How to Implement Security Tests for Resource Controllers

Overview

Motivation

ABAP in Eclipse offers one single service interface which allows access to all existing natively integrated resources in the backend. By use of this interface, it is currently possible to create, read, update, delete and activate development fragments. It is therefore clear that ***proper authorization checks shall be enforced for all resource controllers***, especially for all kind of modifications such as creating, changing or deleting development objects.

There are currently more than 15 domains with up to 7 different resource controllers assigned. Each of them supports one or more HTTP methods such as GET, PUT, DELETE or POST whereas POST can handle different actions such as lock, unlock, activate, check, etc.. This shows that we will end up with a large number of tests, probably more than 100 calls in total.

Wouldn't it be great if we could automate these tests and fulfill the following requirements:

- Maximize the flexibility to adjust the security tests tailored to the specific needs of certain resource controllers
- Keep the development effort to develop ABAP Unit tests for security as low as possible
- Follow a generic approach in order to see which implementations fail to comply with general rules

Often you hear the statement "yes, we already call function module **RS_ACCESS_PERMISSION** which will do the correct authorization check". This may be or may not be true. Security shall be tested. It is not sufficiant to rely on assumptions!

Discussion: Test Concepts

Several concepts have been discussed (see comments):

1. Do manual tests and set a break-point at an authority-check statement. Manipulate the return code in order to see what happens if the authorization is missing
   - It turns out that you have to fill out a very long list of cases you have tested.
   - If you want to reproduce a test in order to show it to someone, the effort is higher than for an automated test.
2. Make the code completely testable (e.g. mock **RS_ACCESS_PERMISSION**) and implement normal ABAP Unit tests
   - This may be the favorite method for completely new code because it is fast and you would only test what's essential for the test
   - The effort to make all code testable is too high, and there may be more and more code needed to be made testable
3. Record authorization traces using SM01 in the system client
   - You have to activate the traces globally.
   - You have to manually or automatically read and interpret the authorization traces.
   - It may still be possible that the return code after the authorization check is not evaluated correctly.
4. Implement an integration test by invoking the service API from outside by simulating different users
   - This would require an additional test client to provide testable functionality. It would be slower than a native ABAP Unit test.
   - If the API is changed, all tests have to be adjusted accordingly.
   - This test will also answer the question whether the service API is consistent. As an integrative test, it simulates the real situation.
   - For the tests, a special setup is necessary (users + RFC destinations)

Finally, we have decided to implement option 4. As you will see, the implementation efforts will be moderate.

Test Framework for Unit Tests

The test framework consists of two classes in package **SADT_REST**: a test client and a basis class for security tests.

Test Client: CL_ADT_RC_TEST_CLIENT_INTEGR

By using the abstract signature of the main class, this class implements an RFC client for integration tests. It provides HTTP methods which are used in the ADT framework. With a parameter, you can simulate different user roles for security tests.

Public Interface:

Method	Description	Parameters
GET_INSTANCE	Return instance of a test client. Currently, there is only one RFC client	**CLIENT**: Kind of test client to be instantiated. Currently only CL_ADT_RC_TEST_CLIENT_INTEGR-CO_CLIENT-RFC_CLIENT **HTTP-Version**: Specifies the HTTP version (default HTTP/1.1)
GET	The method will invoke the calls of the client, e.g. RFC calls to the function SADT_REST_RFC_ENDPOINT	By use of the parameter USER_PROFILE, you can simulate calls in different user contexts: • ***co_user_profile-display_none*** Simulates a user which has at least one development authorization to display development objects but which may not display any of the objects being subject of the unit tests. • ***co_user_profile-display_all*** Simulates a user which has full authorization to display all development objects but which may not modify, create or delete any objects. • ***co_user_profile-modify_all*** Simulates a user which has full authorization to modify, create, delete or activate all development objects.
PUT	(see GET)	(see GET)
DELETE	(see GET)	(see GET)
POST	(see GET)	(see GET)

Basis Class with Test Methods: CL_ADT_RC_TEST_AUTH_BASE

This abstract class implements predefined security tests for any kind of resource. It is recommended to implement the security tests by using this basis class for ABAP unit tests for security.

Find more information how the HTTP status is related to exception classes here.

