SpringAndAuthentication

Implementing a custom security policy with Flowscript and Spring

Objectives

When you have to implement authentication and users management in a Cocoon application you have mainly four solutions :

- Using JAAS security mechanisms specified in J2EE standards and implemented by containers (see [AuthWithTomcat])
- Using the authentication framework
- CoWarp
- Implementing your own security policy.

The first approach lacks flexibility. The second one is very generic and hence very difficult to implement, all the more so as documentation is not very clear on that point :-P I don't know about the third solution. I've just heard of it but I skipped it as I thought implementing my own security policy using flowscript and my Spring layer was easier and more flexible.

Security policy

Here is the activity diagram of the security policy I've implemented :
http://www.epseelon.org/cocoon/Activity_Diagram_security.png

Flowscript

And here is the corresponding flowscript code (to adapt to your environment of course) :

```
importClass(Packages.org.springframework.web.context.WebApplicationContext);
//This one require commons-codec to be in classpath. It's used to hash passwords.
importClass(Packages.org.apache.commons.codec.digest.DigestUtils);
// load CForms support
cocoon.load("resource://org/apache/cocoon/forms/flow/javascript/Form.js");

var securityManager;

// Initializes the security manager service bean we will use to do security business stuff.
function getSecurityManager(){
    if (securityManager == null) {
        var appCtx = cocoon.context.getAttribute(WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE);
        schaman = appCtx.getBean("securityManager");
    }
    return schaman;
}

// This function is called from sitemap when a protected resource is requested (see sitemap snippet below)
function requestProtectedResource(){
    var protectedResource = cocoon.parameters["protected-resource"];
    if(authenticated()){
        checkAuthorization(protectedResource);
    }
    else{
        promptForLogin(protectedResource);
    }
}

/**
 * This function calls Spring business layer to check if currently authenticated user has the right to access
 * the protected resource. I personally use a regular expression pattern matching but as everything happens in
 * business layer it's up to you.
 */
function checkAuthorization(protectedResource){
    if(authorized(protectedResource)){
        displayProtectedResource(protectedResource);
    }
    else{
        displayMessage("not-authorized","error");
    }
```

```
}

/**
 * This function is called only when we are sure that the user is authenticated and authorized to access
 * protectedResource.
 */
function displayProtectedResource(protectedResource){
    if(protectedResource == null) protectedResource = "home";
    cocoon.sendPage("internal/" + protectedResource);
}

/**
 * This one is not compulsory. It's just a utility function to display messages.
 */
function displayMessage(messageKey,messageType){
    cocoon.sendPage("internal/message",{message : messageKey,type:messageType});
}

/**
 * Displays a login form using CForms, which is very difficult to achieve with authentication framework.
 * Or at least I didn't find how to do it :-P
 * It's important to notice that protectedResource is passed as a parameter so that the protected resource
requested
 * in the first place can be displayed after the login procedure.
 */
function promptForLogin(protectedResource){
    // If the user is already authenticated it's pointless to display a login form and we can just move on to
    // authorization. This test is useful for the case when the user accesses directly login procedure using
some
    // "Log on" link for example, instead of requesting a protected resource.

    if(authenticated()){
        checkAuthorization(protectedResource);
        return;
    }
    //We are gonna access securityManager variable so we make sure it is initialized.
    getSecurityManager();
    var user = null;
    //incorrect enables to display a special message when the user is unknown in the security manager, or if
    //the password is wrong for example. These are independant from CForm validation so this trick is very
useful.
    //the incorrect variable can be used in JXTemplate to display a register link for example.
    var incorrect = false;
    while(user == null){
        var form = new Form("forms/login_d.xml");
        form.showForm("internal/login",{incorrect : incorrect});
        var model = form.getModel();
        //This is where we use MD5 hasher, just to be sure that no clear password goes under the presentation
layer
        user = securityManager.authenticateUser(model.username,DigestUtils.md5Hex(model.password));
        if(user == null) incorrect = true;
    }
    setUserName(user.getName());
    checkAuthorization(protectedResource);
}

//Session management
/**
 * When a user is authenticated his name is simply registered in some session context
 * So if there is a name, the user is authenticated.
 */
function authenticated(){
    return getUserName() != null;
}

/**
 * This is where the authorization process takes place.
 * I have a checkAuthorization method in my securityManager bean, which just check if protectedResource matches
at
 * least one of the regular expression patterns associated with this user.
 * But again the business code of this method as well as the parameters you need for it are completely up to
```

```
you.
 */
function authorized(protectedResource){
    getSecurityManager();
    var username = getUserName();
    return securityManager.checkAuthorization(username,protectedResource);
}

/**
 * Registers the user against the session context.
 */
function setUserName(username){
    setData("/user/name",username);
}

/**
 * This function (and the one above) are very useful if I want to change the path organization of my session
context
 */
function getUserName(){
    return getData("/user/name");
}

/**
 * Called when the user logs out.
 */
function logout(){
    destroyContext();
    cocoon.sendPage("");
}

/**
 * The two following functions are wrappers to set data in a cocoon session context called "myApp"
 * Those function can be used as is, except for the context name you can customize of course.
 * This one writes data into the session context.
 * path is a JXPath string, value is the corresponding value we want to write
 */
function setData(path,value) {
    var contextManager=cocoon.getComponent(Packages.org.apache.cocoon.webapps.session.ContextManager.ROLE);
    var myContext;
    if (contextManager.existsContext("myApp")){
        myContext=contextManager.getContext("myApp");
    }
    else{
        myContext=contextManager.createContext("myApp","","");
    }

    myContext.setAttribute(path,value);
    cocoon.releaseComponent(contextManager);
}

/**
 * And this one reads data from the session context.
 * path is a JXPath string
 */
function getData(path) {
    var contextManager=cocoon.getComponent(Packages.org.apache.cocoon.webapps.session.ContextManager.ROLE);
    var data;
    var myContext = contextManager.getContext("myApp");
    if(myContext != null){
        data = myContext.getAttribute(path);
    }
    else{
        data = null;
    }

    cocoon.releaseComponent(contextManager);
    return data;
}

/**
```

```
 * Destroys the session context to clear user authentication information.
 */
function destroyContext() {
    var contextManager=cocoon.getComponent(Packages.org.apache.cocoon.webapps.session.ContextManager.ROLE);
    var data;
    try {
        if(contextManager.existsContext("schaman")){
            contextManager.deleteContext("schaman");
        }
    }
    finally {
        cocoon.releaseComponent(contextManager);
    }
}
```

