Apache Shiro Realms

A `Realm` is a component that can access application-specific security data such as users, roles, and permissions. The `Realm` translates this application-specific data into a format that Shiro understands so Shiro can in turn provide a single easy-to-understand Subject programming API no matter how many data sources exist or how application-specific your data might be.

Realms usually have a 1-to-1 correlation with a data source such as a relational database, LDAP directory, file system, or other similar resource. As such, implementations of the `Realm` interface use data source-specific APIs to discover authorization data (roles, permissions, etc), such as JDBC, File IO, Hibernate or JPA, or any other Data Access API.

A Realm is essentially a security-specific DAO

Because most of these data sources usually store both authentication data (credentials such as passwords) as well as authorization data (such as roles or permissions), every Shiro `Realm` can perform *both* authentication and authorization operations.

Realm Configuration

If using Shiro's INI configuration, you define and reference `Realms` like any other object in the `[main]` section, but they are configured on the `securityManager` in one of two ways: explicitly or implicitly.

Explicit Assignment

Based on knowledge of INI configuration thus far, this is an obvious configuration approach. After defining one or more Realms, you set them as a collection property on the `securityManager` object.

For example:

```
fooRealm = com.company.foo.Realm
barRealm = com.company.another.Realm
bazRealm = com.company.baz.Realm

securityManager.realms = $fooRealm, $barRealm, $bazRealm
```

Explicit assignment is deterministic - you control exactly which realms are used as well as *the order* that they will be used for authentication and authorization. Realm ordering effects are described in detail in the Authentication chapter's Authentication Sequence section.

Implicit Assignment

Not Preferred
Implicit assignment can cause unexpected behavior if you change the order in which realms are defined. It is recommended that you avoid this approach and use Explicit Assignment, which has deterministic behavior. It is likely Implicit Assignment will be deprecated/removed from a future Shiro release.

If for some reason you don't want to explicitly configure the `securityManager.realms` property, you can allow Shiro to detect all configured realms and assign them to the `securityManager` directly.

Using this approach, realms are assigned to the `securityManager` instance in the *order that they are defined*.

That is, for the following `shiro.ini` example:

```
blahRealm = com.company.blah.Realm
fooRealm = com.company.foo.Realm
barRealm = com.company.another.Realm
```

```
# no securityManager.realms assignment here
```

basically has the same effect as if the following line were appended:

```
securityManager.realms = $blahRealm, $fooRealm, $barRealm
```

However, realize that with implicit assignment, just the order that the realms are defined directly affects how they are consulted during authentication and authorization attempts. If you change their definition order, you will change how the master `Authenticator`'s Authentication Sequence functions.

For this reason, and to ensure deterministic behavior, we recommend using Explicit Assignment instead of Implicit Assignment.

Realm Authentication

Once you understand Shiro's master Authentication workflow, it is important to know exactly what happens when the `Authenticator` interacts with a `Realm` during an authentication attempt.

Supporting `AuthenticationTokens`

As mentioned in the authentication sequence, just before a `Realm` is consulted to perform an authentication attempt, its `supports` method is called. If the return value is `true`, only then will its `getAuthenticationInfo(token)` method be invoked.

Typically a realm will check the type (interface or class) of the submitted token to see if it can process it. For example, a Realm that processes biometric data may not understand `UsernamePasswordTokens` at all, in which case it would return `false` from the `supports` method.

Handling supported `AuthenticationTokens`

If a `Realm` `supports` a submitted `AuthenticationToken`, the `Authenticator` will call the Realm's getAuthenticationInfo(token) method. This effectively represents an authentication attempt with the `Realm's` backing data source. The method, in order:

1. Inspects the `token` for the identifying principal (account identifying information)
2. Based on the `principal`, looks up corresponding account data in the data source
3. Ensures that the token's supplied `credentials` matches those stored in the data store
4. If the credentials match, an AuthenticationInfo instance is returned that encapsulates the account data in a format Shiro understands
5. If the credentials DO NOT match, an AuthenticationException is thrown

This is the highest-level workflow for all Realm `getAuthenticationInfo` implementations. Realms are free to do whatever they want during this method, such as record the attempt in an audit log, update data records, or anything else that makes sense for the authentication attempt for that data store.

The only thing required is that, if the credentials match for the given principal(s), that a non-null `AuthenticationInfo` instance is returned that represents Subject account information from that data source.

Save Time
Implementing `Realm` interface directly might be time consuming and error prone. Most people choose to subclass the AuthorizingRealm abstract class instead of starting from scratch. This class implements common authentication and authorization workflow to save you time and effort.

Credentials Matching

In the above realm authentication workflow, a Realm has to verify that the Subject's submitted credentials (e.g. password) must match the credentials stored in the data store. If they match, authentication is considered successful, and the system has verified the end-user's identity.

Realm Credentials Matching
It is each Realm's responsibility to match submitted credentials with those stored in the Realm's backing data store, and not the `Authenticator's` responsibility. Each `Realm` has intimate knowledge of credentials format and storage and can perform detailed credentials matching, whereas the `Authenticator` is a generic workflow component.

The credentials matching process is nearly identical in all applications and usually only differs by the data compared. To ensure this process is pluggable and customizable if necessary, the AuthenticatingRealm and its subclasses support the concept of a CredentialsMatcher to perform the credentials comparison.

After discovering account data, it and the submitted `AuthenticationToken` are presented to a `CredentialsMatcher` to see if what was submitted matches what is stored in the data store.

