Creating a VIVO authorization policy - an example

Overview

The ability of users to access data in VIVO is controlled by a collection of Policy objects. By creating or controlling Policy objects, you can control access to the data.

The Policy objects are instances of Java classes that implement the `PolicyIface` interface. These objects are created when VIVO starts up, and are collected in the `ServletPolicyList`. When code in VIVO needs to know whether a user is authorized to perform a particular action, the code creates a `RequestedAction` object and passes it to the Policy list for approval.

When the list is asked for approval, the first Policy in the list is asked first. It must respond with a decision that is `AUTHORIZED`, `UNAUTHORIZED`, or `INCONCLUSIVE`. If the decision is `AUTHORIZED` or `UNAUTHORIZED`, it is taken to be final, and the other Policies in the list are not consulted. If the decision is `INCONCLUSIVE`, then the next Policy in the list is asked to approve the same request, and the process repeats until a conclusive answer is obtained, or until all policies have answered. If no Policy has answered with `AUTHORIZED`, the request fails.

The code below is an example of such a Policy. The entire class is available in the attached file.

The example

This Policy will check each request to edit an object property statement. The request will be rejected if the statement appears in any graph that is not in the approved set.

The use case is where an individual whose data is stored in the default graph (`vitro-kb2`) links to data in other graphs which were created by ingest and may not be edited. The result of this Policy is that there will be no edit link from the profile page of the individual to that data.

Lines 1-39: imports

```
 /* $This file is distributed under the terms of the license in /doc/license.txt$ */

package edu.cornell.mannlib.vitro.webapp.auth.policy;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

import javax.servlet.ServletContext;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

import com.hp.hpl.jena.query.Dataset;
import com.hp.hpl.jena.query.Query;
import com.hp.hpl.jena.query.QueryExecution;
import com.hp.hpl.jena.query.QueryExecutionFactory;
import com.hp.hpl.jena.query.QueryFactory;
import com.hp.hpl.jena.query.ResultSet;
import com.hp.hpl.jena.query.Syntax;
import com.hp.hpl.jena.rdf.model.RDFNode;
import com.hp.hpl.jena.shared.Lock;

import edu.cornell.mannlib.vitro.webapp.auth.identifier.IdentifierBundle;
import edu.cornell.mannlib.vitro.webapp.auth.identifier.common.IsRootUser;
import edu.cornell.mannlib.vitro.webapp.auth.policy.ifaces.Authorization;
import edu.cornell.mannlib.vitro.webapp.auth.policy.ifaces.PolicyDecision;
import edu.cornell.mannlib.vitro.webapp.auth.policy.ifaces.PolicyIface;
import edu.cornell.mannlib.vitro.webapp.auth.requestedAction.ifaces.RequestedAction;
import edu.cornell.mannlib.vitro.webapp.auth.requestedAction.propstmt.EditObjectPropertyStatement;
import edu.cornell.mannlib.vitro.webapp.dao.jena.QueryUtils;
import edu.cornell.mannlib.vitro.webapp.servlet.setup.JenaDataSourceSetupBase;
import edu.cornell.mannlib.vitro.webapp.startup.StartupStatus;
```

Import statements for the classes used in the Policy

Lines 40-56: Class declaration, variables, constructor

```
 /**
 * Deny authorization to edit a statement from one of the prohibited graphs.
 */
public class RestrictEditingByGraphPolicy implements PolicyIface {
    private static final Log log = LogFactory
            .getLog(RestrictEditingByGraphPolicy.class);

    private static final Syntax SYNTAX = Syntax.syntaxARQ;
    private static final Set<String> PERMITTED_GRAPHS = new HashSet<>(
            Arrays.asList(new String[] { "http://vitro.mannlib.cornell.edu/default/vitro-kb-2" }));

    private final Dataset dataset;

    public RestrictEditingByGraphPolicy(ServletContext ctx) {
        this.dataset = JenaDataSourceSetupBase.getStartupDataset(ctx);
    }
```

The class must implement the `PolicyIface` interface.

The constructor stores a reference to the `startupDataset`, which will be used to execute SPARQL queries. Because this reference is taken from the context, it will contend with all other context-based references for access to a single database connection. It would be more efficient to use a `Dataset` that was provided by the `HttpServletRequest`, but a Policy never has access to the Request. This will be changed in a future release. (See this JIRA issue.)

