Understanding Permissions in Apache Shiro

Shiro defines a Permission as a statement that defines an explicit behavior or action. It is a statement of raw functionality in an application and nothing more. Permissions are the lowest-level constructs in security polices, and they explicitly define only "what" the application can do.

They do *not* at all describe "who" is able to perform the action(s).

Some examples of permissions:

- Open a file
- View the '/user/list' web page
- Print documents
- Delete the 'jsmith' user

Defining "who" (users) is allowed to do "what" (permissions) is an exercise of assigning permissions to users in some way. This is always done by the application's data model and can vary greatly across applications.

For example, permissions can be grouped in a Role and that Role could be associated with one or more User objects. Or some applications can have a Group of users and a Group can be assigned a Role, which by transitive association would mean that all the Users in that Group are implicitly granted the permissions in the Role.

There are many variations for how permissions could be granted to users - the application determines how to model this based on the application requirements.

Related Content

Java Authorization Guide

Learn how Shiro handles access control in Java.
Read More >>

Web App Tutorial

Step-by-step tutorial for securing a web application with Shiro.
Read More >>

Getting Started

Resources, guides and tutorials for new Shiro users.
Read More >>

10-Minute Shiro Tutorial

Try Apache Shiro for yourself in under 10 minutes.
Read More >>

Wildcard Permissions

The above examples of permissions, "Open a file", "View the 'user/list' web page", etc are all valid permission statements. However, it would be very difficult computationally to interpret those natural language strings and determine if a user is allowed to perform that behavior or not.

So to enable easy-to-process yet still readable permission statements, Shiro provides powerful and intuitive permission syntax we refer to as the WildcardPermission.

Simple Usage

Let's you want to protect access to your company's printers such that some people can print to particular printers, while others can query what jobs are currently in the queue.

An extremely simple approach would be to use grant the user a "queryPrinter" permission. Then you could check to see if the user has the queryPrinter permission by calling:

```
subject.isPermitted("queryPrinter")
```

This is (mostly) equivalent to

```
subject.isPermitted( new WildcardPermission("queryPrinter") )
```

but more on that later.

The simple permission string may work for simple applications, but it requires you to have permissions like "printPrinter", "queryPrinter", "managePrinter", etc. You can also grant a user "*" permissions using the wildcard character (giving this permission construct its name), which means they have ***all*** permissions across *the entire application*.

But using this approach there's no way to just say a user has "all printer permissions". For this reason, Wildcard Permissions supports multiple *levels* of permissioning.

Multiple Parts

Wildcard Permissions support the concept of multiple *levels* or *parts*. For example, you could restructure the previous simple example by granting a user the permission

```
printer:query
```

The colon in this example is a special character used to delimit the next part in the permission string.

In this example, the first part is the domain that is being operated on (`printer`) and the second part is the action (`query`) being

performed. The other above examples would be changed to:

```
printer:print
printer:manage
```

There is no limit to the number of parts that can be used, so it is up to your imagination in terms of ways that this could be used in your application.

Multiple Values

Each part can contain multiple values. So instead of granting the user both the "printer:print" and "printer:query" permissions, you could simply grant them one:

```
printer:print,query
```

which gives them the ability to `print` and `query` printers. And since they are granted both those actions, you could check to see if the user has the ability to query printers by calling:

```
subject.isPermitted("printer:query")
```

which would return `true`

All Values

What if you wanted to grant a user *all* values in a particular part? It would be more convenient to do this than to have to manually list every value. Again, based on the wildcard character, we can do this. If the `printer` domain had 3 possible actions (`query`, `print`, and `manage`), this:

```
printer:query,print,manage
```

simply becomes this:

```
printer:*
```

Then, *any* permission check for "printer:XXX" will return `true`. Using the wildcard in this way scales better than explicitly listing actions since, if you added a new action to the application later, you don't need to update the permissions that use the wildcard character in that part.

Finally, it is also possible to use the wildcard token in any part of a wildcard permission string. For example, if you wanted to grant a user the "view" action across *all* domains (not just printers), you could grant this:

```
*:view
```

Then any permission check for "foo:view" would return `true`

Instance-Level Access Control

Another common usage of wildcard permissions is to model instance-level Access Control Lists. In this scenario you use three parts - the first is the *domain*, the second is the *action*(s), and the third is the instance(s) being acted upon.

So for example you could have

```
printer:query:lp7200
printer:print:epsoncolor
```

The first defines the behavior to `query` the `printer` with the ID `lp7200`. The second permission defines the behavior to `print` to the `printer` with ID `epsoncolor`. If you grant these permissions to users, then they can perform specific behavior on *specific instances*. Then you can do a check in code:

```
if ( SecurityUtils.getSubject().isPermitted("printer:query:lp7200") {
    // Return the current jobs on printer lp7200
}
```

This is an extremely powerful way to express permissions. But again, having to define multiple instance IDs for all printers does not scale well, particularly when new printers are added to the system. You can instead use a wildcard:

```
printer:print:*
```

This does scale, because it covers any new printers as well. You could even allow access to all actions on all printers:

```
printer:*:*
```

or all actions on a single printer:

```
printer:*:lp7200
```

or even specific actions:

```
printer:query,print:lp7200
```

The '*' wildcard and ',' sub-part separator can be used in any part of the permission.

