Marcel Schoffelmeer Follow

May 8, 2019 · 10 min read · Listen

Implementing Role Based Security in a Web App

So you built a web application for a growing set of customers. At first, securing the app was as simple as separating users from administrators and applying checks on both the UI and backend to make sure only administrators can do administrative stuff. But over time, the all or reduced access model stopped being sufficient. When this happens, you need to introduce a more granular level of permissions. For example, some groups of users just require (or are allowed) read access, while others may need edit access.

Introducing a New Level of Access: The Quick Way

Let's assume that our starting point is a web app that only has two levels of access: administrator with full permissions to all features in the app and user access, which is restricted to non-administrator features. A new requirement comes up stating that we need to allow a group of users only read or viewing access (e.g. this group will not be able to create, modify or delete things). One way is to just hardcode the concept of read access into the application and add a flag to the users to mark them as having such access. Back-end checks can be extended with additional branches:

```
if user.is_admin:
    # allow full access
elif user.has_read_access:
    # allow read access, deny create/update/delete
else:
    # allow user access
```

On the upside, this approach can be quickly implemented on the back-end, and depending on the complexity of the UI, the front-end as well. The downsides are that every time a new level of access is needed, it requires code changes, testing, deployment etc. In addition, the code enforcing permissions gets more complicated (tried it, got the t-shirt). Finally, read-access is still quite a broad type of permission.

Rethinking Access Management

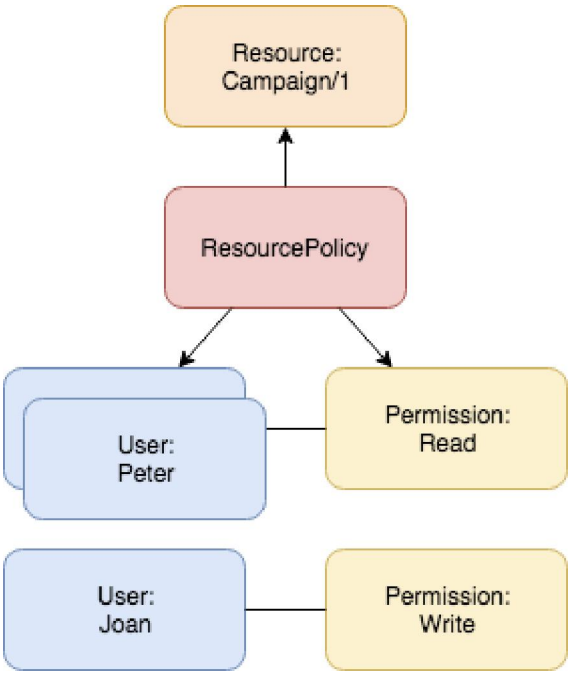
Years ago, we set out to come up with a new way to specify and control fine grained access for both Bluecore employees and our customers, who also require different levels of access to our web app. To be able to quickly adjust application access, the configuration of these permissions should occur without having to make code changes. To help design an approach, we made use of ideas stemming from the security industry called Identity and Access Management (IAM).

IAM

Identity and Access Management is a concept that deals with authentication and authorization, respectively. Authentication and authorization are generally confused but a simple explanation is that authentication deals with identifying a user while authorization makes sure these users have permission to do specific things. In this blog post we'll focus only on authorization. To do that, we'll first define the terms used.

Terminology

- User: An authenticated person or service logged in to the web application
- Resource: These are the things that we want to secure. For Bluecore, these include campaigns (scheduled channel-specific sends) and audiences (people we want to address the campaigns).
- Permission: A specific authorization to do something. Example: "edit", "view" etc.
- Resource Policy: Define who has what permissions for what resources. Example: "Peggy" and "Peter" have "read" permission for "campaigns" or a specific campaign "1", while "Joan" has "write permission:



Implementation

Storage

The primary items to store are the resource policies as these will be the “source of truth” to decide whether we should allow certain actions. We will store a separate resource policy for each resource that we want to secure. For example, for the resource campaign/1 shown above, a JSON representation could be:

```
{
  "read": [
    "Peggy",
    "Peter"
  ],
  "write": [
    "Joan"
  ]
}
```

Another alternative could be to lay out these entries in a relational database, but in our case document storage of JSON blobs worked just fine.

