Spring Security ACL - Reference Documentation

Authors: Burt Beckwith
Version: 2.0-RC2

Table of Contents

1 Introduction to the Spring Security ACL Plugin

The ACL plugin adds Domain Object Security support to a Grails application that uses Spring Security. It depends on the Spring Security Core plugin.

The core plugin and other extension plugins support restricting access to URLs via rules that include checking a user's authentication status, roles, etc. and the ACL plugin extends this by adding support for restricting access to individual domain class instances. The access can be very fine-grained and can define which actions can be taken on an object - these typically include Read, Create, Write, Delete, and Administer but you're free to define whatever actions you like.

To learn about using ACLs in Grails, you can follow guide:3. Tutorial and in addition you can download and run a complete Grails application that uses the plugin. Installing and running the application are described guide:4. Sample Application.

In addition to this document, you should read the Spring Security documentation here.

1.1 History

History

- November 17, 2014
  - 2.0-RC2 release
- October 08, 2013
  - 2.0-RC1 release
- August 20, 2012
  - 1.1.1 release
- February 16, 2011
  - 1.1 release
- February 7, 2011
  - 1.0.2 release
- August 1, 2010
  - 1.0.1 release
- July 27, 2010
  - 1.0 release
- May 22, 2010
  - initial 0.1 release

Authors

Burt Beckwith

Previous work

Stephan February did the first work adding ACL support to the Acegi plugin. At the time the plugin was based on Acegi 1.0.x and around the same time the plugin was converted to use Spring Security 2.0 and the ACL support wasn't converted to use the new package layout and approach.

Work was done in 2009 to create a GORM-based implementation (the standard Spring Security implementation uses JDBC). Around the same time, Phillip Merensky mentioned on the Grails mailing list that he was working on an implementation. He wrote about his approach here and this was merged in with the other approach but never formally released.

This plugin builds on that work but is based on Spring Security 3 and Spring 3.

2 Usage

2.1 Securing Service Methods

There are two primary use cases for ACL security: determining whether a user is allowed to perform an action on an instance before the action is invoked, and restricting access to single or multiple instances after methods are invoked (this is typically implemented by collection filtering). You can call `aclUtilService.hasPermission()` explicitly, but this tends to clutter your code with security logic that often has little to do with business logic. Instead, Spring Security provides some convenient annotations that are used to wrap your method calls in access checks.

There are four annotations:

- @PreAuthorize
- @PreFilter
- @PostAuthorize
- @PostFilter

The annotations use security-specific Spring expression language (SpEL) expressions - see the documentation for the available standard and method expressions.

Here's an example service that manages a `Report` domain class and uses these annotations and expressions:

```
import org.springframework.security.access.prepost.PostFilter
import org.springframework.security.access.prepost.PreAuthorize
import org.springframework.transaction.annotation.Transactional

import com.yourapp.Report

class ReportService {

@PreAuthorize("hasPermission(#id, 'com.yourapp.Report', read) or " +
                  "hasPermission(#id, 'com.yourapp.Report', admin)")
   Report getReport(long id) {
      Report.get(id)
   }

@Transactional
   @PreAuthorize("hasRole('ROLE_USER')")
   Report createReport(params) {
      Report report = new Report(params)
      report.save()
      report
   }

@PreAuthorize("hasRole('ROLE_USER')")
   @PostFilter("hasPermission(filterObject, read) or " +
               "hasPermission(filterObject, admin)")
   List getAllReports(params = [:]) {
      Report.list(params)
   }

@Secured(['ROLE_USER', 'ROLE_ADMIN'])
   String getReportName(long id) {
      Report.get(id).name
   }

@Transactional
   @PreAuthorize("hasPermission(#report, write) or " +
                 "hasPermission(#report, admin)")
   Report updateReport(Report report, params) {
      report.properties = params
      report.save()
      report
   }

@Transactional
   @PreAuthorize("hasPermission(#report, delete) or " +
                 "hasPermission(#report, admin)")
   void deleteReport(Report report) {
      report.delete()
   }
}
```

The configuration specifies these rules:

