M. Nardone, C. Scarioni, *Pro Spring Security*

https://doi-org.ezproxy.sfpl.org/10.1007/979-8-8688-0035-1_7

# 7. Business Object Security with ACLs

Massimo Nardone[1] and Carlo Scarioni[2]

(1) HELSINKI, Finland

(2) Surbiton, UK

This chapter introduces access control lists (ACLs) in the context of Spring Security.

Access control lists can be considered an extension to the business-level security rules reviewed in Chapter 6. This chapter, however, looks at more fine-grained rules to secure individual domain objects instead of the relatively coarse-grained rules used to secure method calls on services.

This means that ACLs are in charge of securing instances of domain classes (such as a `Forum` class, a `Cart` class, and so on), while the standard method-level rules secure entry points determined by methods (like a `Service` method or a `DAO` method).

Securing domain objects with ACLs is conceptually simple. The idea is that any user has a certain level of access (read, write, none, and so on) to each domain object. A user's level of access (*permissions*) to a particular domain object depends on the user or the role or group to which the user belongs.

## ACL Key Concepts

The source code of the Spring Security ACL module or inside the `.jar` file itself: `spring-security-acl-6.1.3.RELEASE.jar`. It is in the `src/main/resources` folder in the source code, which means it is in the root of the classpath.

Spring Security's domain object instance security capabilities center on the concept of an access control list (ACL). Every domain object instance in our system has its own ACL, and the ACL records details of who can and cannot work with that domain object. Spring Security provides three main ACL-related capabilities to your application.

- A way to retrieve ACL entries for our domain objects
- A way to ensure a given principal is permitted to work with our objects before methods are called
- A way to ensure a given principal is permitted to work with our objects after methods are called

Note that one of the main capabilities of the Spring Security ACL module is providing a high-performance way of retrieving ACLs.

This ACL repository capability is extremely important because every domain object instance in our system might have several access control entries, and each ACL might inherit from other ACLs in a tree-like structure.

Spring Security's ACL capability has been carefully designed to provide high-performance retrieval of ACLs, together with pluggable caching, deadlock-minimizing database updates, independence from ORM frameworks (we use JDBC directly), proper encapsulation, and transparent database updating.

The following are the main abstractions in Spring Security's ACL support.

- **Security identity (SID)** is an abstraction that represents a security identity in the system to be used by the ACL infrastructure. A security identity can be a user, role, group, and so forth. It maps to the `ACL_SID` table.

- **Access control entry (ACE)** represents an individual permission in the ACL, making relationships between objects, SIDs, and permissions. It maps to the `ACL_ENTRY` table.
- **Object identity** represents the identity of an individual domain object instance. They are the entities on which the permissions are set. It maps to the `ACL_OBJECT_IDENTITY` table.
- **Entry** stores the individual permissions assigned to each recipient. Columns include a foreign key to the `ACL_OBJECT_IDENTITY`, the recipient (i.e., a foreign key to ACL_SID), whether auditing or not and the integer bit mask that represents the actual permission being granted or denied.

The ACL system uses integer bit masking so that when you have 32 bits, you can switch on or off. Each of these bits represents a permission. By default, the permissions are read (bit 0), write (bit 1), create (bit 2), delete (bit 3), and administer (bit 4). You can implement your own permission instance so that the ACL framework operates without knowledge of your extensions.

You should understand that the number of domain objects in your system has no bearing on the fact that we have chosen to use integer bit masking. While you have 32 bits available for permissions, there could be billions of domain object instances to avoid mistakenly believing that one is needed for each potential domain object, which is not the case.

Now that you have a basic understanding of what the ACL system does and what it looks like at a table-structure level, you need to understand the ACL key interfaces.

- **ACL**: This is every domain object has one and only one ACL object, which internally holds the AccessControlEntry objects and knows the owner of the ACL, which does not refer directly to the domain object but instead to an ObjectIdentity. It is stored in the ACL_OBJECT_IDENTITY table.