Assuming that the resources are already stored using some sort of a unique key, and since we’ll have one policy for each resource, policies can use the same key in combination with resource type for identification (e.g. “campaign/1”).

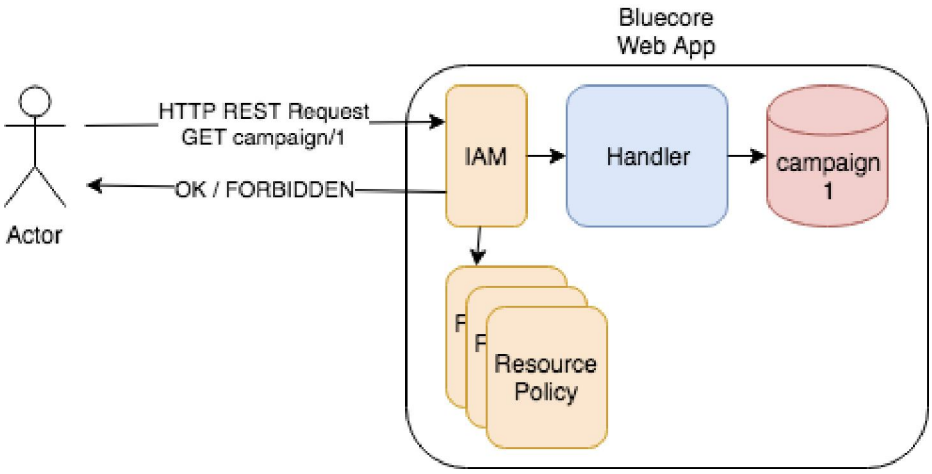
This looks a lot like Access Control Lists (ACLs)!

Enforcement

Having defined and stored our authorization configuration, we then need to make sure the policies get enforced. For web based applications like ours, the REST APIs are the entrance for the UI to our application and a good place to do this. For each REST request we'll need to associate the request with a resource and permission:

Request Method	URL	Permission	Resource
GET	/campaigns/1	read	campaign/1
PUT	/campaigns/5	write	campaign/5

The backend then proceeds to load the resource policy associated with the resource and checks whether the user is listed for the given permission and responds with either a success response (HTTP status code 200 and content of the campaign) or rejection (401):

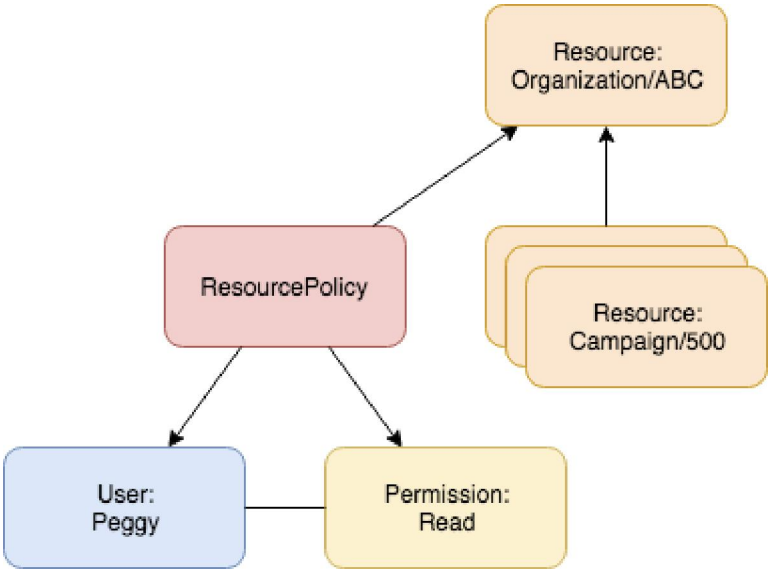


The above is a bit of a hello world version of access control. In the next sections, we'll describe changes to the above model for more realistic scenarios.