Shiro has some `CredentialsMatcher` implementations to get you started out of the box, such as the SimpleCredentialsMatcher and HashedCredentialsMatcher implementations, but if you wanted to configure a custom implementation for custom matching logic, you could do so directly:

```
Realm myRealm = new com.company.shiro.realm.MyRealm();
CredentialsMatcher customMatcher = new com.company.shiro.realm.CustomCredentialsMatcher();
myRealm.setCredentialsMatcher(customMatcher);
```

Or, if using Shiro's INI configuration:

```
[main]
...
customMatcher = com.company.shiro.realm.CustomCredentialsMatcher
myRealm = com.company.shiro.realm.MyRealm
myRealm.credentialsMatcher = $customMatcher
...
```

Simple Equality Check

All of Shiro's out-of-the-box `Realm` implementations default to using a SimpleCredentialsMatcher. The `SimpleCredentialsMatcher` performs a plain direct equality check of the stored account credentials with what was submitted in the `AuthenticationToken`.

For example, if a UsernamePasswordToken was submitted, the `SimpleCredentialsMatcher` verifies that the password submitted is exactly equal to the password stored in the database.

The `SimpleCredentialsMatcher` performs direct equality comparisons for more than just Strings though. It can work with most common byte sources, such as Strings, character arrays, byte arrays, Files and InputStreams. See its JavaDoc for more.

Hashing Credentials

Instead of storing credentials in their raw form and performing raw/plain comparisons, a much more secure way of storing end-user's credentials (e.g. passwords) is to one-way hash them first before storing them in the data store.

This ensures that end-users' credentials are never stored in their raw form and that no one can know the original/raw value. This is a much more secure mechanism than plain-text or raw comparisons, and all security-conscious applications should favor this approach over non-hashed storage.

To support these preferred cryptographic hashing strategies, Shiro provides HashedCredentialsMatcher implementations to be configured on realms instead of the aforementioned `SimpleCredentialsMatcher`.

Hashing credentials and the benefits of salting and multiple hash iterations are outside the scope of this `Realm` documentation, but definitely read the HashedCredentialsMatcher JavaDoc which covers these principles in detail.

Hashing and Corresponding Matchers

So how do you configure a Shiro-enabled application to do this easily?

Shiro provides multiple `HashedCredentialsMatcher` subclass implementations. You must configure the specific implementation on your realm to match the hashing algorithm you use to hash your users' credentials.

For example, let's say your application uses username/password pairs for authentication. And due to the benefits of hashing credentials described above, let's say you want to one-way hash a user's password using the SHA-256 algorithm when you create a user account. You would hash the user's entered plain-text password and save that value:

```
import org.apache.shiro.crypto.hash.Sha256Hash;
import org.apache.shiro.crypto.RandomNumberGenerator;
import org.apache.shiro.crypto.SecureRandomNumberGenerator;
...

//We'll use a Random Number Generator to generate salts.  This
//is much more secure than using a username as a salt or not
//having a salt at all.  Shiro makes this easy.
//
//Note that a normal app would reference an attribute rather
//than create a new RNG every time:
RandomNumberGenerator rng = new SecureRandomNumberGenerator();
Object salt = rng.nextBytes();

//Now hash the plain-text password with the random salt and multiple
//iterations and then Base64-encode the value (requires less space than Hex):
String hashedPasswordBase64 = new Sha256Hash(plainTextPassword, salt, 1024).toBase64();

User user = new User(username, hashedPasswordBase64);
//save the salt with the new account.  The HashedCredentialsMatcher
//will need it later when handling login attempts:
user.setPasswordSalt(salt);
userDAO.create(user);
```

Since you're `SHA-256` hashing your user's passwords, you need to tell Shiro to use the appropriate `HashedCredentialsMatcher` to match your hashing preferences. In this example, we create a random salt and perform 1024 hash iterations for strong security (see the `HashedCredentialsMatcher` JavaDoc for why). Here is the Shiro INI configuration to make this work:

```
[main]
...
credentialsMatcher = org.apache.shiro.authc.credential.Sha256CredentialsMatcher
# base64 encoding, not hex in this example:
credentialsMatcher.storedCredentialsHexEncoded = false
credentialsMatcher.hashIterations = 1024
# This next property is only needed in Shiro 1.0.  Remove it in 1.1 and later:
```

```
credentialsMatcher.hashSalted = true

...
myRealm = com.company.....
myRealm.credentialsMatcher = $credentialsMatcher
...
```

`SaltedAuthenticationInfo`

The last thing to do to ensure this works is that your `Realm` implementation must return a SaltedAuthenticationInfo instance instead of a normal `AuthenticationInfo` one. The `SaltedAuthenticationInfo` interface ensures that the salt that you used when you created the user account (e.g. the `user.setPasswordSalt(salt);` call above) can be referenced by the `HashedCredentialsMatcher`.

The `HashedCredentialsMatcher` needs the salt in order to perform the same hashing technique on the submitted `AuthenticationToken` to see if the token matches what you saved in the data store. So if you use salting for user passwords (and you should!!!), ensure your `Realm` implementation represents that by returning `SaltedAuthenticationInfo` instances.

Disabling Authentication

If for some reason, you don't want a Realm to perform authentication for a data source (maybe because you only want the Realm to perform authorization), you can disable a Realm's support for authentication entirely by always returning `false` from the Realm's `supports` method. Then your realm will never be consulted during an authentication attempt.

Of course at least one configured `Realm` needs to be able to support AuthenticationTokens if you want to authenticate Subjects.

Realm Authorization

TBD

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.