- `getReport` requires that the authenticated user have `BasePermission.READ` or `BasePermission.ADMIN` for the instance
- `createReport` requires `ROLE_USER`
- `getAllReports` requires `ROLE_USER` and will have elements removed from the returned `List` that the user doesn't have an ACL grant for; the user must have `BasePermission.READ` or `BasePermission.ADMIN` for each element in the list; elements that don't have access granted will be removed
- `getReportName` requires that the authenticated user have either `ROLE_USER` or `ROLE_ADMIN` (but no ACL rules)
- `updateReport` has no role restrictions but must satisfy the requirements of the `aclReportWriteVoter` voter (which has the `ACL_REPORT_WRITE` config attribute), i.e. `BasePermission.ADMINISTRATION` or `BasePermission.WRITE`
- `deleteReport` has no role restrictions but must satisfy the requirements of the `aclReportDeleteVoter` voter (which has the `ACL_REPORT_DELETE` config attribute), i.e. `BasePermission.ADMINISTRATION` or `BasePermission.DELETE`

2.2 Working with ACLs

Suggested application changes

To properly display access denied exceptions (e.g. when a user tries to perform an action but doesn't have a grant authorizing it), you should create a mapping in `grails-app/conf/UrlMappings.groovy` for error code 403. In addition, it's possible to trigger a NotFoundException which will create an error 500, but should be treated like a 403 error, so you should add mappings for these conditions:

```
import org.springframework.security.access.AccessDeniedException
import org.springframework.security.acls.model.NotFoundException

class UrlMappings {

static mappings = {
      "/$controller/$action?/$id?"{
         constraints {}
      }

"/"(view:"/index")

"403"(controller: "errors", action: "error403")
      "500"(controller: "errors", action: "error500")
      "500"(controller: "errors", action: "error403",
            exception: AccessDeniedException)
      "500"(controller: "errors", action: "error403",
            exception: NotFoundException)
   }
}
```

These depend on an `ErrorsController`:

```
package com.yourcompany.yourapp

import grails.plugin.springsecurity.annotation.Secured

@Secured(['permitAll'])
class ErrorsController {

def error403() {}

def error500() {
      render view: '/error'
   }
}
```

and a `grails-app/views/errors/error403.gsp` similar to this:

```
<html>
<head>
<title>Access denied!</title>
<meta name='layout' content='main' />
</head>

<body>
<h1>Access Denied</h1>
<p>We're sorry, but you are not authorized to
   perform the requested operation.</p>
</body>
</html>
```

actionSubmit

Grails has a convenient feature where it supports multiple submit actions per form via the `<g:actionSubmit>` tag. This is done by posting to the `index` action but with a special parameter that indicates which action to invoke. This is a problem in general for security since any URL rules for edit, delete, save, etc. will be bypassed. It's an even more significant issue with ACLs because of the way that the access denied exception interacts with the `actionSubmit` processing. If you don't make any adjustments for this, your users will see a blank page when they attempt to submit a form and the action is disallowed. The solution is to remove `actionSubmit` buttons and replace them with regular submit buttons. This requires one form per button, and without adjusting the CSS the buttons will look differently than if they were in-line `actionSubmit` buttons, but that is fixable with the appropriate CSS changes.

It's simple to adjust the `actionSubmit` buttons and you'll need to change them in `show.gsp` and `edit.gsp`; `list.gsp` and `show.gsp` don't need any changes. In `show.gsp`, replace the two actionSubmit buttons with these two forms (maintain the g:message tags; the strings are hard-coded here to reduce clutter):

```
<div class="buttons">
   <g:form action='edit'>
      <g:hiddenField name="id" value="${reportInstance?.id}" />
      <span class="button">
         <g:submitButton class="edit" name="Edit" />
      </span>
   </g:form>
   <g:form action='delete'>
      <g:hiddenField name="id" value="${reportInstance?.id}" />
      <span class="button">
         <g:submitButton class="delete" name="Delete"
                         onclick="return confirm('Are you sure?');" />
      </span>
   </g:form>
</div>
```

In `edit.gsp`, change the `<form>` tag to

```
<g:form action='update'>
```

and convert the update button to a regular submit button:

```
<div class="buttons">
   <span class="button">
      <g:submitButton class="save" name="Update" />
   </span>
</div>
```

and move the delete button out of the form into its own form just below the main form:

```
<g:form action='delete'>
   <g:hiddenField name="id" value="${reportInstance?.id}" />
   <div class="buttons">
      <span class="button">
         <g:submitButton class="delete" name="Delete"
                         onclick="return confirm('Are you sure?');" />
      </span>
   </div>
</g:form>
```

2.3 Domain Classes

The plugin uses domain classes to manage database state. Ordinarily the database structure isn't all that important, but to be compatible with the traditional JDBC-based Spring Security code, the domain classes are configured to generate the table and column names that are used there.