Missing Parts

One final thing to note about permission assignments: missing parts imply that the user has access to all values corresponding to that part. In other words,

```
printer:print
```

is equivalent to

```
printer:print:*
```

and

```
printer
```

is equivalent to

```
printer:*:*
```

However, you can only leave off parts from the *end* of the string, so this:

```
printer:lp7200
```

is ***not*** equivalent to

```
printer:*:lp7200
```

Checking Permissions

While permission assignments use the wildcard construct quite a bit ("printer:print:*" = print to any printer) for convenience and scalability, permission **checks** at runtime should *always* be based on the most specific permission string possible.

For example, if the user had a UI and they wanted to print a document to the `lp7200` printer, you **should** check if the user is permitted to do so by executing this code:

```
if ( SecurityUtils.getSubject().isPermitted("printer:print:lp7200") ) {
    //print the document to the lp7200 printer
}
```

That check is very specific and explicitly reflects what the user is attempting to do at that moment in time.

The following however is much less ideal for a runtime check:

```
if ( SecurityUtils.getSubject().isPermitted("printer:print") ) {
    //print the document
}
```

Why? Because the second example says "You must be able to print to **any** printer for the following code block to execute". But remember that "printer:print" is equivalent to "printer:print:*"!

Therefore, this is an incorrect check. What if the current user does not have the ability to print to any printer, but they **do** have the ability to print to say, the `lp7200` and `epsoncolor` printers. Then the 2nd example above would never allow them to print to the `lp7200` printer even though they have been granted that ability!

So the rule of thumb is to use the most specific permission string possible when performing permission checks. Of course, the 2nd block above might be a valid check somewhere else in the application if you really did only want to execute the code block if the user was allowed to print to any printer (suspect, but possible). Your application will determine what checks make sense, but in general, the more specific, the better.

Implication, not Equality

Why is it that runtime permission checks should be as specific as possible, but permission assignments can be a little more generic? It is because the permission checks are evaluated by *implication* logic - not equality checks.

That is, if a user is assigned the `user:*` permission, this *implies* that the user can perform the `user:view` action. The string "user:*" is clearly not equal to "user:view", but the former implies the latter. "user:*" describes a superset of functionality of that defined by "user:view".

To support implication rules, all permissions are translated in to object instances that implement the `org.apache.shiro.authz.Permission` interface. This is so that implication logic can be executed at runtime and that implication logic is often more complex than a simple string equality check. All of the wildcard behavior described in this document is actually made possible by the `org.apache.shiro.authz.permission.WildcardPermission` class implementation. Here are some more wildcard permission strings that show access by implication:

```
user:*
```

implies the ability to also delete a user:

```
user:delete
```

Similarly,

```
user:*:12345
```

implies the ability to also update user account with ID 12345:

```
user:update:12345
```

and

```
printer
```

implies the ability to print to any printer

```
printer:print
```

Performance Considerations

Permission checks are more complex than a simple equals comparison, so runtime implication logic must execute for each assigned Permission. When using permission strings like the ones shown above, you're implicitly using Shiro's default `WildcardPermission` which executes the necessary implication logic.

Shiro's default behavior for Realm implementations is that, for every permission check (for example, a call to `subject.isPermitted`), *all* of the permissions assigned to that user (in their Groups, Roles, or directly assigned to them) need to be checked individually for implication. Shiro 'short circuits' this process by returning immediately after the first successful check occurs to increase performance, but it is not a silver bullet.

This is usually extremely fast when users, roles and permissions are cached in memory when using a proper CacheManager, which Shiro does support for Realm implementations. Just know that with this default behavior, as the number of permissions assigned to a user or their roles or groups increase, the time to perform the check will necessarily increase.

If a Realm implementor has a more efficient way of checking permissions and performing this implication logic, especially if based on the applicaton's data model, they should implement that as part of their Realm isPermitted* method implementations. The default Realm/WildcardPermission support exists to cover 80-90% of most use cases, but it might not be the best solution for applications that have massive amounts of permissions to store and/or check at runtime.

Lend a hand with documentation

While we hope this documentation helps you with the work you're doing with Apache Shiro, the community is improving and expanding the documentation all the time. If you'd like to help the Shiro project, please consider corrected, expanding, or adding documentation where you see a need. Every little bit of help you provide expands the community and in turn improves Shiro.

The easiest way to contribute your documentation is to send it to the User Forum or the User Mailing List.