aws

User Guide

Amazon Verified Permissions

Amazon Verified Permissions: User Guide

Table of Contents

What is Amazon Verified Permissions?

Amazon Verified Permissions is a scalable, fine-grained permissions management and authorization service for custom applications built by you. Verified Permissions enables your developers to build secure applications faster by externalizing authorization and centralizing policy management and administration. Verified Permissions uses the Cedar policy language to define fine-grained permissions for application users.

Topics

Authorization in Verified Permissions

Verified Permissions provides *authorization* by verifying whether a principal is allowed to perform an action on a resource in a given context in a custom application. Verified Permissions presumes that the principal has been previously identified and authenticated through other means, such as by using protocols like OpenID Connect, a hosted provider like Amazon Cognito, or another authentication solution. Verified Permissions is agnostic to where the user is managed and how the user was authenticated.

Verified Permissions is a service that enables customers to create, maintain, and test policies in the AWS Management Console. Permissions are expressed using the Cedar policy language. The client application calls authorization APIs to evaluate the Cedar policies stored with the service and provide an access decision for whether an action is permitted.

Cedar policy language

Authorization policies in Verified Permissions are written by using the Cedar policy language. Cedar is an open source language for writing authorization policies and making authorization decisions based on those policies. When you create an application, you need to ensure that only authorized users can access the application, and can do only what each user is authorized to do. Using Cedar, you can decouple your business logic from the authorization logic. In your application's code, you preface requests made to your operations with a call to the Cedar authorization engine, asking "Is this request authorized?". Then, the application can either perform the requested operation if the decision is "allow", or return an error message if the decision is "deny".

Verified Permissions currently uses **Cedar version 2.4**.

For more information about Cedar, see the following:

- Cedar policy language Reference Guide
- Cedar GitHub repository

Benefits of Verified Permissions

Accelerate application development

Accelerate application development by decoupling authorization from business logic.

More secure applications

Verified Permissions enables developers to build more secure applications.

End-user features

Verified Permissions allows you to deliver richer end-user features for permissions management.

Related services

- **Amazon Cognito** – Amazon Cognito is an identity platform for web and mobile apps. It's a user directory, an authentication server, and an authorization service for OAuth 2.0 access tokens and AWS credentials. When you create a policy store, you have the option to build your principals and groups from an Amazon Cognito user pool. For more information, see the Amazon Cognito Developer Guide.
- **Amazon API Gateway** – Amazon API Gateway is an AWS service for creating, publishing, maintaining, monitoring, and securing REST, HTTP, and WebSocket APIs at any scale. When you create a policy store, you have the option to build your actions and resources from an API in API Gateway. For more information about API Gateway, see the API Gateway Developer Guide.

- **AWS IAM Identity Center –** With IAM Identity Center, you can manage sign-in security for your workforce identities, also known as workforce users. IAM Identity Center provides one place where you can create or connect workforce users and centrally manage their access across all their AWS accounts and applications. For more information, see the AWS IAM Identity Center User Guide.

Accessing Verified Permissions

You can work with Amazon Verified Permissions in any of the following ways.

AWS Management Console

The console is a browser-based interface to manage Verified Permissions and AWS resources. For more information about accessing Verified Permissions through the console, see How to sign in to AWS in the *AWS Sign-In User Guide*.

- Amazon Verified Permissions console

AWS Command Line Tools

You can use the AWS command line tools to issue commands at your system's command line to perform Verified Permissions and AWS tasks. Using the command line can be faster and more convenient than the console. The command line tools are also useful if you want to build scripts that perform AWS tasks.

AWS provides two sets of command line tools: the AWS Command Line Interface (AWS CLI) and the AWS Tools for Windows PowerShell. For information about installing and using the AWS CLI, see the AWS Command Line Interface User Guide. For information about installing and using the Tools for Windows PowerShell, see the AWS Tools for Windows PowerShell User Guide.

- verifiedpermissions in the AWS CLI Command Reference
- Amazon Verified Permissions in AWS Tools for Windows PowerShell

AWS SDKs

AWS provides SDKs (software development kits) that consist of libraries and sample code for various programming languages and platforms (Java, Python, Ruby, .NET, iOS, Android, etc.). The SDKs provide a convenient way to create programmatic access to Verified Permissions and AWS. For example, the SDKs take care of tasks such as cryptographically signing requests, managing errors, and retrying requests automatically.

To learn more and download AWS SDKs, see Tools for Amazon Web Services.

The following are links to documentation for Verified Permissions resources in various AWS SDKs.

- AWS SDK for .NET
- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java
- AWS SDK for JavaScript
- AWS SDK for PHP
- AWS SDK for Python (Boto)
- AWS SDK for Ruby

Verified Permissions API

You can access Verified Permissions and AWS programmatically by using the Verified Permissions API, which lets you issue HTTPS requests directly to the service. When you use the API, you must include code to digitally sign requests using your credentials.

- Amazon Verified Permissions API Reference Guide

Pricing for Verified Permissions

Verified Permissions provides tiered pricing based on the amount of authorization requests per month made by your applications to Verified Permissions. There is also pricing for policy management actions based on the amount of cURL (client URL) policy API requests per month made by your applications to Verified Permissions.

For a complete list of charges and prices for Verified Permissions see Amazon Verified Permissions pricing.

To see your bill, go to the **Billing and Cost Management Dashboard** in the AWS Billing and Cost Management console. Your bill contains links to usage reports that provide details about your bill. To learn more about AWS account billing, see the *AWS Billing User Guide*.

If you have questions concerning AWS billing, accounts, and events, contact AWS Support.

Amazon Verified Permissions terms and concepts

You should understand the following concepts to use Amazon Verified Permissions.

Verified Permissions concepts

- Authorization model
- Authorization request
- Authorization response
- Considered policies
- Context data
- Determining policies
- Entity data
- Permissions, authorization, and principals
- Policy enforcement
- Policy store
- Satisfied policies
- Differences between Verified Permissions and Cedar

Cedar policy language concepts

- Authorization
- Entity
- Groups and hierarchies
- Namespaces
- Policy
- Policy template
- Schema

Authorization model

The *authorization model* describes the scope of the authorization requests made by the application and the basis for evaluating those requests. It is defined in terms of the different types of resources, the actions taken on those resources, and the types principals that take those actions. It also considers the context in which those actions are being taken.

Role-based Access Control (RBAC) is an evaluation basis in which roles are defined and associated with a set of permissions. These roles can then be assigned to one or more identities. The assigned identity acquires the permissions associated with the role. If the permissions associated with the role are modified, then the modification automatically impacts any identity to which the role has been assigned. Cedar can support RBAC decisions through the use of principal groups.

Attribute-based Access Control (ABAC) is an evaluation basis in which the permissions associated with an identity are determined by attributes of that identity. Cedar can support ABAC decisions through the use of policy conditions that reference attributes of the principal.

The Cedar policy language enables the combination of RBAC and ABAC in a single policy by allowing permissions to be defined for a group of users, which have attribute-based conditions.

Authorization request

An *authorization request* is a request made of Verified Permissions by an application to evaluate a set of policies in order to determine whether a principal may perform an action on a resource for a given context.

Authorization response

The *authorization response* is the response to the authorization request. It includes an allow or deny decision, plus additional information, such as the IDs of the determining policies.

Considered policies

Considered policies are the full set of policies that are selected by Verified Permissions for inclusion when evaluating an authorization request.

Context data

Context data are attribute values that provide additional information to be evaluated.

Determining policies

Determining policies are the policies that determine the authorization response. For example, if there are two satisfied policies, where one is a deny and the other is an allow, then the deny policy will be the determining policy. If there are multiple satisfied permit policies and no satisfied forbid policies, then there are multiple determining policies. In the case that no policies match and the response is deny, there are no determining policies.

Entity data

Entity data are data about the principal, action, and resource. Entity data relevant for policy evaluation are group membership all the way up the entity hierarchy and attribute values of the principal and resource.

Permissions, authorization, and principals

Verified Permissions manages fine-grained *permissions* and *authorization* within custom applications that you build.

A *principal* is user of an application, either human or machine, that has an identity bound to an identifier such as a username or machine ID. The process of authentication determines whether the principal is truly the identity they claim to be.

Associated with that identity are a set of application *permissions* that determine what that principal is permitted to do within that application. *Authorization* is the process of assessing those permissions to determine whether a principal is permitted to perform a particular action in the application. These permissions can be expressed as policies.

Policy enforcement

Policy enforcement is the process of enforcing the evaluation decision within the application outside of Verified Permissions. If Verified Permissions evaluation returns a deny, then enforcement would ensure that the principal was prevented from accessing the resource.

Policy store

A *policy store* is a container for policies and templates. Each store contains a schema that is used to validate policies added to the store. By default, each application has its own policy store, but multiple applications can share a single policy store. When an application makes an authorization request, it identifies the policy store used to evaluate that request. Policy stores provide a way to isolate a set of policies, and can therefore be used in a multi-tenant application to contain the schemas and policies for each tenant. A single application can have separate policy stores for each tenant.

When evaluating an authorization request, Verified Permissions only considers the subset of the policies in the policy store that are relevant to the request. Relevance is determined based on the *scope* of the policy. The scope identifies the specific principal and resource to which the policy applies, and the actions that the principal can perform on the resource. Defining the scope helps improve performance by narrowing the set of considered policies.

Satisfied policies

Satisfied policies are the policies that match the parameters of the authorization request.

Differences between Verified Permissions and Cedar

Amazon Verified Permissions uses the Cedar policy language engine to perform its authorization tasks. However, there are some differences between the native Cedar implementation and the implementation of Cedar found in Verified Permissions. This topic identifies those differences.

Namespace definition

Verified Permissions implementation of Cedar has the following differences from the native Cedar implementation:

- Verified Permissions supports only one namespace in a schema defined in a policy store.
- Verified Permissions doesn't allow you to create a namespace with the following values: `aws`, `amazon`, or `cedar`.

Policy template support

Both Verified Permissions and Cedar allow placeholders in the scope for only the `principal` and `resource`. However, Verified Permissions also requires that neither the `principal` and `resource` are unconstrained.

The following policy is valid in Cedar but is rejected by Verified Permissions because the `principal` is unconstrained.

```
permit(principal, action == Action::"view", resource == ?resource);
```

Both of the following examples are valid in both Cedar and Verified Permissions because both the `principal` and `resource` have constraints.

```
permit(principal == User::"alice", action == Action::"view", resource == ?resource);
```

```
permit(principal == ?principal, action == Action::"a", resource in ?resource);
```

Schema support

Verified Permissions requires all schema JSON key names to be non-empty strings. Cedar allows empty strings in a few cases, such as for properties.

Extension type support

Verified Permissions supports Cedar extension types in policies, but doesn't currently support including them in the definition of a schema or as part of the `entities` parameter of the `IsAuthorized` and `IsAuthorizedWithToken` operations.

Extension types include the fixed point (`decimal`) and IP address (`ipaddr`) data types.

Cedar JSON format for entities

At this time, Verified Permissions requires you to pass the list of entities to be considered in an authorization requestion using the structure defined for the EntitiesDefinition, which is an array of EntityItem elements. Verified Permissions doesn't currently support passing the list of entities to be considered in an authorization request in Cedar JSON format. For specific requirements of formatting your entities for use in Verified Permissions, see Entity formatting in Amazon Verified Permissions.

Action groups definition

The Cedar authorization methods require a list of the entities to be considered when evaluating an authorization request against the policies.

You can define the actions and action groups used by your application in the schema. However, Cedar doesn't include the schema as part of an evaluation request. Instead, Cedar uses the schema only to validate the policies and policy templates that you submit. Because Cedar doesn't reference the schema during evaluation requests, even if you defined action groups in the schema, you must also include the list of any action groups as part of the entities list you must pass to the authorization API operations.

Verified Permissions does this for you. Any action groups that you define in your schema are automatically appended to the entities list that you pass to as a parameter to the `IsAuthorized` or `IsAuthorizedWithToken` operations.

Length and size limits

Verified Permissions supports storage in the form of policy stores to hold your schema, policies, and policy templates. That storage causes Verified Permissions to impose some length and size limits that aren't relevant to Cedar.

Object	Verified Permissions limit (in bytes)	Cedar limit
Policy size[1]	10,000	None
Inline policy description	150	Not applicable to Cedar
Policy template size	10,000	None
Schema size	10,000	None
Entity type	200	None
Policy ID	64	None
Policy template ID	64	None
Entity ID	200	None
Policy store ID	64	Not applicable to Cedar

[1] There is a limit for policies per policy store in Verified Permissions based on the combined size of principals, actions, and resources of policies created in the policy store. The total size of all policies pertaining to a single resource can't exceed 200,000 bytes. For template-linked policies, the size of the policy template is counted only once, plus the size of each set of parameters used to instantiate each template-linked policy.

Getting started with Verified Permissions

Use this tutorial to get started with Amazon Verified Permissions.

Topics

Sign up for an AWS account

If you do not have an AWS account, complete the following steps to create one.

To sign up for an AWS account

1. Open https://portal.aws.amazon.com/billing/signup.
2. Follow the online instructions.

   Part of the sign-up procedure involves receiving a phone call and entering a verification code on the phone keypad.

   When you sign up for an AWS account, an *AWS account root user* is created. The root user has access to all AWS services and resources in the account. As a security best practice, assign administrative access to an administrative user, and use only the root user to perform tasks that require root user access.

AWS sends you a confirmation email after the sign-up process is complete. At any time, you can view your current account activity and manage your account by going to https://aws.amazon.com/ and choosing **My Account**.

Create an administrative user

After you sign up for an AWS account, secure your AWS account root user, enable AWS IAM Identity Center, and create an administrative user so that you don't use the root user for everyday tasks.

Secure your AWS account root user

1. Sign in to the AWS Management Console as the account owner by choosing **Root user** and entering your AWS account email address. On the next page, enter your password.

   For help signing in by using root user, see Signing in as the root user in the *AWS Sign-In User Guide*.
2. Turn on multi-factor authentication (MFA) for your root user.

   For instructions, see Enable a virtual MFA device for your AWS account root user (console) in the *IAM User Guide*.

Create an administrative user

1. Enable IAM Identity Center.

   For instructions, see Enabling AWS IAM Identity Center in the *AWS IAM Identity Center User Guide*.
2. In IAM Identity Center, grant administrative access to an administrative user.

   For a tutorial about using the IAM Identity Center directory as your identity source, see Configure user access with the default IAM Identity Center directory in the *AWS IAM Identity Center User Guide*.

Sign in as the administrative user

- To sign in with your IAM Identity Center user, use the sign-in URL that was sent to your email address when you created the IAM Identity Center user.

  For help signing in using an IAM Identity Center user, see Signing in to the AWS access portal in the *AWS Sign-In User Guide*.

IAM policies for Verified Permissions

Verified Permissions manages the permissions of users within your application. In order for your application to call the Verified Permissions APIs or for AWS Management Console users to be allowed to manage Cedar policies in a Verified Permissions policy store, you must add the necessary IAM permissions.

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, group of users, or role. These policies control what actions users and roles can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Creating IAM policies in the IAM User Guide.

With IAM identity-based policies, you can specify allowed or denied actions and resources as well as the conditions under which actions are allowed or denied (listed below). You can't specify the principal in an identity-based policy because it applies to the user or role to which it is attached. To learn about all of the elements that you can use in a JSON policy, see IAM JSON policy elements reference in the IAM User Guide.

Action	Description
CreatePolicyStore	Action to create a new policy store.
DeletePolicyStore	Action to delete a policy store.
ListPolicyStores	Action to list all policy stores in the AWS account.
CreatePolicy	Action to create a Cedar policy in a policy store. You can create either a static policy or a policy linked to a policy template.
DeletePolicy	Action to delete a policy from a policy store.
GetPolicy	Action to retrieve information about a specified policy.
ListPolicies	Action to list all policies in a policy store.
IsAuthorized	Action to get an authorization response based on the parameters described in the authoriza tion request.

Example IAM policy for permission to the CreatePolicy action:

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

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

Create your first Verified Permissions policy store

When you sign in to the Verified Permissions console for the first time, you can choose how to create your first [policy store](#) and Cedar policy. Follow the sign-in procedure appropriate to your user type as described in the topic [How to sign in to AWS](#) in the *AWS Sign-In User Guide*. On the **Console Home** page, select the Amazon Verified Permissions service. Choose **Get started**.

Creating a sample policy store

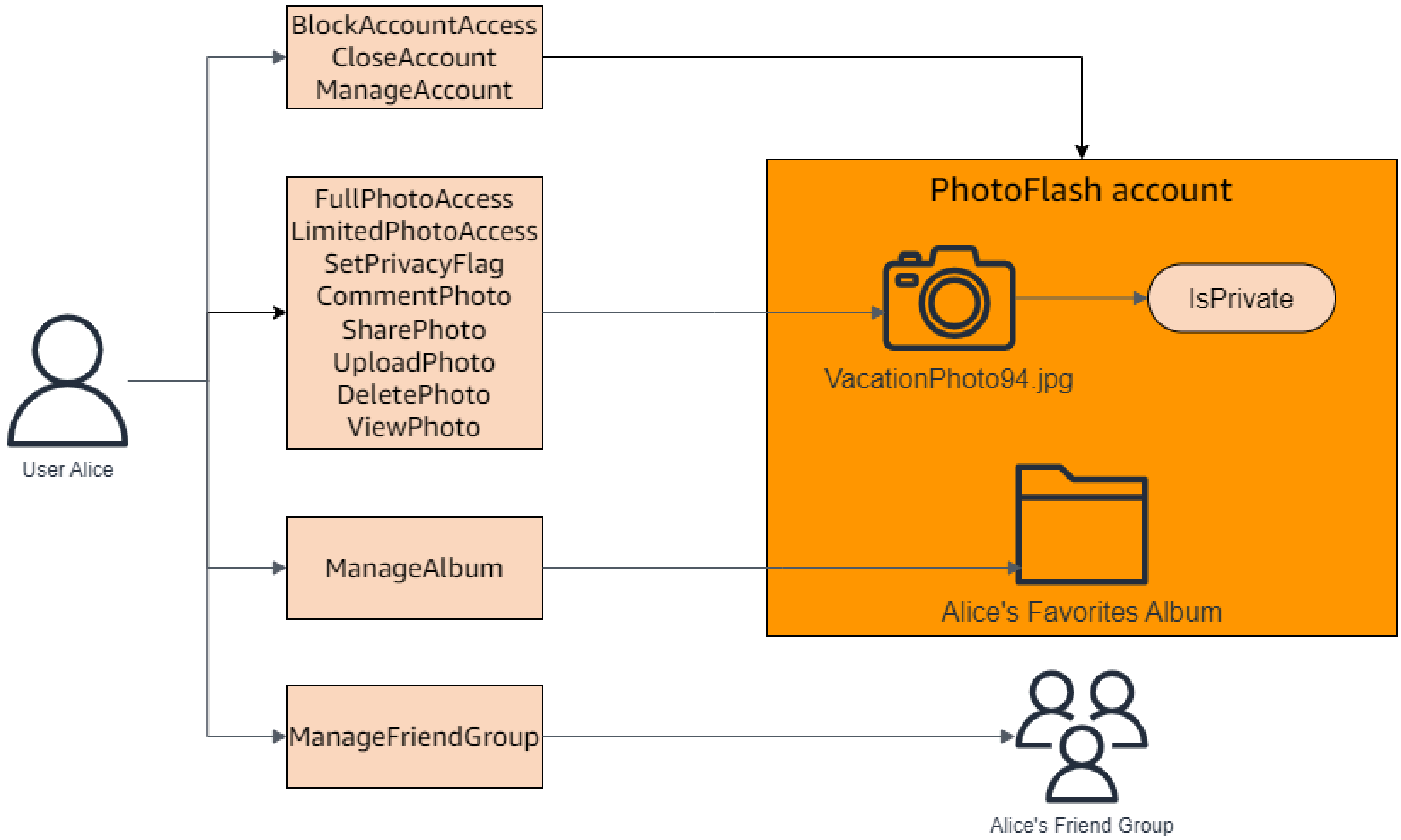
If this is your first time using Verified Permissions, we recommend using one of the sample policy stores to familiarize yourself with how Verified Permissions works. The sample policy stores provide pre-defined policies and a schema.

To create a policy store using the Sample policy store configuration method

1. In the [Verified Permissions console](#), select **Create new policy store**.
2. In the **Starting options** section, choose **Sample policy store**.
3. In the **Sample project** section, choose the type of sample Verified Permissions application to use. For this tutorial, choose the **PhotoFlash** policy store.
4. A namespace for the schema of your sample policy store is automatically generated based on the sample project you chose.
5. Choose **Create policy store**.

   Your policy store is created with policies, policy templates, and a schema for the sample policy store.

The diagram below illustrates the relationships between the PhotoFlash sample policy store actions and the resource types that they apply to.



Creating template-linked policies for a sample policy store

The PhotoFlash sample policy store includes policies, policy templates, and a schema. You can create template-linked policies based on the policy templates included with the sample policy store.

To create template-linked policies for the sample policy store

1. Open the Verified Permissions console at https://console.aws.amazon.com/verifiedpermissions/. Choose your policy store.
2. In the navigation pane on the left, choose **Policies**.
3. Choose **Create policy** and then choose **Create template-linked policy**.
4. Choose the radio button next to the policy template with the description **Grant full access to non-private shared photos** and then choose **Next**.

5. For **Principal**, enter `PhotoFlash::User::"Alice"`. For **Resource**, enter `PhotoFlash::Album::"Bob-Vacation-Album"`.
6. Choose **Create template-linked policy**.

   The new template-linked policy is displayed under **Policies**.
7. Create another template-linked policy for the PhotoFlash sample policy store. Choose **Create policy** and then choose **Create template-linked policy**.
8. Choose the radio button next to the policy template with the description **Grant limited access to non-private shared photos** and then choose **Next**.
9. For **Principal**, enter `PhotoFlash::FriendGroup::"MySchoolFriends"`. For **Resource**, enter `PhotoFlash::Album::"Alice’s favorite album"`.
10. Choose **Create template-linked policy**.

    The new template-linked policy is displayed under **Policies**.

We will test the new template-linked policies in the next section of the tutorial. For more examples of values you can use to create a template-linked policy for PhotoFlash, see PhotoFlash template-linked policy examples.

Testing a sample policy store

After creating your sample policy store and template-linked policies, you can test the sample Verified Permissions static policies and your new template-linked policies by running a simulated authorization request using the Verified Permissions test bench.

Depending on when you created your sample policy store, your policy templates might differ from the references in this procedure. Before you start this part of the tutorial, check that you have each policy template that follows in your PhotoFlash example policy store. If your policy doesn't align with these policies, edit the existing policies or create a new policy store from the **Sample project** option **PhotoFlash**.

Grant full access to non-private shared photos

```
permit (
    principal in ?principal,
    action in PhotoFlash::Action::"FullPhotoAccess",
    resource in ?resource
)
```

```
when { resource.IsPrivate == false };
```

Grant limited access to non-private shared photos

```
permit (
    principal in ?principal,
    action in PhotoFlash::Action::"LimitedPhotoAccess",
    resource in ?resource
)
when { resource.IsPrivate == false };
```

To test sample policy store policies

1. Open the Verified Permissions console at https://console.aws.amazon.com/verifiedpermissions/. Choose your policy store.
2. In the navigation pane on the left, choose **Test bench**.
3. Choose **Visual mode**.
4. In the **Principal** section, choose **PhotoFlash::User** from the principal types in your schema. Type an identifier for the user in the text box. For example, `Alice`.
5. Do not choose **Add a parent** for the principal.
6. For the **Account: Entity** attribute, make sure that the **PhotoFlash::Account** entity is selected. Type an identifier for the account. For example, `Alice-account`.
7. In the **Resource** section, choose the **PhotoFlash::Photo** resource type. Type an identifier for the photo in the text box. For example, `photo.jpeg`.
8. Choose **Add a parent** and choose **PhotoFlash::Account** for the entity type. Type the same identifier for the parent account for the photo that you specified in the **Account: Entity** field for the user. For example, `Alice-account`.
9. In the **Action** section, choose **PhotoFlash::Action::"ViewPhoto"** from the list of valid actions.
10. In the **Additional entities** section, choose **Add this entity** to add the suggested account entity.
11. Choose **Run authorization request** at the top of the page to simulate the authorization request for the Cedar policies in the sample policy store. The test bench should display the decision to allow the request.

The following table provides additional values for the principal, resource, and action you can test with the Verified Permissions test bench. The table includes the authorization request decision based on the static policies included with the PhotoFlash sample policy store and the template-linked policies you created in the previous section.

Principal value	Principal Account: Entity value	Resource value	Resource parent value	Action	Authorization decision
PhotoFlash::User | Alice	**PhotoFlash::Account |** Alice-account	**PhotoFlash::Photo |** photo.jpeg	**PhotoFlash::Account |** Bob-account	**PhotoFlash::Action::"ViewPhoto"**	Deny
PhotoFlash::User | Alice	**PhotoFlash::Account |** Alice-account	**PhotoFlash::Photo |** photo.jpeg	**PhotoFlash::Account |** Alice-account	**PhotoFlash::Action::"ViewPhoto"**	Allow
PhotoFlash::User | Alice	**PhotoFlash::Account |** Alice-account	**PhotoFlash::Photo |** Bob-photo.jpeg	**PhotoFlash::Album |** Bob-Vacation-Album	**PhotoFlash::Action::"ViewPhoto"**	Allow
PhotoFlash::User | Alice	**PhotoFlash::Account |** Alice-account	**PhotoFlash::Photo |** Bob-photo.jpeg	**PhotoFlash::Album |** Bob-Vacation-Album	**PhotoFlash::Action::"DeletePhoto"**	Deny
PhotoFlash::User | Alice	**PhotoFlash::Account |** Alice-account	**PhotoFlash::Photo** | Bob-photo.jpeg, **IsPrivate: Boolean |** true	**PhotoFlash::Album |** Bob-Vacation-Album	**PhotoFlash::Action::"ViewPhoto"**	Deny
PhotoFlash::User | Jane,	**PhotoFlash::Account |** Jane-account	**PhotoFlash::Photo |** photo.jpeg	**PhotoFlash::Album** | Alice’s	**PhotoFlash::Action**	Allow

Principal value	Principal Account: Entity value	Resource value	Resource parent value	Action	Authorization decision
PhotoFlash::FriendGroup | MySchoolFriends			favorite album	**::"ViewPhoto"**	
PhotoFlash::User | Jane, **PhotoFlash::FriendGroup |** MySchoolFriends	**PhotoFlash::Account |** Jane-account	**PhotoFlash::Photo |** photo.jpeg	**PhotoFlash::Album** | Alice's favorite album	**PhotoFlash::Action::"DeletePhoto"**	Deny

Create a policy store with a connected API and identity provider

A common use case for Amazon Verified Permissions is to authorize requests from an application client to a back-end API. AWS has a service for authentication of application users: Amazon Cognito. AWS also has a service for secure hosted APIs: Amazon API Gateway. When you combine a Verified Permissions policy store with these two AWS services, you can connect user pool authentication and API authorization in your application with a consistent, centralized set of policies. Verified Permissions policy stores have built-in support for Amazon Cognito user pool identity sources and API Gateway APIs.

To create a policy store that's linked to an existing user pool and API, choose **Set up with Cognito and API Gateway** when you create a new policy store.

An API-linked policy store automatically provisions your authorization model and resources for authorization requests. The **Set up with Cognito and API Gateway** creation process generates a policy store with a user pool identity source, and a Lambda authorizer that connects API Gateway to Verified Permissions. Initially, you can authorize API requests based on users' group memberships. For example, Verified Permissions can grant access only to users who are members of the `Directors` group.

As your application grows, you can implement fine-grained authorization with user attributes and OAuth 2.0 scopes. For example, Verified Permissions can grant access only to users who have an `email` attribute in the domain `mycompany.co.uk`.

After you have automated the authorization model for your API, your remaining responsibility is to authenticate users and generate API requests in your application, and to maintain your policy store.

To learn more, see API-linked policy stores.

Amazon Verified Permissions policy stores

A policy store is a container for policies and policy templates. Each policy store contains a schema that is used to validate policies added to the policy store. We recommend creating one policy store per application, or one policy store per tenant for multi-tenant applications. You must specify a policy store when making an authorization request.

We recommend using *namespaces* to Cedar entities in your policy stores to prevent ambiguity. A namespace is a string prefix for a type, separated by a pair of colons (`::`) as a delimiter. Verified Permissions supports one namespace per policy store. For more information, see Namespaces in the *Cedar policy language Reference Guide*.

Topics

Creating Verified Permissions policy stores

You can create a policy store using the following methods:

- **Follow a guided setup –** You will define a resource type with valid actions and a principal type before creating your first policy.
- **Set up with Cognito and API Gateway –** Define your principals from an Amazon Cognito user pool and actions and resources from an Amazon API Gateway API. We recommend this option if you want your application to authorize API requests with users' group membership.
- **Start from a sample policy store –** Choose a pre-defined sample project policy store. We recommend this option if you are learning about Verified Permissions and want to view and test example policies.
- **Create an empty policy store –** You will define the schema and all access policies yourself. We recommend this option if you are already familiar with configuring a policy store.

Guided setup

To create a policy store using the Guided setup configuration method

The guided setup wizard leads you through the process of creating the first iteration of your policy store. You will create a schema for your first resource type, describe the actions that are applicable for that resource type, and the principal type for which you are granting permissions. You will then create your first policy. Once you've completed this wizard, you will be able to add to your policy store, extend the schema to describe other resource and principal types, and create additional policies and templates.

1. In the Verified Permissions console, select **Create new policy store**.
2. In the **Starting options** section, choose **Guided setup**.
3. Enter a **Policy store description**. This text can be whatever suits your organization as a friendly reference to the function of the current policy store, for example *Weather updates*.
4. In the **Details** section, type a **Namespace for your schema**.
5. Choose **Next**.
6. On the **Resource type** window, type a name for your resource type.
7. (Optional) Choose **Add an attribute** to add resource attributes. Type the **Attribute name** and choose an **Attribute type** for each attribute of the resource. Choose whether each attribute is **Required**. Verified Permissions uses the specified attribute values when verifying policies against the schema. To remove an attribute that has been added for the resource type, choose **Remove** next to the attribute.
8. In the **Actions** field, type the actions to be authorized for the specified resource type. To add additional actions for the resource type, choose **Add an action**. To remove an action that has been added for the resource type, choose **Remove** next to the action.
9. In the **Name of the principal type** field, type the name for a type of principal that will be using the specified actions for your resource type.
10. Choose **Next**.
11. On the **Principal type** window, choose the identity source for your principal type.

