AWS Verified Access

AWS Verified Access (AVA) is an AWS managed service that enables enterprises to provide VPN-less secure network access to corporate applications, improve their security, and simplify their end-users' experience. IT administrators can use AVA to build a unified set of *fine-grained policies* that define their users' ability to access each of their applications. AVA verifies each access request in real-time and only connects users to the applications they are allowed to access. This eliminates broad access to corporate applications. For example, using AVA, administrators can define a policy to let only employees on the finance team, who are also using a device with malware protection enabled, access core financial applications. AVA utilizes a powerful policy language, Cedar , to enable enterprises to quickly create policy documents that secure access to their applications from any location and any network.

AVA policies using Cedar Policy Language

AVA uses Cedar policy language for the creation of policy documents that govern access to a protected resource. A basic policy document is laid out as follows:

```
permit(
principal,action,resource)
when{
context.<idp-or-edm-reference>.<attribute1>
```

In general, authorization policies in Cedar govern which *principals* (*e.g.* developers) can perform which *actions* (*e.g.* directly write) on a given *resource* (*e.g.* a company's production customer database) under certain *conditions* (*e.g.* during certain scheduled maintenance hours, with a specific security posture, etc).

element	**example**
principal	developers
action	write
resource	production customer database

conditions	during scheduled maintenance hours

When using Cedar with AVA, only conditions are utilized when creating policy documents, as AVA does not work well with the functionality of Cedar's principals, actions, or resources. These entities each must be predefined in Cedar (all "principals" are "roles", or all "principals" are "users") and this would be too limiting given the breadth of use cases that AVA aims to support.

Instead, using conditions alone, AVA users have the flexibility to add any number of rules regarding what is acceptable or unacceptable incoming trust data. "Trust data" here refers to all the attributes of users/devices AVA obtains via third party integrated identity providers/electronic device managers (IdPs/EDMs), such as their email address, username, role, location, software version status, or any other standard or custom attribute that is sent to AVA from the third party IdP/EDM in order to successfully make an access decision to allow or deny according to the policy.

Using Conditions

Consider the following Cedar policy schema, where the conditions are expressed in the form `when{` followed by a new line with `context.<idp-name>.<attribute>`:

```
permit(
principal,action,resource)
when{
context.<idp-name>.<attribute1> &&
context.<idp-name>.<attribute2>
}
```

and as implied by the above schema, more than one conditions clause can be added in a policy document using the `&&` operator. This means that by using Cedar, AVA customers have sufficient expressive power to create policy documents that are not only custom and fine-grained, but extensible. For example, by using the following policy document, someone is only allowed access if they have an email and are an admin:

```
permit(
principal,action,resource)
when{
context.idp_okta.email.size() > 0 &&
```

```
context.idp_okta.roles.contains(“admin”)
}
```

Otherwise, with Cedar, the default behavior is to deny any request that does not satisfy the policy document for the AVA group and AVA resource involved.

Drafting Cedar Policy Documents for AWS Verified Access

All Cedar policy documents with AVA start with either a “permit” or “forbid” and either a “when” or an “unless”. This empowers the user to create access policies that enforce semantic schemas taking the following forms:

Permit access when conditions meet C.

Forbid access unless conditions meet C

So in the example above, with email.size and role.contains as the attributes of context being used, in order to establish trust and gain access:

Permit access only when the size of the email address letter count is greater than 0 character and a role is “admin”.

which means the same as:

Forbid access unless the size of the email address letter count is greater than 0 characters and a role is “admin”.

as both control access equivalently. The choices of which action and operator combination to use (permits/when, permits/unless, forbid/when, and forbids/unless) is at the discretion of the policy author.

Creating Policy Documents in Neutrino

Dot Notation for Accessing Trust Data

In the preceding Cedar policy document examples, "dot" notation is used in the policy document snippets when accessing trust data from the third party IdP/EDM. Unfortunately, Cedar only supports "dot" notation on Cedar strings and not on json

String (AVA's trust input is in json). There will be a feature to allow this in a future release, but in this version "[]" notation is used to access keys (similar to Maps in Java):

```
forbid(
principal,action,resource)
unless{
context.idp_okta[email].size() > 0 &&
context.idp_okta[roles].contains(“admin”) &&
```

Available Trust Data

The attributes (such as `email` or `roles` in the last example here) available to evaluate incoming trust data according to the conditions of a given policy are either supplied by default by the trust provider (in this example the IdP Okta) or defined custom via configuration with the IdP or EDM. For example, consider that in Okta:

- when creating an Okta user profile there are 31 default base attributes for all users in an org
- the only base attributes that can be modified or removed are First Name and Last Name.
- Custom attributes can be made for additional user settings,
- these reserved keywords cannot be changed or removed: "id", "profile", "status", "transitioningtostatus", "created", "activated", "statuschanged", "lastlogin", "lastupdated", "passwordchanged", "type", "realm", "password", "credentials", "_links", "_embedded", "class", "classloader".

In sum, regardless of the IdP or EDM, when writing a Cedar policy document, it is essential to ensure that the attribute being relied on to create that policy document is an available attribute sent to AVA via the trust provider (in this case IdP Okta), either by default or created custom.

Built-in Operators

As mentioned above, when creating the context of the policy using various conditions, we can use the `&&` operator to add additional conditional clauses within the scope of a `when` or `unless` condition. There are also many other built in operators that can be used to add additional expressive power to policy conditions, both operators between conditions (eg &&) as well as operators within a conditional proposition (eg, > , .contains , or in etc). Here are all the built-in operators for reference:

Symbol	Types and overloads	Description
!	bool → bool	logical not
-	int → int	unary negation
==	any → any	equality. Works on arguments of any type, even if the types don't match. (Values of different types are never equal to each other.)
!=	any → any	inequality; the exact inverse of equality (see above)
<	(int, int) → bool	int less-than
<=	(int, int) → bool	int less-than-or-equal-to
>	(int, int) → bool	int greater-than
>=	(int, int) → bool	int greater-than-or-equal-to
in	(entity, entity) → bool	Hierarchy membership (reflexive: A in A is always true)
	(entity, list(entity)) → bool	Hierarchy membership: A in [B, C, ...] is true iff (A in B) || (A in C) || ... error if the list contains a non-entity
&&	(bool, bool) → bool	Logical *and* (short-circuiting)
||	(bool, bool) → bool	Logical *or* (short-circuiting)
.size()	string → int	string length
	list → int	list size (number of elements)
	map → int	map size (number of keys)
.contains()	(string, string) → bool	Substring (is B a substring of A)
	(list, any) → bool	List membership (is B an element of A)
	(map, string) → bool	Map membership (is B a key in A)
.containsAll()	(list, list) → bool	Tests if list A contains all of the elements in list B
.containsAny()	(list, list) → bool	Tests if list A contains any of the elements in list B
.startsWith()	(string, string) → bool	Tests if string A starts with string B
.endsWith()	(string, string) → bool	Tests if string A ends with string B