The PERMITTED_GRAPHS constant holds the set of graph URIs for which editing is permitted. It would be a simple code change to use a PROHIBITED_GRAPHS constant instead.

Lines 57-68: Implement the isAuthorized() method

```
    /**
     * For each request to Edit an ObjectProperty, find out what graph the
     * statement is in. Prohibit editing if the statement is in the wrong graph.
     *
     * Note that this will not work with a DataProperty, since the
     * EditDataProperty object does not contains the value of the property. We
     * didn't anticipate that editing privileges would be determined by the
     * contents of the string.
     */
    @Override
    public PolicyDecision isAuthorized(IdentifierBundle whoToAuth,
            RequestedAction whatToAuth) {
```

Every PolicyIFace class must implement this method.

- whoToAuth is a collection of Identifiers, each one holding a piece of information about the user who is currently logged in.
- whatToAuth is the action being requested.

Lines 69-81: Make quick and easy decisions

```
        if (whoToAuth == null) {
            return inconclusiveDecision("whoToAuth was null");
        }
        if (whatToAuth == null) {
            return inconclusiveDecision("whatToAuth was null");
        }
        if (IsRootUser.isRootUser(whoToAuth)) {
            return inconclusiveDecision("Anything for the root user");
        }
        if (!(whatToAuth instanceof EditObjectPropertyStatement)) {
            return inconclusiveDecision("Only interested in editing object properties");
        }
```

Policies are called very frequently, especially when a large profile page is displayed. Whenever possible, answer the easy questions first before doing more expensive tests.

Checking for null arguments should not be necessary - these arguments should never be null. However, it is simple defensive programming, and not costly.

This policy is only interested in requests to edit object property statements, so we can quickly reject any other type of RequestedAction. Again, the INCONCLUSIVE decision is equivalent to saying "let someone else decide."

This policy does not attempt to restrict the editing of data property statements. This is because the EditDataPropertyStatement class does not include the value of the data property. At one time it was felt that this could not affect the decision of whether to permit the request. This will be changed in a future release (See this JIRA issue).

This policy will not restrict the root account from attempting to edit statements.

We already have RootUserPolicy, which says that the root user is permitted to do anything. So why do we need this test?

We need to consider the order in which policies are called, and to remember that polling ono a RequestedAction will stop when any policy returns a decision that is not INCONCLUSIVE. So, if this Policy is placed before RootUserPolicy, and returns an UNAUTHORIZED decision, then the RootUserPolicy will never been consulted.

The question of "what to do when one Policy would authorize and another Policy would prohibit" is a tricky one.

Lines 82-105: Execute the SPARQL query and test the result

```
        EditObjectPropertyStatement stmt = (EditObjectPropertyStatement) whatToAuth;

        String queryString = assembleQueryString(stmt);
        List<String> graphUris = executeQuery(queryString);
        log.debug("graph URIs: " + graphUris);

        if (graphUris.isEmpty()) {
            log.warn("Can't find this statement in any graph: " + stmt);
            return inconclusiveDecision("Can't find this statement in any graph: "
                    + stmt);
        }

        graphUris.removeAll(PERMITTED_GRAPHS);
        if (graphUris.isEmpty()) {
            log.debug("Permitted: " + stmt);
            return inconclusiveDecision("Statement is only in permitted graphs: "
                    + stmt);
        }

        log.debug("Statement is prohibited: " + stmt + ", graphs=" + graphUris);
        return unauthorizedDecision("Statement is in a prohibited graph, "
                + stmt + " in " + graphUris);
    }
```

Assemble the query and execute it. This results in a list of the URIs of all Graphs that contain this statment. (See the subroutines in the next section).

What to do if we do not find the statement in any graph? It would be possible to err on the side of caution and return an `UNAUTHORIZED` decision. We could even throw a `RuntimeException` of some sort to abort the page display. In this case, we choose to return `INCONCLUSIVE` and write a warning to the log.

If the statement appears only in the permitted graphs, return a decision of `INCONCLUSIVE`, letting some other policy decide.

If the statement appears in other, prohibited graphs, return a decision of `UNAUTHORIZED`, rejecting the requested action.

