Zend_Acl

1. Introduction

`Zend_Acl` provides a lightweight and flexible access control list (ACL) implementation for privileges management. In general, an application may utilize such ACL's to control access to certain protected objects by other requesting objects.

For the purposes of this documentation:

- a *resource* is an object to which access is controlled.
- a *role* is an object that may request access to a Resource.

Put simply, *roles request access to resources*. For example, if a parking attendant requests access to a car, then the parking attendant is the requesting role, and the car is the resource, since access to the car may not be granted to everyone.

Through the specification and use of an ACL, an application may control how roles are granted access to resources.

1.1. Resources

Creating a resource in `Zend_Acl` is very simple. `Zend_Acl` provides the resource, `Zend_Acl_Resource_Interface`, to facilitate creating resources in an application. A class need only implement this interface, which consists of a single method, `getResourceId()`, for `Zend_Acl` to recognize the object as a resource. Additionally, `Zend_Acl_Resource` is provided by `Zend_Acl` as a basic resource implementation for developers to extend as needed.

`Zend_Acl` provides a tree structure to which multiple resources can be added. Since resources are stored in such a tree structure, they can be organized from the general (toward the tree root) to the specific (toward the tree leaves). Queries on a specific resource will automatically search the resource's hierarchy for rules assigned to ancestor resources, allowing for simple inheritance of rules. For example, if a default rule is to be applied to each building in a city, one would simply assign the rule to the city, instead of assigning the same rule to each building. Some buildings may require exceptions to such a rule, however, and this can be achieved in `Zend_Acl` by assigning such exception rules to each building that requires such an exception. A resource may inherit from only one parent resource, though this parent resource can have its own parent resource, etc.

`Zend_Acl` also supports privileges on resources (e.g., "create", "read", "update", "delete"), so the developer can assign rules that affect all privileges or specific privileges on one or more resources.

1.2. Roles

As with resources, creating a role is also very simple. All roles must implement `Zend_Acl_Role_Interface`. This interface consists of a single method, `getRoleId()`, Additionally, `Zend_Acl_Role` is provided by `Zend_Acl` as a basic role implementation for developers to extend as needed.

In `Zend_Acl`, a role may inherit from one or more roles. This is to support inheritance of rules among roles. For example, a user role, such as "sally", may belong to one or more parent roles, such as "editor" and "administrator". The developer can assign rules to "editor" and "administrator" separately, and "sally" would inherit such rules from both, without having to assign rules directly to "sally".

Though the ability to inherit from multiple roles is very useful, multiple inheritance also introduces some degree of complexity. The following example illustrates the ambiguity condition and how `Zend_Acl` solves it.

Example 28. Multiple Inheritance among Roles

The following code defines three base roles - "guest", "member", and "admin" - from which other roles may inherit. Then, a role identified by "someUser" is established and inherits from the three other roles. The order in which these roles appear in the `$parents` array is important. When necessary, `Zend_Acl` searches for access rules defined not only for the queried role (herein, "someUser"), but also upon the roles from which the queried role inherits (herein, "guest", "member", and "admin"):

```
$acl = new Zend_Acl();

$acl->addRole(new Zend_Acl_Role('guest'))
    ->addRole(new Zend_Acl_Role('member'))
    ->addRole(new Zend_Acl_Role('admin'));

$parents = array('guest', 'member', 'admin');
$acl->addRole(new Zend_Acl_Role('someUser'), $parents);

$acl->add(new Zend_Acl_Resource('someResource'));

$acl->deny('guest', 'someResource');
$acl->allow('member', 'someResource');

echo $acl->isAllowed('someUser', 'someResource') ? 'allowed' : 'denied';
```

Since there is no rule specifically defined for the "someUser" role and "someResource", `Zend_Acl` must search for rules that may be defined for roles that "someUser" inherits. First, the "admin" role is visited, and there is no access rule defined for it. Next, the "member" role is visited, and `Zend_Acl` finds that there is a rule specifying that "member" is allowed access to "someResource".

If `Zend_Acl` were to continue examining the rules defined for other parent roles, however, it would find that "guest" is denied access to "someResource". This fact introduces an ambiguity because now "someUser" is both denied and allowed access to "someResource", by reason of having inherited conflicting rules from different parent roles.