Test method	Description	Authorization	Expected HTTP status	Purpose / Background	
SUCCEED_GET_OBJECT	The test invokes the GET method for a user which is authorized to read data.	Read all development objects	**200** (OK) **405** (Method not supported)	This test method is not a security test! It is necessary in order to check whether the resource can be read in general. If this is not the case, maybe the whole test class implementation is incorrect.	
FAIL_GET_OBJECT	The test invokes the GET method for a user which is not authorized to read data.	Read for a non-existing development object (Comment: The specific user has authorization to display specific development objects, e.g. for a package that does not exist. This precondition is necessary in order to pass the check against display authorization for any object ("dummy") at the entry point of the service API.)	**403** (Forbidden) **405** (Method not supported)	The test checks whether the authority check for display authorization works fine. A user which is not authorized to display data should not be able to invoke the GET method successfully.	

FAIL_LOCK_OBJECT_FOR_CREATE	The test invokes the POST method to lock a resource for a user which is not authorized to create data.	Read all development objects (No authorization for create)	**403** (Forbidden) **405** (Method not supported)	Before an object can be created, a logical lock needs to be set. A call of the CREATE method of the resource controller without a valid lock handle will fail. It is therefore essential to check if the lock method implements the right authorization check.	
FAIL_LOCK_OBJECT_FOR_UPDATE	The test invokes the POST method to lock a resource for a user which is not authorized to change data.	Read all development objects (No authorization for change)	**403** (Forbidden) **405** (Method not supported)	Before an object can be created, a logical lock needs to be set. A call of the UPDATE method of the resource controller without a valid lock handle will fail. It is therefore essential to check if the lock method implements the right authorization check.	
FAIL_PUT_OBJECT	The test will first read the object data first by use of an authorized call of the GET method. After a lock is set on the object, it invokes the PUT method and tries to rewrite the data by use of a valid lock handle.	Read all development objects (No authorization for change)	**403** (Forbidden) **423** (Invalid Lock Handle) **405** (Method not supported)	A lock handle is only valid in a user-specific context. It is therefore not possible to use a foreign lock handle and use PUT or POST to change or create an object. This kind of "attack" is therefore only possible if the same user locks an object and the authorization of the user is revoked afterwards. However, the mentioned methods shall implement the check of the lock handle correctly and raise the correct exception. The test expects that the method fails with an invalid lock handle, but a status code 403 would also be a correct result.	
FAIL_CREATE_OBJECT	The test will first read the object data first by use of an authorized call of the GET method. After a lock is set on the object, it invokes the POST method and tries to recreate the data by use of a valid lock handle.	Read all development objects (No authorization for create)	**403** (Forbidden) **405** (Method not supported) **423** (Invalid Lock Handle)	see comment for method FAIL_PUT_OBJECT	
FAIL_DELETE_OBJECT	After a lock is set on the object, it invokes the DELETE method and tries to delete the data by use of a valid lock handle.	Read all development objects	**403** (Forbidden) **405** (Method not supported) **423** (Invalid Lock Handle)	see comment for method FAIL_PUT_OBJECT	

Set up Users and RFC destinations

This chapter is only relevant in the case that the test setup is still missing.

You can find basic information regarding authorization checks for workbench tools under Relevant Authorization Objects

Test Setup

User profile	Activities for S_DEVELOP	User Profile Constant CL_ADT_RC_TEST_CLIENT_INTEGR	RFC Destination Constant LCL_TEST_CLIENT_RFC	RFC Destination for BAP (Configuration may change!)
Read, create, change, delete, activate all development objects	01, 02, 03, 06, 07	CO_USER_PROFILE-MODIFY_ALL	CO_RFC_DEST-MODIFY_ALL	BAPCLNT100
Read all development objects	03	CO_USER_PROFILE-DISPLAY_ALL	CO_RFC_DEST-DISPLAY_ALL	BAP_SEC_TEST_DISPLAY
Read a non-existing development object	(03)	CO_USER_PROFILE-DISPLAY_NONE	CO_RFC_DEST-DISPLAY_NONE	BAP_SEC_TEST_NO_DISPLAY

Step 1: Create Users

In the setup described above, you need two additional users with special authorizations (e.g. ANZ_RFC and AIE_NO_DISP). Both of them shall be authorized for RFC-calls to the function SADT_REST_RFC_ENDPOINT. As one of them is authorized to display any development object, it is convenient to copy the user ANZEIGER or any other pure display user to a new user (e.g. ANZ_RFC).

