole API call, an IAM user can assume a role in another AWS account to perform specific, allowed actions
* This API call is specifically triggered during a role assumption of Cross-Account roles

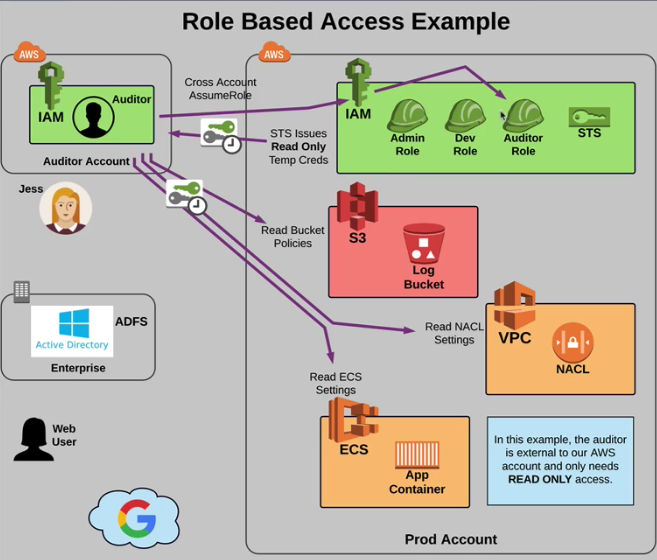
**AssumeRoleWithWebIdentity:**

* Using the AssumeRoleWithWebIdentity API call, our user can assume a role in the target AWS account to perform specific, allowed actions
* The user receives an ID Token from Google (or Facebook, Twitter, Amazon, etc.), which is then embedded in the AssumeRoleWithWebIdentity request to AWS. AWS returns our STS credentials

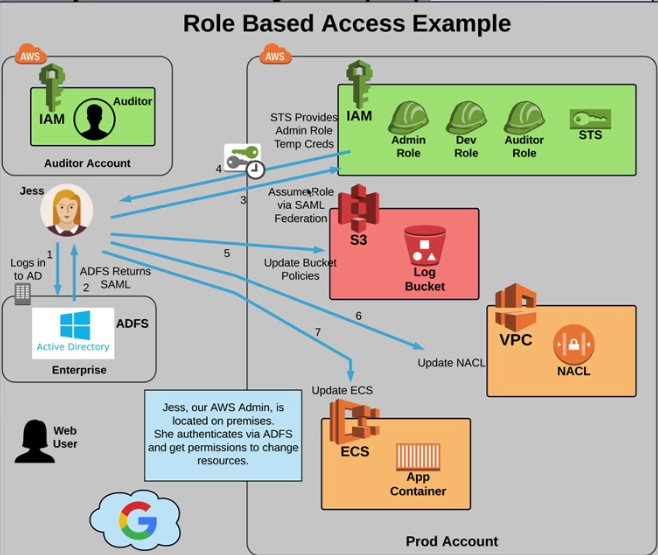
**AssumeRoleWithSAML:**

* Using AssumeRoleWithSAML, our enterprise user, who exists in our Active Directory, is able to authenticate against ADFS and use that response with our SSO page.
* The SAML approach allows you to leverage existing users, avoiding having to create IAM users for each person. Allows for console access.

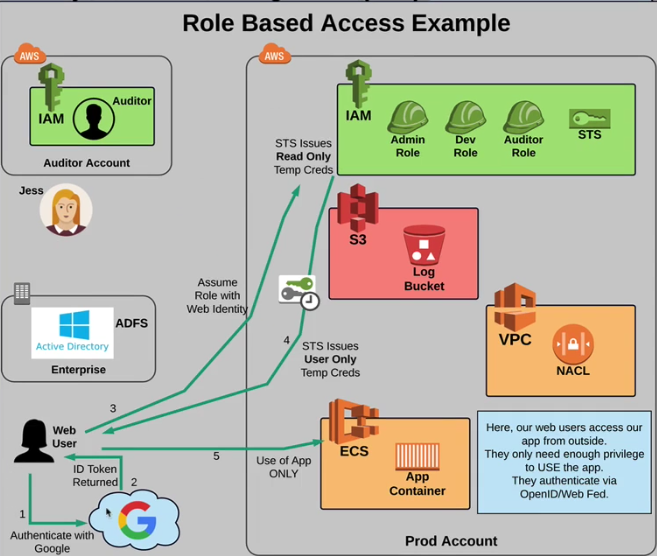
**Auditor Example:**



**Corporate Example:**

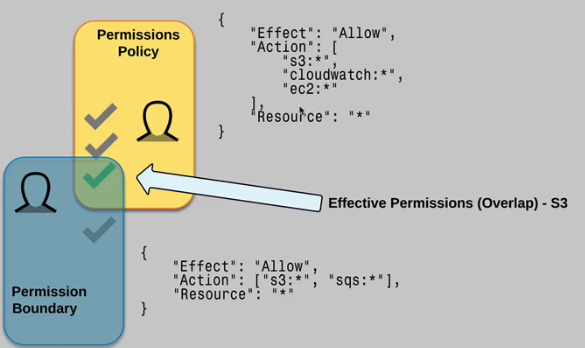
****

**Web user example:**

****

**IAM Permissions Boundaries:**

* Permission Boundaries limit the maximum permission an identity can have
* They can be applied to IAM Users, IAM Roles, or AWS Organizations (Service control policies)
* They do not give any permissions, but rather limit what permission a thing can have
* An entities permissions are the overlap of the IAM Policy and the IAM Permissions Boundary



**Policy Evaluation Logic:**

* Default to deny on policies
* Explicit Allows overrule default denies
* Explicit Denies overrule Explicit Allows
* Boundaries (Service Control Policies and Permission Boundaries) are always processed first. Starting with organizational and then identity.
* Then AWS checks if you have chosen a subset of permissions for a sts:AssumeRole
* Final effective permissions are merge of identity resource, and ACL