`Zend_Acl` resolves this ambiguity by completing a query when it finds the first rule that is directly applicable to the query. In this case, since the "member" role is examined before the "guest" role, the example code would print "allowed".

When specifying multiple parents for a role, keep in mind that the last parent listed is the first one searched for rules applicable to an authorization query.

1.3. Creating the Access Control List

An Access Control List (ACL) can represent any set of physical or virtual objects that you wish. For the purposes of demonstration, however, we will create a basic Content Management System

(CMS) ACL that maintains several tiers of groups over a wide variety of areas. To create a new ACL object, we instantiate the ACL with no parameters:

```
$acl = new Zend_Acl();
```

Until a developer specifies an "allow" rule, `Zend_Acl` denies access to every privilege upon every resource by every role.

1.4. Registering Roles

CMS's will nearly always require a hierarchy of permissions to determine the authoring capabilities of its users. There may be a 'Guest' group to allow limited access for demonstrations, a 'Staff' group for the majority of CMS users who perform most of the day-to-day operations, an 'Editor' group for those responsible for publishing, reviewing, archiving and deleting content, and finally an 'Administrator' group whose tasks may include all of those of the other groups as well as maintenance of sensitive information, user management, back-end configuration data, backup and export. This set of permissions can be represented in a role registry, allowing each group to inherit privileges from 'parent' groups, as well as providing distinct privileges for their unique group only. The permissions may be expressed as follows:

Table 1. Access Controls for an Example CMS

Name	Unique Permissions	Inherit Permissions From
Guest	View	N/A
Staff	Edit, Submit, Revise	Guest
Editor	Publish, Archive, Delete	Staff
Administrator	(Granted all access)	N/A

For this example, `Zend_Acl_Role` is used, but any object that implements `Zend_Acl_Role_Interface` is acceptable. These groups can be added to the role registry as follows:

```
$acl = new Zend_Acl();

// Add groups to the Role registry using Zend_Acl_Role
// Guest does not inherit access controls
$roleGuest = new Zend_Acl_Role('guest');
$acl->addRole($roleGuest);

// Staff inherits from guest
$acl->addRole(new Zend_Acl_Role('staff'), $roleGuest);

/*
Alternatively, the above could be written:
$acl->addRole(new Zend_Acl_Role('staff'), 'guest');
*/

// Editor inherits from staff
$acl->addRole(new Zend_Acl_Role('editor'), 'staff');

// Administrator does not inherit access controls
$acl->addRole(new Zend_Acl_Role('administrator'));
```

1.5. Defining Access Controls

Now that the ACL contains the relevant roles, rules can be established that define how resources may be accessed by roles. You may have noticed that we have not defined any particular resources for this example, which is simplified to illustrate that the rules apply to all resources. `Zend_Acl` provides an implementation whereby rules need only be assigned from general to specific, minimizing the number of rules needed, because resources and roles inherit rules that are defined upon their ancestors.

In general, `Zend_Acl` obeys a given rule if and only if a more specific rule does not apply.

Consequently, we can define a reasonably complex set of rules with a minimum amount of code. To apply the base permissions as defined above:

```
$acl = new Zend_Acl();

$roleGuest = new Zend_Acl_Role('guest');
$acl->addRole($roleGuest);
$acl->addRole(new Zend_Acl_Role('staff'), $roleGuest);
$acl->addRole(new Zend_Acl_Role('editor'), 'staff');
$acl->addRole(new Zend_Acl_Role('administrator'));

// Guest may only view content
$acl->allow($roleGuest, null, 'view');

/*
Alternatively, the above could be written:
$acl->allow('guest', null, 'view');
//*/

// Staff inherits view privilege from guest, but also needs additional
// privileges
$acl->allow('staff', null, array('edit', 'submit', 'revise'));

// Editor inherits view, edit, submit, and revise privileges from
// staff, but also needs additional privileges
$acl->allow('editor', null, array('publish', 'archive', 'delete'));

// Administrator inherits nothing, but is allowed all privileges
$acl->allow('administrator');
```

The `NULL` values in the above `allow()` calls are used to indicate that the allow rules apply to all resources.