General procedure:

1. By use of transaction PFCG, create one or more PFCG roles (example: ZAIE_RFC)
    a. Go to the tab *Authorizations* and press the button *Change authorization data* (confirm any kind of popup)
    b. Press the button to *enter an authorization object manually*
    c. Assign the authorization objects needed (example: S_RFC, or S_DEVELOP)
    d. Expand the node of the authorization object and assign the field values by clicking on the orange change icon. For most of the fields, you may enter just a wildcard, the Activity can be selected explicitly
2. Copy or create a user by use of transaction SU01.
    a. Go to the tab Roles and assign the roles created above
    b. Save the User

Authorizations needed:

User Profile	Procedure
CO_RFC_DEST-MODIFY_ALL	Just use the logon user which shall be authorized to create, change, display, delete and activate objects
CO_USER_PROFILE-DISPLAY_ALL	• Copy a pure display user. • By use of the general procedure described above, add authorization for Authorization object S_RFC *Activity* Execute *Name of RFC to be protected* * *Type of RFC object to be prote* All values

CO_USER_PROFILE-DISPLAY_NONE	• Create a new user or copy an existing one (in this case, remove all existing authorization profiles and roles first) • By use of the general procedure described above, add authorization for authorization object S_RFC *Activity* Execute *Name of RFC to be protected* * *Type of RFC object to be prote* All values • By use of the general procedure described above, add authorization for authorization object S_DEVELOP *Activity* Display *Package* PACK_DOES_NOT_EXIST *Object name* * *Object type* * *Authorization group ABAP/4 pro* *

Step 2: Create RFC destinations

According to your user setup, you probably need two special RFC destinations, e.g. BAP_SEC_TEST_DISPLAY and BAP_SEC_TEST_NO_DISPLAY (see above).

General procedure:

By use of transaction SM59, create a new RFC destination of type '3' - ABAP Connections

1. Enter a description, enter target host and system number (compare with existing RFC destination)
2. On the tab *logon and security*, assign the user and enter the password
3. Save the RFC destination
4. Use the button *connection test* to check if the RFC destination is set up correctly

How to implement the test

For every single resource URI, an own test class needs to be implemented. All tests for a given resource should be grouped in an own test program (see Security Checklist for Implementation of Resource Controllers). Every test class needs to be derived from the base class cl_adt_rc_test_auth_base.

The test implementation must redefine the following methods which are abstract in the basis class:

Method	Purpose	How to Implement
GET_URI	This method will generate an URI of the resource to be tested. For every test, there will be only one single URI. Query parameters are added automatically by the test.	Use the URI builder classes to generate URI's.
GET_OBJECT_NAMES	This method returns object names for resources to be read (object_name_existing) or to be created object_name_non_existing. As a test preparation, you also need to create test objects (e.g. a test domain) in the home package. A constant CO_NUMBER_PLACEHOLDER can be used as a placeholder for a generated number which is increased for every test. If the CREATE method of the test does not fail, it will create objects, and it is therefore convenient to use a new name every time the test is being executed. A third parameter to be returned is the lock handle type. The parameter is necessary in order to generate valid lock handles for the test.	Create a test object and return its name in the implementation as *OBJECT_NAME_EXISTING*. Use a prefix for objects which are not yet existing but may be created by the create method if the test fails. Return the correct lock handle type for the given object.
SET_CONTENT	This method is the most complex one in the test. As the test tries to create new objects, it is necessary to pass valid object data in order to get the test running. Because every object type uses different data types, it is necessary to use the specific data type for the resource to be created. Usually, it is convenient to call method GET_OBJECT and read the existing object as a template for a new object to be created. In the metadata, the name is set to the name OBJECT_NAME_NON_EXISTING. The package may be set to $TMP in order to bypass any kind of transport errors.	The method must be redefined. Finally, the method *SET_CONTENT_FOR_CREATE* has to be called in order to set the object data with a content handler. If just the request header is sufficient in order to create the object, the attribute *NO_CONTENT_FOR_CREATE* can be set to abap_true.