Many Resources

Having a resource policy for each resource becomes problematic in terms of maintenance and enforcement execution speed as the number of resources increases. For example, having hundreds of campaigns would require us to create and maintain an equivalent number of resource policies. If we wanted to give one user read access to all these campaigns, we'd have to add an entry to all campaign resource policies. It would be more beneficial to only have to state (or remove) this access once.

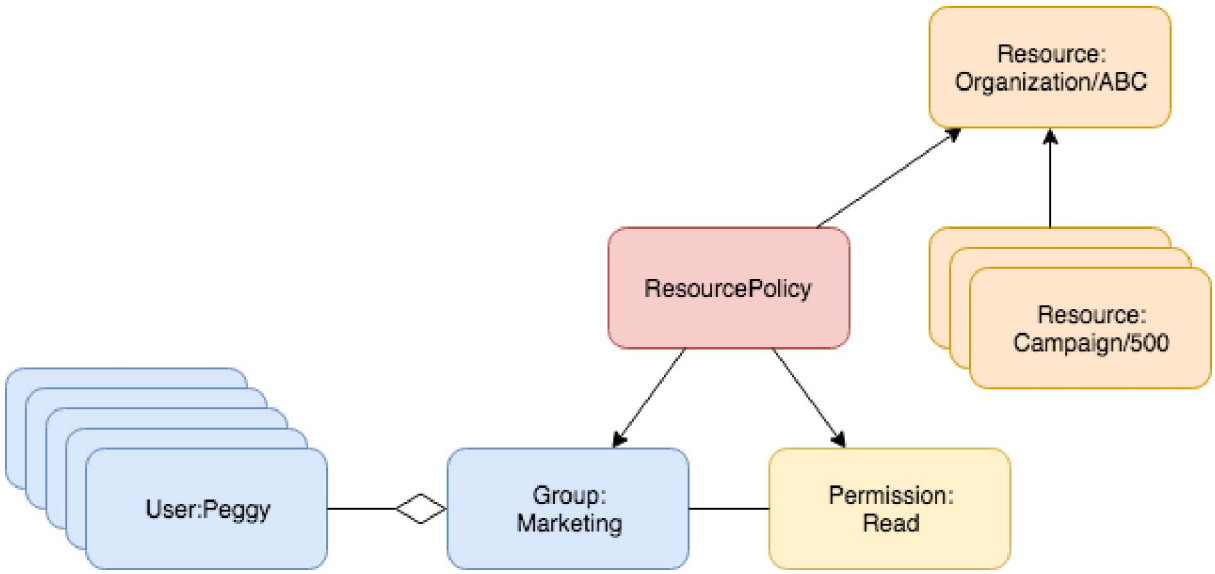
A way to support the above is to introduce the concept of **resource inheritance**, where permissions granted to a person get inherited from parent to child resources. We'll introduce a company resource, where permissions specified in the resource policy are passed down to the child resources such as campaigns:



If in the above example Peggy tried to read campaign 500, read access will be granted as read access is granted for organization ABC and the campaign belongs to this organization.

Many Users

Just like the problems caused by having many resources, having many users creates a similar maintenance/execution problem that can be resolved by introducing user groups and granting access at this level. If we want to grant all people in marketing group read access, it will look like this:



Now we reduced the organization resource policy to one entry:

```
{
  “Read”: [
    “Group:Marketing”
  ]
}
```

Note how this introduces a small change by adding the type of grantee (users vs groups).