The plugin classes related to persistence use these classes, so they're included in the plugin but can be overridden by running the s2-create-acl-domains script.

As you can see, the database structure is highly normalized.

AclClass

The `AclClass` domain class contains entries for the names of each application domain class that has associated permissions:

```
class AclClass {

String className

@Override
   String toString() {
      "AclClass id $id, className $className"
   }

static mapping = {
      className column: 'class'
      version false
   }

static constraints = {
      className unique: true
   }
}
```

AclSid

The `AclSid` domain class contains entries for the names of grant recipients (a principal or authority - SID is an acronym for "security identity"). These are typically usernames (where `principal` is `true`) but can also be a `GrantedAuthority` (role name, where `principal` is `false`). When granting permissions to a role, any user with that role receives that permission:

```
class AclSid {

String sid
   boolean principal

@Override
   String toString() {
      "AclSid id $id, sid $sid, principal $principal"
   }

static mapping = {
      version false
   }

static constraints = {
      principal unique: 'sid'
   }
}
```

AclObjectIdentity

The `AclObjectIdentity` domain class contains entries representing individual domain class instances (OIDs). It has a field for the instance id (`objectId`) and domain class (`aclClass`) that uniquely identify the instance. In addition there are optional nullable fields for the parent OID (`parent`) and owner (`owner`). There's also a flag (`entriesInheriting`) to indicate whether ACL entries can inherit from a parent ACL.

```
class AclObjectIdentity extends AbstractAclObjectIdentity {

Long objectId

@Override
   String toString() {
      "AclObjectIdentity id $id, aclClass $aclClass.className, " +
      "objectId $objectId, entriesInheriting $entriesInheriting"
   }

static mapping = {
      version false
      aclClass column: 'object_id_class'
      owner column: 'owner_sid'
      parent column: 'parent_object'
      objectId column: 'object_id_identity'
   }

static constraints = {
      objectId unique: 'aclClass'
   }
}
```

`AclObjectIdentity` actually extends a base class, `AbstractAclObjectIdentity`:

```
abstract class AbstractAclObjectIdentity {

AclClass aclClass
   AclObjectIdentity parent
   AclSid owner
   boolean entriesInheriting

static constraints = {
      parent nullable: true
      owner nullable: true
   }
}
```

By default it's assumed that domain classes have a numeric primary key, but that's not required. So the default implementation has a `Long objectId` field, but if you want to support other types of ids you can change that field and retain the other standard functionality from the base class.

AclEntry

Finally, the `AclEntry` domain class contains entries representing grants (or denials) of a permission on an object instance to a recipient. The `aclObjectIdentity` field references the domain class instance (since an instance can have many granted permissions). The `sid` field references the recipient. The `granting` field determines whether the entry grants the permission (`true`) or denies it (`false`). The `aceOrder` field specifies the position of the entry, which is important because the entries are evaluated in order and the first matching entry determines whether access is allowed. `auditSuccess` and `auditFailure` determine whether to log success and/or failure events (these both default to `false`).

The `mask` field holds the permission. This can be a source of confusion because the name (and the Spring Security documentation) indicates that it's a bit mask. A value of 1 indicates permission A, a value of 2 indicates permission B, a value of 4 indicates permission C, a value of 8 indicates permission D, etc. So you would think that a value of 5 would indicate a grant of both permission A and C. Unfortunately this is not the case. There is a CumulativePermission class that supports this, but the standard classes don't support it (`AclImpl.isGranted()` checks for == rather than using | (bitwise or) so a combined entry would never match). So rather than grouping all permissions for one recipient on one instances into a bit mask, you must create individual records for each. This will be addressed in Spring Security 3.1 however.

```
class AclEntry {

AclObjectIdentity aclObjectIdentity
   int aceOrder
   AclSid sid
   int mask
   boolean granting
   boolean auditSuccess
   boolean auditFailure

@Override
   String toString() {
      "AclEntry id $id, aceOrder $aceOrder, mask $mask, " +
      "granting $granting, aclObjectIdentity $aclObjectIdentity"
   }

static mapping = {
      version false
      sid column: 'sid'
      aclObjectIdentity column: 'acl_object_identity'
   }

static constraints = {
      aceOrder unique: 'aclObjectIdentity'
   }
}
```