The following methods are also available for tests:

Method	Purpose
SET_CONTENT_FOR_CREATE	This method accepts a content handler and content data to be used by the create method of the test. It shall be used in method SET_CONTENT in order to pass the content to the test class.
SET_CONTENT_DUMMY_SOURCE	There is a large number of resources which are using plain text. Even though plain text cannot be created, the right data format is needed for the create method of the test. The method SET_CONTENT_DUMMY_SOURCE will set a dummy source text as a message body for the create method.

Example

```
*--------------------------------------------------------------------------------------------------------------------
Resource: Meta data of a domain
* URI: /sap/bc/adt/ddic/domains/sddic_adt_test_auth_check_doma
*--------------------------------------------------------------------------------------------------------------------
ltcl_auth_domain definition for testing inheriting from cl_adt_rc_test_auth_base risk level critical
duration medium.
  protected section.
    methods:
      get_uri redefinition,
      get_object_names redefinition,
      set_content redefinition.
endclass.                    "ltcl_auth_domain DEFINITION

*----------------------------------------------------------------------*
*       CLASS ltcl_auth_domain IMPLEMENTATION
*----------------------------------------------------------------------*
*
*----------------------------------------------------------------------*
class ltcl_auth_domain implementation.
  method get_uri.
    result = cl_ddic_adt_uri_builder_doma=>create_uri_for_domain( object_name ).
  endmethod.                    "get_uri
  method get_object_names.
    object_name_existing     = |SDDIC_ADT_TEST_AUTH_CHECK_DOMA|.
    object_name_non_existing = |SDDIC_ADT_TEST_AUTH_ND{ CO_NUMBER_PLACEHOLDER }|.
    object_type_lock_handle  = |DOMA|.
  endmethod.                    "get_object_names
  method set_content.
    data http_status type i.
    data doma_data type sddic_adt_doma_data.
    data content_handler type ref to if_adt_srl_content_handler.
*   Get content handler for domain meta data:
    content_handler = cl_adt_srl_st_handler=>create_instance( st_name =
cl_ddic_adt_rc_doma=>co_transformation_name root_name = cl_ddic_adt_rc_doma=>co_transformation_root ).
*   Read existing object which shall be replicated
    get_object(
      exporting
        content_handler = content_handler
      importing
        content_data = doma_data
        http_status  = http_status ).
    cl_abap_unit_assert=>assert_equals( exp = cl_rest_status_code=>gc_success_ok act = http_status ).
*   Change name of the domain to a non-existing name
    doma_data-name = me->object_name_non_existing.
    doma_data-package_ref-name = '$TMP'.
*   Finally, set the content for the create method
    set_content_for_create(
      exporting
        content_data    = doma_data
        content_handler = content_handler ).
  endmethod.                    "set_content
endclass.                    "ltcl_auth_domain IMPLEMENTATION
```

As you can see, the only thing you need to do is to implement the methods GET_URI and GET_OBJECT_NAMES. Method

GET_OBJECT_NAME will return the names of objects to be used in a URI. There must always be one object name for an object that does not yet exist (to test the methods for creation) and there must be one object that exists permanently to test the methods for read, update and delete.

 Please note that you have to implement **own test classes for every single test URI** you can access. In the following example, you will get 7 test classes.
Example: For classes and class includes you may have the following URI's
/sap/bc/adt/oo/classes/zzcl_test_class
/sap/bc/adt/oo/classes/zzcl_test_class/includes/source/main
/sap/bc/adt/oo/classes/zzcl_test_class/includes/definitions
/sap/bc/adt/oo/classes/zzcl_test_class/includes/implementations
/sap/bc/adt/oo/classes/zzcl_test_class/includes/macros
/sap/bc/adt/oo/classes/zzcl_test_class/includes/testclasses
/sap/bc/adt/oo/classes/zzcl_test_class/includes/localtypes

How to fix test errors

Usually, all test methods shall be green. As this is a security test, the tests will provide tolerable results if the HTTP status returned is 404 instead of 403. If you need to investigate the issue further, you can set break-points in the test methods of the parent class CL_ADT_RC_TEST_AUTH_BASE to debug the test methods.