1.6. Querying an ACL

We now have a flexible ACL that can be used to determine whether requesters have permission to perform functions throughout the web application. Performing queries is quite simple using the `isAllowed()` method:

```
echo $acl->isAllowed('guest', null, 'view') ?
     "allowed" : "denied";
// allowed

echo $acl->isAllowed('staff', null, 'publish') ?
```

```
     "allowed" : "denied";
// denied

echo $acl->isAllowed('staff', null, 'revise') ?
     "allowed" : "denied";
// allowed

echo $acl->isAllowed('editor', null, 'view') ?
     "allowed" : "denied";
// allowed because of inheritance from guest

echo $acl->isAllowed('editor', null, 'update') ?
     "allowed" : "denied";
// denied because no allow rule for 'update'

echo $acl->isAllowed('administrator', null, 'view') ?
     "allowed" : "denied";
// allowed because administrator is allowed all privileges

echo $acl->isAllowed('administrator') ?
     "allowed" : "denied";
// allowed because administrator is allowed all privileges

echo $acl->isAllowed('administrator', null, 'update') ?
     "allowed" : "denied";
// allowed because administrator is allowed all privileges
```

2. Refining Access Controls

2.1. Precise Access Controls

The basic ACL as defined in the previous section shows how various privileges may be allowed upon the entire ACL (all resources). In practice, however, access controls tend to have exceptions and varying degrees of complexity. `Zend_Acl` allows to you accomplish these refinements in a straightforward and flexible manner.

For the example CMS, it has been determined that whilst the 'staff' group covers the needs of the vast majority of users, there is a need for a new 'marketing' group that requires access to the newsletter and latest news in the CMS. The group is fairly self-sufficient and will have the ability to publish and archive both newsletters and the latest news.

In addition, it has also been requested that the 'staff' group be allowed to view news stories but not to revise the latest news. Finally, it should be impossible for anyone (administrators included) to archive any 'announcement' news stories since they only have a lifespan of 1-2 days.

First we revise the role registry to reflect these changes. We have determined that the 'marketing' group has the same basic permissions as 'staff', so we define 'marketing' in such a way that it inherits permissions from 'staff':

```
// The new marketing group inherits permissions from staff
$acl->addRole(new Zend_Acl_Role('marketing'), 'staff');
```

Next, note that the above access controls refer to specific resources (e.g., "newsletter", "latest news", "announcement news"). Now we add these resources:

```
// Create Resources for the rules
// newsletter
$acl->addResource(new Zend_Acl_Resource('newsletter'));

// news
$acl->addResource(new Zend_Acl_Resource('news'));

// latest news
$acl->addResource(new Zend_Acl_Resource('latest'), 'news');

// announcement news
$acl->addResource(new Zend_Acl_Resource('announcement'), 'news');
```

Then it is simply a matter of defining these more specific rules on the target areas of the ACL:

```
// Marketing must be able to publish and archive newsletters and the
// latest news
$acl->allow('marketing',
            array('newsletter', 'latest'),
            array('publish', 'archive'));

// Staff (and marketing, by inheritance), are denied permission to
// revise the latest news
$acl->deny('staff', 'latest', 'revise');

// Everyone (including administrators) are denied permission to
// archive news announcements
$acl->deny(null, 'announcement', 'archive');
```

We can now query the ACL with respect to the latest changes:

```
echo $acl->isAllowed('staff', 'newsletter', 'publish') ?
     "allowed" : "denied";
// denied

echo $acl->isAllowed('marketing', 'newsletter', 'publish') ?
     "allowed" : "denied";
// allowed

echo $acl->isAllowed('staff', 'latest', 'publish') ?
     "allowed" : "denied";
// denied

echo $acl->isAllowed('marketing', 'latest', 'publish') ?
     "allowed" : "denied";
// allowed

echo $acl->isAllowed('marketing', 'latest', 'archive') ?
     "allowed" : "denied";
// allowed

echo $acl->isAllowed('marketing', 'latest', 'revise') ?
     "allowed" : "denied";
// denied

echo $acl->isAllowed('editor', 'announcement', 'archive') ?
     "allowed" : "denied";
// denied

echo $acl->isAllowed('administrator', 'announcement', 'archive') ?
```

```
     "allowed" : "denied";
// denied
```