Lines 106-171: Subroutines

```
    private static final String QUERY_TEMPLATE = "" + //
            "SELECT ?graph WHERE{" + //
            "   GRAPH ?graph{" + //
            "     ?s ?p ?o ." + //
            "   } " + //
            "} LIMIT 10"; //

        private String assembleQueryString(EditObjectPropertyStatement stmt) {
        String q = QUERY_TEMPLATE;
        q = QueryUtils.subUriForQueryVar(q, "s", stmt.getSubjectUri());
        q = QueryUtils.subUriForQueryVar(q, "p", stmt.getPredicateUri());
        q = QueryUtils.subUriForQueryVar(q, "o", stmt.getObjectUri());
        return q;
    }
```

We have a template for the SPARQL query. Substitute the values for this statement into the query. The only unresolved variable will be `?graph`.

```
    private List<String> executeQuery(String queryStr) {
        log.debug("select query is: '" + queryStr + "'");
        QueryExecution qe = null;
        dataset.getLock().enterCriticalSection(Lock.READ);
        try {
            Query query = QueryFactory.create(queryStr, SYNTAX);
            qe = QueryExecutionFactory.create(query, dataset);
            return parseResults(queryStr, qe.execSelect());
        } catch (Exception e) {
            log.error("Failed to execute the Select query: " + queryStr, e);
            return Collections.emptyList();
        } finally {
            if (qe != null) {
                qe.close();
            }
            dataset.getLock().leaveCriticalSection();
        }
    }

    private List<String> parseResults(String queryStr, ResultSet results) {
        List<String> uris = new ArrayList<>();
        if (results.hasNext()) {
            try {
                RDFNode node = results.next().get("graph");
                if ((node != null) && node.isResource()) {
                    uris.add(node.asResource().getURI());
                }
            } catch (Exception e) {
                log.warn("Failed to parse the query result" + queryStr, e);
            }
        }
        return uris;
    }
```

Execute the SPARQL query against the `Dataset`. Extract the graph URIs from the result.

```
    /**
     * An UNAUTHORIZED decision says
     * "Not allowed. Don't bother asking anyone else".
     */
    private PolicyDecision unauthorizedDecision(String message) {
        return new BasicPolicyDecision(Authorization.UNAUTHORIZED, getClass()
                .getSimpleName() + ": " + message);
    }

    /**
     * An INCONCLUSIVE decision says "Let someone else decide".
     */
    private PolicyDecision inconclusiveDecision(String message) {
        return new BasicPolicyDecision(Authorization.INCONCLUSIVE, getClass()
                .getSimpleName() + ": " + message);
    }
```

Convenience methods for creating `PolicyDecision` return values.

Setup when VIVO starts

When VIVO starts execution, the `StartupManager` processes the file `startup_listeners.txt`, and instantiating each class that is named in the file, and invoking the `contextsInitialized()` method on each class.

Lines 172-193: The Setup class

```
    // ----------------------------------------------------------------------
    // Setup class - must be specified in startup_listeners.txt before any
    // policy that might be more permissive.
    // ----------------------------------------------------------------------
    public static class Setup implements ServletContextListener {

        @Override
        public void contextInitialized(ServletContextEvent sce) {
            ServletContext ctx = sce.getServletContext();
            StartupStatus ss = StartupStatus.getBean(ctx);

            RestrictEditingByGraphPolicy p = new RestrictEditingByGraphPolicy(
                    ctx);
            ServletPolicyList.addPolicy(ctx, p);
            ss.info(this,
                    "Editing object properties is only permitted in these graphs: "
                            + RestrictEditingByGraphPolicy.PERMITTED_GRAPHS);
        }

        @Override
        public void contextDestroyed(ServletContextEvent sce) { /* nothing */
        }
    }
```

The Setup class must implement `ServletContextListener`.

On startup, create an instance of the Policy, and add it to the `ServletPolicyList`. Produce an informative message for the startup status screen.

On shutdown, there is nothing to be done. If there were resources to be freed or files to be closed, this would be the place to do it.

Invoking the Setup class

Initialize the policy in startup_listeners.txt
`edu.cornell.mannlib.vitro.webapp.auth.policy.RestrictEditingByGraphPolicy$Setup`

Add this line to `startup_listeners.txt`. Consult the note above regarding placement of this Policy relative to the other Policies.

A more complicated example

For another example of writing a policy, look at A more elaborate authorization policy