    - Choose **Custom** if the principal's ID and attributes will be provided directly by your Verified Permissions application. Choose **Add an attribute** to add principal attributes. Type the **Attribute name** and choose an **Attribute type** for each attribute of the prinicpal. Verified Permissions uses the specified attribute values when verifying policies against the schema. To remove an attribute that has been added for the prinicpal type, choose **Remove** next to the attribute.

- Choose **Cognito User Pool** if the principal's ID and attributes will be provided from an ID or access token generated by Amazon Cognito. Choose **Connect user pool**. Select the **AWS Region** and type **User pool ID** of the Amazon Cognito user pool to connect to. Choose **Connect**. For more information, see Authorization with Amazon Verified Permissions in the *Amazon Cognito Developer Guide*.

12. Choose **Next**.
13. In the **Policy details** section, type an optional **Policy description** for your first Cedar policy.
14. In the **Principals scope** field, choose the principals that will be granted permissions from the policy.
    - Choose **Specific principal** to apply the policy to a specific principal. Choose the principal in the **Principal that will be permitted to take actions** field and type an entity identifier for the principal.
    - Choose **All principals** to apply the policy to all principals in your policy store.
15. In the **Resources scope** field, choose which resources that the specified principals will be authorized to act on.
    - Choose **Specific resource** to apply the policy to a specific resource. Choose the resource in the **Resource this policy should apply to** field and type an entity identifier for the resource.
    - Choose **All resources** to apply the policy to all resources in your policy store.
16. In the **Actions scope** field, choose which actions that the specified principals will be authorized to perform.
    - Choose **Specific set of actions** to apply the policy to specific actions. Select the check boxes next to the actions in the **Action(s) this policy should apply to** field.
    - Choose **All actions** to apply the policy to all actions in your policy store.
17. Review the policy in the **Policy preview** section. Choose **Create policy store**.

Set up with Cognito and API Gateway

To create a policy store using the Set up with Amazon Cognito and API Gateway configuration method

The Amazon Cognito and API Gateway option builds a policy store for testing authorization with user pool groups and an API with a Lambda authorizer.

The users and their groups in an Amazon Cognito user pool become your principals. The methods and paths in an API Gateway API become the actions that your policies authorize. Your application becomes the resource and the provider of context. As a result of this workflow, Verified Permissions creates a policy store, a Lambda function, and an API Lambda authorizer. You must assign the Lambda authorizer to your API after you finish this workflow.

1. In the Verified Permissions console, select **Create new policy store**.
2. In the **Starting options** section, choose **Set up with Cognito and API Gateway** and select **Next**.
3. In the **Import resources and actions** step, under **API**, choose an API that will function as the model to your policy store resources and actions.
    a. Choose a **Deployment stage** from the stages configured in your API and select **Import API**.
    b. Preview your **Map of imported resources and actions**.
    c. To update resources or actions, modify your API paths or methods and select **Import API**.
    d. When you are satisfied with your choices, choose **Next**.
4. In the **Choose identity source** step, under **Amazon Cognito user pools**, configure your identity source.
    a. Choose a user pool in the same AWS Region and AWS account as your policy store.
    b. Choose the **Token type to pass to API** that you want to submit for authorization. Either token types contains user groups, the foundation of this API-linked authorization model.
    c. Under **App client validation**, you can limit the scope of a policy store to a subset of the Amazon Cognito app clients in a multi-tenant user pool. To require that user authenticate with one or more specified app clients in your user pool, select **Only**

accept tokens with expected app client IDs. To accept any user who authenticates with the user pool, select **Don't validate app client IDs**.

d. Choose **Next**.

5. Verified Permissions queries your user pool for groups. The **Assign actions to groups** step creates policies for your policy store that permit group members to perform actions.

    a. Choose the **Groups from user pool** that you want to include in your policies.

    b. Assign actions to each of the groups that you selected.

    c. Choose **Next**.

6. In the **Deploy app integration**, review the steps that Verified Permissions will take to create your policy store and Lambda authorizer.

7. When you're ready to create the new resources, choose **Create and deploy**.

8. Keep the **Policy store status** step open in your browser to monitor the progress of resource creation by Verified Permissions.

9. After about an hour, or when the **Deploy Lambda authorizer** step shows **Success**, configure your authorizer.

    Verified Permissions will have created a Lambda function and a Lambda authorizer in your API. Choose **Open API** to navigate to your API.

    To learn how to assign a Lambda authorizer, see Use API Gateway Lambda authorizers in the *Amazon API Gateway Developer Guide*.

    a. Navigate to **Authorizers** for your API and note the name of the authorizer that Verified Permissions created.

    b. Navigate to **Resources** and select a top-level method in your API.

    c. Select **Edit** under **Method request settings**.

    d. Set the **Authorizer** to be the authorizer name you noted earlier.

    e. Expand **HTTP request headers**, enter a **Name** or `AUTHORIZATION`, and select **Required**.

    f. Deploy the API stage.

    g. **Save** your changes.

10. Test your authorizer with a user pool token of the **Token type** that you selected in the **Choose identity source** step. For more information about user pool sign-in and retrieving tokens, see User pool authentication flow in the *Amazon Cognito Developer Guide*.

11. Test authentication again with a user pool token in the `AUTHORIZATION` header of a request to your API.

12. Examine your new policy store. Add and refine policies.

Sample policy store

To create a policy store using the Sample policy store configuration method

1. In the **Starting options** section, choose **Sample policy store**.

2. In the **Sample project** section, choose the type of sample Verified Permissions application to use.

   - **PhotoFlash** is a sample customer-facing web application that enables users to share individual photos and albums with friends. Users can set fine-grained permissions on who is allowed to view, comment on, and re-share their photos. Account owners can also create groups of friends and organize photos into albums.

   - **DigitalPetStore** is a sample application where anyone can register and become a customer. Customers can add pets for sale, search pets, and place orders. Customers who have added a pet are recorded as the pet owner. Pet owners can update the pet's details, upload a pet image, or delete the pet listing. Customers who have placed an order are recorded as the order owner. Order owners can get details on the order or cancel it. Pet store managers have administrative access.

     > **Note**
     >
     > The **DigitalPetStore** sample policy store does not include policy templates. The **PhotoFlash** and **TinyTodo** sample policy stores include policy templates.

   - **TinyTodo** is a sample application that enables users to create taks and task lists. List owners can manage and share their lists and specify who can view or edit their lists.

3. A namespace for the schema of your sample policy store is automatically generated based on the sample project you chose.

4. Choose **Create policy store**.

   Your policy store is created with policies and a schema for the sample policy store you chose. For more information on template-linked policies you can create for the sample policy stores, see Example template-linked policies for Verified Permissions sample policy stores.

Empty policy store

To create a policy store using the Empty policy store configuration method

1. In the **Starting options** section, choose **Empty policy store**.
2. Choose **Create policy store**.

An empty policy store is created without a schema, which means policies are not validated. For more information about updating the schema for your policy store, see Amazon Verified Permissions policy store schema.

For more information about creating policies for your policy store, see Creating Amazon Verified Permissions static policies and Creating template-linked policies.

AWS CLI

To create an empty policy store by using the AWS CLI.

You can create a policy store by using the `create-policy-store` operation.

> **Note**
>
> A policy store that you create by using the AWS CLI is empty.
>
> - To add schema, see Amazon Verified Permissions policy store schema.
> - To add policies, see Creating Amazon Verified Permissions static policies.
> - To add policy templates, see Creating policy templates.

```
$ aws verifiedpermissions create-policy-store \
    --validation-settings "mode=STRICT"
{
    "arn": "arn:aws:verifiedpermissions::123456789012:policy-store/
PSEXAMPLEabcdefg111111",
    "createdDate": "2023-05-16T17:41:29.103459+00:00",
    "lastUpdatedDate": "2023-05-16T17:41:29.103459+00:00",
    "policyStoreId": "PSEXAMPLEabcdefg111111"
```

```
}
```

AWS SDKs

You can create a policy store using the `CreatePolicyStore` API. For more information, see CreatePolicyStore in the Amazon Verified Permissions API Reference Guide.

API-linked policy stores

When you create a new policy store in the Amazon Verified Permissions console, you can choose the **Set up with Cognito and API Gateway** option. With this option, you build an *API-linked policy store*, an authorization model for applications that authenticate with Amazon Cognito user pools and get data from Amazon API Gateway APIs. To get started, see Create a policy store with a connected API and identity provider.

Topics

> **Important**
>
> Policy stores that you create with the **Set up with Cognito and API Gateway** option in the Verified Permissions console aren't intended for immediate deployment to production. With your initial policy store, finalize your authorization model and export the policy store resources to CloudFormation. Deploy Verified Permissions to production programmatically with the AWS Cloud Development Kit (CDK). For more information, see Moving to production with AWS CloudFormation.

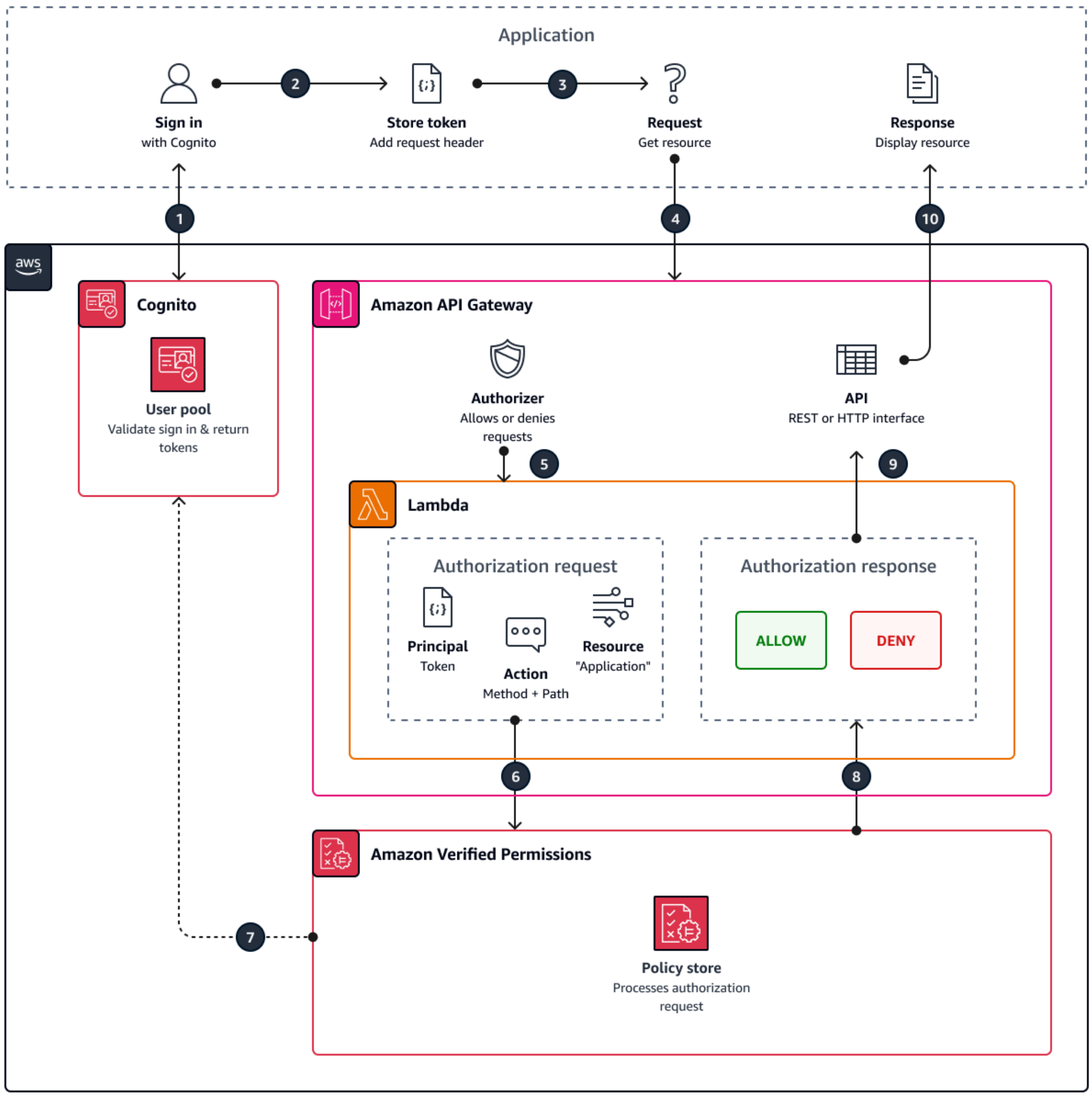
In a policy store that's linked to an API and user pool, your application presents a user pool token in an authorization header when it makes a request to the API. The identity source of your policy store, a user pool, provides token validation for Verified Permissions. The token forms the `principal` in authorization requests with the IsAuthorizedWithToken API. Verified Permissions builds policies around the group membership of your users, as presented in the `cognito:groups` claim in identity (ID) and access tokens. Your API processes the token from your application in a Lambda authorizer and submits it to Verified Permissions for an authorization decision. When your API receives the authorization decision from the Lambda authorizer, it passes the request on to your data source or denies the request.

Components of Amazon Cognito and API Gateway authorization with Verified Permissions

- An Amazon Cognito user pool that authenticates and groups users. Users' tokens populate the principal and group membership that Verified Permissions evaluates in your policy store.
- An API Gateway REST API. Verified Permissions defines actions from API paths and API methods, for example `MyAPI::Action::get /photo`.
- A Lambda function and a Lambda authorizer for your API. The Lambda function takes in bearer tokens from your user pool, requests authorization from Verified Permissions, and returns a decision to API Gateway. The **Set up with Cognito and API Gateway** workflow automatically creates this Lambda authorizer for you.
- A Verified Permissions policy store. The policy store identity source is your user pool. The policy store schema reflects the configuration of your API, and the policies link user groups to permitted API actions.
- An application that authenticates users with Amazon Cognito and appends tokens to API requests.

How Verified Permissions authorizes API requests

When you create a new policy store and select the **Set up with Cognito and API Gateway** option, Verified Permissions creates policy store schema and policies. The schema and policies reflect API actions and the user pool groups that you want to authorize to take the actions. Verified Permissions also creates the Lambda function and authorizer. You must configure the new authorizer on a method in your API.



1. Your user signs in with your application. Amazon Cognito issues ID and access tokens with the user's information.
2. Your application stores the JWTs. For more information, see [Using tokens with user pools]() in the *Amazon Cognito Developer Guide*..
3. Your user requests data that your application must retrieve from an external API.

4. Your application requests data from a REST API in API Gateway. It appends an ID or access token as a request header.
5. If your API has a cache for the authorization decision, it returns the previous response. If caching is disabled or the API has no current cache, API Gateway passes the request parameters to a token-based Lambda authorizer.
6. The Lambda function sends an authorization request to a Verified Permissions policy store with the IsAuthorizedWithToken API. The Lambda function passes the elements of an authorization decision:
   a. The user pool token as the principal.
   b. The API method combined with the API path, for example `GetPhoto`, as the action.
   c. The term `Application` as the resource.
7. Verified Permissions validates the user pool token. For more information, see Authorization with Amazon Verified Permissions in the *Amazon Cognito Developer Guide*.
8. Verified Permissions evaluates the authorization request against the policies in your policy store and returns an authorization decision.
9. The Lambda authorizer returns an `Allow` or `Deny` response to API Gateway.
10. The API returns data or an `ACCESS_DENIED` response to your application. Your application processes and displays the results of the API request.

Adding attribute-based access control (ABAC)

A typical authentication session with an Amazon Cognito user pool returns ID and access tokens. You can pass either of these token types as a bearer token in application requests to your API. Depending on your choices when you create your policy store, Verified Permissions expects one of the two types of tokens. Both types carry information about the user's group membership. For more information about token types in Amazon Cognito, see Using tokens with user pools in the *Amazon Cognito Developer Guide*.

After you create a policy store, you can add and extend policies. For example, you can add new groups to your policies as you add them to your user pool. Because your policy store is already aware of the way that your user pool presents groups in tokens, you can permit a set of actions for any new group with a new policy.

You might also want to extend the group-based model of policy evaluation into a more precise model based on user properties. User pool tokens contain additional user information that can contribute to authorization decisions.

ID tokens

ID tokens represent a user’s attributes and have the highest level of fine-grained access control. To evaluate email addresses, phone numbers, or custom attributes like department and manager, evaluate the ID token.

Access tokens

Access tokens represent a user’s permissions with OAuth 2.0 scopes. To add a layer of authorization or to set up requests for additional resources, evaluate the access token. For example, you can validate that a user is in the appropriate groups *and* carries a scope like `PetStore.read` that generally authorizes access to the API. User pools can add custom scopes to tokens with resource servers and with token customization at runtime.

See Mapping Amazon Cognito tokens to Verified Permissions schema for example policies that process claims in ID and access tokens.

Considerations for API-linked policy stores

When you build an API-linked policy store in the Verified Permissions console, you're creating a test for an eventual production deployment. Before you move to production, establish a fixed configuration for your API and user pool. Consider the following factors:

API Gateway caches responses

In API-linked policy stores, Verified Permissions creates a Lambda authorizer with an **Authorization caching** TTL of 120 seconds. You can adjust this value or turn off caching in your authorizer. In an authorizer with caching enabled, your authorizer returns the same response each time until the TTL expires. This can extend the effective lifetime of user pool tokens by a duration that equals the caching TTL of the requested stage.

Amazon Cognito groups can be reused

Amazon Verified Permissions determines group membership for user pool users from the `cognito:groups` claim in a user's ID or access token. The value of this claim is an array of the friendly names of the user pool groups that the user belongs to. You can't associate user pool groups with a unique identifier.

User pool groups that you delete and recreate with the same name present to your policy store as the same group. When you delete a group from a user pool, delete all references to the group from your policy store.

API-derived namespace and schema are point-in-time

Verified Permissions captures your API at a *point in time*: it only queries your API when you create your policy store. When the schema or name of your API changes, you must update your policy store and Lambda authorizer, or create a new API-linked policy store. Verified Permissions derives the policy store namespace from the name of your API.

Lambda function has no VPC configuration

The Lambda function that Verified Permissions creates for your API authorizer isn't connected to a VPC. By default. APIs that have network access restricted to private VPCs can't communicate with the Lambda function that authorizes access requests with Verified Permissions.

Verified Permissions deploys authorizer resources in CloudFormation

To create an API-linked policy store, you must sign in a highly-privileged AWS principal to the Verified Permissions console. This user deploys an AWS CloudFormation stack that creates resources across several AWS services. This principal must have the permission to add and modify resources in Verified Permissions, IAM, Lambda, and API Gateway. As a best practice, don't share these credentials with other administrators in your organization.

See Moving to production with AWS CloudFormation for an overview of the resources that Verified Permissions creates.

Moving to production with AWS CloudFormation

API-linked policy stores are a way to quickly build an authorization model for an API Gateway API. They are designed to serve as a testing environment for the authorization component of your application. After you create your test policy store, spend time refining the policies, schema, and Lambda authorizer.

You might adjust the architecture of your API, requiring equivalent adjustments to your policy store schema and policies. API-linked policy stores don't automatically update their schema from API architecture–Verified Permissions only polls the API at the time you create a policy store. If your API changes sufficiently, you might have to repeat the process with a new policy store.

When your application and authorization model are ready for deployment to production, integrate the API-linked policy store that you developed with your automation processes. As a best practice, we recommend that you export the policy store schema and policies into a AWS CloudFormation template that you can deploy to other AWS accounts and AWS Regions.

The results of the API-linked policy store process are an initial policy store and a Lambda authorizer. The Lambda authorizer has several dependent resources. Verified Permissions deploys these resources in an automatically-generated CloudFormation stack. To deploy to production, you must collect the policy store and the Lambda authorizer resources into a template. An API-linked policy store is made of the following resources:

1. AWS::VerifiedPermissions::PolicyStore: Copy your schema to the `SchemaDefinition` object. Escape `"` characters as `\"`.
2. AWS::VerifiedPermissions::IdentitySource: Copy values from the output of GetIdentitySource from your test policy store and modify as needed.
3. One or more of AWS::VerifiedPermissions::Policy: Copy your policy statement to the `Definition` object. Escape `"` characters as `\"`.
4. AWS::Lambda::Function, AWS::IAM::Role, AWS::IAM::Policy, AWS::ApiGateway::Authorizer, AWS::Lambda::Permission: Copy the template from the **Template** tab of the stack that Verified Permissions deployed when you created your policy store.

The following template is an example policy store. You can append the Lambda authorizer resources from your existing stack to this template.

```
{
    "AWSTemplateFormatVersion": "2010-09-09",
    "Resources": {
        "MyExamplePolicyStore": {
            "Type": "AWS::VerifiedPermissions::PolicyStore",
            "Properties": {
                "ValidationSettings": {
                    "Mode": "STRICT"
                },
                "Description": "ApiGateway: PetStore/test",
                "Schema": {
                    "CedarJson": "{\"PetStore\":{\"actions\":{\"get /pets\":
{\"appliesTo\":{\"principalTypes\":[\"User\"],\"resourceTypes\":[\"Application\"],
\"context\":{\"type\":\"Record\",\"attributes\":{}}}},\"get /\":{\"appliesTo\":
{\"principalTypes\":[\"User\"],\"resourceTypes\":[\"Application\"],\"context\":{\"type
```

```
\":\"Record\",\"attributes\":{}}}},\"get /pets/{petId}\":{\"appliesTo\":{\"context
\":{\"type\":\"Record\",\"attributes\":{}},\"resourceTypes\":[\"Application\"],
\"principalTypes\":[\"User\"]}},\"post /pets\":{\"appliesTo\":{\"principalTypes\":
[\"User\"],\"resourceTypes\":[\"Application\"],\"context\":{\"type\":\"Record\",
\"attributes\":{}}}}},\"entityTypes\":{\"Application\":{\"shape\":{\"type\":\"Record\",
\"attributes\":{}}},\"User\":{\"memberOfTypes\":[\"UserGroup\"],\"shape\":{\"attributes
\":{},\"type\":\"Record\"}},\"UserGroup\":{\"shape\":{\"type\":\"Record\",\"attributes
\":{}}}}}}"
                }
            }
        },
        "MyExamplePolicy": {
            "Type": "AWS::VerifiedPermissions::Policy",
            "Properties": {
                "Definition": {
                    "Static": {
                        "Description": "Policy defining permissions for testgroup
 cognito group",
                        "Statement": "permit(\nprincipal in PetStore::UserGroup::
\"us-east-1_EXAMPLE|testgroup\",\naction in [\n  PetStore::Action::\"get /\",
\n  PetStore::Action::\"post /pets\",\n  PetStore::Action::\"get /pets\",\n
 PetStore::Action::\"get /pets/{petId}\"\n],\nresource);"
                    }
                },
                "PolicyStoreId": {
                    "Ref": "MyExamplePolicyStore"
                }
            },
            "DependsOn": [
                "MyExamplePolicyStore"
            ]
        },
        "MyExampleIdentitySource": {
            "Type": "AWS::VerifiedPermissions::IdentitySource",
            "Properties": {
                "Configuration": {
                    "CognitoUserPoolConfiguration": {
                        "ClientIds": [
                            "1example23456789"
                        ],
                        "GroupConfiguration": {
                            "GroupEntityType": "PetStore::UserGroup"
                        },
```

```
                        "UserPoolArn": "arn:aws:cognito-idp:us-
east-1:123456789012:userpool/us-east-1_EXAMPLE"
                    }
                },
                "PolicyStoreId": {
                    "Ref": "MyExamplePolicyStore"
                },
                "PrincipalEntityType": "PetStore::User"
            },
            "DependsOn": [
                "MyExamplePolicyStore"
            ]
        }
    }
}
```

Troubleshooting API-linked policy stores

Use the information here to help you diagnose and fix common issues when you build Amazon Verified Permissions API-linked policy stores.

Topics

- I updated my policy but the authorization decision didn't change
- I attached the Lambda authorizer to my API but it's not generating authorization requests
- I received an unexpected authorization decision and want to review the authorization logic
- I want to find logs from my Lambda authorizer
- My Lambda authorizer doesn't exist
- My API is in a private VPC and can't invoke the authorizer
- I want to process additional user attributes in my authorization model
- I want to add new actions, action context attributes, or resource attributes

I updated my policy but the authorization decision didn't change

By default, Verified Permissions configures the Lambda authorizer to cache authorization decisions for 120 seconds. Try again after two minutes, or disable cache on your authorizer. For more information, see Enabling API caching to enhance responsiveness in the *Amazon API Gateway Developer Guide*.

I attached the Lambda authorizer to my API but it's not generating authorization requests

To begin processing requests, you must deploy the API stage that you attached your authorizer to. For more information, see Deploying a REST API in the *Amazon API Gateway Developer Guide*.

I received an unexpected authorization decision and want to review the authorization logic

The API-linked policy store process creates a Lambda function for your authorizer. Verified Permissions automatically builds the logic of your authorization decisions into the authorizer function. You can go back after you create your policy store to review and update the logic in the function.

To locate your Lambda function from the AWS CloudFormation console, choose the **Check deployment** button on the **Overview** page of your new policy store.

You can also locate your function in the AWS Lambda console. Navigate to the console in the AWS Region of your policy store and search for a function name with a prefix of `AVPAuthorizerLambda`. If you have create more than one API-linked policy store, use the **Last modified** time of your functions to correlate them with policy store creation.

I want to find logs from my Lambda authorizer

Lambda functions collect metrics and log their invocation results in Amazon CloudWatch. To review your logs, locate your function in the Lambda console and choose the **Monitor** tab. Select **View CloudWatch logs** and review the entries in the log group.

For more information about Lambda function logs, see Using Amazon CloudWatch Logs with AWS Lambda in the *AWS Lambda Developer Guide*.

My Lambda authorizer doesn't exist

After you complete setup of an API-linked policy store, you must attach the Lambda authorizer to your API. If you can't locate your authorizer in the API Gateway console, the additional resources for your policy store might have failed or not deployed yet. API-linked policy stores deploy these resources in an AWS CloudFormation stack.

Verified Permissions displays a link with the label **Check deployment** at the end of the creation process. If you already navigated away from this screen, go to the CloudFormation console and search recent stacks for a name that's prefixed with `AVPAuthorizer-<policy store ID>`. CloudFormation provides valuable troubleshooting information in the output of a stack deployment.

For help troubleshooting CloudFormation stacks, see Troubleshooting CloudFormation in the *AWS CloudFormation User Guide*.

My API is in a private VPC and can't invoke the authorizer

Verified Permissions doesn't support access to Lambda authorizers through VPC endpoints. You must open a network path between your API and the Lambda function that serves as your authorizer.

I want to process additional user attributes in my authorization model

The API-linked policy store process derives Verified Permissions policies from the groups claim in users' tokens. To update your authorization model to consider additional user attributes, integrate those attributes in your policies.

You can map many claims in ID and access tokens from Amazon Cognito user pools to Verified Permissions policy statements. For example, most users have an `email` claim in their ID token. For more information about adding claims from your identity source to policies, see Mapping Amazon Cognito tokens to Verified Permissions schema.

I want to add new actions, action context attributes, or resource attributes

An API-linked policy store and the Lambda authorizer that it creates are a point-in-time resource. They reflect the state of your API at the time of creation. The policy store schema doesn't assign any context attributes to actions, nor any attributes or parents to the default `Application` resource.

When you add actions—paths and methods—to your API, you must update your policy store to be aware of the new actions. You must also update your Lambda authorizer to process authorization requests for the new actions. You can start again with a new policy store or you can update your existing policy store.

To update your existing policy store, locate your function. Examine the logic in the automatically-generated function and update it to process the new actions, attributes, or context. Then edit your schema to include the new actions and attributes.

Switching Verified Permissions policy stores

AWS Management Console

To switch policy stores or create additional policy stores

1. Open the Verified Permissions console at https://console.aws.amazon.com/verifiedpermissions/. Choose your policy store.
2. In the navigation pane on the left, choose **switch** next to **Current policy store**.
3. You can switch between existing policy stores or create additional policy stores.
   - To switch policy stores, choose the policy store ID of the policy store to switch to.
   - To create a new policy store, choose **Create new policy store**. Follow the instructions in Creating Verified Permissions policy stores.

AWS CLI

To switch policy stores or create additional policy stores

The AWS CLI doesn't maintain a "default" policy store. Instead, most AWS CLI commands use the `--policy-store-id` to specify which policy store to use for each command.

To create a new policy store, use the create-policy-store command.

Deleting Verified Permissions policy stores

AWS Management Console

To delete a policy store

1. Open the Verified Permissions console at https://console.aws.amazon.com/verifiedpermissions/. Choose your policy store.
2. In the navigation pane on the left, choose **Settings**.
3. Choose **Delete this policy store**.
4. Type `delete` in the text box and choose **Delete**.

AWS CLI

To delete a policy store

You can delete a policy store by using the `delete-policy-store` operation.

```
$ aws verifiedpermissions delete-policy-store \
    --policy-store-id PSEXAMPLEabcdefg111111
```

This command produces no output if successful.

Amazon Verified Permissions policy store schema

A *schema* is a declaration of the structure of the entity types supported by your application, and the actions your application may provide in authorization requests.

For more information, see Cedar schema format in the Cedar policy language Reference Guide.

> **Note**
>
> The use of schemas in Verified Permissions is optional, but they are highly recommended for production software. When you create a new policy, Verified Permissions can use the schema to validate the entities and attributes referenced in the scope and conditions to avoid typos and mistakes in policies that can lead to confusing system behavior. If you activate policy validation, then all new policies must conform with the schema.

AWS Management Console

To create a schema

1. Open the Verified Permissions console at https://console.aws.amazon.com/verifiedpermissions/. Choose your policy store.
2. In the navigation pane on the left, choose **Schema**.
3. Choose **Create schema**.