2.4 Configuration

Creating, editing, or deleting permissions requires an authenticated user. In most cases if the authenticated user is the owner of the ACL then access is allowed, but granted roles also affect whether access is allowed. The default required role is `ROLE_ADMIN` for all actions, but this can be configured in `grails-app/conf/Config.groovy`. This table summarizes the attribute names and the corresponding actions that are allowed for it:

Attribute	Affected methods
`grails.plugin.springsecurity.acl.authority.modifyAuditingDetails`	`AuditableAcl.updateAuditing()`
`grails.plugin.springsecurity.acl.authority.changeOwnership`	`OwnershipAcl.setOwner()`
`grails.plugin.springsecurity.acl.authority.changeAclDetails`	`MutableAcl.deleteAce()`, `MutableAcl.insertAce()`, `MutableAcl.setEntriesInheriting()`, `MutableAcl.setParent()`, `MutableAcl.updateAce()`

You can leave the attributes set to `ROLE_ADMIN` or change them to have separate values, e.g.

```
grails.plugin.springsecurity.acl.authority.
        modifyAuditingDetails = 'ROLE_ACL_MODIFY_AUDITING'

grails.plugin.springsecurity.acl.authority.
        changeOwnership = 'ROLE_ACL_CHANGE_OWNERSHIP'

grails.plugin.springsecurity.acl.authority.
        changeAclDetails = 'ROLE_ACL_CHANGE_DETAILS'
```

Run-As Authentication Replacement

There are also two options to configure Run-As Authentication Replacement:

Attribute	Meaning
`grails.plugin.springsecurity.useRunAs`	change to `true` to enable; defaults to `false`
`grails.plugin.springsecurity.runAs.key`	a shared key between the two standard implementation classes, used to verify that a third party hasn't created a token for the user; should be changed from its default value

Example:

```
grails.plugin.springsecurity.useRunAs = true
grails.plugin.springsecurity.runAs.key = 'your run-as key'
```

2.5 Run-As Authentication Replacement

Although not strictly related to ACLs, the plugin implements Run-As Authentication Replacement since it's related to method security in general. This feature is similar to the Switch User feature of the Spring Security Core plugin, but instead of running as another user until you choose to revert to your original `Authentication`, the temporary authentication switch only lasts for one method invocation.

For example, in this service `someMethod()` requires that the authenticated user have `ROLE_ADMIN` and will also be granted `ROLE_RUN_AS_SUPERUSER` for the duration of the method only:

```
class SecureService {

@Secured(['ROLE_ADMIN', 'RUN_AS_SUPERUSER'])
   def someMethod() {
      …
   }
}
```

2.6 Custom Permissions

By default there are 5 permissions available from the `org.springframework.security.acls.domain.BasePermission` class: `READ`, `WRITE`, `CREATE`, `DELETE`, and `ADMINISTRATION`. You can also add your own permissions if these aren't sufficient.

The easiest approach is to create a subclass of `BasePermission` and add your new permissions there. This way you retain the default permissions and can use them if you need. For example, here's a subclass that adds a new `APPROVE` permission:

```
package com.mycompany.myapp;

import org.springframework.security.acls.domain.BasePermission;
import org.springframework.security.acls.model.Permission;

public class MyPermission extends BasePermission {

public static final Permission APPROVE = new MyPermission(1 << 5, 'V');

protected MyPermission(int mask) {
      super(mask);
   }

protected MyPermission(int mask, char code) {
      super(mask, code);
   }
}
```

It sets the mask value to 32 (1 << 5) since the values up to 16 are defined in the base class.