2.2. Removing Access Controls

To remove one or more access rules from the ACL, simply use the available `removeAllow()` or `removeDeny()` methods. As with `allow()` and `deny()`, you may provide a `NULL` value to indicate application to all roles, resources, and/or privileges:

```
// Remove the denial of revising latest news to staff (and marketing,
// by inheritance)
$acl->removeDeny('staff', 'latest', 'revise');

echo $acl->isAllowed('marketing', 'latest', 'revise') ?
     "allowed" : "denied";
// allowed

// Remove the allowance of publishing and archiving newsletters to
// marketing
$acl->removeAllow('marketing',
                  'newsletter',
                  array('publish', 'archive'));

echo $acl->isAllowed('marketing', 'newsletter', 'publish') ?
     "allowed" : "denied";
// denied

echo $acl->isAllowed('marketing', 'newsletter', 'archive') ?
     "allowed" : "denied";
// denied
```

Privileges may be modified incrementally as indicated above, but a `NULL` value for the privileges overrides such incremental changes:

```
// Allow marketing all permissions upon the latest news
$acl->allow('marketing', 'latest');

echo $acl->isAllowed('marketing', 'latest', 'publish') ?
     "allowed" : "denied";
// allowed

echo $acl->isAllowed('marketing', 'latest', 'archive') ?
     "allowed" : "denied";
// allowed

echo $acl->isAllowed('marketing', 'latest', 'anything') ?
     "allowed" : "denied";
// allowed
```

3. Advanced Usage

3.1. Storing ACL Data for Persistence

`Zend_Acl` was designed in such a way that it does not require any particular backend technology such as a database or cache server for storage of the ACL data. Its complete PHP implementation enables customized administration tools to be built upon `Zend_Acl` with relative ease and flexibility. Many situations require some form of interactive maintenance of the ACL, and `Zend_Acl` provides methods for setting up, and querying against, the access controls of an application.

Storage of ACL data is therefore left as a task for the developer, since use cases are expected to vary widely for various situations. Because `Zend_Acl` is serializable, ACL objects may be serialized with PHP's `serialize()` function, and the results may be stored anywhere the developer should desire, such as a file, database, or caching mechanism.

3.2. Writing Conditional ACL Rules with Assertions

Sometimes a rule for allowing or denying a role access to a resource should not be absolute but dependent upon various criteria. For example, suppose that certain access should be allowed, but only between the hours of 8:00am and 5:00pm. Another example would be denying access because a request comes from an IP address that has been flagged as a source of abuse. `Zend_Acl` has built-in support for implementing rules based on whatever conditions the developer needs.

`Zend_Acl` provides support for conditional rules with `Zend_Acl_Assert_Interface`. In order to use the rule assertion interface, a developer writes a class that implements the `assert()` method of the interface:

```
class CleanIPAssertion implements Zend_Acl_Assert_Interface
{
    public function assert(Zend_Acl $acl,
                           Zend_Acl_Role_Interface $role = null,
                           Zend_Acl_Resource_Interface $resource = null,
                           $privilege = null)
    {
        return $this->_isCleanIP($_SERVER['REMOTE_ADDR']);
    }

    protected function _isCleanIP($ip)
    {
        // ...
    }
}
```

Once an assertion class is available, the developer must supply an instance of the assertion class when assigning conditional rules. A rule that is created with an assertion only applies when the assertion method returns `TRUE`.

```
$acl = new Zend_Acl();
$acl->allow(null, null, null, new CleanIPAssertion());
```

The above code creates a conditional allow rule that allows access to all privileges on everything by everyone, except when the requesting IP is "blacklisted." If a request comes in from an IP that is not considered "clean," then the allow rule does not apply. Since the rule applies to all roles, all resources, and all privileges, an "unclean" IP would result in a denial of access. This is a special case, however, and it should be understood that in all other cases (i.e., where a specific role, resource, or privilege is specified for the rule), a failed assertion results in the rule not applying, and other rules would be used to determine whether access is allowed or denied.

The `assert()` method of an assertion object is passed the ACL, role, resource, and privilege to which the authorization query (i.e., `isAllowed()`) applies, in order to provide a context for the assertion class to determine its conditions where needed.