AWS CLI

To submit a new schema, or overwrite an existing schema by using the AWS CLI.

You can create a policy store by running a AWS CLI command similar to the following example.

Consider a schema that contains the following Cedar content:

```
{
    "MySampleNamespace": {
        "actions": {
            "remoteAccess": {
```

```
                "appliesTo": {
                    "principalTypes": [ "Employee" ]
                }
            }
        },
        "entityTypes": {
            "Employee": {
                "shape": {
                    "type": "Record",
                    "attributes": {
                        "jobLevel": {"type": "Long"},
                        "name": {"type": "String"}
                    }
                }
            }
        }
    }
}
```

You must first escape the JSON into a single line string, and preface it with a declaration of its data type: `cedarJson`. The following example uses the following contents of `schema.json` file that contains the escaped version of the JSON schema.

Note

The example here is line wrapped for readability. You must have the entire file on a single line for the command to accept it.

```
{"cedarJson": "{\"MySampleNamespace\": {\"actions\": {\"remoteAccess\": {\"appliesTo
\":
{\"principalTypes\": [\"Employee\"]}}},\"entityTypes\": {\"Employee\": {\"shape\":
{\"attributes\": {\"jobLevel\": {\"type\": \"Long\"},\"name\": {\"type\": \"String
\"}},
\"type\": \"Record\"}}}}}"}
```

```
$ aws verifiedpermissions put-schema \
    --definition file://schema.json \
    --policy-store PSEXAMPLEabcdefg111111
{
    "policyStoreId": "PSEXAMPLEabcdefg111111",
```

```
    "namespaces": [
        "MySampleNamespace"
    ],
    "createdDate": "2023-07-17T21:07:43.659196+00:00",
    "lastUpdatedDate": "2023-08-16T17:03:53.081839+00:00"
}
```

AWS SDKs

You can create a policy store using the `PutSchema` API. For more information, see PutSchema in the Amazon Verified Permissions API Reference Guide.

Editing schemas in Visual mode

When you select **Schema** in the Verified Permissions console, the **Visual mode** displays the **Entity types** and **Actions** that make up your schema. At this top-level view or from within the details of any entity, you can choose **Edit schema** to begin to make updates to your schema. Visual mode isn’t available with some schema formats like nested records.

The visual schema editor begins with a series of diagrams that illustrate the relationships between the entities in your schema.Choose **Expand** to maximize your view of the entity relationships of your schema.

Actions diagram

The **Actions diagram** view lists the types of **Principals** you have configured in your policy store, the **Actions** they are eligible to perform, and the **Resources** that they are eligible to perform actions on. The lines between entities indicate your ability to create a policy that allows a principal to take an action on a resource. If your actions diagram doesn't indicate a relationship between two entities, you must create that relationship between them before you can allow or deny it in policies. Select an entity to see a properties overview and drill down to view full details. Choose **Filter by this [action | resource type | principal type]** to see an entity in a view with only its own connections.

Entity types diagram

The **Entity types diagram** focuses on the relationships between principals and resources. When you want to understand the complex nested parent relationships in your schema, review this diagram. Hover over an entity to drill down into the parent relationships that it has.

Under the diagrams are list views of the **Entity types** and **Actions** in your schema. The list view is useful when you want to immediately view the details of a specific action or entity type. Select any entity to view details.

To edit a Verified Permissions schema in Visual mode

1. Open the Verified Permissions console at https://console.aws.amazon.com/verifiedpermissions/. Choose your policy store.
2. In the navigation pane on the left, choose **Schema**.
3. Choose **Visual mode**. Review the entity-relationship diagrams and plan the changes that you want to make to your schema. You can optionally **Filter by** one entity to examine its individual connections to other entities.
4. Choose **Edit schema**.
5. In the **Details** section, type a **Namespace** for your schema.
6. In the **Entity types** section, choose **Add new entity type**.
7. Type the name of the entity.
8. (Optional) Choose **Add a parent** to add parent entities that the new entity is a member of. To remove a parent that has been added to the entity, choose **Remove** next to the name of the parent.
9. Choose **Add an attribute** to add attributes to the entity. Type the **Attribute name** and choose the **Attribute type** for each attribute of the entity. Verified Permissions uses the specified attribute values when verifying policies against the schema. Select whether each attribute is **Required**. To remove an attribute that has been added to the entity, choose **Remove** next to the attribute.
10. Choose **Add entity type** to add the entity to the schema.
11. In the **Actions** section, choose **Add new action**.
12. Type the name of the action.
13. (Optional) Choose **Add a resource** to add resource types for which the action applies to. To remove a resource type that has been added to the action, choose **Remove** next to the name of the resource type.
14. (Optional) Choose **Add a principal** to add a principal type that the action applies to. To remove a principal type that has been added to the action, choose **Remove** next to the name of the principal type.

15. Choose **Add an attribute** to add attributes that can be added to the context of an action in your authorization requests. Enter the **Attribute name** and choose the **Attribute type** for each attribute. Verified Permissions uses the specified attribute values when verifying policies against the schema. Select whether each attribute is **Required**. To remove an attribute that has been added to the action, choose **Remove** next to the attribute.
16. Choose **Add action**.
17. After all the entity types and actions have been added to the schema, choose **Save changes**.

Editing schemas in JSON mode

To edit a Verified Permissions schema in JSON mode

1. Open the Verified Permissions console at https://console.aws.amazon.com/verifiedpermissions/. Choose your policy store.
2. In the navigation pane on the left, choose **Schema**.
3. Choose **JSON mode** and then choose **Edit schema**.
4. Enter the content of your JSON schema in the **Contents** field. You can't save updates to your schema until you resolve all syntax errors. You can choose **Format JSON** to format the JSON syntax of your schema with the recommended spacing and indentation.
5. Choose **Save changes**.

Deleting a schema

AWS Management Console

To delete a Verified Permissions schema

1. Open the Verified Permissions console at https://console.aws.amazon.com/verifiedpermissions/. Choose your policy store.
2. In the navigation pane on the left, choose **Schema**.
3. Choose **Delete schema**.

AWS CLI

To delete a Verified Permissions schema

There isn't a delete schema command. You can delete the schema in a policy store by using the `put-schema` command with an empty schema in `cedarJson` field. An empty schema is represented by a pair of curly braces '**{}**'.

```
$ aws verifiedpermissions put-schema \
    --policy-store-id PSEXAMPLEabcdefg111111 \
    --definition cedarJson='{}'{
    "policyStoreId": "PSEXAMPLEabcdefg111111",
    "namespaces": [],
    "createdDate": "2023-06-14T21:55:27.347581Z",
    "lastUpdatedDate": "2023-06-19T17:55:04.95944Z"
}
```

Amazon Verified Permissions policy validation mode

You can set the policy validation mode in Verified Permissions to control whether policy changes are validated against the schema in your policy store.

> **Important**
>
> When you turn on policy validation, all attempts to create or update a policy or policy template are validated against the schema in the policy store. Verified Permissions rejects the request if validation fails.

AWS Management Console

To set the policy validation mode for a policy store

1. Open the Verified Permissions console at https://console.aws.amazon.com/verifiedpermissions/. Choose your policy store.
2. Choose **Settings**.
3. In the **Policy validation mode** section, choose **Modify**.
4. Do one of the following:
   - To activate policy validation and enforce that all policy changes must be validated against your schema, choose the **Strict (recommended)** radio button.
   - To turn off policy validation for policy changes, choose the **Off** radio button. Type `confirm` to confirm that updates to policies will no longer be validated against your schema.
5. Choose **Save changes**.

AWS CLI

To set the validation mode for a policy store

You can change the validation mode for a policy store by using the UpdatePolicyStore operation and specifying a different value for the ValidationSettings parameter.

```
$ aws verifiedpermissions update-policy-store \
```

```
    --validation-settings "mode=OFF",
    --policy-store-id PSEXAMPLEabcdefg111111
{
    "createdDate": "2023-05-17T18:36:10.134448+00:00",
    "lastUpdatedDate": "2023-05-17T18:36:10.134448+00:00",
    "policyStoreId": "PSEXAMPLEabcdefg111111",
    "validationSettings": {
        "Mode": "OFF"
    }
}
```

For more information, see Policy validation in the *Cedar policy language Reference Guide*.

Amazon Verified Permissions policies

A *policy* is a statement that either permits or forbids a *principal* to take one or more *actions* on a *resource*. Each policy is evaluated independently of any other policy. For more information about how Cedar policies are structured and evaluated, see [Cedar policy validation against schema] in the Cedar policy language Reference Guide.

> **Important**
>
> When you write Cedar policies that reference principals, resources and actions, you can define the unique identifiers used for each of those elements. We strongly recommend that you follow these best practices:
>
> - **Use values like universally unique identifiers (UUIDs) for all principal and resource identifiers.**
>
>   For example, if user `jane` leaves the company, and you later let someone else use the name `jane`, then that new user automatically gets access to everything granted by policies that still reference `User::"jane"`. Cedar can't distinguish between the new user and the old. This applies to both principal and resource identifiers. Always use identifiers that are guaranteed unique and never reused to ensure that you don't unintentionally grant access because of the presence of an old identifier in a policy.
>
>   Where you use a UUID for an entity, we recommend that you follow it with the // comment specifier and the 'friendly' name of your entity. This helps to make your policies easier to understand. For example: principal == User::"a1b2c3d4-e5f6-a1b2-c3d4-EXAMPLE11111", // alice
> - **Do not include personally identifying, confidential, or sensitive information as part of the unique identifier for your principals or resources.** These identifiers are included in log entries shared in AWS CloudTrail trails.

Entity formatting in Amazon Verified Permissions

Amazon Verified Permissions uses the Cedar policy language to create policies. The syntax of policies and the data types supported match the syntax and data types outlined in [Basic policy construction in Cedar] and [Data types supported by Cedar] topics in the Cedar policy language

Reference Guide. However, there are differences between Verified Permissions and Cedar in the formatting of entities when making an authorization request.

The JSON formatting of entities in Verified Permissions differs from Cedar in the following ways:

- In Verified Permissions, a JSON object must have all of its key-value pairs wrapped in a JSON object with the name of `Record`.
- A JSON list in Verified Permissions must be wrapped in a JSON key-value pair where the key name is `Set` and the value is the original JSON list from Cedar.
- For `String`, `Long`, and `Boolean` type names, each key-value pair from Cedar is replaced by a JSON object in Verified Permissions. The name of the object is the original key name. Inside the JSON object, there is one key-value pair where the key name is the type name of the scalar value (`String`, `Long`, or `Boolean`) and the value is the value from the Cedar entity.
- The syntax formatting of Cedar entities and Verified Permissions entities differs in the following ways:

Cedar format	Verified Permissions format
`uid`	`Identifier`
`type`	`EntityType`
`id`	`EntityId`
`attrs`	`Attributes`
`parents`	`Parents`

The following example shows how entities in a list are formatted using Cedar.

```
[
  {
    "number": 1
  },
  {
    "sentence": "Here is an example sentence"
  },
  {
    "Question": false
```

```
  }
]
```

The following exmple shows how the same entities from the previous Cedar list example are formatted in Verified Permissions.

```
{
  "Set": [
    {
      "Record": {
        "number": {
          "Long": 1
        }
      }
    },
    {
      "Record": {
        "sentence": {
          "String": "Here is an example sentence"
        }
      }
    },
    {
      "Record": {
        "question": {
          "Boolean": false
        }
      }
    }
  ]
}
```

The following example shows how Cedar entities are formatted for evaluating a policy in an authorization request.

```
[
    {
        "uid": {
            "type": "PhotoApp::User",
            "id": "alice"
        },
        "attrs": {
            "age": 25,
```

```
            "name": "alice",
            "userId": "123456789012"
        },
        "parents": [
            {
                "type": "PhotoApp::UserGroup",
                "id": "alice_friends"
            },
            {
                "type": "PhotoApp::UserGroup",
                "id": "AVTeam"
            }
        ]
    },
    {
        "uid": {
            "type": "PhotoApp::Photo",
            "id": "vacationPhoto.jpg"
        },
        "attrs": {
            "private": false,
            "account": {
                "__entity": {
                    "type": "PhotoApp::Account",
                    "id": "ahmad"
                }
            }
        },
        "parents": []
    },
    {
        "uid": {
            "type": "PhotoApp::UserGroup",
            "id": "alice_friends"
        },
        "attrs": {},
        "parents": []
    },
    {
        "uid": {
            "type": "PhotoApp::UserGroup",
            "id": "AVTeam"
        },
        "attrs": {},
```

```
        "parents": []
    }
]
```

The following example shows how the same entities from the previous Cedar example are formatted in Verified Permissions.

```
[
    {
        "Identifier": {
            "EntityType": "PhotoApp::User",
            "EntityId": "alice"
        },
        "Attributes": {
            "age": {
                "Long": 25
            },
            "name": {
                "String": "alice"
            },
            "userId": {
                "String": "123456789012"
            }
        },
        "Parents": [
            {
                "EntityType": "PhotoApp::UserGroup",
                "EntityId": "alice_friends"
            },
            {
                "EntityType": "PhotoApp::UserGroup",
                "EntityId": "AVTeam"
            }
        ]
    },
    {
        "Identifier": {
            "EntityType": "PhotoApp::Photo",
            "EntityId": "vacationPhoto.jpg"
        },
        "Attributes": {
            "private": {
                "Boolean": false
```

```
            },
            "account": {
                "EntityIdentifier": {
                    "EntityType": "PhotoApp::Account",
                    "EntityId": "ahmad"
                }
            }
        },
        "Parents": []
    },
    {
        "Identifier": {
            "EntityType": "PhotoApp::UserGroup",
            "EntityId": "alice_friends"
        },
        "Parents": []
    },
    {
        "Identifier": {
            "EntityType": "PhotoApp::UserGroup",
            "EntityId": "AVTeam"
        },
        "Parents": []
    }
]
```


Creating Amazon Verified Permissions static policies

You can create a Cedar static policy to allow or deny principals to perform specified actions on specified resources for your application.

AWS Management Console

To create a static policy

1. Open the Verified Permissions console at https://console.aws.amazon.com/verifiedpermissions/. Choose your policy store.
2. In the navigation pane on the left, choose **Policies**.
3. Choose **Create policy** and then choose **Create static policy**.
4. In the **Policy effect** section, choose whether the policy will **Permit** or **Forbid** when a request matches the policy.

5. In the **Principals scope** field, choose the scope of the principals that the policy will apply to.

   - Choose **Specific principal** to apply the policy to a specific principal. Specify the entity type and identifier for the principal that will be permitted for forbidden to take the actions specified in the policy.
   - Choose **Group of principals** to apply the policy to a group of principals. Type the principal group name in the **Group of principals** field.
   - Choose **All principals** to apply the policy to all principals in your policy store.

6. In the **Resources scope** field, choose the scope of the resources that the policy will apply to.

   - Choose **Specific resources** to apply the policy to a specific resource. Specify the entity type and identifier for the resource that the policy should apply to.
   - Choose **Group of resources** to apply the policy to a group of resources. Type the resource group name in the **Group of resources** field.
   - Choose **All resources** to apply the policy to all resources in your policy store.

7. In the **Actions scope** section, choose the scope of the resources that the policy will apply to.

   - Choose **Specific set of actions** to apply the policy to a set of actions. Select the check boxes next to the actions to apply the policy.
   - Choose **All actions** to apply the policy to all actions in your policy store.

8. Choose **Next**.
9. In the **Policy** section, review your Cedar policy. You can choose **Format** to format the syntax of your policy with the recommended spacing and indentation. For more information, see Basic policy construction in Cedar in the Cedar policy language Reference Guide.
10. In the **Details** section, type an optional description of the policy.
11. Choose **Create policy**.

AWS CLI

To create a static policy

You can create a static policy by using the CreatePolicy operation. The following example creates a simple static policy.

```
$ aws verifiedpermissions create-policy \
    --definition "{ \"static\": { \"Description\": \"MyTestPolicy\", \"Statement\":
 \"permit(principal,action,resource) when {principal.owner == resource.owner};\"}}"
 \
    --policy-store-id PSEXAMPLEabcdefg111111
{
"Arn": "arn:aws:verifiedpermissions::123456789012:policy/PSEXAMPLEabcdefg111111/
SPEXAMPLEabcdefg111111",
    "createdDate": "2023-05-16T20:33:01.730817+00:00",
    "lastUpdatedDate": "2023-05-16T20:33:01.730817+00:00",
    "policyId": "SPEXAMPLEabcdefg111111",
    "policyStoreId": "PSEXAMPLEabcdefg111111",
    "policyType": "STATIC"
}
```

Editing Amazon Verified Permissions static policies

You can edit an existing Cedar static policy in your policy store. You can directly update only static policies. You can change only certain elements of a static policy:

- The `action` referenced by the policy.
- A condition clause, such as `when` and `unless`.

You can't change these elements of a static policy:

- Changing a policy from a static policy to a template-linked policy.
- Changing the effect of a static policy from `permit` or `forbid`.
- The `principal` referenced by a static policy.
- The `resource` referenced by a static policy.

To change a template-linked policy, you must update the template instead. For more information, see Editing policy templates.

AWS Management Console

To edit a static policy

1. Open the Verified Permissions console at https://console.aws.amazon.com/verifiedpermissions/. Choose your policy store.
2. In the navigation pane on the left, choose **Policies**.
3. Choose the radio button next to the static policy to edit and then choose **Edit**.
4. In the **Policy body** section, update the `action` or condition clause of your static policy. You can't update the policy effect, `principal`, or `resource` of the policy.
5. Choose **Update policy**.

> **Note**
>
> If policy validation is enabled in the policy store, then updating a static policy causes Verified Permissions to validate the policy against the schema in the policy store. If the updated static policy doesn't pass validation, the operation fails and the update isn't saved.

AWS CLI

To edit a static policy

You can edit a static policy by using the UpdatePolicy operation. The following example edits a simple static policy.

The example uses the file `definition.txt` to contain the policy definition.

```
{
    "static": {
        "description":  "Grant everyone of janeFriends UserGroup access to the
 vacationFolder Album",
        "statement": "permit(principal in UserGroup::\"janeFriends\", action,
 resource in Album::\"vacationFolder\" );"
    }
}
```

The following command references that file.

```
$ aws verifiedpermissions create-policy \
    --definition file://definition.txt \
    --policy-store-id PSEXAMPLEabcdefg111111

{
    "createdDate": "2023-06-12T20:33:37.382907+00:00",
    "lastUpdatedDate": "2023-06-12T20:33:37.382907+00:00",
    "policyId": "SPEXAMPLEabcdefg111111",
    "policyStoreId": "PSEXAMPLEabcdefg111111",
    "policyType": "STATIC",
    "principal": {
        "entityId": "janeFriends",
        "entityType": "UserGroup"
    },
    "resource": {
        "entityId": "vacationFolder",
        "entityType": "Album"
    }
}
```

Viewing policies

AWS Management Console

To view your Verified Permissions policies

1. Open the Verified Permissions console at https://console.aws.amazon.com/verifiedpermissions/. Choose your policy store.
2. In the navigation pane on the left, choose **Policies**. All of the policies you have created are displayed.
3. Choose the **Search** text box to filter policies by **Principal** or **Resource**.
4. Choose the radio button next to a policy to display details about the policy, such as when the policy was created, updated, and the policy contents.
5. You can delete a policy by choosing the radio button next to a policy and then choosing **Delete**. Choose **Delete policy** to confirm deleting the policy.

AWS CLI

To list all of the available policies in a policy store

You can view the list of policies by using the GetPolicy operation. The following example retrieves a list that includes a static policy and a template-linked policy.

```
$ aws verifiedpermissions list-policies \
    --policy-store-id PSEXAMPLEabcdefg111111
{
    "Policies": [
        {
            "createdDate": "2023-05-17T18:38:31.359864+00:00",
            "definition": {
                "static": {
                    "Description": "Grant everyone of janeFriends UserGroup access
 to the vacationFolder Album"
                }
            },
            "lastUpdatedDate": "2023-05-18T16:15:04.366237+00:00",
            "policyId": "SPEXAMPLEabcdefg111111",
            "policyStoreId": "PSEXAMPLEabcdefg111111",
            "policyType": "STATIC",
            "resource": {
                "entityId": "publicFolder",
                "entityType": "Album"
            }
        },
        {
            "createdDate": "2023-05-22T18:57:53.298278+00:00",
            "definition": {
                "templateLinked": {
                    "policyTemplateId": "PTEXAMPLEabcdefg111111"
                }
            },
            "lastUpdatedDate": "2023-05-22T18:57:53.298278+00:00",
            "policyId": "TPEXAMPLEabcdefg111111",
            "policyStoreId": "PSEXAMPLEabcdefg111111",
            "policyType": "TEMPLATELINKED",
            "principal": {
                "entityId": "alice",
                "entityType": "User"
            },
            "resource": {
```

```
                "entityId": "VacationPhoto94.jpg",
                "entityType": "Photo"
            }
        }
    ]
}
```

To view the details for an individual policy

You can retrieve the details for a policy by using the GetPolicy operation. The following example retrieves details for a template-linked policy.

```
$ aws verifiedpermissions get-policy \
    --policy-id TPEXAMPLEabcdefg111111
    --policy-store-id PSEXAMPLEabcdefg111111

{
    "arn": "arn:aws:verifiedpermissions::123456789012:policy/PSEXAMPLEabcdefg111111/
TPEXAMPLEabcdefg111111",
    "createdDate": "2023-03-15T16:03:07.620867Z",
    "lastUpdatedDate": "2023-03-15T16:03:07.620867Z",
    "policyDefinition": {
        "templatedPolicy": {
            "policyTemplateId": "PTEXAMPLEabcdefg111111",
            "principal": {
                "entityId": "alice",
                "entityType": "User"
            },
            "resource": {
                "entityId": "Vacation94.jpg",
                "entityType": "Photo"
            }
        }
    },
    "policyId": "TPEXAMPLEabcdefg111111",
    "policyStoreId": "PSEXAMPLEabcdefg111111",
    "policyType": "TEMPLATELINKED",
    "principal": {
        "entityId": "alice",
        "entityType": "User"
    },
    "resource": {
        "entityId": "Vacation94.jpg",
```

```
        "entityType": "Photo"
    }
}
```

Amazon Verified Permissions example policies

The following Verified Permissions policy examples are based on the schema defined for the hypothetical application called PhotoFlash described in the Example schema section of the Cedar policy language Reference Guide. For more information about Cedar policy syntax, see Basic policy construction in Cedar in the Cedar policy language Reference Guide.

Policy examples

- Allows access to individual entities
- Allows access to groups of entities
- Allows access for any entity
- Allows access for attributes of an entity (ABAC)
- Denies access

Allows access to individual entities

This example shows how you might create a policy that allows the user `alice` to view the photo `VacationPhoto94.jpg`.

```
permit(
  principal == User::"alice",
  action == Action::"view",
  resource == Photo::"VacationPhoto94.jpg"
);
```

Allows access to groups of entities

This example shows how you might create a policy that allows anyone in the group `alice_friends` to view the photo `VacationPhoto94.jpg`.

```
permit(
  principal in Group::"alice_friends",
```

```
  action == Action::"view",
  resource == Photo::"VacationPhoto94.jpg"
);
```

This example shows how you might create a policy that allows the user `alice` to view any photo in the album `alice_vacation`.

```
permit(
  principal == User::"alice",
  action == Action::"view",
  resource in Album::"alice_vacation"
);
```

This example shows how you might create a policy that allows the user `alice` to view, edit, or delete any photo in the album `alice_vacation`.

```
permit(
  principal == User::"alice",
  action in [Action::"view", Action::"edit", Action::"delete"],
  resource in Album::"alice_vacation"
);
```

This example shows how you might create a policy that allows permissions for the user `alice` in the album `alice_vacation`, where `admin` is a group defined in the schema hierarchy that contains the permissions to view, edit, and delete a photo.

```
permit(
  principal == User::"alice",
  action in PhotoflashRole::"admin",
  resource in Album::"alice_vacation"
);
```

This example shows how you might create a policy that allows permissions for the user `alice` in the album `alice_vacation`, where `viewer` is a group defined in the schema hierarchy that contains the permission to view and comment on a photo. The user `alice` is also granted the `edit` permission by the second action listed in the policy.

```
permit(
  principal == User::"alice",
  action in [PhotoflashRole::"viewer", Action::"edit"],
```

```
  resource in Album::"alice_vacation"
)
```

Allows access for any entity

This example shows how you might create a policy that allows any authenticated principal to view the album `alice_vacation`.

```
permit(
  principal,
  action == Action::"view",
  resource in Album::"alice_vacation"
);
```

This example shows how you might create a policy that allows the user `alice` list all the albums in the `jane` account, list the photos in each album, and view photos in the account.

```
permit(
  principal == User::"alice",
  action in [Action::"listAlbums", Action::"listPhotos", Action::"view"],
  resource in Account::"jane"
);
```

This example shows how you might create a policy that allows the user `alice` to perform any action on resources in the album `jane_vaction`.

```
permit(
  principal == User::"alice",
  action,
  resource in Album::"jane_vacation"
);
```

Allows access for attributes of an entity (ABAC)

Attribute-based access control (ABAC) is an authorization strategy that defines permissions based on attributes. Verified Permissions allows attributes to be attached to principals, actions, and resources. These attributes can then be referenced within the `when` and `unless` clauses of policies that evaluate the attributes of the principals, actions, and resources that make up the context of the request.

The following examples use the attributes defined in the hypothetical application called PhotoFlash described in the [Example schema] section of the Cedar policy language Reference Guide.

This example shows how you might create a policy that allows any principal in the `HardwareEngineering` department with a job level of greater than or equal to 5 to view and list photos in the album `device_prototypes`.

```
permit(
  principal,
  action in [Action::"listPhotos", Action::"view"],
  resource in Album::"device_prototypes"
)
when {
  principal.department == "HardwareEngineering" &&
  principal.jobLevel >= 5
};
```

This example shows how you might create a policy that allows the user `alice` to view any resource of file type `JPEG`.

```
permit(
  principal == User::"alice",
  action == Action::"view",
  resource
)
when {
  resource.fileType == "JPEG"
};
```

Actions have *context attributes*. You must pass these attributes in the `context` of an authorization request. This example shows how you might create a policy that allows the user `alice` to perform any `readOnly` action. You can also set an `appliesTo` property for actions in your schema. This specifies valid actions for a resource when you want to ensure that, for example, users can only attempt to authorize `ViewPhoto` for a resource of type `PhotoFlash::Photo`.

```
permit(
  principal == PhotoFlash::User::"alice",
  action,
  resource
) when {
```

```
    context has readOnly &&
    context.readOnly == true
};
```

A better way to set the properties of actions in your schema, however, is to arrange them into functional action groups. For example, you can create an action named `ReadOnlyPhotoAccess` and set `PhotoFlash::Action::"ViewPhoto"` to be a member of `ReadOnlyPhotoAccess` as an action group. This example shows how you might create a policy that grants Alice access to the read-only actions in that group.

```
permit(
  principal == PhotoFlash::User::"alice",
  action,
  resource
) when {
    action in PhotoFlash::Action::"ReadOnlyPhotoAccess"
};
```

This example shows how you might create a policy that allows all principals to perform any action on resources for which they have `owner` attribute.

```
permit(
  principal,
  action,
  resource
)
when {
  principal == resource.owner
};
```

This example shows how you might create a policy that allows any principal to view any resource if the `department` attribute for the principal matches the `department` attribute of the resource.

> **Note**
>
> If an entity doesn't have an attribute mentioned in a policy condition, then the policy will be ignored when making an authorization decision and evaluation of that policy fails for that entity. For example, any principal that does not have a `department` attribute cannot be granted access to any resource by this policy.

```
permit(
  principal,
  action == Action::"view",
  resource
)
when {
  principal.department == resource.owner.department
};
```

This example shows how you might create a policy that allows any principal to perform any action on a resource if the principal is the `owner` of the resource OR if the principal is part of the `admins` group for the resource.

```
permit(
  principal,
  action,
  resource,
)
when {
  principal == resource.owner |
  resource.admins.contains(principal)
};
```

Denies access

If a policy contains `forbid` for the effect of the policy, it constrains permissions instead of granting permissions.

> **Important**
>
> During authorization, if both a `permit` and `forbid` policy are enforced, the `forbid` takes precedence.

The following examples use the attributes defined in the hypothetical application called PhotoFlash described in the [Example schema](#) section of the Cedar policy language Reference Guide.

This example shows how you might create a policy that denies the user `alice` from performing all actions except `readOnly` on any resource.

```
forbid (
  principal == User::"alice",
  action,
  resource
)
unless {
  action.readOnly
};
```

This example shows how you might create a policy that denies access to all resources that have a `private` attribute unless the principal has the `owner` attribute for the resource.

```
forbid (
  principal,
  action,
  resource
)
when {
  resource.private
}
unless {
  principal == resource.owner
};
```

Amazon Verified Permissions policy templates

You can create Cedar policy templates in Verified Permissions to define an access control rule for your system. Policy templates are Cedar policies with placeholders for the `principal`, `resource`, or both. Policy templates allow a policy to be defined once and then attached to multiple principals and resources. Updates to the policy template are reflected across all principals and resources that use the template. For more information, see [Cedar policy templates](#) in the Cedar policy language Reference Guide.

We recommend using policy templates to create policies that can be shared throughout your application. For example, you could create a policy template for an editor that provides read, edit, and comment permissions for the principal and resource that use the policy template.

```
permit(
  principal == ?principal,
  action in [Action::"Read", Action::"Edit", Action::"Comment"],
  resource == ?resource
);
```

When a principal is designated as an editor for a resource, your application could instantiate a policy using the template to provide permissions for the principal to perform the read, edit, and comment actions on the resource.

Creating policy templates

AWS Management Console

To create a policy template

1. Open the Verified Permissions console at https://console.aws.amazon.com/verifiedpermissions/. Choose your policy store.
2. In the navigation pane on the left, choose **Policy templates**.
3. Choose **Create policy template**.
4. In the **Details** section, type a **Policy template description**.
5. In the **Policy template body** section, use placeholders `?principal` and `?resource` to allow policies created based on this template to customize permissions they grant. You

can choose **Format** to format the syntax of your policy template with the recommended spacing and indentation.