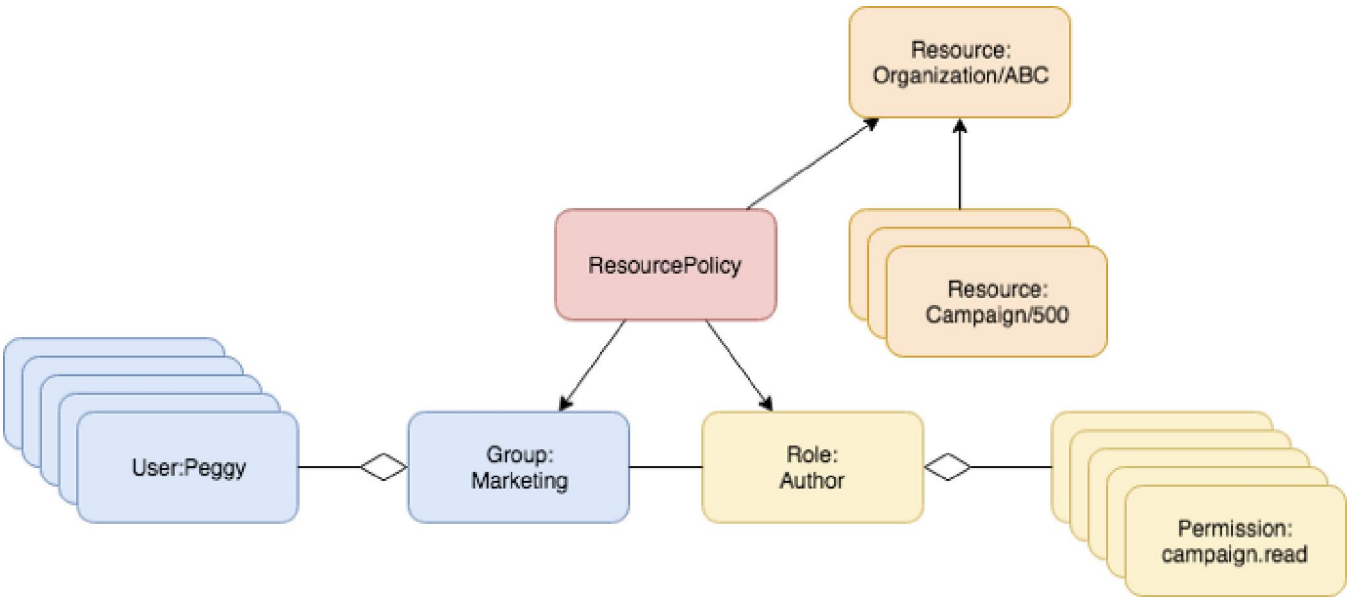
In addition to reducing the amount of information stored in the policy, the usage of groups also allows us to make use of existing systems that store and control group information, such as Google Groups (if your company administers its users in Google). External identity providers have similar functionality and corresponding APIs to obtain group membership information. Finally, external group/user management usage prevents the need for us to store and manage this information inside our application and makes life easier for IT administrators (e.g. on-boarding a new marketing person is a matter of adding the person to the group in the directory).

Many Permissions

The last optimization, and the most visible part to application users, is to combine permissions into roles. As the number of permissions in the application grows, granting access for each individual permission becomes cumbersome and error prone, so here we’ll introduce the final construct: roles. A role is the easiest part of our model to understand as it, from an implementation perspective, can be described as a collection of permissions and, from a usage perspective, a reflection of the responsibility of the grantee.

Role	Permissions
Viewer	campaign.get audience.get
Author	campaign.get campaign.update campaign.delete campaign.publish audience.get audience.update

If we then want to give the marketing team full permissions for campaign and audience resources, we can grant the marketing team the author role at the organization level like we did earlier:



Wait, did you just add the resource type to the permission?

Yes! With supporting resource inheritance we need to make permissions more specific; otherwise it would not be possible to limit permissions to the inheriting resources. If we inherited just “read” permissions from the organization resource, it would be mean the grantee would have read access to both campaign and audience resources. We pay the price of muddying the distinction between resources and permissions for the benefit of being able to grant permissions at a higher level than individual resources. Google IAM for example, takes the additional step to include the service name in the permission, where permission is comprised of the service name, resource and verb.

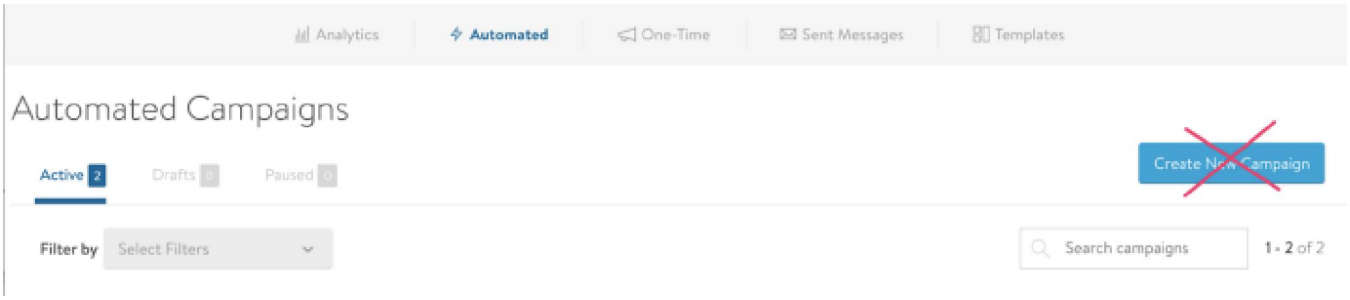
Do we not need resources then?