To use your class instead of the default, specify it in with the `grails.plugin.springsecurity.acl. permissionClass` attribute either as a Class or a String, for example

```
import com.mycompany.myapp.MyPermissions
…
grails.plugin.springsecurity.acl. permissionClass = MyPermissions
```

or

```
grails.plugin.springsecurity.acl.permissionClass =
'com.mycompany.myapp.MyPermissions'
```

You can also override the `aclPermissionFactory` bean in `grails-app/conf/spring/resources.groovy`, keeping the `org.springframework.security.acls.domain. DefaultPermissionFactory` class but passing your class as the constructor argument to keep it from defaulting to `BasePermission`, or do a more complex override to more fully reconfigure the behavior:

```
import org.springframework.security.acls.domain.DefaultPermissionFactory
import com.mycompany.myapp.MyPermission

beans = {
   aclPermissionFactory(DefaultPermissionFactory, MyPermission)
}
```

Once this is done you can use the permission like any other, specifying its quoted lowercase name in an expression, e.g.

```
@PreAuthorize("hasPermission(#id, 'com.testacl.Report', 'approve')")
Report get(long id) {
   Report.get id
}
```

3 Tutorial

First create a test application:

```
$ grails create-app acltest
$ cd acltest
```

Install the plugin by adding it to the plugins section in BuildConfig.groovy:

```
plugins {
   …
   runtime ':spring-security-acl:2.0-RC2'
}
```

This will install the Spring Security Core plugin, so you'll need to configure that by running the `s2-quickstart` script:

```
$ grails s2-quickstart com.testacl User Role
```

The ACL support uses domain classes but to allow customizing the domain classes (e.g. to enable Hibernate 2nd-level caching) there's a script that copies the domain classes into your application, `s2-create-acl-domains`. This script is run when the plugin is installed (otherwise the plugin code wouldn't compile) but you can run it again to re-create the domain classes:

```
$ grails s2-create-acl-domains
```

Note that you cannot change the domain class names or packages since they're used by the plugin. The domain class mappings are configured to generate the same DDL as is required by the standard Spring Security JDBC implementation for portability.

We'll need a domain class to test with, so create a `Report` domain class:

```
$ grails create-domain-class com.testacl.Report
```

and add a `name` property for testing:

```
package com.testacl

class Report {
   String name
}
```

Next we'll create a service to test ACLs:

```
$ grails create-service com.testacl.Report
```

and add some methods that work with `Reports`:

```
package com.testacl

import org.springframework.security.access.prepost.PostFilter
import org.springframework.security.access.prepost.PreAuthorize
import org.springframework.security.acls.domain.BasePermission
import org.springframework.security.acls.model.Permission
import org.springframework.transaction.annotation.Transactional

class ReportService {

def aclPermissionFactory
   def aclService
   def aclUtilService
   def springSecurityService

void addPermission(Report report, String username, int permission) {
      addPermission report, username,
           aclPermissionFactory.buildFromMask(permission)
   }

@PreAuthorize("hasPermission(#report, admin)")
   @Transactional
   void addPermission(Report report, String username,
                      Permission permission) {
      aclUtilService.addPermission report, username, permission
   }

@Transactional
   @PreAuthorize("hasRole('ROLE_USER')")
   Report create(String name) {
      Report report = new Report(name: name)
      report.save()

// Grant the current principal administrative permission
      addPermission report, springSecurityService.authentication.name,
                    BasePermission.ADMINISTRATION

report
   }

@PreAuthorize("hasPermission(#id, 'com.testacl.Report', read) or " +
                  "hasPermission(#id, 'com.testacl.Report', admin)")
   Report get(long id) {
      Report.get id
   }

@PreAuthorize("hasRole('ROLE_USER')")
   @PostFilter("hasPermission(filterObject, read) or " +
               "hasPermission(filterObject, admin)")
   List<Report> list(Map params) {
      Report.list params
   }

int count() {
      Report.count()
   }

@Transactional
   @PreAuthorize("hasPermission(#report, write) or " +
                  "hasPermission(#report, admin)")
   void update(Report report, String name) {
      report.name = name
   }
```

```
@Transactional
   @PreAuthorize("hasPermission(#report, delete) or " +
                 "hasPermission(#report, admin)")
   void delete(Report report) {
      report.delete()

// Delete the ACL information as well
      aclUtilService.deleteAcl report
   }

@Transactional
   @PreAuthorize("hasPermission(#report, admin)")
   void deletePermission(Report report, String username, Permission
permission) {
      def acl = aclUtilService.readAcl(report)

// Remove all permissions associated with this particular
      // recipient (string equality to KISS)
      acl.entries.eachWithIndex { entry, i ->
         if (entry.sid.equals(recipient) &&
               entry.permission.equals(permission)) {
            acl.deleteAce i
         }
      }

aclService.updateAcl acl
   }
}
```