6. Choose **Create policy template**.

AWS CLI

To create a policy template

You can create a policy template by using the [CreatePolicyTemplate](#) operation. The following example creates a policy template with a placeholder for the principal.

The file `template1.txt` contains the following.

```
"VacationAccess"
permit(
    principal in ?principal,
    action == Action::"view",
    resource == Photo::"VacationPhoto94.jpg"
);
```

```
$ aws verifiedpermissions create-policy-template \
    --description "Template for vacation picture access"
    --statement file://template1.txt
    --policy-store-id PSEXAMPLEabcdefg111111
{
    "createdDate": "2023-05-18T21:17:47.284268+00:00",
    "lastUpdatedDate": "2023-05-18T21:17:47.284268+00:00",
    "policyStoreId": "PSEXAMPLEabcdefg111111",
    "policyTemplateId": "PTEXAMPLEabcdefg111111"
}
```

Creating template-linked policies

You can create template-linked policies to link to a policy template. Template-linked policies stay linked to their policy templates. If you change the policy statement in the policy template, any policies linked to that template automatically use the new statement for all authorization decisions made from that moment forward.

AWS Management Console

To create a template-linked policy by instantiating a policy template

1. Open the Verified Permissions console at https://console.aws.amazon.com/verifiedpermissions/. Choose your policy store.
2. In the navigation pane on the left, choose **Policies**.
3. Choose **Create policy** and then choose **Create template-linked policy**.
4. Choose the radio button next to the policy template to use and then choose **Next**.
5. Type the **Principal** and **Resource** to be used for this specific instance of the template-linked policy. The specified values are displayed in the **Policy statement preview** field.

   > **Note**
   >
   > The **Principal** and **Resource** values must have the same formatting as static policies. For example, to specify the `AdminUsers` group for the principal, type `Group::"AdminUsers"`. If you type `AdminUsers`, a validation error is displayed.

6. Choose **Create template-linked policy**.

   The new template-linked policy is displayed under **Policies**.

AWS CLI

To create a template-linked policy by instantiating a policy template

You can create a template-linked policy that references an existing policy template and that specifies values for any placeholders used by the template.

The following example creates a template-linked policy that uses a template with the following statement:

```
permit(
    principal in ?principal,
    action == Action::"view",
    resource == Photo::"VacationPhoto94.jpg"
);
```

It also uses the following `definition.txt` file to supply the value for the `definition` parameter:

```
{
    "templateLinked": {
        "policyTemplateId": "pt-4651be67-c128-4d22-8e67-9b068980c631",
        "principal": {
            "entityType": "User",
            "entityId": "alice"
        }
    }
}
```

The output shows both the resource, which it gets from the template, and the principal, which it gets from the definition parameter

```
$ aws verifiedpermissions create-policy \
    --definition file://definition.txt
    --policy-store-id PSEXAMPLEabcdefg111111
{
    "createdDate": "2023-05-22T18:57:53.298278+00:00",
    "lastUpdatedDate": "2023-05-22T18:57:53.298278+00:00",
    "policyId": "TPEXAMPLEabcdefg111111",
    "policyStoreId": "PSEXAMPLEabcdefg111111",
    "policyType": "TEMPLATELINKED",
    "principal": {
        "entityId": "alice",
        "entityType": "User"
    },
    "resource": {
        "entityId": "VacationPhoto94.jpg",
        "entityType": "Photo"
    }
}
```

Editing policy templates

AWS Management Console

To edit your policy templates

1. Open the Verified Permissions console at https://console.aws.amazon.com/verifiedpermissions/. Choose your policy store.
2. In the navigation pane on the left, choose **Policy templates**. The console displays all of the policy templates you created in the current policy store.
3. Choose the radio button next to a policy template to display details about the policy template, such as when the policy template was created, updated, and the policy template contents.
4. Choose **Edit** to edit your policy template. Update the **Policy description** and **Policy body** as necessary and then choose **Update policy template**.
5. You can delete a policy template by choosing the radio button next to a policy template and then choosing **Delete**. Choose **OK** to confirm deleting the policy template.

AWS CLI

To update a policy template

You can create a static policy by using the UpdatePolicy operation. The following example updates the specified policy template by replacing its policy body with a new policy defined in a file.

Contents of file `template1.txt`:

```
permit(
    principal in ?principal,
    action == Action::"view",
    resource in ?resource)
when {
    principal has department && principal.department == "research"
};
```

```
$ aws verifiedpermissions update-policy-template \
    --policy-template-id PTEXAMPLEabcdefg111111 \
    --description "My updated template description" \
```

```
    --statement file://template1.txt \
    --policy-store-id PSEXAMPLEabcdefg111111
{
    "createdDate": "2023-05-17T18:58:48.795411+00:00",
    "lastUpdatedDate": "2023-05-17T19:18:48.870209+00:00",
    "policyStoreId": "PSEXAMPLEabcdefg111111",
    "policyTemplateId": "PTEXAMPLEabcdefg111111"
}
```

Example template-linked policies for Verified Permissions sample policy stores

When you create a policy store in Verified Permissions using the **Sample policy store** method, your policy store is created with predefined policies, policy templates, and a schema for the sample project you chose. The following Verified Permissions template-linked policy examples can be used with the sample policy stores and their respective policies, policy templates, and schemas.

PhotoFlash template-linked policy examples

This example shows how you might create a template-linked policy that uses the policy template **Grant limited access to non-private shared photos** with an individual user and photo.

> **Note**
>
> Cedar policy language considers an entity to be `in` itself. Therefore, `principal in User::"Alice"` is equivalent to `principal == User::"Alice"`.

```
permit (
 principal in PhotoFlash::User::"Alice",
 action in PhotoFlash::Action::"SharePhotoLimitedAccess",
 resource in PhotoFlash::Photo::"VacationPhoto94.jpg"
 );
```

This example shows how you might create a template-linked policy that uses the policy template **Grant limited access to non-private shared photos** with an individual user and album.

```
permit (
```

```
 principal in PhotoFlash::User::"Alice",
 action in PhotoFlash::Action::"SharePhotoLimitedAccess",
 resource in PhotoFlash::Album::"Italy2023"
 );
```

This example shows how you might create a template-linked policy that uses the policy template **Grant limited access to non-private shared photos** with a friend group and individual photo.

```
permit (
 principal in PhotoFlash::FriendGroup::"Jane::MySchoolFriends",
 action in PhotoFlash::Action::"SharePhotoLimitedAccess",
 resource in PhotoFlash::Photo::"VacationPhoto94.jpg"
 );
```

This example shows how you might create a template-linked policy that uses the policy template **Grant limited access to non-private shared photos** with a friend group and album.

```
permit (
 principal in PhotoFlash::FriendGroup::"Jane::MySchoolFriends",
 action in PhotoFlash::Action::"SharePhotoLimitedAccess",
 resource in PhotoFlash::Album::"Italy2023"
 );
```

This example shows how you might create a template-linked policy that uses the policy template **Grant full access to non-private shared photos** with a friend group and an individual photo.

```
permit (
 principal in PhotoFlash::UserGroup::"Jane::MySchoolFriends",
 action in PhotoFlash::Action::"SharePhotoFullAccess",
 resource in PhotoFlash::Photo::"VacationPhoto94.jpg"
 );
```

This example shows how you might create a template-linked policy that uses the policy template **Block user from an account**.

```
forbid(
 principal == PhotoFlash::User::"Bob",
 action,
 resource in PhotoFlash::Account::"Alice-account"
 );
```

DigitalPetStore

The DigitalPetStore sample policy store does not include any policy templates. You can view the policies included with the policy store by choosing **Policies** in the navigation pane on the left after creating the **DigitalPetStore** sample policy store.

TinyToDo template-linked policy examples

This example shows how you might create a template-linked policy that uses the policy template that gives viewer access for an individual user and task list.

```
permit (
    principal == TinyTodo::User::"https://cognito-idp.us-east-1.amazonaws.com/us-
east-1_h2aKCU1ts|5ae0c4b1-6de8-4dff-b52e-158188686f31|bob",
    action in [TinyTodo::Action::"ReadList", TinyTodo::Action::"ListTasks"],
    resource == TinyTodo::List::"1"
);
```

This example shows how you might create a template-linked policy that uses the policy template that gives editor access for an individual user and task list.

```
permit (
    principal == TinyTodo::User::"https://cognito-idp.us-east-1.amazonaws.com/us-
east-1_h2aKCU1ts|5ae0c4b1-6de8-4dff-b52e-158188686f31|bob",
    action in [
        TinyTodo::Action::"ReadList",
        TinyTodo::Action::"UpdateList",
        TinyTodo::Action::"ListTasks",
        TinyTodo::Action::"CreateTask",
        TinyTodo::Action::"UpdateTask",
        TinyTodo::Action::"DeleteTask"
    ],
    resource == TinyTodo::List::"1"
);
```

Using Amazon Verified Permissions with identity providers

You can use OpenID Connect (OIDC) identity providers (IdPs) with Verified Permissions to pass user attributes to use as principals in Verified Permissions policies.

Working with Amazon Cognito and identity sources

An *identity source* is a collection of user information referenced by an identity provider for simplifying authorization requests to your policy stores. You can create an identity source to provide information about principals for your Verified Permissions applications. You can specify the AWS Region, Amazon Cognito user pool ID, and principal type of your identity sources. Because Verified Permissions only works with Amazon Cognito user pools in the same AWS account, you can't specify an identity source in another account.

User pool JSON Web Token (JWT) *claims* contain user attributes. You can add custom claims to inform the authorization decisions that Verified Permissions makes. Identity token claims include `cognito:username` and `cognito:groups`. For more information, see Using tokens with user pools in the *Amazon Cognito Developer Guide*.

> **Important**
>
> Although you can revoke Amazon Cognito tokens before they expire, JWTs are considered to be stateless resources that are self-contained with a signature and validity. Services that conform with the JSON Web Token RFC 7519 are expected to validate tokens remotely and aren't required to validate them with the issuer. This means that it is possible for Verified Permissions to grant access based on a token that was revoked or issued for user that was later deleted. To mitigate this risk, we recommend that you create your tokens with the shortest possible validity duration and revoke refresh tokens when you want to remove authorization to continue a user's session.

You can also add custom attributes to your user pool. Custom attributes require the `custom:` prefix to distinguish them from standard attributes. For example, you can add a `custom:joblevel` attribute to a user pool. For more information, see Custom attributes in the *Amazon Cognito Developer Guide*.

When writing Cedar policies in Verified Permissions using Amazon Cognito user pool claims and attributes that contain a `:` character, you must reference them in Cedar policies with a period (`.`) instead of a colon (`:`) to comply with Cedar policy syntax. For example, you must change `cognito:username` and `cognito:groups` to `cognito.username` and `cognito.groups` respectively.

> **Note**
>
> If a token contains a claim with a `cognito:` or `custom:` prefix and a `cognito` or `custom` claim, an authorization request with [IsAuthorizedWithToken](#) will fail with a `ValidationException`.

This example shows how you might create a policy that references Amazon Cognito user pools custom claims corresponding to a principal.

```
permit(
     principal == ExampleCo::User::"us-east-1_example|4fe90f4a-ref8d9-4033-
a750-4c8622d62fb6",
     action,
     resource == ExampleCo::Photo::"VacationPhoto94.jpg"
) when { principal.cognito.username == "alice" };
```

For more information, see [Authorization with Amazon Verified Permissions](#) in the *Amazon Cognito Developer Guide*.

Working with other identity providers

You can extract your entity attributes from a JSON Web Token (JWT) from any OpenID Connect (OIDC) provider (IdP) and parse it into Verified Permissions.

This example shows how you might call Verified Permissions from an OIDC-compliant IdP.[1]

```
async function authorizeUsingJwtToken(jwtToken) {

    const payload = await verifier.verify(jwtToken);

    var principalEntity = {
        entityType: "PhotoFlash::User", // the application needs to fill in the
 relevant user type
```

```
        entityId: payload["sub"], // the application need to use the claim that
 represents the user-id
    };
    var resourceEntity = {
        entityType: "PhotoFlash::Photo", //the application needs to fill in the
 relevant resource type
        entityId: "jane_photo_123.jpg", // the application needs to fill in the
 relevant resource id
    };
    var action = {
        actionType: "PhotoFlash::Action", //the application needs to fill in the
 relevant action id
        actionId: "GetPhoto", //the application needs to fill in the relevant action
 type
    };
    var entities = {
        entityList: [],
    };
    entities.entityList.push(...getUserEntitiesFromToken(payload));
    var policyStoreId = "PSEXAMPLEabcdefg111111"; // set your own policy store id

    const authResult = await client
        .isAuthorized({
        policyStoreId: policyStoreId,
        principal: principalEntity,
        resource: resourceEntity,
        action: action,
        entities,
        })
        .promise();

    return authResult;

}

function getUserEntitiesFromToken(payload) {
  let attributes = {};
  let claimsNotPassedInEntities = ['aud', 'sub', 'exp', 'jti', 'iss'];
  Object.entries(payload).forEach(([key, value]) => {
    if (claimsNotPassedInEntities.includes(key)) {
        return;
    }
    if (Array.isArray(value)) {
      var attibuteItem = [];
```

```
      value.forEach((item) => {
        attibuteItem.push({
          string: item,
        });
      });
      attributes[key] = {
        set: attibuteItem,
      };
    } else if (typeof value === 'string') {
      attributes[key] = {
        string: value,
      }
    } else if (typeof value === 'bigint' || typeof value ==='number') {
        attributes[key] = {
            long: value,
          }
    } else if (typeof value === 'bigint' || typeof value ==='number') {
        attributes[key] = {
            long: value,
          }
    } else if (typeof value === 'boolean') {
        attributes[key] = {
            boolean: value,
       }
    }

  });

  let entityItem = {
    attributes: attributes,
    identifier: {
      entityType: "PhotoFlash::User",
      entityId: payload["sub"], // the application need to use the claim that
 represents the user-id
    }
  };
  return [entityItem];
}
```

[1] This code example uses the aws-jwt-verify library for verifying JWTs signed by OIDC-compatible IdPs.

Creating Amazon Verified Permissions identity sources

The following procedure adds an Amazon Cognito identity source to your policy store.

> **Note**
>
> **Identity sources** is not available in the navigation pane on the left until you have created a policy store. Identity sources that you create are associated with the current policy store.

You can leave out the principal entity type when you create an identity source with [create-identity-source](#) in the AWS CLI or [CreateIdentitySource](#) in the Verified Permissions API. However, a blank entity type creates an identity source with an entity type of `AWS::Cognito`. This entity name isn't compatible with policy store schema. To integrate Amazon Cognito identities with your policy store schema, you must set the principal entity type to a supported policy store entity.

AWS Management Console

To create an Amazon Cognito user pools identity source

1. Open the Verified Permissions console at https://console.aws.amazon.com/verifiedpermissions/. Choose your policy store.
2. In the navigation pane on the left, choose **Identity sources**.
3. Choose **Create identity source**.
4. In the **Cognito user pool details** section, select the AWS Region and type the **User pool ID** for your identity source.
5. In the **Principal details** section, choose the **Principal type** for the identity source. Identities from the connected Amazon Cognito user pools will be mapped to the selected principal type.
6. In the **Client application validation** section, choose whether to validate client application IDs.

   • To validate client application IDs, choose **Only accept tokens with matching client application IDs**. Choose **Add new client application ID** for each client application ID to validate. To remove a client application ID that has been added, choose **Remove** next to the client application ID.

- Choose **Do not validate client application IDs** if you do not want to validate client application IDs.

7. Choose **Create identity source**.
8. Before you can reference attributes you extract from identity or access tokens in your Cedar policies, you must update your schema to make Cedar aware of the type of principal that your identity source creates. That addition to the schema must include the attributes that you want to reference in your Cedar policies. For more information about mapping Amazon Cognito token attributes to Cedar principal attributes, see Mapping Amazon Cognito tokens to Verified Permissions schema.

AWS CLI

To create an Amazon Cognito user pools identity source

You can an create an identity source by using the CreateIdentitySource operation. The following example creates an identity source that can access authenticated identities from a Amazon Cognito user pool.

The following `config.txt` file contains the details of the Amazon Cognito user pool for use by the --configuration parameter in the `create-identity-source` command.

```
{
    "cognitoUserPoolConfiguration": {
        "userPoolArn": "arn:aws:cognito-idp:us-west-2:123456789012:userpool/us-
west-2_1a2b3c4d5",
        "clientIds":["a1b2c3d4e5f6g7h8i9j0kalbmc"]
    }
}
```

Command:

```
$ aws verifiedpermissions create-identity-source \
    --configuration file://config.txt \
    --principal-entity-type "User" \
    --policy-store-id 123456789012
{
    "createdDate": "2023-05-19T20:30:28.214829+00:00",
    "identitySourceId": "ISEXAMPLEabcdefg111111",
    "lastUpdatedDate": "2023-05-19T20:30:28.214829+00:00",
    "policyStoreId": "PSEXAMPLEabcdefg111111"
```

```
}
```

Before you can reference attributes you extract from identity or access tokens in your Cedar policies, you must update your schema to make Cedar aware of the type of principal that your identity source creates. That addition to the schema must include the attributes that you want to reference in your Cedar policies. For more information about mapping Amazon Cognito token attributes to Cedar principal attributes, see Mapping Amazon Cognito tokens to Verified Permissions schema.

For more information about using Amazon Cognito access and identity tokens for authenticated users in Verified Permissions, see Authorization with Amazon Verified Permissions in the *Amazon Cognito Developer Guide*.

Editing Amazon Verified Permissions identity sources

AWS Management Console

To update an Amazon Cognito user pools identity source

1. Open the Verified Permissions console at https://console.aws.amazon.com/verifiedpermissions/. Choose your policy store.
2. In the navigation pane on the left, choose **Identity sources**.
3. Choose the ID of the identity source to edit.
4. Choose **Edit**.
5. In the **Cognito user pool details** section, select the AWS Region and type the **User pool ID** for your identity source.
6. In the **Principal details** section, you can update the **Principal type** for the identity source. Identities from the connected Amazon Cognito user pools will be mapped to the selected principal type.
7. In the **Client application validation** section, choose whether to validate client application IDs.

    - To validate client application IDs, choose **Only accept tokens with matching client application IDs**. Choose **Add new client application ID** for each client application ID to validate. To remove a client application ID that has been added, choose **Remove** next to the client application ID.

- Choose **Do not validate client application IDs** if you do not want to validate client application IDs.

8. Choose **Save changes**.
9. If you changed the principal type for the identity source, you must update your schema to correctly reflect the updated principal type.

You can delete an identity source by choosing the radio button next to an identity source and then choosing **Delete identity source**. Type `delete` in the text box and then choose **Delete identity source** to confirm deleting the identity source.

AWS CLI

To update an Amazon Cognito user pools identity source

You can update an identity source by using the [UpdateIdentitySource](#) operation. The following example updates the specified identity source to use a different Amazon Cognito user pool.

The following `config.txt` file contains the details of the Amazon Cognito user pool for use by the --configuration parameter in the `create-identity-source` command.

```
{
    "cognitoUserPoolConfiguration": {
        "userPoolArn": "arn:aws:cognito-idp:us-west-2:123456789012:userpool/us-
west-2_1a2b3c4d5",
        "clientIds":["a1b2c3d4e5f6g7h8i9j0kalbmc"]
    }
}
```

Command:

```
$ aws verifiedpermissions update-identity-source \
    --update-configuration file://config.txt \
    --policy-store-id 123456789012
{
    "createdDate": "2023-05-19T20:30:28.214829+00:00",
    "identitySourceId": "ISEXAMPLEabcdefg111111",
    "lastUpdatedDate": "2023-05-19T20:30:28.214829+00:00",
    "policyStoreId": "PSEXAMPLEabcdefg111111"
}
```

If you change the principal type for the identity source, you must update your schema to correctly reflect the updated principal type.

Mapping Amazon Cognito tokens to Verified Permissions schema

Amazon Cognito identity tokens have four types of attributes that affect the naming of attributes in your Verified Permissions schema. To use Amazon Cognito as an identity source in your Verified Permissions policy store and write policies using attributes from the Amazon Cognito token that will be verified using the isAuthorizedWithToken API action, you must add Amazon Cognito attributes to your schema.

- Amazon Cognito-specific claims that have the `cognito` namespace prefix, such as `cognito:username`.

> **Note**
>
> The `cognito:groups` Amazon Cognito-specific claim is not currently mapped in Verified Permissions schemas. Amazon Cognito groups can use identifiers that are not guaranteed to be unique and that can be reused. Including them in an authorization decision could introduce security risks to your application.

- Custom claims that have the `custom` namespace prefix, such as `custom:employmentStoreCode`.
- Standard claims that do not have a namespace prefix, such as `email`.
- Transient claims that are added during token customization, such as `tenant`, `department`, and `clearance`.

For more information about using Amazon Cognito access and identity tokens for authenticated users in Verified Permissions, see Authorization with Amazon Verified Permissions in the *Amazon Cognito Developer Guide*.

Mapping Amazon Cognito ID tokens

The following example identity token has each of the four types of attributes. It includes the Amazon Cognito-specific claim `cognito:username`, the custom claim `custom:employmentStoreCode`, the standard claim `email`, and the transient claim `tenant`.

```
{
    "sub": "91eb4550-XXX",
    "cognito:groups": [
        "Store-Owner-Role",
        "Customer"
    ],
    "email_verified": true,
    "clearance": "confidential",
    "iss": "https://cognito-idp.us-east-2.amazonaws.com/us-east-2_wBEbEZKaX",
    "cognito:username": "alice",
    "custom:employmentStoreCode": "petstore-dallas",
    "origin_jti": "5b9f50a3-05da-454a-8b99-b79c2349de77",
    "aud": "52n97d5afhfiuXXX",
    "event_id": "0ed5ad5c-7182-4ecf-XXX",
    "token_use": "id",
    "auth_time": 1687885407,
    "department": "engineering",
    "exp": 1687889006,
    "iat": 1687885407,
    "tenant": "x11app-tenant-1",
    "jti": "a1b2c3d4-e5f6-a1b2-c3d4-TOKEN1111111",
    "email": "alice@example.com"
}
```

When you create an identity store and associate it with your Amazon Cognito user pool, you specify the type of principal that Cedar generates when a token is passed to an `IsAuthorizedWithToken` evaluation request. Your policies can then test attributes of that principal as part of evaluating that request. You must define that principal type and the attributes that you want to be able to reference from you Cedar policies.

The following example shows how to reflect the attributes from the example identity token in your Verified Permissions schema. For more information about editing your schema, see Editing schemas in JSON mode. If your identity source configuration specifies the principal type `CognitoUser`, then you can include something similar to the following example to make those attributes available to Cedar.

```
"CognitoUser": {
   "shape": {
      "type": "Record",
      "attributes": {
         "cognito": {
            "type": "Record",
            "required": true,
            "attributes": {
               "username": {
                  "type": "String",
                  "required": true
               }
            }
         },
         "custom": {
            "type": "Record",
            "required": true,
            "attributes": {
               "employmentStoreCode": {
                  "type": "String",
                  "required": true
               }
            }
         },
         "email": {
            "type": "String"
         },
         "tenant": {
            "type": "String",
            "required": true
         }
      }
   }
}
```

After updating your schema to reflect the Amazon Cognito attributes, you can create policies that reference the attributes.

```
permit(principal, action, resource)
when {
    principal.cognito.username == "alice" &&
    principal.custom.employmentStoreCode == "petstore-dallas" &&
    principal.tenant == "x11app-tenant-1" &&
```

```
    principal has email && principal.email == "alice@example.com"
};
```

Mapping Amazon Cognito access tokens

Amazon Cognito access tokens have claims that can be used for authorization:

- `client_id` represents the application client ID. This value can be used in machine-to-machine authorization scenarios to represent identity of the machine.
- `scope` is the OAuth 2.0 standard scope that represents the authorized scopes for the bearer of the token.

An Amazon Cognito access token is mapped to a context object when passed to Verified Permissions. Attributes of the access token can be referenced using `context.token.`*`attribute_name`*. The following example access token includes both the `client_id` and `scope` claims.

```
{
    "sub": "91eb4550-9091-708c-a7a6-9758ef8b6b1e",
    "cognito:groups": [
        "Store-Owner-Role",
        "Customer"
    ],
    "iss": "https://cognito-idp.us-east-2.amazonaws.com/us-east-2_wBEbEZKaX",
    "client_id": "52n97d5afhfiu1c4di1k5m8f60",
    "origin_jti": "a1b2c3d4-e5f6-a1b2-c3d4-TOKEN1111111",
    "event_id": "bda909cb-3e29-4bb8-83e3-ce6808f49011",
    "token_use": "access",
    "scope": "aws.cognito.signin.user.admin",
    "auth_time": 1688092966,
    "exp": 1688096566,
    "iat": 1688092966,
    "jti": "a1b2c3d4-e5f6-a1b2-c3d4-TOKEN2222222",
    "username": "alice"
}
```

The following example shows how to reflect the attributes from the example access token in your Verified Permissions schema. For more information about editing your schema, see Editing schemas in JSON mode.

```
{
   "MyApplication": {
      "actions": {
         "Read": {
            "appliesTo": {
               "context": {
                  "type": "ReusedContext"
               },
               "resourceTypes": [
                  "Application"
               ],
               "principalTypes": [
                  "User"
               ]
            }
         }
      },
      ...
      ...
      "commonTypes": {
         "ReusedContext": {
            "attributes": {
               "token": {
                  "type": "Record",
                  "attributes": {
                     "scope": {
                        "type": "Set",
                        "element": {
                           "type": "String"
                        }
                     },
                     "client_id": {
                        "type": "String"
                     }
                  }
               }
            },
            "type": "Record"
         }
      }
   }
}
```

After updating your schema to reflect the Amazon Cognito attributes, you can create policies that reference the attributes.

```
permit(principal, action in [MyApplication::Action::"Read",
 MyApplication::Action::"GetStoreInventory"], resource)
when {
    context.token.client_id == "52n97d5afhfiu1c4di1k5m8f60" &&
    context.token.scope.contains("aws.cognito.signin.user.admin")
};
```

Designing an authorization model for your application

As you prepare to use the Amazon Verified Permissions service within a software application, it can be challenging to leap immediately into writing policy statements as a first step. This would be similar to beginning development of other portions of an application by writing SQL statements or API specifications before fully deciding what the application should do. Instead, you should begin with a user experience, gathering a clear understanding of what end-users should see when managing permissions in the application UI. Then, work backwards from that experience to arrive at an implementation approach.

As you do this work, you'll find yourself asking questions such as:

- What are my resources? Do they have relationships to each other? For example, do files reside within a folder?
- What actions can principals perform on each resource?
- How do principals acquire those permissions?
- Do you want your end-users to choose from predefined permissions such as "Admin", "Operator", or "ReadOnly", or should they create ad-hoc policy statements? Or both?
- Should permissions inherit across resources, such as files inheriting permissions from a parent folder?
- What types of queries are necessary to render the user experience? For example, do you need to list all of the resources that a principal can access to render that user's home page?
- Can users accidentally lock themselves out of their own resources? Does that need to be avoided?

The end result of this exercise is referred to as an **authorization model**; it defines the principals, resources, actions, and how they interrelate to each other. Producing this model doesn't require unique knowledge of Cedar or the Verified Permissions service. Instead, it is first and foremost a user experience design exercise, much like any other, and can manifest in artifacts such as interface mockups, logical diagrams, and an overall description of how permissions influence what users see in the product. Cedar is designed to be flexible enough to meet customers at a model, rather than forcing the model to bend unnaturally to comply with a Cedar's implementation. As a result, gaining a crisp understanding of the desired user experience is the best way to arrive at an optimal model.

This section provides general guidance on how to approach the design exercise, things to watch out for, and a collection of best practices for using Verified Permissions successfully.

In addition to the guidelines presented here, remember to consider the best practices in the Cedar policy language reference guide.

Topics

There isn't a canonical “correct” model

When you design an authorization model, there is no single, uniquely correct answer. Different applications can effectively use different authorization models for similar concepts, and this is OK. For example, consider the representation of a computer's file system. When you create a file in a Unix-like operating system, it doesn't automatically inherit permissions from the parent folder. In contrast, in many other operating systems and most online file-sharing services, files do inherit permissions from its parent folder. Both choices are valid depending upon the circumstances the application is optimizing for.

The correctness of an authorization solution isn’t absolute, but should be viewed in terms of how it delivers the experience that your customers want, and whether it protects their resources in the way they expect. If your authorization model delivers on this, then it is successful.

This is why beginning your design with the desired user experience is the most helpful prerequisite to the creation of an effective authorization model.

Focus on your resources beyond API operations

In most consumer-facing applications, permissions are modeled around the resources supported by the application. For example, a file-sharing application might represent permissions as actions that can be performed on a file or a folder. This is a good, simple model that abstracts away the underlying implementation and the backend API operations.

In contrast, other types of applications, particularly web services, frequently design permissions around the API operations themselves. For example, if a web service provides an API named `createThing()`, the authorization model might define a corresponding permission, or an `action` in Cedar named `createThing`. This works in many situations and makes it easy to understand the permissions. To invoke the `createThing` operation, you need the `createThing` action permission. Seems simple, right?

You'll find that the getting started process in the Verified Permissions console includes the option to build your resources and actions directly from an API. This is a useful baseline: a direct mapping between your policy store and the API that it authorizes for.

However, this API-focused approach can be less than optimal, because APIs are merely a proxy for what your customers are truly trying to protect: the underlying data and resources. If multiple APIs control access to the same resources, it can be difficult for administrators to reason about the paths to those resources and manage access accordingly.

For example, consider a user directory that contains the members of an organization. Users can be organized into groups, and one of the security goals is to prohibit discovery of group memberships by unauthorized parties. The service managing this user directory provides two API operations:

- `listMembersOfGroup`
- `listGroupMembershipsForUser`

Customers can use either of these operations to discover group membership. Therefore, the permissions administrator must remember to coordinate access to *both* operations. This is complicated further if you later choose to add a new API operation to address additional use cases, such as the following.