- **AccessControlEntry** holds multiple AccessControlEntry objects, often abbreviated as ACEs in the framework. Each ACE refers to a specific tuple of permission, SID, and ACL. An ACE can also be granting or non-granting and contain audit settings. The ACE is stored in the ACL_ENTRY table.
- **Permission** represents a particular immutable bit mask and offers convenience functions for bit masking and outputting information. The basic permissions (bits from 0 to 4) are contained in the BasePermission class.
- **SID** is the abbreviation for *security identity*. The ACL module must refer to principals and GrantedAuthority instances, and the SID interface provides a level of indirection. Common classes include PrincipalSid, which represents the principal inside an Authentication object, and GrantedAuthoritySid. The SID information is stored in the ACL_SID table.
- **ObjectIdentity** internally represents each domain object within the ACL module. The default implementation is ObjectIdentityImpl.
- **AclService** retrieves the ACL applicable for a given ObjectIdentity. In the included implementation (JdbcAclService), retrieval operations are delegated to a LookupStrategy, which provides a highly optimized strategy for retrieving ACL information, using batched retrievals (BasicLookupStrategy) and supporting custom implementations that use materialized views, hierarchical queries, and similar performance-centric, non-ANSI SQL capabilities.
- **MutableAclService** presents a modified ACL for persistence; this interface is optional.

Note that our AclService and related database classes use ANSI SQL, which should work with all major databases. At the time of writing, the system had been successfully tested with Hypersonic SQL, PostgreSQL, Microsoft SQL Server, and Oracle.

The following is the Maven dependency in our code.

```
<dependency>

    <groupId>org.springframework.security</groupId>

    <artifactId>spring-security-acl</artifactId>

    <version>6.1.3</version>

</dependency>
```

To get started with Spring Security’s ACL capability, you must first store your ACL information somewhere. This means that an instantiation of a DataSource in Spring is needed. It is then injected into a JdbcMutableAclService and a BasicLookupStrategy instance.

Next, you need to populate the database with the four ACL-specific tables.

Finally, once you have created the required schema and instantiated JdbcMutableAclService, you need to ensure that our domain model supports interoperability with the Spring Security ACL package like ObjectIdentityImpl proves sufficient because it provides many ways in which it can be used.

If you use domain objects that contain a public serializable getId() method, it returns a type that is long or compatible with long (such as an int), and you may find that you need not give further consideration to ObjectIdentity issues.

In general, many parts of the ACL module rely on long identifiers, and if you do not use long (or an int, byte, and so on), you might need to re-implement several classes.

We avoid using and supporting non-long identifiers in Spring Security's ACL module, as longs are already compatible with all data-base sequences, are the most common identifier data type, and are of sufficient length to accommodate all common usage scenarios.

Listing 7-1 shows how to create an ACL or modify an existing one.

```
// Prepare the information to be in the access control entry (ACE)

ObjectIdentity oi = new ObjectIdentityImpl(Foo.class, new Long(44));

Sid sid = new PrincipalSid("Massimo");

Permission p = BasePermission.ADMINISTRATION;



// Create or update the relevant ACL

MutableAcl acl = null;
```

```
try {

acl = (MutableAcl) aclService.readAclById(oi);

} catch (NotFoundException nfe) {

acl = aclService.createAcl(oi);

}



// Granting permissions via an access control entry (ACE)

acl.insertAce(acl.getEntries().length, p, sid, true);

aclService.updateAcl(acl);
```

***Listing 7-1***
Example of ACL Java Code

In this simple example, prepare the information you want in the access control entry (ACE) with ObjectIdentityImpl.

Then, retrieve the ACL associated with the Foo domain object with identifier 44.

Next, add an ACE so a principal named "Massimo" can "administer" the object. The code fragment is relatively self-explanatory, except for the insertAce method, where the first argument determines the position in the ACL at which the new entry is inserted.

Then, add code to create or update the relevant ACL.

Finally, grant permissions via an access control entry (ACE) at the end of the existing ACEs. The final argument is a Boolean indicating whether the ACE is granting or denying, which it usually grants (true), but if it denies (false), the permissions are blocked.

Spring Security does not provide any special integration to automatically create, update, or delete ACLs as part of your DAO or repository operations. Instead, you must write code similar to the preceding example for your domain objects. You should consider using AOP on your services layer to automatically integrate the ACL information with your services layer operations. We have found this approach to be effective.

Once you have used the techniques described here to store some ACL information in the database, the next step is to use the ACL information as part of authorization decision logic, which can be done by writing your own AccessDecisionVoter or AfterInvocationProvider that (respectively) fires before or after a method invocation. Such classes would use AclService to retrieve the relevant ACL and then call Acl.isGranted(Permission[] permission, Sid[] sids, boolean administrativeMode) to decide whether permission is granted or denied.

Alternatively, you could use the following classes, which provide a declarative-based approach to evaluating ACL information at runtime, freeing you from needing to write any code.

- AclEntryVoter
- AclEntryAfterInvocationProvider
- AclEntryAfterInvocationCollectionFilteringProvider

## Summary

This chapter provided general information on how to use Spring Security’s support for ACLs.