Service Authentication

Problem

To access the data storage in the Resource Tree and/or the JCR Repository authentication is required to properly setup access control and guard sensitive data from unauthorized access. For regular request processing this authentication step is handled by the Sling Authentication subsystem.

On the other hand there are also some background tasks to be executed with access to the resources. Such tasks cannot in general be configured with user names and passwords: Neither hard coding the passwords in the code nor having the passwords in – more or less – plain text in some configuration is considered good practice.

To solve this problem for services to identify themselves and authenticate with special users properly configured to support those services.

The solution presented here serves the following goals:

- Prevent over-use and abuse of administrative ResourceResolvers and/ro JCR Sessions
- Allow services access to ResourceResolvers and/or JCR Sessions without requiring to hard-code or configure passwords
- Allow services to use *service users* which have been specially configured for service level access (as is usually done on unixish systems)
- Allow administrators to configure the assignment of service users to services

Concept

A *Service* is a piece or collection of functionality. Examples of services are the Sling queuing system, Tenant Administration, or some Message Transfer System. Each service is identified by a unique *Service Name*. Since a service will be implemented in an OSGi bundle (or a collection of OSGi bundles), services are named by the bundles providing them.

A Service may be comprised of multiple parts, so each part of the service may be further identified by a *Subservice Name*. This Subservice Name is optional, though. Examples of *Subservice Name* are names for subsystems in a Message Transfer System such as accepting messages, queueing messages, delivering messages.

Ultimately, the combination of the *Service Name* and *Subservice Name* defines the *Service ID*. It is the *Service ID* which is finally mapped to a Resource Resolver and/or JCR Repository user ID for authentication.

Thus the actual service identification (service ID) is defined as:

```
service-id = service-name [ ":" subservice-name ] .
```

The service-name is the symbolic name of the bundle providing the service.

Example: Tenant Administration

Tenant Administration mostly deals with creating and managing groups and some other user administration tasks. Instead of just using an administrative session for Tenant administration this feature could define itself as being the tenant-admin service and leverage a properly configured Tenant Administration account.

Example: Mail Transfer System

Consider a Mail Transfer System which may be comprised of the following sub systems:

- Accepting mail for processing — for example the SMTP server daemon
- Queing and processing the messages
- Delivering messages to mailboxes

You could conceive that all these functions serve different purposes and thus should have different access rights to the repository to persist messages while they are being processed.

Using the Service Authentication framework, the Mail Transfer System would be consituting the `mta` service. The sub systems would be called `smtp`, `queue`, and `deliver`.

Thus the SMTP server daemon would be represented by a user for the `mta:smtp` Service. queueing with `mta:queue`, and delivery with `mta:deliver`.

Implementation

The implementation in Sling of the *Service Authentication* concept described above consists of three parts:

ServiceUserMapper

The first part is a new OSGi Service `ServiceUserMapper`. The ServiceUserMapper service allows for mapping *Service IDs* comprised of the *Service Names* defined by the providing bundles and optional *Subservice Name* to ResourceResolver and/or JCR Repository user IDs or principals (SLING-6939). This mapping is configurable such that system administrators are in full control of assigning users to services.

The ServiceUserMapper defines the following API:

```
String getServiceUserID(Bundle bundle, String subServiceName);
```

The implementation uses two fallbacks in case no mapping can be found for the given subServiceName

1. Use user/principal mapping for the serviceName only (not considering subServiceName)
2. Use default user (if one is configured in the OSGi configuration for PID org.apache.sling.serviceusermapping.impl.ServiceUserMapperImpl).
3. Use default mapping (if it is enabled in the OSGi configuration for PID org.apache.sling.serviceusermapping.impl.ServiceUserMapperImpl) which looks up a user with id serviceuser--<bundleId>[--<subservice-name>] (since Service User Mapper 1.3.0, SLING-6227).

In addition a service named `ServiceUserMapped` is registered for each bundle and subservice name for which a service user mapping is explicitly configured (SLING-4312). By explicitly defining a (static) reference towards ServiceUserMapped one can defer starting the service until that service user mapping is available. Please note, that the two last default mappings are not represented as a ServiceUserMapped service and therefore the above mentioned reference does not work prior to version 1.4.4 (SLING-7930). Also since version 1.4.4 the ServiceUserMapped is only registered in case there is a valid user/principal found in the underlying repository which is given in the mapping (SLING-7930).

ResourceResolverFactory

The second part is support for service access to the Resource Tree. To this avail, the `ResourceResolverFactory` service is enhanced with a new factory method

```
ResourceResolver getServiceResourceResolver(Map<String, Object> authenticationInfo)
    throws LoginException;
```

This method allows for access to the resource tree for services where the service bundle is the bundle actually using the ResourceResolverFactory service. The optional Subservice Name may be provided as an entry in the authenticationInfo map.

In addition to having new API on the `ResourceResolverFactory` service to be used by services, the ResourceProviderFactory service is updated with support for Service Authentication: Now new API is required, though but additional properties are defined to convey the service to authenticate for.

The default implementation leverages `ServiceUserMapper.getServiceUserID()` to resolve the right user id and throws a LoginException in case no mapping has been setup (and none of the fallbacks returned a user id `!= null` either).