- `isUserInGroups` *(a new API to quickly test if a user belongs in one or more groups)*

From a security perspective, this API opens a third path for discovering group memberships, disrupting the carefully crafted permissions of the administrator.

We recommend that you ignore the API semantics and instead focus on the underlying data and resources and their association operations. Applying this approach to the group membership example would lead to an abstract permission, such as `viewGroupMembership`, which each of the three API operations must consult.

API Name	Permissions
`listMembersOfGroup`	requires `viewGroupMembership` permission on the group
`listGroupMembershipsForUser`	requires `viewGroupMembership` permission on the user
`isUserInGroups`	requires `viewGroupMembership` permission on the user

By defining this one permission, the administrator successfully controls access to discovering group memberships, now and forever. As a tradeoff, each API operation must now document the possibly several permissions that it requires, and the administrator must consult this documentation when crafting permissions. This can be a valid tradeoff when necessary to meet your security requirements.

Compound authorization is normal

Compound authorization occurs when a single user activity, such as clicking a button in your application's interface, requires multiple individual authorization queries to determine whether that activity is permitted. For example, moving a file to a new directory in a file system might require three different permissions: the ability to delete a file from the source directory, the ability to add a file to the destination directory, and possibly the ability to touch the file itself (depending on the application).

If you're new to designing an authorization model, you might think that every authorization decision must be resolvable in a single authorization query. But this can lead to overly complex models and convoluted policy statements. In practice, using compound authorizations can be useful in helping you to produce a simpler authorization model. One measure of a well-designed authorization model is that when you have sufficiently decomposed individual actions, your compound operations, such as moving a file, can be represented by an intuitive aggregation of primitives.

Another situation where compound authorization occurs is when multiple parties are involved in the process of granting a permission. Consider an organizational directory where users can be members of groups. A simple approach is to give the group owner permission to add anyone. However, what if you want your users to first consent to being added? This introduces a handshake agreement in which both the user and the group must consent to the membership. To accomplish this, you can introduce another permission that is bound to the user and specifies whether the user can be added to any group, or to a particular group. When a caller subsequently attempts to add members to a group, the application must enforce both sides of the permissions: that the caller has permission to add members to the specified group, and that the individual user being added has the permissions to be added. When `N`-way handshakes exist, it is common to observe `N` compound authorization queries to enforce each portion of the agreement.

If you find yourself with a design challenge where multiple resources are involved and it is unclear how to model the permissions, it can be a sign that you have a compound authorization scenario. In this case, a solution might be found by decomposing the operation into multiple, individual authorization checks.

Multi-tenancy considerations

You might want to develop applications for use by multiple customers - businesses that consume your application, or *tenants* - and integrate them with Amazon Verified Permissions. Before you develop your authorization model, develop a multi-tenant strategy. You can manage the policies of your customers in *one shared policy store*, or assign each a *per-tenant policy store*.

1. **One shared policy store**

   All tenants share a single policy store. The application sends all authorization requests to the shared policy store.

2. **Per-tenant policy store**

   Each tenant has a dedicated policy store. The application will query different policy stores for an authorization decision, depending on the tenant that makes the request.

Neither strategy creates a relatively-higher volume of authorization requests that might have an impact on your AWS bill. So how, then, should you design your approach? The following are common conditions that might contribute to your Verified Permissions multi-tenancy authorization strategy.

Tenant policies isolation

Isolation of the policies of each tenant from the others is important to protect tenant data. When each tenant has their own policy store, they each have their own isolated set of policies.

Authorization flow

You can identify a tenant making an authorization request with a policy store ID in the request, with per-tenant policy stores. With a shared policy store, all requests use the same policy store ID.

Templates and schema management

Your policy templates and a policy store schema add a level of design and maintenance overhead in each policy store.

Global policies management

You might want to apply some *global* policies to every tenant. The level of overhead for management of global policies varies between shared and per-tenant policy store models.

Tenant off-boarding

Some tenants will contribute elements to your schema and policies that are specific to their case. When a tenant is no longer active with your organization and you want to remove their data, the level of effort varies with their level of isolation from other tenants.

Service resource quotas

Verified Permissions has resource and request-rate quotas that might influence your multi-tenancy decision. For more information about quotas, see Quotas for resources.

Comparing shared policy stores and per-tenant policy stores

Each consideration requires its own level of time and resource commitment in shared and per-tenant policy store models.

Consideration	Effort level in a shared policy store	Effort level in per-tenant policy stores
Tenant policies isolation	*Medium.* Must include tenant identifiers in policies and authorization requests.	*Low.* Isolation is default behavior. Tenant-specific

		policies are inaccessible to other tenants.
Authorization flow	*Low.* All queries target one policy store.	*Medium.* Must maintain mappings between each tenant and their policy store ID.
Templates and schema management	*Low.* Must make one schema work for all tenants.	*High.* Schemas and templates might be less complex individually, but changes require more coordination and complexity.
Global policies management	*Low.* All policies are global and can be centrally updated.	*High.* You must add global policies to each policy store in onboarding. Replicate global policy updates between many policy stores.
Tenant off-boarding	*Medium.* Must identify and delete only tenant-specific policies.	*Low.* Delete the policy store.
Service resource quotas	*High.* Tenants share resource quotas that affect policy stores like schema size, policy size per resource, and identity sources per policy store.	*Low.* Each tenant has dedicated resource quotas.

How to choose

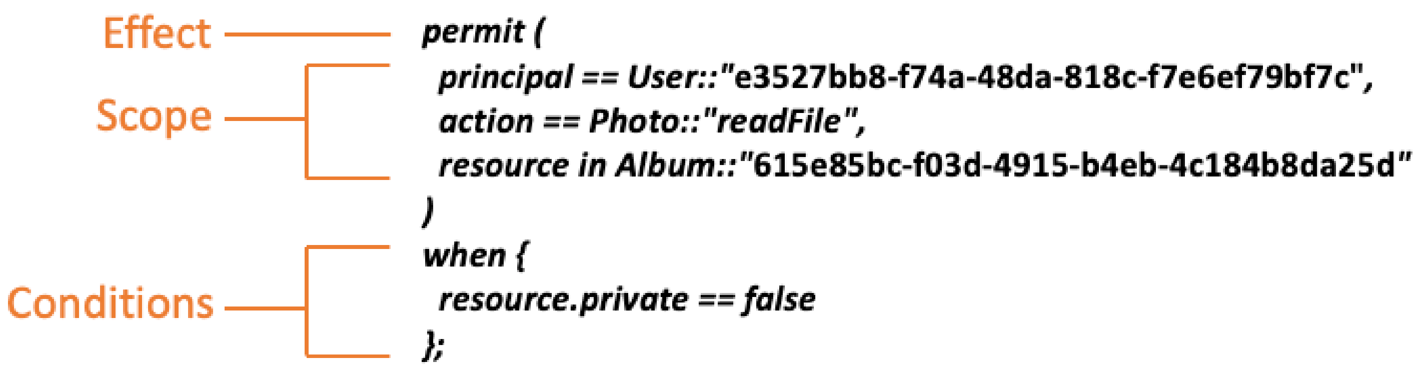
Each multi-tenant application is different. Carefully compare the two approaches and their considerations before making an architectural decision.

If your application doesn't require tenant-specific policies and uses a single identity source, one shared policy store for all tenants is likely to be the most effective solution. This results in a simpler authorization flow and global policy management. Off-boarding a tenant using one shared policy store requires less effort because the application does not need to delete tenant-specific policies.

But if your application requires many tenant-specific policies, or uses multiple identity sources, per-tenant policy stores are likely to be most effective. You can control access to tenant policies with IAM policies that grant per-tenant permissions to each policy store. Off-boarding a tenant involves deleting their policy store; in a shared-policy-store environment, you must find and delete tenant-specific policies.

When possible, populate the policy scope

The policy scope is the portion of a Cedar policy statement after the `permit` or `forbid` keywords and between the opening parenthesis.



We recommend that you populate the values for `principal` and `resource` whenever possible. This lets Verified Permissions index the policies for more efficient retrieval and therefore improves performance. If you need to grant the same permissions to many different principals or resources, we recommend that you use a policy template and attach it to each principal and resource pair.

Avoid creating one large policy that contains lists of principals and resources in a `when` clause. Doing so will likely cause you to run into scalability limits or operational challenges. For example, in order to add or remove a single user from a large list within a policy, it is necessary to read the whole policy, edit the list, write the new policy in full, and handle concurrency errors if one administrator overwrites another's changes. In contrast, by using many fine-grained permissions, adding or removing a user is as simple as adding or removing the single policy that applies to them.

Every resource lives in a container

When you design an authorization model, every action must be associated with a particular resource. With an action such as `viewFile`, the resource that you can apply it to is intuitive: an individual file, or perhaps a collection of files within a folder. However, an operation such as `createFile` is less intuitive. When modeling the capability to create a file, what resource does it apply to? It can't be the file itself, because the file doesn’t exist yet.

This is an example of the generalized problem of resource creation. Resource creation is a bootstrapping problem. There must be a way for something to have permission to create resources even when no resources exist yet. The solution is to recognize that every resource must exist within some container, and it is the container itself that acts as the anchor point for permissions. For example, if a folder already exists in the system, the ability to create a file can be modeled as a permission on that folder, since that is the location where permissions are necessary to instantiate the new resource.

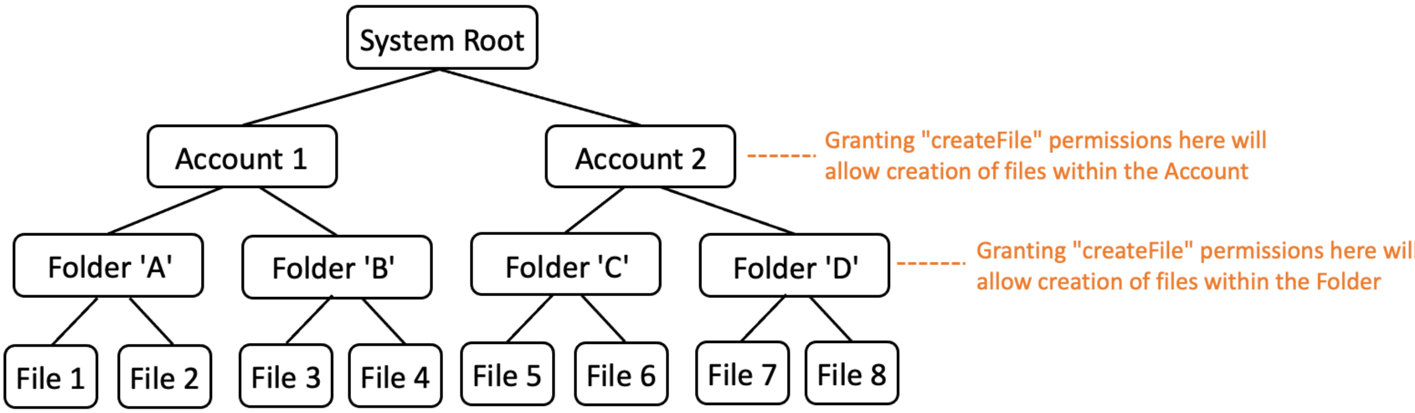
```
permit (
    principal == User::"6688f676-1aa9-456a-acf4-228340b54e9d",
    action == Action::"createFile",
    resource == Folder::"c863f89b-461f-4fc2-b638-e5fa5f79a48b"
);
```

But what if no folder exists? Perhaps this is a brand new customer account in an application where no resources exist yet. In this situation, there is still a context that can be intuitively understood by asking: where can the customer create new files? You don't want them to be able to create files inside any random customer account. Rather, there is an implied context: the customer’s own account boundary. Therefore, the account itself represents the container for resource creation, and this can be explicitly modeled in a policy similar to the following example.

```
// Grants permission to create files within an account,
// or within any sub-folder inside the account.
permit (
    principal == User::"6688f676-1aa9-456a-acf4-228340b54e9d",
    action == Action::"createFile",
    resource in Account::"c863f89b-461f-4fc2-b638-e5fa5f79a48b"
);
```

Yet, what if no accounts exist either? You might choose to design the customer sign-up workflow so that the it creates new accounts in the system. If so, you’ll need a container to hold the outermost boundary in which the process can create the accounts. This root level container represents the system as a whole and might be named something like “system root”. However, the decision for whether this is needed, and what to name it is up to you, the application owner.

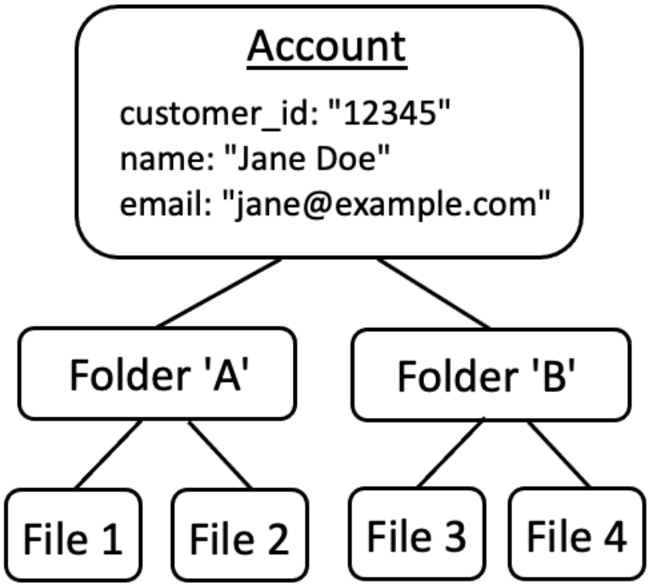
For this sample application, the resulting container hierarchy would therefore appears as follows:



This is one sample hierarchy. Others are valid as well. The thing to remember is that resource creation always happens within the context of a resource container. These containers can be implicit, such as an account boundary, and it can be easy to overlook them. When designing your authorization model, be sure to note these implicit assumptions so they can be formally documented and represented in the authorization model.

Separate the principals from the resource containers

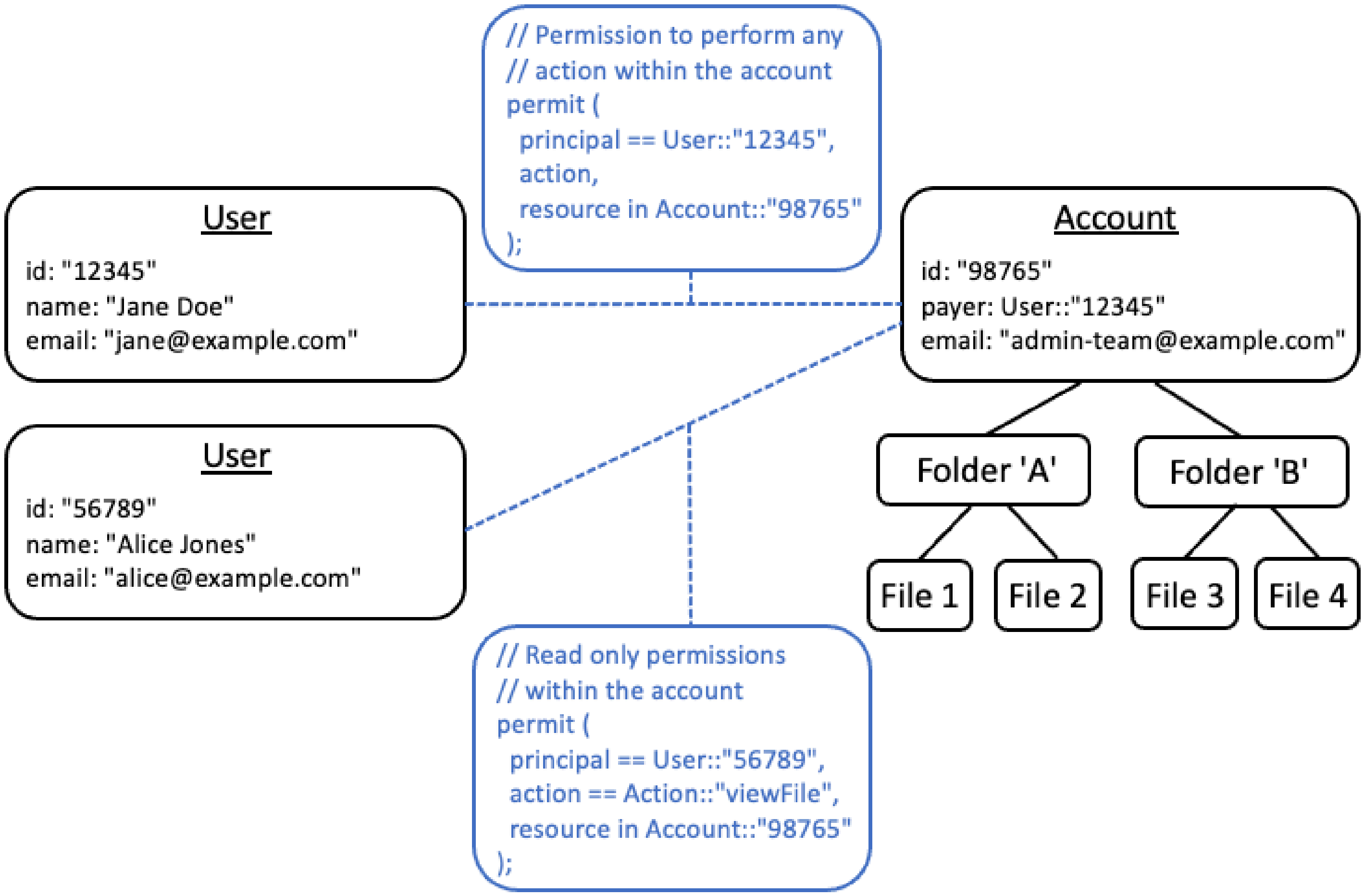
When you are designing a resource hierarchy, one of the common inclinations, especially for consumer-facing applications, is to use the customer's user identity as the container for resources within a customer account.



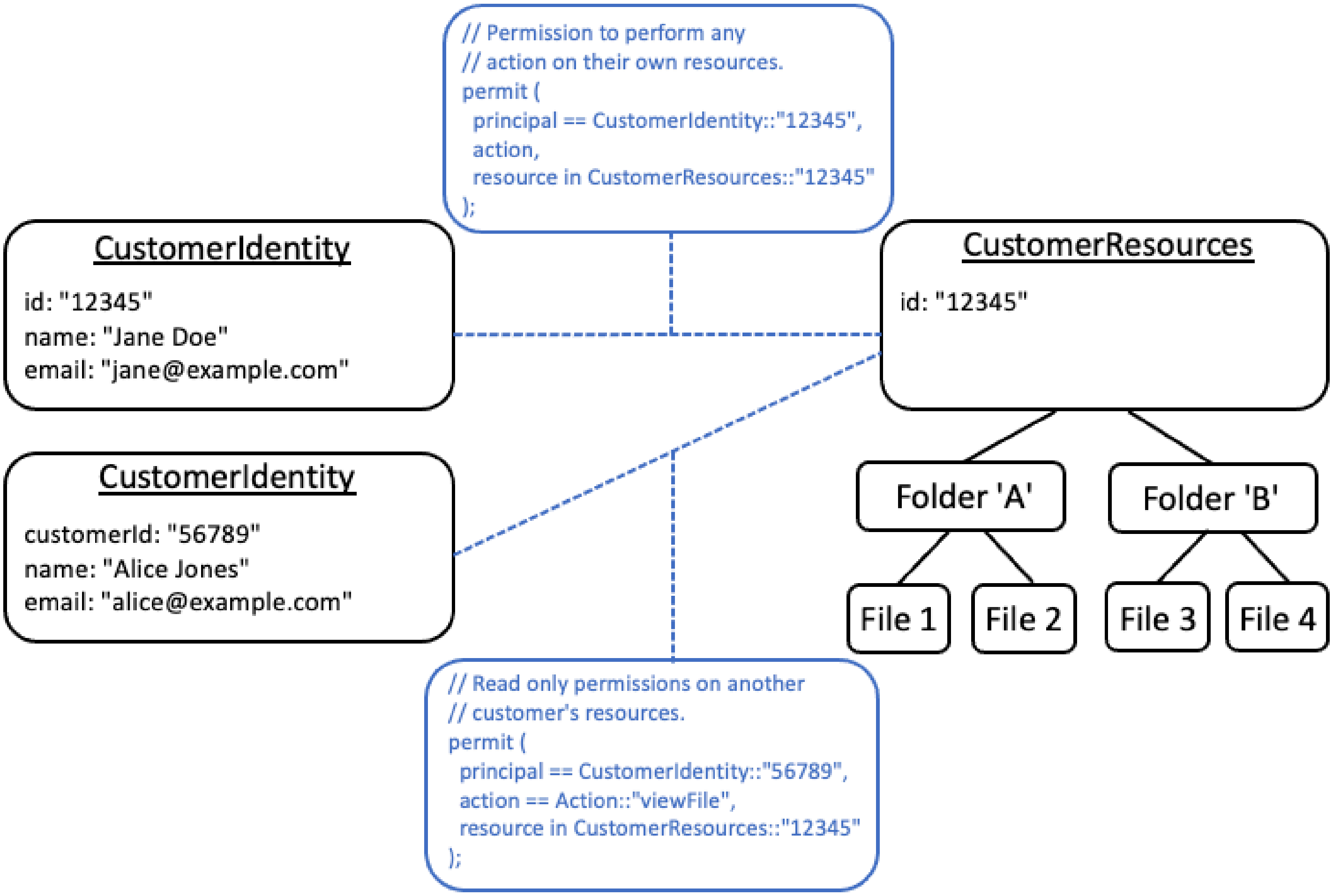
We recommend that you treat this strategy as an anti-pattern. This is because there is a natural tendency in richer applications to delegate access to additional users. For example, you might choose to introduce "family" accounts, where other users can share account resources. Similarly, enterprise customers sometimes want to designate multiple members of the workforce as operators for portions of the account. You might also need to transfer ownership of an account to a different user, or merge the resources of multiple accounts together.

When a user identity is used as the resource container for an account, the previous scenarios become more difficult to achieve. More alarming, if others are granted access to the account container in this approach, they might inadvertently be granted access to modify the user identity itself, such as changing Jane's email or login credentials.

Therefore, when possible to do so, a more resilient approach is to separate the principals from the resource containers, and model the connection between them by using concepts such as "admin permissions" or "ownership".



Where you have an existing application that is unable to pursue this decoupled model, we recommend that you consider mimicking it as much as possible when designing an authorization model. For example, an application that possesses only a single concept named `Customer` that encapsulates the user identity, login credentials, and resources that they own, could map this to an authorization model that contains one logical entity for `Customer Identity` (containing name, email, etc) and a separate logical entity for `Customer Resources` or `Customer Account`, acting as the parent node for all the resources they own. Both entities can share the same `Id`, but with a different `Type`.



Don't embed permissions inside attributes

Attributes are best used as an *input* to the authorization decision. Don't use attributes to represent the permissions themselves, such as by declaring an attribute named “permittedFolders” on a User:

```
// ANTI-PATTERN: comingling permissions into user attributes
{
    "id": "df82e4ad-949e-44cb-8acf-2d1acda71798",
    "name": "alice",
    "email": "alice@example.com",
    "permittedFolders": [
        "Folder::\"c943927f-d803-4f40-9a53-7740272cb969\"",
        "Folder::\"661817a9-d478-4096-943d-4ef1e082d19a\"",
        "Folder::\"b8ee140c-fa09-46c3-992e-099438930894\""
    ]
}
```

And, subsequently using the attribute within a policy:

```
// ANTI-PATTERN
permit (
    principal,
    action == Action::"readFile",
    resource
)
when {
    resource in principal.permittedFolders
};
```

This approach transforms what would otherwise be a simple authorization model, where a specific principal has access to a specific folder, into an attributes-based access control (ABAC) model with the accompanying tradeoffs. One such tradeoff is that it becomes more difficult to quickly determine who has permission to a resource. In the preceding example, to determine who has access to a particular folder, it is necessary to iterate over every user to check if that folder is listed in their attributes, and doing so with the special awareness that there is a policy that grants access when they do.

Another risk with this approach is the scaling factors when permissions are packed together inside a single `User` record. If the user has access to many things, the cumulative size of their `User` record will grow and perhaps approach the maximum limit of whatever system is storing the data.

Instead, we recommend that you represent this scenario using multiple individual policies, perhaps using policy templates to minimize repetition.

```
//BETTER PATTERN
permit (
    principal == User::"df82e4ad-949e-44cb-8acf-2d1acda71798",
    action == Action::"readFile",
    resource in Folder::"c943927f-d803-4f40-9a53-7740272cb969"
);

permit (
    principal == User::"df82e4ad-949e-44cb-8acf-2d1acda71798",
    action == Action::"readFile",
    resource in Folder::"661817a9-d478-4096-943d-4ef1e082d19a"
);

permit (
```

```
    principal == User::"df82e4ad-949e-44cb-8acf-2d1acda71798",
    action == Action::"readFile",
    resource in Folder::"b8ee140c-fa09-46c3-992e-099438930894"
);
```

Verified Permissions can efficiently handle many individual, fine-grained policies during authorization evaluation. Modeling things in this way is more manageable and auditable over time.

Prefer fine-grained permissions in the model and aggregate permissions in the user interface

One strategy that designers often regret later is designing an authorization model with very broad actions, such as `Read` and `Write`, and realizing later that finer-grained actions are necessary. The need for finer granularity can be driven by customer feedback for more granular access controls, or by compliance and security auditors who encourage least-privilege permissions.

If fine-grained permissions are not defined upfront, it can require a complicated conversion to modify the application code and policy statements to user finer grained permissions. For example, application code that previously authorized against a course-grained action will need to be modified to use the fine-grained actions. In addition, policies will need to be updated to reflect the migration:

```
permit (
    principal == User::"6688f676-1aa9-456a-acf4-228340b54e9d",
    // action  ==  Action::"read",              -- coarse-grained permission --
 commented out
    action in [                       //        -- finer grained permissions
        Action::"listFolderContents",
        Action::"viewFile"
    ],
    resource in Account::"c863f89b-461f-4fc2-b638-e5fa5f79a48b"
);
```

To avoid this costly migration, it's better to define fine-grained permissions upfront. However, this can result in a tradeoff if your end-users are subsequently forced to understand a larger number of fine-grained permissions, especially if most customers would be satisfied with course-grained controls such as `Read` and `Write`. To attain the best of both worlds, you can group fine-grained permissions into predefined collections such as `Read` and `Write` using mechanisms like policy templates or action groups. By using this approach, customers see only the course-grained permissions. But behind the scenes, you've future-proofed your application by modeling the course-grained permissions as a collection of fine-grained actions. When either customers or auditors ask for it, the fine-grained permissions can be exposed.

Consider other reasons to query authorization

We usually associate authorization checks with user requests. The check is a way to determine whether the user has permission to perform that request. However, you can also use authorization data to influence the design of the application's interface. For example, you might want to display a home screen that shows a list of only those resources that the end-user can access. When viewing the details of a resource, you might want the interface to show only those operations that the user can perform on that resource.

These situations can introduce tradeoffs into the authorization model. For example, heavy reliance on attributed-based access control (ABAC) policies can make it more difficult to quickly answer the question "who has access to what?" This is because answering that question requires examining each rule against every principal and resource to determine if there is a match. As a result, a product that needs to optimize for listing only those resources accessible by the user might choose to use a role-based access control (RBAC) model. By using RBAC, it can be easier to iterate over all the policies attached to a user to determine resource access.

Test bench

The Verified Permissions test bench allows you to test and troubleshoot Verified Permissions policies by running authorization requests against them. The test bench uses the parameters that you specify to determine whether the Cedar policies in your policy store would authorize the request. You can toggle between **Visual mode** and **JSON mode** while testing authorization requests. For more information about how Cedar policies are structured and evaluated, see Basic policy construction in Cedar in the Cedar policy language Reference Guide.

> **Note**
>
> When you make an authorization request using Verified Permissions, you can provide the list of principals and resources as part of the request in the **Additional entities** section. However, you can't include the details about the actions. They must be specified in the schema or inferred from the request. You can't put an action in the **Additional entities** section.

For a visual overview and demonstration of the test bench, see this video.

Visual mode

> **Note**
>
> You must have a schema defined in your policy store to use the **Visual mode** of the test bench.

To test policies in Visual mode

1. Open the Verified Permissions console at https://console.aws.amazon.com/verifiedpermissions/. Choose your policy store.
2. In the navigation pane on the left, choose **Test bench**.
3. Choose **Visual mode**.
4. In the **Principal** section, choose the **Principal taking action** from the principal types in your schema. Type an identifier for the principal in the text box.

5. (Optional) Choose **Add a parent** to add parent entities for the specified principal. To remove a parent that has been added to the principal, choose **Remove** next to the name of the parent.
6. Specify the **Attribute value** for each attribute of the specified principal. The test bench uses the specified attribute values in the simulated authorization request.
7. In the **Resource** section, choose the **Resource that principal is acting on**. Type an identifier for the resource in the text box.
8. (Optional) Choose **Add a parent** to add parent entities for the specified resource. To remove a parent that has been added to the resource, choose **Remove** next to the name of the parent.
9. Specify the **Attribute value** for each attribute of the specified resource. The test bench uses the specified attribute values in the simulated authorization request.
10. In the **Action** section, choose the **Action that principal is taking** from the list of valid actions for the specified principal and resource.
11. Specify the **Attribute value** for each attribute of the specified action. The test bench uses the specified attribute values in the simulated authorization request.
12. (Optional) In the **Additional entities** section, choose **Add entity** to add entities to be evaluated for the authorization decision.
13. Choose the **Entity Identifier** from the dropdown list and type the entity identifier.
14. (Optional) Choose **Add a parent** to add parent entities for the specified entity. To remove a parent that has been added to the entity, choose **Remove** next to the name of the parent.
15. Specify the **Attribute value** for each attribute of the specified entity. The test bench uses the specified attribute values in the simulated authorization request.
16. Choose **Confirm** to add the entity to the test bench.
17. Choose **Run authorization request** to simulate the authorization request for the Cedar policies in your policy store. The test bench displays the decision to allow or deny the request along with information about the policies satisfied or the errors encountered during evaluation.