Not so fast. A permission like “campaign.read” applies to all resources if granted at the organization level. However, we need the concept of a single resource to allow granting permissions to only one. For example, a test campaign may just be shared with a limited list of people.

Making the UI permission aware

From a security perspective, enforcing permissions on REST/RPC endpoints would be sufficient to implement. However, keeping the UI unchanged regardless of permissions would not be a very good user experience. We would show options on pages that result in red HTTP 401 status error messages after selection. Instead we only want to show

functionality that is permitted for the given user. In the screenshot below, the “Create New Campaign” button will be hidden for users that are granted the viewer role:



To accomplish this, the UI asks the backend to get a list of permissions for the current user. The backend will obtain the roles associated with the user, load the permissions and return the set back. The UI will apply hiding rules using these permissions. Note that for this purpose, the UI does not need to know what roles the user has, just the permissions will do. The roles are just there to simplify the administration of permissions.

Note that sending a list of permissions to the UI does not create a security issue, as we are just adjusting the UI based on what the user is authorized to access. In the end, it is the backend’s responsibility to make sure policies are not violated.

Applying role based access to the entire application

Depending on the complexity of the application in terms of features and REST handlers, this can be a small or large effort. In Bluecore’s case, our product manager helped us out by going through every feature/page/action and creating a spreadsheet assigning permissions to a set of roles for each item. The spreadsheet was then crowdsourced across the different back- and front-end development teams to implement.

Testing and rollout

To switch from the old style (admin — non-admin access) to the role based access control, a carefully planned testing and rollout process is needed. A possible order of activities:

Implement role based access control that mimics the original access

This way you limit the changes to just the underlying security mechanism while nothing changes for the end user. In addition, we can run the new permission checks in the background, compare the results with the existing permission checks and validate the new code before switching to it.

Switch on role based access checks

Turn on the new code and monitor for errors (checking the REST access logs for 401s is a good start).

Introduce new roles

To make sure new roles work as expected, each development team responsible for the roles that apply to them can test/verify the behavior using test users granted these roles.

Remove legacy code

The engineering reward of removing lots of admin flags and checks from both the UI and backend code.

Benefits & concerns

Implementing role based security can be a large development effort with a significant amount of risk (denying access to users or opening up a security hole). In some cases, it can be a requirement for certain customers or prospects. Large corporations, especially, want to make sure users only have access to fulfill their job, known in security circles as "Principle of least privilege." In other cases, the requirement can also be driven from internal growth, as in Bluecore's case. Regardless, moving to a system as described here has several benefits:

- Changing permissions can be reduced to changing group membership or role assignments. In other words, no code changes/deployment/engineering involvement needed.
- Determining who has access to what becomes easier as the permissions are stored in resource policies only, while earlier one had to either scour through GitHub or test access using different user types. Security audits become simpler as a result.
- Allowing customers to share access to 3rd parties (for example with outsourced work) by using roles with reduced access.
- Introducing a source of new features: for example the ability to share specific resources with others through specific policies. Another might be to share access across geographical regions for large companies using resource inheritance. Allowing customers to define their own custom roles!

But there is a cost

Role based security is not something that is implemented and forgotten about, however. UI facing development teams need to now be aware of assigning the right permissions to UI features and REST endpoints. Product managers need to think about access permissions for existing and/or new roles based on customer needs. To make sure development teams get out of the admin/non-admin mindset, development project checklists should have items regarding role based security and testing needs to include the various roles.