The configuration specifies these rules:

- `addPermission` requires that the authenticated user have admin permission on the report instance to grant a permission to someone else
- `create` requires that the authenticated user have `ROLE_USER`
- `get` requires that the authenticated user have read or admin permission on the specified Report
- `list` requires that the authenticated user have ROLE_USER and read or admin permission on each returned Report; instances that don't have granted permissions will be removed from the returned List
- `count` has no restrictions
- `update` requires that the authenticated user have write or admin permission on the report instance to edit it
- `delete` requires that the authenticated user have delete or admin permission on the report instance to edit it
- `deletePermission` requires that the authenticated user have admin permission on the report instance to delete a grant

To test this out we'll need some users; create those and their grants in BootStrap.groovy:

```
import com.testacl.Report
import com.testacl.Role
import com.testacl.User
import com.testacl.UserRole

import static
org.springframework.security.acls.domain.BasePermission.ADMINISTRATION
import static org.springframework.security.acls.domain.BasePermission.DELETE
import static org.springframework.security.acls.domain.BasePermission.READ
import static org.springframework.security.acls.domain.BasePermission.WRITE
```

```
import org.springframework.security.authentication.
UsernamePasswordAuthenticationToken
import org.springframework.security.core.authority.AuthorityUtils
import org.springframework.security.core.context.SecurityContextHolder as SCH

class BootStrap {

def aclService
   def aclUtilService
   def objectIdentityRetrievalStrategy
   def sessionFactory

def init = { servletContext ->
      createUsers()
      loginAsAdmin()
      grantPermissions()
      sessionFactory.currentSession.flush()

// logout
      SCH.clearContext()
   }

private void loginAsAdmin() {
      // have to be authenticated as an admin to create ACLs
      SCH.context.authentication = new UsernamePasswordAuthenticationToken(
            'admin', 'admin123',
            AuthorityUtils.createAuthorityList('ROLE_ADMIN'))
   }

private void createUsers() {
      def roleAdmin = new Role(authority: 'ROLE_ADMIN').save()
      def roleUser = new Role(authority: 'ROLE_USER').save()

3.times {
         long id = it + 1
         def user = new User(username: "user$id", enabled: true,
            password: "password$id").save()
         UserRole.create user, roleUser
      }

def admin = new User(username: 'admin', enabled: true,
         password: 'admin123').save()

UserRole.create admin, roleUser
      UserRole.create admin, roleAdmin, true
   }

private void grantPermissions() {
      def reports = []
      100.times {
         long id = it + 1
         def report = new Report(name: "report$id").save()
         reports << report
         aclService.createAcl(
               objectIdentityRetrievalStrategy.getObjectIdentity(report))
      }

// grant user 1 admin on 11,12 and read on 1-67
      aclUtilService.addPermission reports[10], 'user1', ADMINISTRATION
      aclUtilService.addPermission reports[11], 'user1', ADMINISTRATION
      67.times {
         aclUtilService.addPermission reports[it], 'user1', READ
      }

// grant user 2 read on 1-5, write on 5
      5.times {
         aclUtilService.addPermission reports[it], 'user2', READ
      }
      aclUtilService.addPermission reports[4], 'user2', WRITE

// user 3 has no grants

// grant admin admin on all
      for (report in reports) {
         aclUtilService.addPermission report, 'admin', ADMINISTRATION
      }
```

```
  // grant user 1 ownership on 1,2 to allow the user to grant
        aclUtilService.changeOwner reports[0], 'user1'
        aclUtilService.changeOwner reports[1], 'user1'
    }
  }
```

And to have a UI to test with, let's create a `Report` controller and GSPs:

```
$ grails generate-all com.testacl.Report
```