JSON mode

To test policies in JSON mode

1. Open the Verified Permissions console at https://console.aws.amazon.com/verifiedpermissions/. Choose your policy store.
2. In the navigation pane on the left, choose **Test bench**.
3. Choose **JSON mode**.
4. In the **Request details** section, if you have a schema defined, choose the **Principal taking action** from the principal types in your schema. Type an identifier for the principal in the text box.

   If you do not have a schema defined, type the principal in the **Principal taking action** text box.
5. If you have a schema defined, choose the **Resource** from the resource types in your schema. Type an identifier for the resource in the text box.

   If you do not have a schema defined, type the resource in the **Resource** text box.
6. If you have a schema defined, choose the **Action** from the list of valid actions for the specified principal and resource.

   If you do not have a schema defined, type the action in the **Action** text box.
7. Enter the context of the request to simulate in the **Context** field. The request context is additional information that can be used for authorization decisions.
8. In the **Entities** field, enter the hierarchy of the entities and their attributes to be evaluated for the authorization decision.
9. Choose **Run authorization request** to simulate the authorization request for the Cedar policies in your policy store. The test bench displays the decision to allow or deny the request along with information about the policies satisfied or the errors encountered during evaluation.

Implementing authorization in Amazon Verified Permissions

After you build your policy store, policies, templates, schema, and authorization model, you're ready to start authorizing requests against Amazon Verified Permissions. To implement Verified Permissions authorization, you must combine configuration of policies in AWS with integration in an application. To integrate Verified Permissions with your application, add an AWS SDK and implement the methods that invoke the Verified Permissions API and generate authorization decisions against your policy store.

Authorization with Verified Permissions is useful for *UX permissions* and *API permissions* in your applications.

UX permissions

Control user access to your application UX. You can permit a user to view only the exact forms, buttons, graphics and other resources that they need to access. For example, when a user signs in, you might want to determine whether a "Transfer funds" button is visible in their account. You can also control actions that a user can take. For example, in same banking app you might want to determine whether your user is permitted to change the category of a transaction.

API permissions

Control user access to data. Applications are often often part of a distributed system and bring in information from external APIs. In the example of the banking app where Verified Permissions has permitted the display of a "Transfer funds" button, a more complex authorization decision must be made when your user initiates a transfer. Verified Permissions can authorize the API request that lists the destination accounts that are eligible transfer targets, and then the request to push the transfer to the other account.

The examples that illustrate this content come from a sample policy store. To follow along, create the **DigitalPetStore** sample policy store in your testing environment.

API operations for authorization

The Verified Permissions API has the following authorization operations.

IsAuthorized

The `IsAuthorized` API operation is the entry point to authorization requests with Verified Permissions. You must submit principal, action, resource, context, and entities elements. Verified Permissions validates the entities in your request against your policy store schema. Verified Permissions then evaluates your request against all policies in the requested policy store that apply to the entities in the request.

IsAuthorizedWithToken

The `IsAuthorizedWithToken` operation generates an authorization request from user data in Amazon Cognito JSON web tokens (JWTs). Verified Permissions works directly with Amazon Cognito as an identity source in your policy store. Verified Permissions populates all attributes to the principal in your request from the claims in users' ID or access tokens. You can authorize actions and resources from user attributes or group membership in an Amazon Cognito user pool.

You can't include information about group or user principal types in an `IsAuthorizedWithToken` request. You must populate all principal data to the JWT that you provide.

BatchIsAuthorized

The `BatchIsAuthorized` operation processes multiple authorization decisions for a single principal or resource in a single API request. This operation groups requests into a single batch operation that minimizes quota usage and returns authorization decisions for each of up to 30 complex nested actions. With batch authorization for a single resource, you can filter the actions that a user can take on a resource. With batch authorization for a single principal, you can filter for the resources that a user can take action on.

BatchIsAuthorizedWithToken

The `BatchIsAuthorizedWithToken` operation processes multiple authorization decisions for a single principal in one API request. The principal is provided by your policy store identity source in an ID or access token. This operation groups requests into a single batch operation that minimizes quota usage and returns authorization decisions for each of up to 30 requests for actions and resources. In your policies, you can authorize their access from their attributes or their group membership in an Amazon Cognito user pool.

Like with `IsAuthorizedWithToken`, you can't include information about group or user principal types in a `BatchIsAuthorizedWithToken` request. You must populate all principal data to the JWT that you provide.

Testing your authorization model

To understand the effect of Verified Permissions authorization decision when you deploy your application, you can evaluate your policies as you develop them with the Test bench and with HTTPS REST API requests to Verified Permissions. The test bench is a tool in the AWS Management Console to evaluate authorization requests and responses in your policy store.

The Verified Permissions REST API is the next step in your development as you move from a conceptual understanding to application design. The Verified Permissions API accepts authorization requests with IsAuthorized, IsAuthorizedWithToken, and BatchIsAuthorized as signed AWS API requests to Regional service endpoints. To test your authorization model, you can generate requests with any API client and verify that your policies are returning authorization decisions as expected.

For example, you can test `IsAuthorized` in a sample policy store with the following procedure.

Test bench

1. Open the Verified Permissions console at https://console.aws.amazon.com/verifiedpermissions/. Create a policy store from the **Sample policy store** with the name **DigitalPetStore**.
2. Select **Test bench** in your new policy store.
3. Populate your test bench request from IsAuthorized in the Verified Permissions API reference. The following details replicate the conditions in **Example 4** that references the **DigitalPetStore** sample.

   a. Set Alice as the principal. For **Principal taking action**, choose `DigitalPetStore::User` and enter `Alice`.

   b. Set Alice's role as customer. Choose **Add a parent**, choose `DigitalPetStore::Role`, and enter Customer.

   c. Set the resource as order "1234." For **Resource that the principal is acting on**, choose `DigitalPetStore::Order` and enter `1234`.

   d. The `DigitalPetStore::Order` resource requires an `owner` attribute. Set Alice as the owner of the order. Choose `DigitalPetStore::User` and enter `Alice`

   e. Alice requested to view the order. For **Action that principal is taking**, choose `DigitalPetStore::Action::"GetOrder"`.

4. Choose **Run authorization request**. In an unmodified policy store, this request results in an `ALLOW` decision. Note the **Satisfied policy** that returned the decision.
5. Choose **Policies** from the left navigation bar. Review the static policy with the description **Customer Role - Get Order**.
6. Observe that Verified Permissions allowed the request because the principal was in a customer role and was the owner of the resource.

REST API

1. Open the Verified Permissions console at https://console.aws.amazon.com/verifiedpermissions/. Create a policy store from the **Sample policy store** with the name **DigitalPetStore**.
2. Note the **Policy store ID** of your new policy store.
3. From IsAuthorized in the Verified Permissions API reference, copy the request body of **Example 4** that references the **DigitalPetStore** sample.
4. Open your API client and create a request to the Regional service endpoint for your policy store. Populate the headers as shown in the example.
5. Paste in the sample request body and change the value of `policyStoreId` to the policy store ID you noted earlier.
6. Submit the request and review the results. In a default **DigitalPetStore** policy store, this request returns an `ALLOW` decision.

You can make changes to policies, schema, and requests in your test environment to change the outcomes and produce more complex decisions.

1. Change the request in a way that changes the decision from Verified Permissions. For example, change Alice's role to `Employee` or change the `owner` attribute of order 1234 to `Bob`.
2. Change policies in ways that affect authorization decisions. For example, modify the policy with the description **Customer Role - Get Order** to remove the condition that the `User` must be the owner of the `Resource` and modify the request so that `Bob` wants to view the order.
3. Change the schema to allow policies to make a more complex decision. Update the request entities so that Alice can satisfy the new requirements. For example, edit the schema to allow `User` to be a member of `ActiveUsers` or `InactiveUsers`. Update the policy so that only active users can view their own orders. Update the request entities so that Alice is an active or inactive user.

Integrating with apps and AWS SDKs

To implement Amazon Verified Permissions in your application, you must define the policies and schema that you want your app to enforce. With your authorization model in place and tested, your next step is to start generating API requests from the point of enforcement. To do this, you must set up application logic to collect user data and populate it to authorization requests.

How an app authorizes requests with Verified Permissions

1. Gather information about the current user. Typically, a user's details are provided in the details of an authenticated session, like a JWT or web session cookie. This user data might originate from an Amazon Cognito identity source linked to your policy store or from another OpenID Connect (OIDC) provider.
2. Gather information about the resource that a user wants to access. Typically, your application will receive information about the resource when a user makes a selection that requires your app to load a new asset.
3. Determine the action that your user wants to take.
4. Generate an authorization request to Verified Permissions with the principal, action, resource, and entities for your user's attempted operation.Verified Permissions evaluates the request against the policies in your policy store and returns an authorization decision.
5. Your application reads the allow or deny response from Verified Permissions and enforces the decision on the user's request.

Verified Permissions API operations are built into AWS SDKs. To include Verified Permissions in an app, integrate the AWS SDK for your chosen language into the app package.

To learn more and download AWS SDKs, see Tools for Amazon Web Services.

The following are links to documentation for Verified Permissions resources in various AWS SDKs.

- AWS SDK for .NET
- AWS SDK for C++
- AWS SDK for Go
- AWS SDK for Java
- AWS SDK for JavaScript
- AWS SDK for PHP

- AWS SDK for Python (Boto)
- AWS SDK for Ruby

The following AWS SDK for JavaScript example for `IsAuthorized` originates from Simplify fine-grained authorization with Amazon Verified Permissions and Amazon Cognito.

```
const authResult = await avp.isAuthorized({
    principal: 'User::"alice"',
    action: 'Action::"view"',
    resource: 'Photo::"VacationPhoto94.jpg"',
    // whenever our policy references attributes of the entity,
    // isAuthorized needs an entity argument that provides
    // those attributes
    entities: {
       entityList: [
         {
            "identifier": {
                "entityType": "User",
                "entityId": "alice"
            },
            "attributes": {
                "location": {
                    "String": "USA"
                }
            }
         }
       ]
    }
});
```

More developer resources

- Amazon Verified Permissions workshop
- Amazon Verified Permissions - Resources
- Implement custom authorization policy provider for ASP.NET Core apps using Amazon Verified Permissions
- Build an entitlement service for business applications using Amazon Verified Permissions
- Simplify fine-grained authorization with Amazon Verified Permissions and Amazon Cognito

Adding context

Context is the information that's relevant to policy decisions, but not part of the identity of your principal, action, or resource. You might want to allow an action only from a set of source IP addresses, or only if your user has signed in with MFA. Your application has access to this contextual session data and must populate it to authorization requests. The context data in a Verified Permissions authorization request must be JSON-formatted in a `contextMap` element.

The examples that illustrate this content come from a [sample policy store](). To follow along, create the **DigitalPetStore** sample policy store in your testing environment.

The following context object declares one of each Cedar data type for an application based on the sample **DigitalPetStore** policy store.

```
"context": {
   "contextMap": {
      "MfaAuthorized": {
         "boolean": true
      },
      "AccountCodes": {
         "set": [
            {
               "long": 111122223333
            },
            {
               "long": 444455556666
            },
            {
               "long": 123456789012
            }
         ]
      },
      "UserAgent": {
         "string": "My UserAgent 1.12"
      },
      "RequestedOrderCount": {
         "long": 4
      },
      "NetworkInfo": {
         "record": {
            "IPAddress": {
```

```
                "string": "192.0.2.178"
            },
            "Country": {
                "string": "United States of America"
            },
            "SSL": {
                "boolean": true
            }
        }
    },
    "approvedBy": {
        "entityIdentifier": {
            "entityId": "Bob",
            "entityType": "DigitalPetStore::User"
        }
    }
  }
}
```

Data types in authorization context

Boolean

A binary `true` or `false` value. In the example, the boolean value of `true` for `MfaAuthenticated` indicates that the customer has performed multi-factor authentication before requesting to view their order.

Set

A collection of context elements. Set members can be all the same type, like in this example, or of different types, including a nested set. In the example, the customer is associated with 3 different accounts.

String

A sequence of letters, numbers, or symbols, enclosed in `"` characters. In the example, the `UserAgent` string represents the browser that the customer used to request to view their order.

Long

An integer. In the example, the `RequestedOrderCount` indicates that this request is part of a batch that resulted from the customer asking to view four of their past orders.

Record

A collection of attributes. You must declare these attributes in the request context. A policy store with a schema must include this entity and the attributes of the entity in the schema. In the example, the `NetworkInfo` record contains information about the user's originating IP, the geolocation of that IP as determined by the client, and encryption in transit.

EntityIdentifier

A reference to an entity and attributes declared in the `entities` element of the request. In the example, the user's order was approved by employee `Bob`.

To test this example context in the example **DigitalPetStore** app, you must update your request `entities`, your policy store schema, and the static policy with the description **Customer Role - Get Order**.

Modifying DigitalPetStore to accept authorization context

Initially, **DigitalPetStore** is not a very complex policy store. It doesn't include any preconfigured policies or context attributes to support the context that we have presented. To evaluate an example authorization request with this context information, make the following modifications to your policy store and your authorization request.

Schema

Apply the following updates to your policy store schema to support the new context attributes. Update `GetOrder` in `actions` as follows.

```
"GetOrder": {
   "memberOf": [],
   "appliesTo": {
      "resourceTypes": [
         "Order"
      ],
      "context": {
         "type": "Record",
         "attributes": {
            "UserAgent": {
               "required": true,
               "type": "String"
            },
```

```
            "approvedBy": {
               "name": "User",
               "required": true,
               "type": "Entity"
            },
            "AccountCodes": {
               "type": "Set",
               "required": true,
               "element": {
                  "type": "Long"
               }
            },
            "RequestedOrderCount": {
               "type": "Long",
               "required": true
            },
            "MfaAuthorized": {
               "type": "Boolean",
               "required": true
            }
         }
      },
      "principalTypes": [
         "User"
      ]
   }
}
```

To reference the `record` data type named `NetworkInfo` in your request context, create a [commonType](https://docs.cedarpolicy.com/schema/json-schema.html#schema-commonTypes) construct in your schema as follows. A `commonType` construct is a shared set of attributes that you can apply to different entities.

> **Note**
>
> The Verified Permissions visual schema editor currently doesn't support `commonType` constructs. When you add them to your schema, you can no longer view your schema in **Visual mode**.

```
"commonTypes": {
   "NetworkInfo": {
      "attributes": {
```

```
         "IPAddress": {
            "type": "String",
            "required": true
         },
         "SSL": {
            "required": true,
            "type": "Boolean"
         },
         "Country": {
            "required": true,
            "type": "String"
         }
      },
      "type": "Record"
   }
}
```

Policy

The following policy sets up conditions that must be fulfilled by each of the provided context elements. It builds on the existing static policy with the description **Customer Role - Get Order**. This policy initially only requires that the principal that makes a request is the owner of the resource.

```
permit (
    principal in DigitalPetStore::Role::"Customer",
    action in [DigitalPetStore::Action::"GetOrder"],
    resource
) when {
    principal == resource.owner &&
    context.MfaAuthorized == true &&
    context.UserAgent like "*My UserAgent*" &&
    context.RequestedOrderCount <= 4 &&
    context.AccountCodes.contains(111122223333) &&
    context.NetworkInfo.Country like "*United States*" &&
    context.NetworkInfo.SSL == true &&
    context.NetworkInfo.IPAddress like "192.0.2.*" &&
    context.approvedBy in DigitalPetStore::Role::"Employee"
};
```

We have now required that the request to retrieve an order meets the additional context conditions that we added to the request.

1. The user must have signed in with MFA.
2. The user's web browser `User-Agent` must contain the string `My UserAgent`.
3. The user must have requested to view 4 or fewer orders.
4. One of the user's account codes must be `111122223333`.
5. The user's IP address must originate in the United States, they must be on an encrypted session, and their IP address must begin with `192.0.2.`.
6. An employee must have approved their order. In the `entities` element of the authorization request, we will declare a user `Bob` who has the role of `Employee`.

Request body

After you configure your policy store with the appropriate schema and policy, you can present this authorization request to the Verified Permissions API operation [IsAuthorized](). Note that the `entities` segment contains a definition of `Bob`, a user with a role of `Employee`.

```
{
   "principal": {
      "entityType": "DigitalPetStore::User",
      "entityId": "Alice"
   },
   "action": {
      "actionType": "DigitalPetStore::Action",
      "actionId": "GetOrder"
   },
   "resource": {
      "entityType": "DigitalPetStore::Order",
      "entityId": "1234"
   },
   "context": {
     "contextMap": {
      "MfaAuthorized": {
       "boolean": true
      },
      "UserAgent": {
       "string": "My UserAgent 1.12"
      },
      "RequestedOrderCount":{
       "long": 4
      },
      "AccountCodes": {
```

```
        "set": [
         {"long": 111122223333},
         {"long": 444455556666},
         {"long": 123456789012}
         ]
       },
       "NetworkInfo": {
         "record": {
           "IPAddress": {"string": "192.0.2.178"},
           "Country": {"string": "United States of America"},
           "SSL": {"boolean": true}
         }
       },
       "approvedBy": {
         "entityIdentifier": {
          "entityId": "Bob",
          "entityType": "DigitalPetStore::User"
         }
       }
      }
    },
    "entities": {
      "entityList": [
        {
          "identifier": {
             "entityType": "DigitalPetStore::User",
             "entityId": "Alice"
          },
          "attributes": {
             "memberId": {
               "string": "801b87f2-1a5c-40b3-b580-eacad506d4e6"
               }
          },
          "parents": [
             {
             "entityType": "DigitalPetStore::Role",
             "entityId": "Customer"
             }
          ]
        },
        {
          "identifier": {
             "entityType": "DigitalPetStore::User",
             "entityId": "Bob"
```

```
          },
          "attributes": {
             "memberId": {
               "string": "49d9b81e-735d-429c-989d-93bec0bcfd8b"
               }
          },
          "parents": [
             {
             "entityType": "DigitalPetStore::Role",
             "entityId": "Employee"
             }
          ]
        },
        {
          "identifier": {
             "entityType": "DigitalPetStore::Order",
             "entityId": "1234"
          },
          "attributes": {
             "owner": {
               "entityIdentifier": {
                 "entityType": "DigitalPetStore::User",
                 "entityId": "Alice"
               }
            }
          },
          "parents": []
        }
      ]
    },
    "policyStoreId": "PSEXAMPLEabcdefg111111"
}
```

Security in Amazon Verified Permissions

Cloud security at AWS is the highest priority. As an AWS customer, you benefit from data centers and network architectures that are built to meet the requirements of the most security-sensitive organizations.

Security is a shared responsibility between AWS and you. The shared responsibility model describes this as security *of* the cloud and security *in* the cloud:

- **Security of the cloud** – AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services that you can use securely. Third-party auditors regularly test and verify the effectiveness of our security as part of the AWS Compliance Programs. To learn about the compliance programs that apply to Amazon Verified Permissions, see AWS Services in Scope by Compliance Program.
- **Security in the cloud** – Your responsibility is determined by the AWS service that you use. You are also responsible for other factors including the sensitivity of your data, your company's requirements, and applicable laws and regulations.

This documentation helps you understand how to apply the shared responsibility model when using Verified Permissions. The following topics show you how to configure Verified Permissions to meet your security and compliance objectives. You also learn how to use other AWS services that help you to monitor and secure your Verified Permissions resources.

Topics

Data protection in Amazon Verified Permissions

The AWS shared responsibility model applies to data protection in Amazon Verified Permissions. As described in this model, AWS is responsible for protecting the global infrastructure that runs all of the AWS Cloud. You are responsible for maintaining control over your content that is hosted on this infrastructure. This content includes the security configuration and management tasks for the

AWS services that you use. For more information about data privacy, see the [Data Privacy FAQ](). For information about data protection in Europe, see the [AWS Shared Responsibility Model and GDPR]() blog post on the *AWS Security Blog*.

- For data protection purposes, we recommend that you protect AWS account credentials and set up individual users with AWS IAM Identity Center or AWS Identity and Access Management (IAM). That way, each user is given only the permissions necessary to fulfill their job duties.
- We recommend that you secure your data in the following ways:
  - Use multi-factor authentication (MFA) with each account.
  - Use SSL/TLS to communicate with AWS resources. We require TLS 1.2.
  - Set up API and user activity logging with AWS CloudTrail.
  - Use AWS encryption solutions, along with all default security controls within AWS services.
  - Use advanced managed security services such as Amazon Macie, which assists in discovering and securing sensitive data that is stored in Amazon S3.
  - If you require FIPS 140-2 validated cryptographic modules when accessing AWS through a command line interface or an API, use a FIPS endpoint. For more information about the available FIPS endpoints, see [Federal Information Processing Standard (FIPS) 140-2]().
- We strongly recommend that you never put confidential or sensitive information, such as your customers' email addresses, into tags or free-form text fields such as a **Name** field. This includes when you work with Verified Permissions or other AWS services using the console, API, AWS CLI, or AWS SDKs. Any data that you enter into tags or free-form text fields used for names may be used for billing or diagnostic logs. If you provide a URL to an external server, we strongly recommend that you do not include credentials information in the URL to validate your request to that server.
- Your action names should not include any sensitive information.
- We also strongly recommend that you always use unique, non-mutable, and non-reusable identifiers for your entities (resources and principals). In a test environment, you might choose to use simple entity identifiers, such as `jane` or bob for the name of an entity of type `User`. However, in a production system, it's critical for security reasons that you use unique values that can't be reused. We recommend that you use values like universally unique identifiers (UUIDs). For example, consider the user `jane` who leaves the company. Later, you let someone else use the name `jane`. That new user gets access automatically to everything granted by policies that still reference `User::"jane"`. Verified Permissions and Cedar can't distinguish between the new user and the previous user.

This guidance applies to both principal and resource identifiers. Always use identifiers that are guaranteed unique and never reused to ensure that you don’t grant access unintentionally because of the presence of an old identifier in a policy.

- Ensure that the strings that you provide to define `Long` and `Decimal` values are within the valid range of each type. Also, ensure that your use of any arithmetic operators don't result in a value outside of the valid range. If the range is exceeded, the operation results in an overflow exception. A policy that results in an error is ignored, meaning that a Permit policy might unexpectedly fail to allow access, or a Forbid policy might unexpectedly fail to block access.

Data encryption

Amazon Verified Permissions automatically encrypts all customer data such as policies with an AWS managed key, so the use of a customer managed key is neither necessary nor supported.

Identity and access management for Amazon Verified Permissions

AWS Identity and Access Management (IAM) is an AWS service that helps an administrator securely control access to AWS resources. IAM administrators control who can be *authenticated* (signed in) and *authorized* (have permissions) to use Verified Permissions resources. IAM is an AWS service that you can use with no additional charge.

Topics

- Audience
- Authenticating with identities
- Managing access using policies
- How Amazon Verified Permissions works with IAM
- Identity-based policy examples for Amazon Verified Permissions
- Troubleshooting Amazon Verified Permissions identity and access

Audience

How you use AWS Identity and Access Management (IAM) differs, depending on the work that you do in Verified Permissions.

Service user – If you use the Verified Permissions service to do your job, then your administrator provides you with the credentials and permissions that you need. As you use more Verified Permissions features to do your work, you might need additional permissions. Understanding how access is managed can help you request the right permissions from your administrator. If you cannot access a feature in Verified Permissions, see Troubleshooting Amazon Verified Permissions identity and access.

Service administrator – If you're in charge of Verified Permissions resources at your company, you probably have full access to Verified Permissions. It's your job to determine which Verified Permissions features and resources your service users should access. You must then submit requests to your IAM administrator to change the permissions of your service users. Review the information on this page to understand the basic concepts of IAM. To learn more about how your company can use IAM with Verified Permissions, see How Amazon Verified Permissions works with IAM.

IAM administrator – If you're an IAM administrator, you might want to learn details about how you can write policies to manage access to Verified Permissions. To view example Verified Permissions identity-based policies that you can use in IAM, see Identity-based policy examples for Amazon Verified Permissions.

Authenticating with identities

Authentication is how you sign in to AWS using your identity credentials. You must be *authenticated* (signed in to AWS) as the AWS account root user, as an IAM user, or by assuming an IAM role.

You can sign in to AWS as a federated identity by using credentials provided through an identity source. AWS IAM Identity Center (IAM Identity Center) users, your company's single sign-on authentication, and your Google or Facebook credentials are examples of federated identities. When you sign in as a federated identity, your administrator previously set up identity federation using IAM roles. When you access AWS by using federation, you are indirectly assuming a role.

Depending on the type of user you are, you can sign in to the AWS Management Console or the AWS access portal. For more information about signing in to AWS, see How to sign in to your AWS account in the *AWS Sign-In User Guide*.

If you access AWS programmatically, AWS provides a software development kit (SDK) and a command line interface (CLI) to cryptographically sign your requests by using your credentials. If you don't use AWS tools, you must sign requests yourself. For more information about using the recommended method to sign requests yourself, see Signing AWS API requests in the *IAM User Guide*.

Regardless of the authentication method that you use, you might be required to provide additional security information. For example, AWS recommends that you use multi-factor authentication (MFA) to increase the security of your account. To learn more, see Multi-factor authentication in the *AWS IAM Identity Center User Guide* and Using multi-factor authentication (MFA) in AWS in the *IAM User Guide*.

AWS account root user

When you create an AWS account, you begin with one sign-in identity that has complete access to all AWS services and resources in the account. This identity is called the AWS account *root user* and is accessed by signing in with the email address and password that you used to create the account. We strongly recommend that you don't use the root user for your everyday tasks. Safeguard your root user credentials and use them to perform the tasks that only the root user can perform. For the complete list of tasks that require you to sign in as the root user, see Tasks that require root user credentials in the *IAM User Guide*.

Federated identity

As a best practice, require human users, including users that require administrator access, to use federation with an identity provider to access AWS services by using temporary credentials.

A *federated identity* is a user from your enterprise user directory, a web identity provider, the AWS Directory Service, the Identity Center directory, or any user that accesses AWS services by using credentials provided through an identity source. When federated identities access AWS accounts, they assume roles, and the roles provide temporary credentials.

For centralized access management, we recommend that you use AWS IAM Identity Center. You can create users and groups in IAM Identity Center, or you can connect and synchronize to a set of users and groups in your own identity source for use across all your AWS accounts and applications. For information about IAM Identity Center, see What is IAM Identity Center? in the *AWS IAM Identity Center User Guide*.

IAM users and groups

An *IAM user* is an identity within your AWS account that has specific permissions for a single person or application. Where possible, we recommend relying on temporary credentials instead of creating IAM users who have long-term credentials such as passwords and access keys. However, if you have specific use cases that require long-term credentials with IAM users, we recommend that you rotate access keys. For more information, see Rotate access keys regularly for use cases that require long-term credentials in the *IAM User Guide*.

An *IAM group* is an identity that specifies a collection of IAM users. You can't sign in as a group. You can use groups to specify permissions for multiple users at a time. Groups make permissions easier to manage for large sets of users. For example, you could have a group named *IAMAdmins* and give that group permissions to administer IAM resources.

Users are different from roles. A user is uniquely associated with one person or application, but a role is intended to be assumable by anyone who needs it. Users have permanent long-term credentials, but roles provide temporary credentials. To learn more, see When to create an IAM user (instead of a role) in the *IAM User Guide*.

IAM roles

An *IAM role* is an identity within your AWS account that has specific permissions. It is similar to an IAM user, but is not associated with a specific person. You can temporarily assume an IAM role in the AWS Management Console by switching roles. You can assume a role by calling an AWS CLI or AWS API operation or by using a custom URL. For more information about methods for using roles, see Using IAM roles in the *IAM User Guide*.

IAM roles with temporary credentials are useful in the following situations:

- **Federated user access** – To assign permissions to a federated identity, you create a role and define permissions for the role. When a federated identity authenticates, the identity is associated with the role and is granted the permissions that are defined by the role. For information about roles for federation, see Creating a role for a third-party Identity Provider in the *IAM User Guide*. If you use IAM Identity Center, you configure a permission set. To control what your identities can access after they authenticate, IAM Identity Center correlates the permission set to a role in IAM. For information about permissions sets, see Permission sets in the *AWS IAM Identity Center User Guide*.
- **Temporary IAM user permissions** – An IAM user or role can assume an IAM role to temporarily take on different permissions for a specific task.

- **Cross-account access** – You can use an IAM role to allow someone (a trusted principal) in a different account to access resources in your account. Roles are the primary way to grant cross-account access. However, with some AWS services, you can attach a policy directly to a resource (instead of using a role as a proxy). To learn the difference between roles and resource-based policies for cross-account access, see How IAM roles differ from resource-based policies in the *IAM User Guide*.
- **Applications running on Amazon EC2** – You can use an IAM role to manage temporary credentials for applications that are running on an EC2 instance and making AWS CLI or AWS API requests. This is preferable to storing access keys within the EC2 instance. To assign an AWS role to an EC2 instance and make it available to all of its applications, you create an instance profile that is attached to the instance. An instance profile contains the role and enables programs that are running on the EC2 instance to get temporary credentials. For more information, see Using an IAM role to grant permissions to applications running on Amazon EC2 instances in the *IAM User Guide*.

To learn whether to use IAM roles or IAM users, see When to create an IAM role (instead of a user) in the *IAM User Guide*.

Managing access using policies

You control access in AWS by creating policies and attaching them to AWS identities or resources. A policy is an object in AWS that, when associated with an identity or resource, defines their permissions. AWS evaluates these policies when a principal (user, root user, or role session) makes a request. Permissions in the policies determine whether the request is allowed or denied. Most policies are stored in AWS as JSON documents. For more information about the structure and contents of JSON policy documents, see Overview of JSON policies in the *IAM User Guide*.

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

By default, users and roles have no permissions. To grant users permission to perform actions on the resources that they need, an IAM administrator can create IAM policies. The administrator can then add the IAM policies to roles, and users can assume the roles.

IAM policies define permissions for an action regardless of the method that you use to perform the operation. For example, suppose that you have a policy that allows the `iam:GetRole` action. A user with that policy can get role information from the AWS Management Console, the AWS CLI, or the AWS API.

Identity-based policies

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, group of users, or role. These policies control what actions users and roles can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Creating IAM policies in the *IAM User Guide*.