SlingRepository

The third part is an extension to the `SlingRepository`service interface to support JCR Repository access for services:

```
Session loginService(String subServiceName, String workspace)
    throws LoginException, RepositoryException;
```

This method allows for access to the JCR Repository for services where the service bundle is the bundle actually using the SlingRepository service. The additional Subservice Name may be provided with the subServiceName parameter.

Configuration

Service User Mappings

For each service/subservice name combination an according mapping needs to be provided. The mapping binds a service name/subservice name to a JCR system user or a principal (since version 1.3.4, see SLING-6939). This is configured through an OSGi configuration for the factory configuration with PID org.apache.sling.serviceusermapping.impl.ServiceUserMapperImpl.amended (added in SLING-3578). There you can set one configuration property named user.mapping getting a String array as value where each entry must stick to the following format:

```
<service-name>[:<subservice-name>]=<authorizable id of a JCR system user>|"["<principal name>{","<princi
```

The principal based mapping (enclosed in square brackets) is in general faster than the authorizable id based one. Also it allows to directly reference multiple principals which are all considered to calculate the access rights for the configured service session/resource resolver. The principal based mapping does not consider group memberships of the underlying users, though.

The according user/principal must exist at the point in time where ResourceResolverFactory.getServiceResourceResolver(...) or SlingRepository.loginService(...) is called. If you rely on one of those methods in your activate method of an OSGi component you should make sure that you defer starting your OSGi component until the according service user mapping is in place. For that you can reference the OSGi service ServiceUserMapped (see Section ServiceUserMapper above for details), optionally with a target filter on property subServiceName (in case such a subservice name is used). The service ServiceUserMapped does not expose any methods but is only a marker interface exclusively used to defer starting of other OSGi components.

Example OSGi DS Component

```
@Component(
    reference = {
        // this waits with the activation of this component until a service user mapping with the service name = current
        // you can leave out "target" if the sub service name is not used.
        // Please note that this only waits for the mapping to be available, it does not wait for the service user itself to l
        @Reference(name ="scriptsServiceUser", target="(subServiceName=my-subservice-name)", service=Service
    }
)
class MyComponent {
}
```

Deprecation of administrative authentication

Originally the ResourceResolverFactory.getAdministrativeResourceResolver and SlingRepository.loginAdministrative methods have been defined to provide access to the resource tree and JCR Repository. These methods proved to be inappropriate because they allow for much too broad access.

Consequently these methods are being deprecated and will be removed in future releases of the service implementations.

The following methods are deprecated:

- ResourceResolverFactory.getAdministrativeResourceResolver
- ResourceProviderFactory.getAdministrativeResourceProvider
- SlingRepository.loginAdministrative

The implementations we have in Sling's bundle will remain implemented in the near future. But there will be a configuration switch to disable support for these methods: If the method is disabled, a LoginException is always thrown from these methods. The JavaDoc of the methods is extended with this information.

Whitelisting bundles for administrative login

In order to be able to manage few (hopefully legit) uses of the above deprecated methods, a whitelisting

mechanism was introduced with SLING-5153 (*JCR Base 2.4.2*).

The recommended way to whitelist a bundle for administrative login is via a *whitelist fragment configuration*. It can be created as an OSGi factory configuration with the factoryPID org.apache.sling.jcr.base.internal.LoginAdminWhitelist.fragment. E.g. a typical configuration file might be called org.apache.sling.jcr.base.internal.LoginAdminWhitelist.fragment-myapp.config and could look as follows:

```
whitelist.name="myapp"
whitelist.bundles=[
   "com.myapp.core",
   "com.myapp.commons"
]
```

Property	Type	Default	Description
whitelist.name	String	[unnamed]	Purely informational property that allows easy identification of different fragments.
whitelist.bundles	String[]	[]	An array of bundle symbolic names that should be allowed to make use of the administrative login functionality.

All configured whitelist fragments are taken into account. This makes it easy to separate whitelists for different application layers and purposes.

For example, some Sling bundles need to be whitelisted, which could be done in a whitelist fragment named sling. In addition myapp adds a whitelist fragment called myapp. For integration tests and additional whitelist fragment myapp-integration-testing may be added.

Furthermore, there is a global configuration with PID org.apache.sling.jcr.base.internal.LoginAdminWhitelist, which should only be used in exceptional cases. It has a switch to turn administrative login on globally (whitelist.bypass) and it allows supplying a regular expression to whitelist matching bundle symbolic names (whitelist.bundles.regexp).

The regular expression is most useful for running PaxExam based tests, where bundle symbolic names follow a set pattern but have randomly generated parts.

Example: to whitelist all bundles generated by PaxExam a configuration file named org.apache.sling.jcr.base.internal.LoginAdminWhitelist.config might look as follows:

```
whitelist.bypass=B"false"
whitelist.bundles.regexp="^PAXEXAM.*$"
```

The configuration PID is org.apache.sling.jcr.base.internal.LoginAdminWhitelist. It supports the following configuration properties.

Property	Type	Default	Description
whitelist.bypass	Boolean	false	Allow all bundles to use administrative login. This is **NOT** recommended for production and warnings will be logged.
whitelist.bundles.regexp	String	""	A regular expression that whitelists all matching bundle symbolic names. This is **NOT** recommended for production and warnings will be logged.

Last modified by Konrad Windszus on Wed Dec 12 10:07:41 2018 +0100