But to use the controller, it will have to be reworked to use `ReportService`. It's a good idea to put all create/edit/delete code in a transactional service, but in this case we need to move all database access to the service to ensure that appropriate access checks are made:

```
  package com.testacl

  import org.springframework.dao.DataIntegrityViolationException
  import org.springframework.security.acls.model.Permission

  import grails.plugin.springsecurity.annotation.Secured

  @Secured(['ROLE_USER'])
  class ReportController {

  static defaultAction = 'list'

  def reportService

  def list() {
        params.max = Math.min(params.max ? params.int('max') : 10, 100)
        [reportInstanceList: reportService.list(params),
         reportInstanceTotal: reportService.count()]
     }

  def create() {
        [reportInstance: new Report(params)]
     }

  def save() {
        def report = reportService.create(params.name)
        if (!renderWithErrors('create', report)) {
           redirectShow "Report $report.id created", report.id
        }
     }

  def show() {
        def report = findInstance()
        if (!report) return

  [reportInstance: report]
     }

  def edit() {
        def report = findInstance()
        if (!report) return

  [reportInstance: report]
     }

  def update() {
        def report = findInstance()
        if (!report) return

  reportService.update report, params.name
        if (!renderWithErrors('edit', report)) {
           redirectShow "Report $report.id updated", report.id
        }
     }
```

```
def delete() {
      def report = findInstance()
      if (!report) return

try {
         reportService.delete report
         flash.message = "Report $params.id deleted"
         redirect action: list
      }
      catch (DataIntegrityViolationException e) {
         redirectShow "Report $params.id could not be deleted", params.id
      }
   }

def grant() {

def report = findInstance()
      if (!report) return

if (!request.post) {
         return [reportInstance: report]
      }

reportService.addPermission(report, params.recipient,
               params.int('permission'))

redirectShow "Permission $params.permission granted on Report $report.id " +
                  "to $params.recipient", report.id
   }

private Report findInstance() {
      def report = reportService.get(params.long('id'))
      if (!report) {
         flash.message = "Report not found with id $params.id"
         redirect action: list
      }
      report
   }

private void redirectShow(message, id) {
      flash.message = message
      redirect action: show, id: id
   }

private boolean renderWithErrors(String view, Report report) {
      if (report.hasErrors()) {
         render view: view, model: [reportInstance: report]
         return true
      }
      false
   }
}
```

Note that the controller is annotated to require either `ROLE_USER` or `ROLE_ADMIN`. Since services have nothing to do with HTTP, when access is blocked you cannot be redirected to the login page as when you try to access a URL that requires an authentication. So you need to configure URLs with similar role requirements to give the user a chance to attempt a login before calling secured service methods.

Finally, we'll make a few adjustments so errors are handled gracefully.

First, edit `grails-app/conf/UrlMappings.groovy` and add some error code mappings:

```
import org.springframework.security.access.AccessDeniedException
import org.springframework.security.acls.model.NotFoundException

class UrlMappings {

static mappings = {
      "/$controller/$action?/$id?"{
         constraints {}
      }

"/"(view:"/index")

"403"(controller: "errors", action: "error403")
      "404"(controller: "errors", action: "error404")
      "500"(controller: "errors", action: "error500")
      "500"(controller: "errors", action: "error403",
            exception: AccessDeniedException)
      "500"(controller: "errors", action: "error403",
            exception: NotFoundException)
   }
}
```

Then create the `ErrorsController` that these reference:

```
$ grails create-controller com.testacl.Errors
```

and add this code:

```
package com.testacl

import grails.plugin.springsecurity.annotation.Secured

@Secured(['permitAll'])
class ErrorsController {

def error403() {}

def error404() {}

def error500() {
      render view: '/error'
   }
}
```

and create the GSPs:

Add this to `grails-app/views/errors/error403.gsp`:

```
<html>
<head>
<title>Access denied!</title>
<meta name='layout' content='main' />
</head>

<body>
<h1>Access Denied</h1>
<p>We're sorry, but you are not authorized
   to perform the requested operation.</p>
</body>
</html>
```

and this to `grails-app/views/errors/error404.gsp`:

```
<html>
<head>
<title>Not Found</title>
<meta name='layout' content='main' />
</head>

<body>
<h1>Not Found</h1>
<p>We're sorry, but that page doesn't exist.</p>
</body>
</html>
```

actionSubmit issues

Grails has a convenient feature where it supports multiple submit actions per form. This is done by posting to the `index` action but with a special parameter that indicates which action to invoke. This is a problem in general for security since any URL rules for edit, delete, save, etc. will be bypassed. It's an even more significant issue with ACLs because of the way that the access denied exception interacts with the `actionSubmit` processing. If you don't make any adjustments for this, your users will see a blank page when they attempt to submit a form and the action is disallowed. The solution is to remove `actionSubmit` buttons and replace them with regular submit buttons. This requires one form per button, and without adjusting the CSS the buttons will look differently than if they were in-line `actionSubmit` buttons, but that is fixable with the appropriate CSS changes.