Identity-based policies can be further categorized as *inline policies* or *managed policies*. Inline policies are embedded directly into a single user, group, or role. Managed policies are standalone policies that you can attach to multiple users, groups, and roles in your AWS account. Managed policies include AWS managed policies and customer managed policies. To learn how to choose between a managed policy or an inline policy, see Choosing between managed policies and inline policies in the *IAM User Guide*.

Resource-based policies

Resource-based policies are JSON policy documents that you attach to a resource. Examples of resource-based policies are IAM *role trust policies* and Amazon S3 *bucket policies*. In services that support resource-based policies, service administrators can use them to control access to a specific resource. For the resource where the policy is attached, the policy defines what actions a specified principal can perform on that resource and under what conditions. You must specify a principal in a resource-based policy. Principals can include accounts, users, roles, federated users, or AWS services.

Resource-based policies are inline policies that are located in that service. You can't use AWS managed policies from IAM in a resource-based policy.

Access control lists (ACLs)

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

Amazon S3, AWS WAF, and Amazon VPC are examples of services that support ACLs. To learn more about ACLs, see Access control list (ACL) overview in the *Amazon Simple Storage Service Developer Guide*.

Other policy types

AWS supports additional, less-common policy types. These policy types can set the maximum permissions granted to you by the more common policy types.

- **Permissions boundaries** – A permissions boundary is an advanced feature in which you set the maximum permissions that an identity-based policy can grant to an IAM entity (IAM user or role). You can set a permissions boundary for an entity. The resulting permissions are the intersection of an entity's identity-based policies and its permissions boundaries. Resource-based policies that specify the user or role in the `Principal` field are not limited by the permissions boundary. An explicit deny in any of these policies overrides the allow. For more information about permissions boundaries, see Permissions boundaries for IAM entities in the *IAM User Guide*.
- **Service control policies (SCPs)** – SCPs are JSON policies that specify the maximum permissions for an organization or organizational unit (OU) in AWS Organizations. AWS Organizations is a service for grouping and centrally managing multiple AWS accounts that your business owns. If you enable all features in an organization, then you can apply service control policies (SCPs) to any or all of your accounts. The SCP limits permissions for entities in member accounts, including each AWS account root user. For more information about Organizations and SCPs, see How SCPs work in the *AWS Organizations User Guide*.
- **Session policies** – Session policies are advanced policies that you pass as a parameter when you programmatically create a temporary session for a role or federated user. The resulting session's permissions are the intersection of the user or role's identity-based policies and the session policies. Permissions can also come from a resource-based policy. An explicit deny in any of these policies overrides the allow. For more information, see Session policies in the *IAM User Guide*.

Multiple policy types

When multiple types of policies apply to a request, the resulting permissions are more complicated to understand. To learn how AWS determines whether to allow a request when multiple policy types are involved, see Policy evaluation logic in the *IAM User Guide*.

How Amazon Verified Permissions works with IAM

Before you use IAM to manage access to Verified Permissions, learn what IAM features are available to use with Verified Permissions.

IAM features you can use with Amazon Verified Permissions

IAM feature	Verified Permissions support
Identity-based policies	Yes
Resource-based policies	No
Policy actions	Yes
Policy resources	Yes
Policy condition keys	No
ACLs	No
ABAC (tags in policies)	No
Temporary credentials	Yes
Principal permissions	Yes
Service roles	No
Service-linked roles	No

To get a high-level view of how Verified Permissions and other AWS services work with most IAM features, see AWS services that work with IAM in the *IAM User Guide*.

Identity-based policies for Verified Permissions

Supports identity-based policies	Yes

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, group of users, or role. These policies control what actions users and roles can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Creating IAM policies in the *IAM User Guide*.

With IAM identity-based policies, you can specify allowed or denied actions and resources as well as the conditions under which actions are allowed or denied. You can't specify the principal in an identity-based policy because it applies to the user or role to which it is attached. To learn about all of the elements that you can use in a JSON policy, see IAM JSON policy elements reference in the *IAM User Guide*.

Identity-based policy examples for Verified Permissions

To view examples of Verified Permissions identity-based policies, see Identity-based policy examples for Amazon Verified Permissions.

Resource-based policies within Verified Permissions

Supports resource-based policies	No

Resource-based policies are JSON policy documents that you attach to a resource. Examples of resource-based policies are IAM *role trust policies* and Amazon S3 *bucket policies*. In services that support resource-based policies, service administrators can use them to control access to a specific resource. For the resource where the policy is attached, the policy defines what actions a specified principal can perform on that resource and under what conditions. You must specify a principal in a resource-based policy. Principals can include accounts, users, roles, federated users, or AWS services.

To enable cross-account access, you can specify an entire account or IAM entities in another account as the principal in a resource-based policy. Adding a cross-account principal to a resource-based policy is only half of establishing the trust relationship. When the principal and the resource are in different AWS accounts, an IAM administrator in the trusted account must also grant the principal entity (user or role) permission to access the resource. They grant permission by attaching an identity-based policy to the entity. However, if a resource-based policy grants access to a principal in the same account, no additional identity-based policy is required. For more information, see How IAM roles differ from resource-based policies in the *IAM User Guide*.

Policy actions for Verified Permissions

Supports policy actions	Yes

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Action` element of a JSON policy describes the actions that you can use to allow or deny access in a policy. Policy actions usually have the same name as the associated AWS API operation. There are some exceptions, such as *permission-only actions* that don't have a matching API operation. There are also some operations that require multiple actions in a policy. These additional actions are called *dependent actions*.

Include actions in a policy to grant permissions to perform the associated operation.

To see a list of Verified Permissions actions, see [Actions defined by Amazon Verified Permissions] in the *Service Authorization Reference*.

Policy actions in Verified Permissions use the following prefix before the action:

```
verifiedpermissions
```

To specify multiple actions in a single statement, separate them with commas.

```
"Action": [
      "verifiedpermissions:action1",
      "verifiedpermissions:action2"
         ]
```

You can specify multiple actions using wildcards (*). For example, to specify all actions that begin with the word `Get`, include the following action:

```
"Action": "verifiedpermissions:Get*"
```

To view examples of Verified Permissions identity-based policies, see [Identity-based policy examples for Amazon Verified Permissions].

Policy resources for Verified Permissions

Supports policy resources	Yes

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Resource` JSON policy element specifies the object or objects to which the action applies. Statements must include either a `Resource` or a `NotResource` element. As a best practice, specify a resource using its Amazon Resource Name (ARN). You can do this for actions that support a specific resource type, known as *resource-level permissions*.

For actions that don't support resource-level permissions, such as listing operations, use a wildcard (*) to indicate that the statement applies to all resources.

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

To see a list of Verified Permissions resource types and their ARNs, see Resource types defined by Amazon Verified Permissions in the *Service Authorization Reference*. To learn with which actions you can specify the ARN of each resource, see Actions defined by Amazon Verified Permissions.

Policy condition keys for Verified Permissions

Supports service-specific policy condition keys	No

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Condition` element (or `Condition` *block*) lets you specify conditions in which a statement is in effect. The `Condition` element is optional. You can create conditional expressions that use condition operators, such as equals or less than, to match the condition in the policy with values in the request.

If you specify multiple `Condition` elements in a statement, or multiple keys in a single `Condition` element, AWS evaluates them using a logical `AND` operation. If you specify multiple values for a single condition key, AWS evaluates the condition using a logical `OR` operation. All of the conditions must be met before the statement's permissions are granted.

You can also use placeholder variables when you specify conditions. For example, you can grant an IAM user permission to access a resource only if it is tagged with their IAM user name. For more information, see IAM policy elements: variables and tags in the *IAM User Guide*.

AWS supports global condition keys and service-specific condition keys. To see all AWS global condition keys, see AWS global condition context keys in the *IAM User Guide*.

ACLs in Verified Permissions

Supports ACLs	No

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

ABAC with Verified Permissions

Supports ABAC (tags in policies)	No

Attribute-based access control (ABAC) is an authorization strategy that defines permissions based on attributes. In AWS, these attributes are called *tags*. You can attach tags to IAM entities (users or roles) and to many AWS resources. Tagging entities and resources is the first step of ABAC. Then you design ABAC policies to allow operations when the principal's tag matches the tag on the resource that they are trying to access.

ABAC is helpful in environments that are growing rapidly and helps with situations where policy management becomes cumbersome.

To control access based on tags, you provide tag information in the condition element of a policy using the `aws:ResourceTag/key-name`, `aws:RequestTag/key-name`, or `aws:TagKeys` condition keys.

If a service supports all three condition keys for every resource type, then the value is **Yes** for the service. If a service supports all three condition keys for only some resource types, then the value is **Partial**.

For more information about ABAC, see What is ABAC? in the *IAM User Guide*. To view a tutorial with steps for setting up ABAC, see Use attribute-based access control (ABAC) in the *IAM User Guide*.

Using temporary credentials with Verified Permissions

Supports temporary credentials	Yes

Some AWS services don't work when you sign in using temporary credentials. For additional information, including which AWS services work with temporary credentials, see AWS services that work with IAM in the *IAM User Guide*.

You are using temporary credentials if you sign in to the AWS Management Console using any method except a user name and password. For example, when you access AWS using your company's single sign-on (SSO) link, that process automatically creates temporary credentials. You also automatically create temporary credentials when you sign in to the console as a user and then switch roles. For more information about switching roles, see Switching to a role (console) in the *IAM User Guide*.

You can manually create temporary credentials using the AWS CLI or AWS API. You can then use those temporary credentials to access AWS. AWS recommends that you dynamically generate temporary credentials instead of using long-term access keys. For more information, see Temporary security credentials in IAM.

Cross-service principal permissions for Verified Permissions

Supports principal permissions	Yes

When you use an IAM user or role to perform actions in AWS, you are considered a principal. When you use some services, you might perform an action that then initiates another action in a different service. FAS uses the permissions of the principal calling an AWS service, combined with the requesting AWS service to make requests to downstream services. FAS requests are only made when a service receives a request that requires interactions with other AWS services or resources to complete. In this case, you must have permissions to perform both actions. For policy details when making FAS requests, see Forward access sessions.

Service roles for Verified Permissions

Supports service roles	No

A service role is an IAM role that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For more information, see Creating a role to delegate permissions to an AWS service in the *IAM User Guide*.

Service-linked roles for Verified Permissions

Supports service-linked roles	No

A service-linked role is a type of service role that is linked to an AWS service. The service can assume the role to perform an action on your behalf. Service-linked roles appear in your AWS account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.

For details about creating or managing service-linked roles, see AWS services that work with IAM. Find a service in the table that includes a `Yes` in the **Service-linked role** column. Choose the **Yes** link to view the service-linked role documentation for that service.

Identity-based policy examples for Amazon Verified Permissions

By default, users and roles don't have permission to create or modify Verified Permissions resources. They also can't perform tasks by using the AWS Management Console, AWS Command Line Interface (AWS CLI), or AWS API. An IAM administrator must create IAM policies that grant users and roles permission to perform actions on the resources that they need. The administrator must then attach those policies for users that require them.

To learn how to create an IAM identity-based policy by using these example JSON policy documents, see Creating IAM policies in the *IAM User Guide*.

For details about actions and resource types defined by Verified Permissions, including the format of the ARNs for each of the resource types, see see Actions, resources, and condition keys for Amazon Verified Permissions in the *Service Authorization Reference*.

Topics

Policy best practices

Identity-based policies determine whether someone can create, access, or delete Verified Permissions resources in your account. These actions can incur costs for your AWS account. When you create or edit identity-based policies, follow these guidelines and recommendations:

- **Get started with AWS managed policies and move toward least-privilege permissions** – To get started granting permissions to your users and workloads, use the *AWS managed policies* that grant permissions for many common use cases. They are available in your AWS account. We recommend that you reduce permissions further by defining AWS customer managed policies that are specific to your use cases. For more information, see AWS managed policies or AWS managed policies for job functions in the *IAM User Guide*.
- **Apply least-privilege permissions** – When you set permissions with IAM policies, grant only the permissions required to perform a task. You do this by defining the actions that can be taken on specific resources under specific conditions, also known as *least-privilege permissions*. For more information about using IAM to apply permissions, see Policies and permissions in IAM in the *IAM User Guide*.
- **Use conditions in IAM policies to further restrict access** – You can add a condition to your policies to limit access to actions and resources. For example, you can write a policy condition to specify that all requests must be sent using SSL. You can also use conditions to grant access to service actions if they are used through a specific AWS service, such as AWS CloudFormation. For more information, see IAM JSON policy elements: Condition in the *IAM User Guide*.
- **Use IAM Access Analyzer to validate your IAM policies to ensure secure and functional permissions** – IAM Access Analyzer validates new and existing policies so that the policies adhere to the IAM policy language (JSON) and IAM best practices. IAM Access Analyzer provides more than 100 policy checks and actionable recommendations to help you author secure and functional policies. For more information, see IAM Access Analyzer policy validation in the *IAM User Guide*.
- **Require multi-factor authentication (MFA)** – If you have a scenario that requires IAM users or a root user in your AWS account, turn on MFA for additional security. To require MFA when API operations are called, add MFA conditions to your policies. For more information, see Configuring MFA-protected API access in the *IAM User Guide*.

For more information about best practices in IAM, see Security best practices in IAM in the *IAM User Guide*.

Using the Verified Permissions console

To access the Amazon Verified Permissions console, you must have a minimum set of permissions. These permissions must allow you to list and view details about the Verified Permissions resources in your AWS account. If you create an identity-based policy that is more restrictive than the minimum required permissions, the console won't function as intended for entities (users or roles) with that policy.

You don't need to allow minimum console permissions for users that are making calls only to the AWS CLI or the AWS API. Instead, allow access to only the actions that match the API operation that they're trying to perform.

To ensure that users and roles can still use the Verified Permissions console, also attach the Verified Permissions *ConsoleAccess* or *ReadOnly* AWS managed policy to the entities. For more information, see [Adding permissions to a user](#) in the *IAM User Guide*.

Allow users to view their own permissions

This example shows how you might create a policy that allows IAM users to view the inline and managed policies that are attached to their user identity. This policy includes permissions to complete this action on the console or programmatically using the AWS CLI or AWS API.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ViewOwnUserInfo",
            "Effect": "Allow",
            "Action": [
                "iam:GetUserPolicy",
                "iam:ListGroupsForUser",
                "iam:ListAttachedUserPolicies",
                "iam:ListUserPolicies",
                "iam:GetUser"
            ],
            "Resource": ["arn:aws:iam::*:user/${aws:username}"]
        },
        {
            "Sid": "NavigateInConsole",
            "Effect": "Allow",
            "Action": [
                "iam:GetGroupPolicy",
```

```
                "iam:GetPolicyVersion",
                "iam:GetPolicy",
                "iam:ListAttachedGroupPolicies",
                "iam:ListGroupPolicies",
                "iam:ListPolicyVersions",
                "iam:ListPolicies",
                "iam:ListUsers"
            ],
            "Resource": "*"
        }
    ]
}
```

Troubleshooting Amazon Verified Permissions identity and access

Use the following information to help you diagnose and fix common issues that you might encounter when working with Verified Permissions and IAM.

Topics

I am not authorized to perform an action in Verified Permissions

If you receive an error that you're not authorized to perform an action, your policies must be updated to allow you to perform the action.

The following example error occurs when the `mateojackson` IAM user tries to use the console to view details about a fictional *`my-example-widget`* resource but doesn't have the fictional `verifiedpermissions:`*`GetWidget`* permissions.

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
 verifiedpermissions:GetWidget on resource: my-example-widget
```

In this case, the policy for the `mateojackson` user must be updated to allow access to the *`my-example-widget`* resource by using the `verifiedpermissions:`*`GetWidget`* action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

I am not authorized to perform iam:PassRole

If you receive an error that you're not authorized to perform the `iam:PassRole` action, your policies must be updated to allow you to pass a role to Verified Permissions.

Some AWS services allow you to pass an existing role to that service instead of creating a new service role or service-linked role. To do this, you must have permissions to pass the role to the service.

The following example error occurs when an IAM user named `marymajor` tries to use the console to perform an action in Verified Permissions. However, the action requires the service to have permissions that are granted by a service role. Mary does not have permissions to pass the role to the service.

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
 iam:PassRole
```

In this case, Mary's policies must be updated to allow her to perform the `iam:PassRole` action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

I want to allow people outside of my AWS account to access my Verified Permissions resources

You can create a role that users in other accounts or people outside of your organization can use to access your resources. You can specify who is trusted to assume the role. For services that support resource-based policies or access control lists (ACLs), you can use those policies to grant people access to your resources.

To learn more, consult the following:

- To learn whether Verified Permissions supports these features, see How Amazon Verified Permissions works with IAM.
- To learn how to provide access to your resources across AWS accounts that you own, see Providing access to an IAM user in another AWS account that you own in the *IAM User Guide*.

- To learn how to provide access to your resources to third-party AWS accounts, see Providing access to AWS accounts owned by third parties in the *IAM User Guide*.
- To learn how to provide access through identity federation, see Providing access to externally authenticated users (identity federation) in the *IAM User Guide*.
- To learn the difference between using roles and resource-based policies for cross-account access, see How IAM roles differ from resource-based policies in the *IAM User Guide*.

Compliance validation for Amazon Verified Permissions

To learn whether an AWS service is within the scope of specific compliance programs, see AWS services in Scope by Compliance Program and choose the compliance program that you are interested in. For general information, see AWS Compliance Programs.

You can download third-party audit reports using AWS Artifact. For more information, see Downloading Reports in AWS Artifact.

Your compliance responsibility when using AWS services is determined by the sensitivity of your data, your company's compliance objectives, and applicable laws and regulations. AWS provides the following resources to help with compliance:

- Security and Compliance Quick Start Guides – These deployment guides discuss architectural considerations and provide steps for deploying baseline environments on AWS that are security and compliance focused.
- Architecting for HIPAA Security and Compliance on Amazon Web Services – This whitepaper describes how companies can use AWS to create HIPAA-eligible applications.

> **Note**
>
> Not all AWS services are HIPAA eligible. For more information, see the HIPAA Eligible Services Reference.

- AWS Compliance Resources – This collection of workbooks and guides might apply to your industry and location.
- AWS Customer Compliance Guides – Understand the shared responsibility model through the lens of compliance. The guides summarize the best practices for securing AWS services and map the guidance to security controls across multiple frameworks (including National Institute of

Standards and Technology (NIST), Payment Card Industry Security Standards Council (PCI), and International Organization for Standardization (ISO)).

- Evaluating Resources with Rules in the *AWS Config Developer Guide* – The AWS Config service assesses how well your resource configurations comply with internal practices, industry guidelines, and regulations.
- AWS Security Hub – This AWS service provides a comprehensive view of your security state within AWS. Security Hub uses security controls to evaluate your AWS resources and to check your compliance against security industry standards and best practices. For a list of supported services and controls, see Security Hub controls reference.
- AWS Audit Manager – This AWS service helps you continuously audit your AWS usage to simplify how you manage risk and compliance with regulations and industry standards.

Resilience in Amazon Verified Permissions

The AWS global infrastructure is built around AWS Regions and Availability Zones. AWS Regions provide multiple physically separated and isolated Availability Zones, which are connected with low-latency, high-throughput, and highly redundant networking. With Availability Zones, you can design and operate applications and databases that automatically fail over between zones without interruption. Availability Zones are more highly available, fault tolerant, and scalable than traditional single or multiple data center infrastructures.

When you create a Verified Permissions policy store , it is created within an individual AWS Region, and is automatically replicated across the data centers that make up that Region's Availability Zones. At this time, Verified Permissions doesn't support any cross-region replication.

For more information about AWS Regions and Availability Zones, see AWS Global Infrastructure.

Monitoring Amazon Verified Permissions

Monitoring is an important part of maintaining the reliability, availability, and performance of Amazon Verified Permissions and your other AWS solutions. AWS provides the following monitoring tools to watch Verified Permissions, report when something is wrong, and take automatic actions when appropriate:

- *AWS CloudTrail* captures API calls and related events made by or on behalf of your AWS account and delivers the log files to an Amazon S3 bucket that you specify. You can identify which users and accounts called AWS, the source IP address from which the calls were made, and when the calls occurred. For more information, see the AWS CloudTrail User Guide.

Logging Amazon Verified Permissions API calls using AWS CloudTrail

Amazon Verified Permissions is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service in Verified Permissions. CloudTrail captures all API calls for Verified Permissions as events. The calls captured include calls from the Verified Permissions console and code calls to the Verified Permissions API operations. If you create a trail, you can enable continuous delivery of CloudTrail events to an Amazon S3 bucket, including events for Verified Permissions. If you don't configure a trail, you can still view the most recent events in the CloudTrail console in **Event history**. Using the information collected by CloudTrail, you can determine the request that was made to Verified Permissions, the IP address from which the request was made, who made the request, when it was made, and additional details.

To learn more about CloudTrail, see the AWS CloudTrail User Guide.

Verified Permissions information in CloudTrail

CloudTrail is enabled on your AWS account when you create the account. When activity occurs in Verified Permissions, that activity is recorded in a CloudTrail event along with other AWS service events in **Event history**. You can view, search, and download recent events in your AWS account. For more information, see Viewing events with CloudTrail Event history.

For an ongoing record of events in your AWS account, including events for Verified Permissions, create a trail. A *trail* enables CloudTrail to deliver log files to an Amazon S3 bucket. By default, when you create a trail in the console, the trail applies to all AWS Regions. The trail logs events from all Regions in the AWS partition and delivers the log files to the Amazon S3 bucket that you specify. Additionally, you can configure other AWS services to further analyze and act upon the event data collected in CloudTrail logs. For more information, see the following:

- Overview for creating a trail
- CloudTrail supported services and integrations
- Configuring Amazon SNS notifications for CloudTrail
- Receiving CloudTrail log files from multiple regions and Receiving CloudTrail log files from multiple accounts

All Verified Permissions actions are logged by CloudTrail and are documented in the Amazon Verified Permissions API Reference Guide. For example, calls to the `CreateIdentitySource`, `DeletePolicy`, and `ListPolicyStores` actions generate entries in the CloudTrail log files.

Every event or log entry contains information about who generated the request. The identity information helps you determine the following:

- Whether the request was made with root or AWS Identity and Access Management (IAM) user credentials.
- Whether the request was made with temporary security credentials for a role or federated user.
- Whether the request was made by another AWS service.

For more information, see the CloudTrail userIdentity element.

Data events like IsAuthorized and IsAuthorizedWithToken are not logged by default when you create a trail or event data store. To record CloudTrail data events, you must explicitly add the supported resources or resource types for which you want to collect activity. For more information, see Data events in the *AWS CloudTrail User Guide*.

Understanding Verified Permissions log file entries

A trail is a configuration that enables delivery of events as log files to an Amazon S3 bucket that you specify. CloudTrail log files contain one or more log entries. An event represents a single request from any source and includes information about the requested action, the date and time of the action, request parameters, and so on. CloudTrail log files aren't an ordered stack trace of the public API calls, so they don't appear in any specific order.

Topics

> **Note**
>
> Some fields have been redacted from the examples for data privacy.

IsAuthorized


```
{
    "eventVersion": "1.08",
    "userIdentity": {
  "type": "AssumedRole",
  "principalId": "EXAMPLE_PRINCIPAL_ID",
  "arn": "arn:aws:iam::123456789012:role/ExampleRole",
  "accountId": "123456789012",
  "accessKeyId": "AKIAIOSFODNN7EXAMPLE"
    },
    "eventTime": "2023-11-20T22:55:03Z",
    "eventSource": "verifiedpermissions.amazonaws.com",
```

```
    "eventName": "IsAuthorized",
    "awsRegion": "us-west-2",
    "sourceIPAddress": "203.0.113.0",
    "userAgent": "aws-cli/2.11.18 Python/3.11.3 Linux/5.4.241-160.348.amzn2int.x86_64
 exe/x86_64.amzn.2 prompt/off command/verifiedpermissions.is-authorized",
    "requestParameters": {
        "principal": {
            "entityType": "PhotoFlash::User",
            "entityId": "alice"
        },
        "action": {
            "actionType": "PhotoFlash::Action",
            "actionId": "ViewPhoto"
        },
        "resource": {
            "entityType": "PhotoFlash::Photo",
            "entityId": "VacationPhoto94.jpg"
        },
        "policyStoreId": "PSEXAMPLEabcdefg111111"
    },
    "responseElements": null,
    "additionalEventData": {
        "decision": "ALLOW"
    },
    "requestID": "346c4b6a-d12f-46b6-bc06-6c857bd3b28e",
    "eventID": "8a4fed32-9605-45dd-a09a-5ebbf0715bbc",
    "readOnly": true,
    "resources": [
        {
    "accountId": "123456789012",
    "type": "AWS::VerifiedPermissions::PolicyStore",
    "ARN": "arn:aws:verifiedpermissions::123456789012:policy-store/
PSEXAMPLEabcdefg111111"
        }
    ],
    "eventType": "AwsApiCall",
    "managementEvent": false,
    "recipientAccountId": "123456789012",
    "eventCategory": "Data"
}
```

BatchIsAuthorized

```
{
    "eventVersion": "1.08",
    "userIdentity": {
  "type": "AssumedRole",
  "principalId": "EXAMPLE_PRINCIPAL_ID",
  "arn": "arn:aws:iam::123456789012:role/ExampleRole",
  "accountId": "123456789012",
  "accessKeyId": "AKIAIOSFODNN7EXAMPLE"
    },
    "eventTime": "2023-11-20T23:02:33Z",
    "eventSource": "verifiedpermissions.amazonaws.com",
    "eventName": "BatchIsAuthorized",
    "awsRegion": "us-west-2",
    "sourceIPAddress": "203.0.113.0",
    "userAgent": "aws-cli/2.11.18 Python/3.11.3 Linux/5.4.241-160.348.amzn2int.x86_64
 exe/x86_64.amzn.2 prompt/off command/verifiedpermissions.is-authorized",
    "requestParameters": {
        "requests": [
            {
                "principal": {
                    "entityType": "PhotoFlash::User",
                    "entityId": "alice"
                },
                "action": {
                    "actionType": "PhotoFlash::Action",
                    "actionId": "ViewPhoto"
                },
                "resource": {
                    "entityType": "PhotoFlash::Photo",
                    "entityId": "VacationPhoto94.jpg"
                }
            },
            {
                "principal": {
                    "entityType": "PhotoFlash::User",
                    "entityId": "annalisa"
                },
                "action": {
                    "actionType": "PhotoFlash::Action",
                    "actionId": "DeletePhoto"
                },
```

```
                "resource": {
                    "entityType": "PhotoFlash::Photo",
                    "entityId": "VacationPhoto94.jpg"
                }
            }
        ],
        "policyStoreId": "PSEXAMPLEabcdefg111111"
    },
    "responseElements": null,
    "additionalEventData": {
        "results": [
            {
                "request": {
                    "principal": {
                        "entityType": "PhotoFlash::User",
                        "entityId": "alice"
                    },
                    "action": {
                        "actionType": "PhotoFlash::Action",
                        "actionId": "ViewPhoto"
                    },
                    "resource": {
                        "entityType": "PhotoFlash::Photo",
                        "entityId": "VacationPhoto94.jpg"
                    }
                },
                "decision": "ALLOW"
            },
            {
                "request": {
                    "principal": {
                        "entityType": "PhotoFlash::User",
                        "entityId": "annalisa"
                    },
                    "action": {
                        "actionType": "PhotoFlash::Action",
                        "actionId": "DeletePhoto"
                    },
                    "resource": {
                        "entityType": "PhotoFlash::Photo",
                        "entityId": "VacationPhoto94.jpg"
                    }
                },
                "decision": "DENY"
```

```
            }
        ]
    },
    "requestID": "a8a5caf3-78bd-4139-924c-7101a8339c3b",
    "eventID": "7d81232f-f3d1-4102-b9c9-15157c70487b",
    "readOnly": true,
    "resources": [
        {
    "accountId": "123456789012",
    "type": "AWS::VerifiedPermissions::PolicyStore",
    "ARN": "arn:aws:verifiedpermissions::123456789012:policy-store/
PSEXAMPLEabcdefg111111"
        }
    ],
    "eventType": "AwsApiCall",
    "managementEvent": false,
    "recipientAccountId": "123456789012",
    "eventCategory": "Data"
}
```

CreatePolicyStore

```
{
  "eventVersion": "1.08",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "EXAMPLE_PRINCIPAL_ID",
    "arn": "arn:aws:iam::123456789012:role/ExampleRole",
    "accountId": "123456789012",
    "accessKeyId": "AKIAIOSFODNN7EXAMPLE"
  },
  "eventTime": "2023-05-22T07:43:33Z",
  "eventSource": "verifiedpermissions.amazonaws.com",
  "eventName": "CreatePolicyStore",
  "awsRegion": "us-west-2",
  "sourceIPAddress": "203.0.113.0",
  "userAgent": "aws-sdk-rust/0.55.2 os/linux lang/rust/1.69.0",
  "requestParameters": {
    "clientToken": "a1b2c3d4-e5f6-a1b2-c3d4-TOKEN1111111",
    "validationSettings": {
      "mode": "OFF"
    }
  },
```

```
  "responseElements": {
    "policyStoreId": "PSEXAMPLEabcdefg111111",
    "arn": "arn:aws:verifiedpermissions::123456789012:policy-store/
PSEXAMPLEabcdefg111111",
    "createdDate": "2023-05-22T07:43:33.962794Z",
    "lastUpdatedDate": "2023-05-22T07:43:33.962794Z"
  },
  "requestID": "1dd9360e-e2dc-4554-ab65-b46d2cf45c29",
  "eventID": "b6edaeee-3584-4b4e-a48e-311de46d7532",
  "readOnly": false,
  "eventType": "AwsApiCall",
  "managementEvent": true,
  "recipientAccountId": "123456789012",
  "eventCategory": "Management"
}
```


ListPolicyStores


