Building Blocks

The Chef Server permission system is made of of the following:
Objects, Actors, Groups, Containers, ACLs, and ACEs.

Legend:

Object: An entity that can be acted upon
Actor: An entity that can make actions
Group: Collections of actors or other groups

ACL: Access Control List. Stores ACES for all CRUDG permissions
ACE: Access Control Entity. Lists of actors and groups who have the given permission and can therefore take the given action.
CRUDG: Create, Read, Update, Delete, Grant

The Objects and Actors in the system are:

Objects:

Orgs
Nodes
Cookbooks
Containers
Sandboxes
Clients
Users
Groups
Data Bags
Data Bag Items
Roles
Environments
Policies
Policy Groups

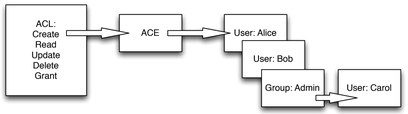
Actors:

Users
Clients

All Actors are also Objects (meaning the actors can be acted upon and have their own ACLs).

ACLs, ACEs, and Groups

Each Object contains an ACL (Access Control List).
This ACL contains an entry for each of the CRUDG permissions (Create, Read, Update, Delete, Grant).
Each permission then contains a list of ACEs (Access Control Entities).
These ACEs are lists of Actors and Groups.



A Group is a entity that contains lists of Actors and other Groups.
It is a way to link Actors in the system that should share the same permissions on an Object. An example is the Admin group.

Groups can contain other Groups. When resolving permissions, the system just walks down the chain until it reaches the end and finds the Actors contained in the Group.

Default Groups:

admins
billing-admins
clients
users

There are four default groups created for any Chef server.

Note that while billing-admins exists as a group in every Chef server, it only has significance and is used in Hosted Chef. In all other cases it can be ignored.

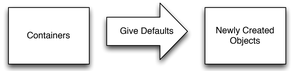
As the Chef server changes over time it is possible the default groups will also change, although this tends to happen infrequently.

Containers

Containers are special objects that contain the default permissions used in the system. They can be thought of as the prototypes for all objects in the system. Any new object that is created inherits the permissions from its container. So for example, a new user inherits its default permissions from the user's container.

Modifying a container's permissions changes the default permissions all new objects of that container type. It does not change the permissions of any object that already exists of that type.

It should be noted that the permissions on a container determine if an actor can list or create objects of that container's type. For example, a User would need the read permission on the roles container to list roles and the create permission on the roles container to create a role.



Default Containers:

clients
containers
cookbooks
data
environments
groups
nodes
roles
sandboxes
policies
policy_groups
cookbook_artifacts

There are many default containers in a Chef server. These tend to change as new functionality is added to the Chef server. For instance, the policies and policy_groups containers are relatively new additions to support the policy file functionality.

Global Objects

There are two special types of objects in the system: orgs and users. They are special in that they are global. Everything else in the system is scoped to an org.

What this means is that when an org is created, any new object created under that org will only be visible under that org, with the exception of users.

Because users are global, this is why users have to be invited to and associated with orgs, since users are not scoped to an org.

Global Objects

Orgs
Users

Org Creation

When a user first creates an org, the org is created from the org container and some initial setup is done to ensure the default groups and containers are in place.
We then rely on Chef Manage or the chef-server-ctl org creation commands to ensure the user who created the org is appropriately associated with the org and added to the org's admin group.

Chef server versions using ACLs

The permissions system talked about in this doc pertains to Chef server 12 and later.

This same permissions system also applies to the older Enterprise Chef server. It does not apply to any version of the old open source Chef server, which used a much simplified permission system without ACLs.

As of this writing, the code that implements this permission system in Chef server 12 is entirely in Erlang. Previous version of the Enterprise Chef server used a version of the ACL system written in Ruby.

What kind of permissions system does the Chef server implement?

The Chef Server permission system is a DAC (Discretionary Access Control) system implemented using ACLs (Access Control Lists). This is in contrast to a MAC (Mandatory Access Control) system.

A DAC system is one where the actors of the system can also grant or pass on permissions to other actors or objects of the system. In a MAC all permissions are centrally controlled. This answer is simplified and of course these types of systems blur in actual use. If you want further information, the DAC and MAC pages on Wikipedia are suggested reading.

We often call the Chef Server permission system a RBAC (Role Based Access Control) system, but this isn't technically true. In an RBAC system permissions are defined based on specific operations (roles) within the system. The classic example is the permissions a bank teller has as compared to the permissions that a bank manager has. By contrast, in an ACL system, the objects themselves are assigned the permissions (so to strain the analogy, the bank vault would store if the bank teller and/or manager could open it, vs. the teller or manager having the permission to open the bank vault).