It's simple to adjust the `actionSubmit` buttons; in `grails-app/views/report/show.gsp`, replace the two actionSubmit buttons with these two forms (maintain the g:message tags; the strings are hard-coded here to reduce clutter):

```
<div class="buttons">
    <g:form action='edit'>
        <g:hiddenField name="id" value="${reportInstance?.id}" />
        <span class="button">
            <g:submitButton class="edit" name="Edit" />
        </span>
    </g:form>
    <g:form action='delete'>
        <g:hiddenField name="id" value="${reportInstance?.id}" />
        <span class="button">
            <g:submitButton class="delete" name="Delete"
                            onclick="return confirm('Are you sure?');" />
        </span>
    </g:form>
</div>
```

In `grails-app/views/report/edit.gsp`, change the `<form>` tag to

```
<g:form action='update'>
```

and convert the update button to a regular submit button:

```
<div class="buttons">
    <span class="button"><g:submitButton class="save" name="Update" /></span>
</div>
```

and move the delete button out of the form into its own form just below the main form:

```
<g:form action='delete'>
    <g:hiddenField name="id" value="${reportInstance?.id}" />
    <div class="buttons">
        <span class="button">
            <g:submitButton class="delete" name="Delete"
                            onclick="return confirm('Are you sure?');" />
        </span>
    </div>
</g:form>
```

`list.gsp` and `show.gsp` are fine as they are.

Testing

Now start the app:

```
$ grails run-app
```

and open http://localhost:8080/acltest/report/list

Login as user1/password1 and you should see the first page of results. But if you click on page 7 or higher, you'll see that you can only see a subset of the `Reports`. This illustrates one issue with using ACLs to restrict view access to instances; you would have to add joins in your query to the ACL database tables to get an accurate count of the total number of visible instances.

Click on any of the report instance links (e.g. http://localhost:8080/acltest/report/show/63) to verify that you can view the instance. You can test that you have no view access to the filtered instances by navigating to http://localhost:8080/acltest/report/show/83.

Verify that user1 has admin permission on report #11 by editing it and deleting it.

Verify that user1 doesn't have admin permission on report #13 by trying to editing or delete it and you should see the error page when you submit the form.

Logout (by navigating to http://localhost:8080/acltest/logout) and login as user2/password2. You should only see the first five reports. Verify that you can edit #5 but not any of the others, and that you can't delete any.

Finally. logout and login as admin/admin123. You should be able to view, edit, and delete all instances.

Database tables

```
select e.id, e.ace_order, c.class, oi.object_id_identity, e.audit_failure,
e.audit_success, e.granting, e.mask, s.sid
from acl_entry e
join acl_object_identity oi on e.acl_object_identity=oi.id
join acl_class c on oi.object_id_class=c.id
join acl_sid s on e.sid=s.id
```

4 Sample Application

Working with ACLs in Spring Security is complex but it will be easier to understand with a sample application. To help get you started, there's a Grails application that uses the plugin to test with. It's based on the Contacts application that comes with Spring Security. But where the Spring Security application uses SpringMVC, JDBC, etc., this application is 100% Grails. Download it from here.

Unpack the zip file, for example in `/opt/workspace/grails-contacts`

Upgrade the application to make sure it's compatible with the version of Grails you're using (note that all of the Spring Security plugins require at least version 2.0 of Grails) and start the app:

```
$ grails run-app
```

Open http://localhost:8080/grails-contacts/ in a browser to get started. The main functionality is at http://localhost:8080/grails-contacts/secure. The login page lists the various configured users and their passwords; the "rod" user is an admin and has full access and the other users have various grants and ownership.