```
{
  "eventVersion": "1.08",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "EXAMPLE_PRINCIPAL_ID",
    "arn": "arn:aws:iam::123456789012:role/ExampleRole",
    "accountId": "123456789012",
    "accessKeyId": "AKIAIOSFODNN7EXAMPLE"
  },
  "eventTime": "2023-05-22T07:43:33Z",
  "eventSource": "verifiedpermissions.amazonaws.com",
  "eventName": "ListPolicyStores",
  "awsRegion": "us-west-2",
  "sourceIPAddress": "203.0.113.0",
  "userAgent": "aws-sdk-rust/0.55.2 os/linux lang/rust/1.69.0",
  "requestParameters": {
    "maxResults": 10
  },
  "responseElements": null,
  "requestID": "5ef238db-9f87-4f37-ab7b-6cf0ba5df891",
  "eventID": "b0430fb0-12c3-4cca-8d05-84c37f99c51f",
  "readOnly": true,
  "eventType": "AwsApiCall",
  "managementEvent": true,
  "recipientAccountId": "123456789012",
```

```
  "eventCategory": "Management"
}
```


DeletePolicyStore


```
{
  "eventVersion": "1.08",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "EXAMPLE_PRINCIPAL_ID",
    "arn": "arn:aws:iam::123456789012:role/ExampleRole",
    "accountId": "123456789012",
    "accessKeyId": "AKIAIOSFODNN7EXAMPLE"
  },
  "eventTime": "2023-05-22T07:43:32Z",
  "eventSource": "verifiedpermissions.amazonaws.com",
  "eventName": "DeletePolicyStore",
  "awsRegion": "us-west-2",
  "sourceIPAddress": "203.0.113.0",
  "userAgent": "aws-sdk-rust/0.55.2 os/linux lang/rust/1.69.0",
  "requestParameters": {
    "policyStoreId": "PSEXAMPLEabcdefg111111"
  },
  "responseElements": null,
  "requestID": "1368e8f9-130d-45a5-b96d-99097ca3077f",
  "eventID": "ac482022-b2f6-4069-879a-dd509123d8d7",
  "readOnly": false,
  "resources": [
    {
      "accountId": "123456789012",
      "type": "AWS::VerifiedPermissions::PolicyStore",
      "arn": "arn:aws:verifiedpermissions::123456789012:policy-store/
PSEXAMPLEabcdefg111111"
    }
  ],
  "eventType": "AwsApiCall",
  "managementEvent": true,
  "recipientAccountId": "123456789012",
  "eventCategory": "Management"
}
```

PutSchema


```
{
  "eventVersion": "1.08",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "EXAMPLE_PRINCIPAL_ID",
    "arn": "arn:aws:iam::123456789012:role/ExampleRole",
    "accountId": "123456789012",
    "accessKeyId": "AKIAIOSFODNN7EXAMPLE"
  },
  "eventTime": "2023-05-16T12:58:57Z",
  "eventSource": "verifiedpermissions.amazonaws.com",
  "eventName": "PutSchema",
  "awsRegion": "us-west-2",
  "sourceIPAddress": "203.0.113.0",
  "userAgent": "aws-sdk-rust/0.55.2 os/linux lang/rust/1.69.0",
  "requestParameters": {
    "policyStoreId": "PSEXAMPLEabcdefg111111"
  },
  "responseElements": {
    "lastUpdatedDate": "2023-05-16T12:58:57.513442Z",
    "namespaces": "[some_namespace]",
    "createdDate": "2023-05-16T12:58:57.513442Z",
    "policyStoreId": "PSEXAMPLEabcdefg111111",
  },
  "requestID": "631fbfa1-a959-4988-b9f8-f1a43ff5df0d",
  "eventID": "7cd0c677-733f-4602-bc03-248bae581fe5",
  "readOnly": false,
  "resources": [
    {
      "accountId": "123456789012",
      "type": "AWS::VerifiedPermissions::PolicyStore",
      "ARN": "arn:aws:verifiedpermissions::123456789012:policy-store/
PSEXAMPLEabcdefg111111"
    }
  ],
  "eventType": "AwsApiCall",
  "managementEvent": true,
  "recipientAccountId": "123456789012",
  "eventCategory": "Management"
}
```

GetSchema

```
{
  "eventVersion": "1.08",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "EXAMPLE_PRINCIPAL_ID",
    "arn": "arn:aws:iam::222222222222:role/ExampleRole",
    "accountId": "222222222222",
    "accessKeyId": "AKIAIOSFODNN7EXAMPLE"
  },
  "eventTime": "2023-05-25T01:12:07Z",
  "eventSource": "verifiedpermissions.amazonaws.com",
  "eventName": "GetSchema",
  "awsRegion": "us-west-2",
  "sourceIPAddress": "203.0.113.0",
  "userAgent": "aws-sdk-rust/0.55.2 os/linux lang/rust/1.69.0",
  "requestParameters": {
    "policyStoreId": "PSEXAMPLEabcdefg111111"
  },
  "responseElements": null,
  "requestID": "a1f4d4cd-6156-480a-a9b8-e85a71dcc7c2",
  "eventID": "0b3b8e3d-155c-46f3-a303-7e9e8b5f606b",
  "readOnly": true,
  "resources": [
    {
      "accountId": "222222222222",
      "type": "AWS::VerifiedPermissions::PolicyStore",
      "ARN": "arn:aws:verifiedpermissions::222222222222:policy-store/
PSEXAMPLEabcdefg111111"
    }
  ],
  "eventType": "AwsApiCall",
  "managementEvent": true,
  "recipientAccountId": "222222222222",
  "eventCategory": "Management"
}
```

CreatePolicyTemplate

```
{
  "eventVersion": "1.08",
  "userIdentity": {
```

```
    "type": "AssumedRole",
    "principalId": "EXAMPLE_PRINCIPAL_ID",
    "arn": "arn:aws:iam::123456789012:role/ExampleRole",
    "accountId": "123456789012",
    "accessKeyId": "AKIAIOSFODNN7EXAMPLE"
  },
  "eventTime": "2023-05-16T13:00:24Z",
  "eventSource": "verifiedpermissions.amazonaws.com",
  "eventName": "CreatePolicyTemplate",
  "awsRegion": "us-west-2",
  "sourceIPAddress": "203.0.113.0",
  "userAgent": "aws-sdk-rust/0.55.2 os/linux lang/rust/1.69.0",
  "requestParameters": {
    "policyStoreId": "PSEXAMPLEabcdefg111111"
  },
  "responseElements": {
    "lastUpdatedDate": "2023-05-16T13:00:23.444404Z",
    "createdDate": "2023-05-16T13:00:23.444404Z",
    "policyTemplateId": "PTEXAMPLEabcdefg111111",
    "policyStoreId": "PSEXAMPLEabcdefg111111",
  },
  "requestID": "73953bda-af5e-4854-afe2-7660b492a6d0",
  "eventID": "7425de77-ed84-4f91-a4b9-b669181cc57b",
  "readOnly": false,
  "resources": [
    {
      "accountId": "123456789012",
      "type": "AWS::VerifiedPermissions::PolicyStore",
      "arn": "arn:aws:verifiedpermissions::123456789012:policy-store/
PSEXAMPLEabcdefg111111"
    }
  ],
  "eventType": "AwsApiCall",
  "managementEvent": true,
  "recipientAccountId": "123456789012",
  "eventCategory": "Management"
}
```


DeletePolicyTemplate


```
{
  "eventVersion": "1.08",
  "userIdentity": {
```

```
    "type": "AssumedRole",
    "principalId": "EXAMPLE_PRINCIPAL_ID",
    "arn": "arn:aws:iam::222222222222:role/ExampleRole",
    "accountId": "222222222222",
    "accessKeyId": "AKIAIOSFODNN7EXAMPLE"
  },
  "eventTime": "2023-05-25T01:11:48Z",
  "eventSource": "verifiedpermissions.amazonaws.com",
  "eventName": "DeletePolicyTemplate",
  "awsRegion": "us-west-2",
  "sourceIPAddress": "203.0.113.0",
  "userAgent": "aws-sdk-rust/0.55.2 os/linux lang/rust/1.69.0",
  "requestParameters": {
    "policyStoreId": "PSEXAMPLEabcdefg111111",
    "policyTemplateId": "PTEXAMPLEabcdefg111111"
  },
  "responseElements": null,
  "requestID": "5ff0f22e-6bbd-4b85-a400-4fb74aa05dc6",
  "eventID": "c0e0c689-369e-4e95-a9cd-8de113d47ffa",
  "readOnly": false,
  "resources": [
    {
      "accountId": "222222222222",
      "type": "AWS::VerifiedPermissions::PolicyStore",
      "ARN": "arn:aws:verifiedpermissions::222222222222:policy-store/
PSEXAMPLEabcdefg111111"
    }
  ],
  "eventType": "AwsApiCall",
  "managementEvent": true,
  "recipientAccountId": "222222222222",
  "eventCategory": "Management"
}
```


CreatePolicy


```
{
  "eventVersion": "1.08",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "EXAMPLE_PRINCIPAL_ID",
    "arn": "arn:aws:iam::123456789012:role/ExampleRole",
    "accountId": "123456789012",
```

```
    "accessKeyId": "AKIAIOSFODNN7EXAMPLE"
  },
  "eventTime": "2023-05-22T07:42:30Z",
  "eventSource": "verifiedpermissions.amazonaws.com",
  "eventName": "CreatePolicy",
  "awsRegion": "us-west-2",
  "sourceIPAddress": "203.0.113.0",
  "userAgent": "aws-sdk-rust/0.55.2 os/linux lang/rust/1.69.0",
  "requestParameters": {
    "clientToken": "a1b2c3d4-e5f6-a1b2-c3d4-TOKEN1111111",
    "policyStoreId": "PSEXAMPLEabcdefg111111"
  },
  "responseElements": {
    "policyStoreId": "PSEXAMPLEabcdefg111111",
    "policyId": "SPEXAMPLEabcdefg111111",
    "policyType": "STATIC",
    "principal": {
        "entityType": "PhotoApp::Role",
        "entityId": "PhotoJudge"
    },
    "resource": {
        "entityType": "PhotoApp::Application",
        "entityId": "PhotoApp"
    },
    "lastUpdatedDate": "2023-05-22T07:42:30.70852Z",
    "createdDate": "2023-05-22T07:42:30.70852Z"
  },
  "requestID": "93ffa151-3841-4960-9af6-30a7f817ef93",
  "eventID": "30ab405f-3dff-43ff-8af9-f513829e8bde",
  "readOnly": false,
  "resources": [
    {
      "accountId": "123456789012",
      "type": "AWS::VerifiedPermissions::PolicyStore",
      "arn": "arn:aws:verifiedpermissions::123456789012:policy-store/
PSEXAMPLEabcdefg111111"
    }
  ],
  "eventType": "AwsApiCall",
  "managementEvent": true,
  "recipientAccountId": "123456789012",
  "eventCategory": "Management"
}
```

GetPolicy


```
{
  "eventVersion": "1.08",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "EXAMPLE_PRINCIPAL_ID",
    "arn": "arn:aws:iam::123456789012:role/ExampleRole",
    "accountId": "123456789012",
    "accessKeyId": "AKIAIOSFODNN7EXAMPLE"
  },
  "eventTime": "2023-05-22T07:43:29Z",
  "eventSource": "verifiedpermissions.amazonaws.com",
  "eventName": "GetPolicy",
  "awsRegion": "us-west-2",
  "sourceIPAddress": "203.0.113.0",
  "userAgent": "aws-sdk-rust/0.55.2 os/linux lang/rust/1.69.0",
  "requestParameters": {
    "policyStoreId": "PSEXAMPLEabcdefg111111",
    "policyId": "SPEXAMPLEabcdefg111111"
  },
  "responseElements": null,
  "requestID": "23022a9e-2f5c-4dac-b653-59e6987f2fac",
  "eventID": "9b4d5037-bafa-4d57-b197-f46af83fc684",
  "readOnly": true,
  "resources": [
    {
      "accountId": "123456789012",
      "type": "AWS::VerifiedPermissions::PolicyStore",
      "arn": "arn:aws:verifiedpermissions::123456789012:policy-store/
PSEXAMPLEabcdefg111111"
    }
  ],
  "eventType": "AwsApiCall",
  "managementEvent": true,
  "recipientAccountId": "123456789012",
  "eventCategory": "Management"
}
```


CreateIdentitySource


```
{
  "eventVersion": "1.08",
```

```
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "EXAMPLE_PRINCIPAL_ID",
    "arn": "arn:aws:iam::333333333333:role/ExampleRole",
    "accountId": "333333333333",
    "accessKeyId": "AKIAIOSFODNN7EXAMPLE"
  },
  "eventTime": "2023-05-19T01:27:44Z",
  "eventSource": "verifiedpermissions.amazonaws.com",
  "eventName": "CreateIdentitySource",
  "awsRegion": "us-west-2",
  "sourceIPAddress": "203.0.113.0",
  "userAgent": "aws-sdk-rust/0.55.2 os/linux lang/rust/1.69.0",
  "requestParameters": {
    "clientToken": "a1b2c3d4-e5f6-a1b2-c3d4-TOKEN1111111",
    "configuration": {
      "cognitoUserPoolConfiguration": {
        "userPoolArn": "arn:aws:cognito-idp:000011112222:us-east-1:userpool/us-
east-1_aaaaaaaaaa"
      }
    },
    "policyStoreId": "PSEXAMPLEabcdefg111111",
    "principalEntityType": "User"
  },
  "responseElements": {
    "createdDate": "2023-07-14T15:05:01.599534Z",
    "identitySourceId": "ISEXAMPLEabcdefg111111",
    "lastUpdatedDate": "2023-07-14T15:05:01.599534Z",
    "policyStoreId": "PSEXAMPLEabcdefg111111"
    },
  "requestID": "afcc1e67-d5a4-4a9b-a74c-cdc2f719391c",
  "eventID": "f13a41dc-4496-4517-aeb8-a389eb379860",
  "readOnly": false,
  "resources": [
    {
      "accountId": "333333333333",
      "type": "AWS::VerifiedPermissions::PolicyStore",
      "arn": "arn:aws:verifiedpermissions::333333333333:policy-store/
PSEXAMPLEabcdefg111111"
    }
  ],
  "eventType": "AwsApiCall",
  "managementEvent": true,
  "recipientAccountId": "333333333333",
```

```
  "eventCategory": "Management"
}
```


GetIdentitySource


```
{
  "eventVersion": "1.08",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "EXAMPLE_PRINCIPAL_ID",
    "arn": "arn:aws:iam::333333333333:role/ExampleRole",
    "accountId": "333333333333",
    "accessKeyId": "AKIAIOSFODNN7EXAMPLE"
  },
  "eventTime": "2023-05-24T19:55:31Z",
  "eventSource": "verifiedpermissions.amazonaws.com",
  "eventName": "GetIdentitySource",
  "awsRegion": "us-west-2",
  "sourceIPAddress": "203.0.113.0",
  "userAgent": "aws-sdk-rust/0.55.2 os/linux lang/rust/1.69.0",
  "requestParameters": {
    "identitySourceId": "ISEXAMPLEabcdefg111111",
    "policyStoreId": "PSEXAMPLEabcdefg111111"
  },
  "responseElements": null,
  "requestID": "7a6ecf79-c489-4516-bb57-9ded970279c9",
  "eventID": "fa158e6c-f705-4a15-a731-2cdb4bd9a427",
  "readOnly": true,
  "resources": [
    {
      "accountId": "333333333333",
      "type": "AWS::VerifiedPermissions::PolicyStore",
      "arn": "arn:aws:verifiedpermissions::333333333333:policy-store/
PSEXAMPLEabcdefg111111"
    }
  ],
  "eventType": "AwsApiCall",
  "managementEvent": true,
  "recipientAccountId": "333333333333",
  "eventCategory": "Management"
}
```

ListIdentitySources

```
{
  "eventVersion": "1.08",
  "userIdentity": {
    "type": "AssumedRole",
    "principalId": "EXAMPLE_PRINCIPAL_ID",
    "arn": "arn:aws:iam::333333333333:role/ExampleRole",
    "accountId": "333333333333",
    "accessKeyId": "AKIAIOSFODNN7EXAMPLE"
  },
  "eventTime": "2023-05-24T20:05:32Z",
  "eventSource": "verifiedpermissions.amazonaws.com",
  "eventName": "ListIdentitySources",
  "awsRegion": "us-west-2",
  "sourceIPAddress": "203.0.113.0",
  "userAgent": "aws-sdk-rust/0.55.2 os/linux lang/rust/1.69.0",
  "requestParameters": {
    "policyStoreId": "PSEXAMPLEabcdefg111111"
  },
  "responseElements": null,
  "requestID": "95d2a7bc-7e9a-4efe-918e-97e558aacaf7",
  "eventID": "d3dc53f6-1432-40c8-9d1d-b9eeb75c6193",
  "readOnly": true,
  "resources": [
    {
      "accountId": "333333333333",
      "type": "AWS::VerifiedPermissions::PolicyStore",
      "arn": "arn:aws:verifiedpermissions::333333333333:policy-store/
PSEXAMPLEabcdefg111111"
    }
  ],
  "eventType": "AwsApiCall",
  "managementEvent": true,
  "recipientAccountId": "333333333333",
  "eventCategory": "Management"
}
```

DeleteIdentitySource

```
{
  "eventVersion": "1.08",
  "userIdentity": {
```

```
    "type": "AssumedRole",
    "principalId": "EXAMPLE_PRINCIPAL_ID",
    "arn": "arn:aws:iam::333333333333:role/ExampleRole",
    "accountId": "333333333333",
    "accessKeyId": "AKIAIOSFODNN7EXAMPLE"
  },
  "eventTime": "2023-05-24T19:55:32Z",
  "eventSource": "verifiedpermissions.amazonaws.com",
  "eventName": "DeleteIdentitySource",
  "awsRegion": "us-west-2",
  "sourceIPAddress": "203.0.113.0",
  "userAgent": "aws-sdk-rust/0.55.2 os/linux lang/rust/1.69.0",
  "requestParameters": {
    "identitySourceId": "ISEXAMPLEabcdefg111111",
    "policyStoreId": "PSEXAMPLEabcdefg111111"
  },
  "responseElements": null,
  "requestID": "d554d964-0957-4834-a421-c417bd293086",
  "eventID": "fe4d867c-88ee-4e5d-8d30-2fbc208c9260",
  "readOnly": false,
  "resources": [
    {
      "accountId": "333333333333",
      "type": "AWS::VerifiedPermissions::PolicyStore",
      "arn": "arn:aws:verifiedpermissions::333333333333:policy-store/
PSEXAMPLEabcdefg111111"
    }
  ],
  "eventType": "AwsApiCall",
  "managementEvent": true,
  "recipientAccountId": "333333333333",
  "eventCategory": "Management"
}
```

Creating Amazon Verified Permissions resources with AWS CloudFormation

Amazon Verified Permissions is integrated with AWS CloudFormation, a service that helps you to model and set up your AWS resources so that you can spend less time creating and managing your resources and infrastructure. You create a template that describes all the AWS resources that you want (such as policy stores), and AWS CloudFormation provisions and configures those resources for you.

When you use AWS CloudFormation, you can reuse your template to set up your Verified Permissions resources consistently and repeatedly. Describe your resources once, and then provision the same resources over and over in multiple AWS accounts and Regions.

> **Important**
>
> Amazon Cognito Identity is not available in all of the same AWS Regions as Amazon Verified Permissions. If you receive an error from AWS CloudFormation regarding Amazon Cognito Identity, such as `Unrecognized resource types: AWS::Cognito::UserPool, AWS::Cognito::UserPoolClient`, we recommend that you create the Amazon Cognito user pool and client in the geographically closest AWS Region where Amazon Cognito Identity is available. Use this newly created user pool when creating the Verified Permissions identity source.

Verified Permissions and AWS CloudFormation templates

To provision and configure resources for Verified Permissions and related services, you must understand AWS CloudFormation templates. Templates are formatted text files in JSON or YAML. These templates describe the resources that you want to provision in your AWS CloudFormation stacks. If you're unfamiliar with JSON or YAML, you can use AWS CloudFormation Designer to help you get started with AWS CloudFormation templates. For more information, see What is AWS CloudFormation Designer? in the *AWS CloudFormation User Guide*.

Verified Permissions supports creating identity sources, policies, policy stores, and policy templates in AWS CloudFormation. For more information, including examples of JSON and YAML templates for Verified Permissions resources, see the Amazon Verified Permissions resource type reference in the *AWS CloudFormation User Guide*.

Learn more about AWS CloudFormation

To learn more about AWS CloudFormation, see the following resources:

- AWS CloudFormation
- AWS CloudFormation User Guide
- AWS CloudFormation API Reference
- AWS CloudFormation Command Line Interface User Guide

Access Amazon Verified Permissions using an interface endpoint (AWS PrivateLink)

You can use AWS PrivateLink to create a private connection between your VPC and Amazon Verified Permissions. You can access Verified Permissions as if it were in your VPC, without the use of an internet gateway, NAT device, VPN connection, or AWS Direct Connect connection. Instances in your VPC don't need public IP addresses to access Verified Permissions.

You establish this private connection by creating an *interface endpoint*, powered by AWS PrivateLink. We create an endpoint network interface in each subnet that you enable for the interface endpoint. These are requester-managed network interfaces that serve as the entry point for traffic destined for Verified Permissions.

For more information, see Access AWS services through AWS PrivateLink in the *AWS PrivateLink Guide*.

Considerations for Verified Permissions

Before you set up an interface endpoint for Verified Permissions, review Considerations in the *AWS PrivateLink Guide*.

Verified Permissions supports making calls to all of its API actions through the interface endpoint.

VPC endpoint policies are not supported for Verified Permissions. By default, full access to Verified Permissions is allowed through the interface endpoint. Alternatively, you can associate a security group with the endpoint network interfaces to control traffic to Verified Permissions through the interface endpoint.

Create an interface endpoint for Verified Permissions

You can create an interface endpoint for Verified Permissions using either the Amazon VPC console or the AWS Command Line Interface (AWS CLI). For more information, see Create an interface endpoint in the *AWS PrivateLink Guide*.

Create an interface endpoint for Verified Permissions using the following service name:

```
com.amazonaws.region.verifiedpermissions
```

If you enable private DNS for the interface endpoint, you can make API requests to Verified Permissions using its default Regional DNS name. For example, `verifiedpermissions.us-east-1.amazonaws.com`.

Quotas for Amazon Verified Permissions

Your AWS account has default quotas, formerly referred to as limits, for each AWS service. Unless otherwise noted, each quota is Region-specific. You can request increases for some quotas, and other quotas cannot be increased.

To view the quotas for Verified Permissions, open the Service Quotas console. In the navigation pane, choose **AWS services** and select **Verified Permissions**.

To request a quota increase, see Requesting a Quota Increase in the *Service Quotas User Guide*. If the quota is not yet available in Service Quotas, use the limit increase form.

Your AWS account has the following quotas related to Verified Permissions.

Topics

Quotas for resources

Name	Default	Adjustable	Description
Policy stores per Region per account	Each supported Region: 1,000	Yes	The maximum number of policy stores.
Policy templates per policy store	Each supported Region: 40	Yes	The maximum number of policy templates in a policy store.
Identity sources per policy store	1	No	The maximum number of identity sources that you can define for a policy store.

Name	Default	Adjustable	Description
Authorization request size[1]	1 MB	No	The maximum size of an authorization request.
Policy size	10,000 bytes	No	The maximum size of an individual policy.
Schema size	100,000 bytes	No	The maximum size of the schema of a policy store.
Policy size per resource	200,000 bytes[2]	No	The maximum size of all policies that reference a specific resource.

[1] The quota for an authorization request is the same for both IsAuthorized and IsAuthorizedWithToken.

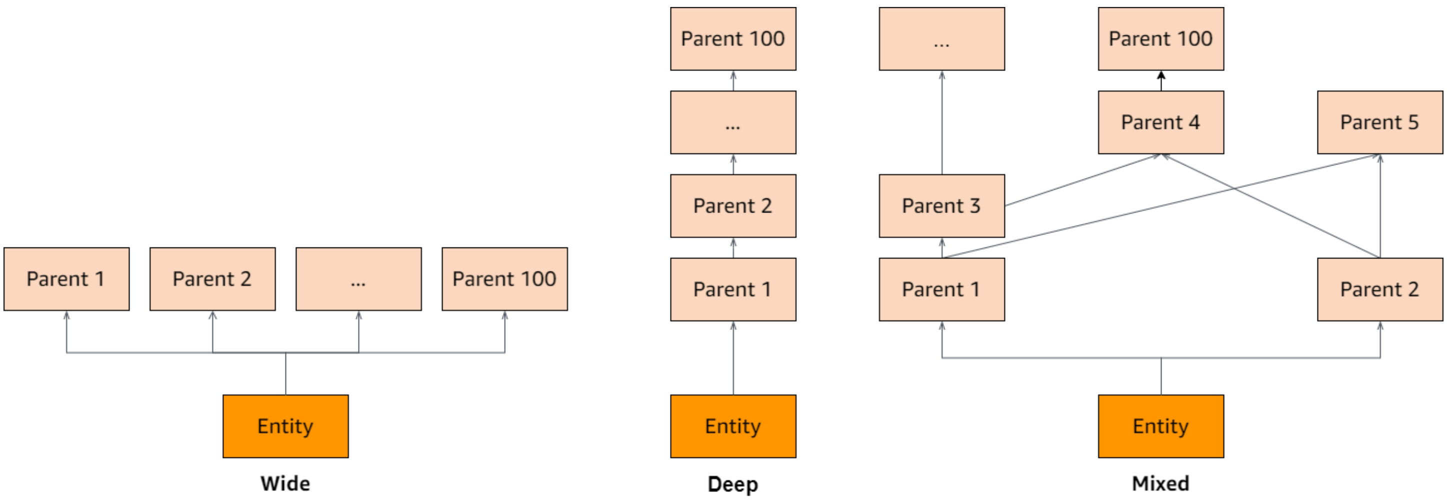
[2] The total size of all policies pertaining to a single resource can't exceed 200,000 bytes. For template-linked policies, the size of the policy template is counted only once, plus the size of each set of parameters used to instantiate each template-linked policy.

Quotas for hierarchies

Name	Default	Adjustable	Description
Transitive parents per principal	100	No	The maximum number of transitive parents for each principal.
Transitive parents per action	100	No	The maximum number of transitive parents for each action.

Name	Default	Adjustable	Description
Transitive parents per resource	100	No	The maximum number of transitive parents for each resource.

The diagram below illustrates how transitive parents can be defined for an entity (principal, action, or resource).



Quotas for operations per second

Verified Permissions throttles requests to service endpoints in an AWS Region when application requests exceed the quota for an API operation. Verified Permissions might return an exception when you exceed the quota in requests per second, or you attempt simultaneous write operations. You can view your current RPS quotas in Service Quotas. To prevent applications from exceeding the quota for an operation, you must optimize them for retries and exponential backoff. For more information, see Retry with backoff pattern and Managing and monitoring API throttling in your workloads.

Name	Default	Adjus e	Description
BatchIsAuthorized requests per second per Region per account	Each supported Region: 30	Yes	The maximum number of BatchIsAuthorized requests per second.
BatchIsAuthorizedWithToken requests per second per Region per account	Each supported Region: 30	Yes	maximum number of BatchIsAuthorizedW ithToken requests per second.
CreatePolicy requests per second per Region per account	Each supported Region: 10	Yes	The maximum number of CreatePolicy requests per second.
CreatePolicyStore requests per second per Region per account	Each supported Region: 1	No	The maximum number of CreatePolicyStore requests per second.
CreatePolicyTemplate requests per second per Region per account	Each supported Region: 10	Yes	The maximum number of CreatePolicyTemplate requests per second.
DeletePolicy requests per second per Region per account	Each supported Region: 10	Yes	The maximum number of DeletePolicy requests per second.
DeletePolicyStore requests per second per Region per account	Each supported Region: 1	No	The maximum number of DeletePolicyStore requests per second.
DeletePolicyTemplate requests per second per Region per account	Each supported Region: 10	Yes	The maximum number of DeletePolicyTemplate requests per second.

Name	Default	Adjustable	Description
GetPolicy requests per second per Region per account	Each supported Region: 10	Yes	The maximum number of GetPolicy requests per second.
GetPolicyTemplate requests per second per Region per account	Each supported Region: 10	Yes	The maximum number of GetPolicyTemplate requests per second.
GetSchema requests per second per Region per account	Each supported Region: 10	Yes	The maximum number of GetSchema requests per second.
IsAuthorized requests per second per Region per account	Each supported Region: 200	Yes	The maximum number of IsAuthorized requests per second.
IsAuthorizedWithToken requests per second per Region per account	Each supported Region: 200	Yes	The maximum number of IsAuthorizedWithToken requests per second.
ListPolicies requests per second per Region per account	Each supported Region: 10	Yes	The maximum number of ListPolicies requests per second.
ListPolicyStores requests per second per Region per account	Each supported Region: 10	Yes	The maximum number of ListPolicyStores requests per second.
ListPolicyTemplates requests per second per Region per account	Each supported Region: 10	Yes	The maximum number of ListPolicyTemplates requests per second.
PutSchema requests per second per Region per account	Each supported Region: 10	Yes	The maximum number of PutSchema requests per second.

Name	Default	Adjustable	Description
UpdatePolicy requests per second per Region per account	Each supported Region: 10	Yes	The maximum number of UpdatePolicy requests per second.
UpdatePolicyTemplate requests per second per Region per account	Each supported Region: 10	Yes	The maximum number of UpdatePolicyTemplate requests per second.

Document history for the Amazon Verified Permissions User Guide

The following table describes the documentation releases for Verified Permissions.

Change	Description	Date
Batch authorization with identity source tokens	You can now authorize users from a Amazon Cognito user pool in a single `BatchIsAuthorizedWithToken` API request.	April 5, 2024
Creating a policy store with API Gateway	You can now create a policy store from an existing API and Amazon Cognito user pool.	April 1, 2024
Context concepts and example	Added information about context in authorization requests with Verified Permissions.	February 1, 2024
Authorization concepts and example	Added information about authorization requests with Verified Permissions.	February 1, 2024
AWS CloudFormation integration	Verified Permissions supports creating identity sources, policies, policy stores, and policy templates in AWS CloudFormation.	June 30, 2023
Initial release	Initial release of the Amazon Verified Permissions User Guide	June 13, 2023