An ACL system that has groups can mimic RBAC, so it is therefore possible for the Chef Server to implement a full RBAC system. At the moment however it does not implement the permissions in such a way as to mimic true RBAC. Our use of the term RBAC for describing the permissions system is inaccurate.

USAGS

USAGS (User Specific Association Groups) are a concept that is mentioned from time to time with the Chef server permission system. The idea behind a USAG is that it is a group that contains only a single user. So if we had a user Bob, there would exist a group, let's say it's called Bob's Group, that contains only Bob.

Why would USAGS be useful? To answer that, consider how the system currently works.
Users are directly added to the ACLs of objects that they have permission to act upon.
If a user then needs to be denied access to an object they are removed from the appropriate ACL.
Consider though: what if a user needs to be denied access to an entire org and all objects under it?

It's tempting to say just delete the user, but remember that a user is a global object. Users might exist in more than one org, so the user can't simply be deleted.

Because of the way the ACL system works, by building up permissions, the only way to deny a user access to all objects under an org is to find and remove the user from the ACLs of all the objects under the org.

USAGS are a potential answer to this problem. The user would be assigned to their specifc group, (the USAG) and then when the user needed permissions on an object, instead of adding the user directly, the USAG would be added to the ACLs of the object instead.

This would have the effect of giving the user the permissions needed, but in the event a user needed to be denied all permissions on an org a simple solution now exists: delete the USAG. Once the USAG is deleted the user will no longer be linked to any ACLs and will have no more permissions.

USAGs, knife-acl, and Manage

Unfortunately, USAGs were never fully implemented in the system. Some places attempted to make use of USAGs, while others did not. This meant the fully power of USAGs was never realized. This is also why tools such as knife-acl and Manage clash and could not be used together. knife-acl was attempting to use USAGs while Manage was not. This resulted in competing paradigms being used with difficult to understand results in practice.

Updates to these systems will hopefully resolve this conflict in the future.

Code

The Chef server permission system is primarily implemented in three places:

oc_chef_authz
oc_bifrost
oc_chef_wm

All of these systems are part of the Chef server and can be found in the Chef server repository on GitHub. oc_chef_authz and oc_chef_wm are part of Erchef, while oc_bifrost (often just referred to as bifrost) is a separate component that lives along side Erchef in the Chef server repo.

The main entry point for ACL modification starts at oc_chef_wm_acl.erl and oc_chef_wm_acl_permission.erl. Consult the dispatch.conf file in the priv directory under oc_chef_wm for the exact routes that are used.

oc_chef_wm calls into oc_chef_authz, which then calls into oc_bifrost.

For full implementation details it is best to consult the code and corresponding tests.

oc-id

How does oc-id factor into this picture?

oc-id stands for opcode-identity. oc-id ships as part of the Chef server. It is an OAuth2 provider and is intended to be the place that a Chef user will manage their identity.

oc-id does not factor into the permissions here. Instead the way it is is used is that other applications that interact with the Chef server, such as the Supermarket and Analytics, call out to oc-id when a user signs on. The user enters their login information into oc-id, which then verifies this with the Chef server auth system. Once verified, oc-id issues a token to the calling application and the application then uses this token as a user's authorization.

This allows the Chef server to act as a central auth point without each application needing to provide their own auth system. oc-id however is a blunt instrument. It simply provides a yes or no answer on if a user is who they say they are to the Chef server. It does not provide any of the finer grain permissions that the Chef server uses to any of the other applications.

Document Versions

v1.0 - Mark Mzyk - compiles initial version
v1.1 - Mark Mzyk - add in missing object types; expand code explanation by adding in oc_chef_wm; add oc-id section
v1.2 - Mark Mzyk - clarify details of user association in org creation section; fix spelling of contrast; add section clarifying which server versions this doc applies to; add note to containers section about needed permission on container to create or list objects
v1.3 - Mark Mzyk - add container as an object in object list, it was inadvertently left off

TODO:

Verify and add details on:

- In addition to the container permissions, the actor who created an object is also inserted into all 5 ACEs of the new object's ACL. An exception to this rule are validator clients.

- May want to add a section on the "pivotal" user

- Include section on ORGNAME_global_admins (this was possibly recently renamed):

ORGNAME_global_admins is woefully misnamed. It is a group that we create when we construct a new organization. It contains the admins group (currently, soon it will contain the users group as well, this is what I'm working on). Anytime we add a user into an organization, we add the ORGNAME_global_admins group into the READ ace for that user.
This allows admins in the org to read the user objects in their org. The change I'm making will add the users group into that group as well, which will mean that users within the same org will be able to see each other's user objects

https://github.com/chef/chef-server/pull/244

- Document exactly where USAGS are used in the system

- Add a diagram that shows the entire system - chef server, oc-id, analytics, supermarket, etc, and how it all fits together