Note

Special thanks to NachoJimenez for the session framework wrappers 😉

Sitemap snippet

```
        <map:pipeline internal-only="true">
                <map:match pattern="internal/*">
                    <map:generate src="pages/{1}.xml" type="jx"/>
                    <!--
                    the "apply-theme" resource is used to wrap the page is the general layout with menus,
                    headers, etc.
                    It also contains all the necessary transformers like forms, i18n, jx, etc.
                    -->
                    <map:call resource="apply-theme">
                        <map:parameter name="includemenu" value="true"/>
                    </map:call>
                </map:match>
        </map:pipeline>

        <map:pipeline>
            <!--<map:act type="locale">-->
                <map:match pattern="">
                    <map:call function="index"/>
                </map:match>

                <map:match pattern="login">
                    <map:call function="promptForLogin"/>
                </map:match>

                <map:match pattern="logout">
                    <map:call function="logout"/>
                </map:match>

                <map:match pattern="*">
                    <map:call function="requestProtectedResource">
                        <map:parameter name="protected-resource" value="{1}"/>
                    </map:call>
                </map:match>
            <!--</map:act>-->
        </map:pipeline>
```

Caution

Of course I'm not sure this solution is optimized and it has the great drawback to be coupled with the application, making it impossible to use alternative authentication and authorization methods like LDAP or database for example. But it's enough for a small application like the one I'm working one and I'm sure it can fit in most of simple projects.

Anyway feel free to contact me if you have any question, comment or suggestion to improve this proposition.

Sébastien Arbogast

sebastien.arbogast@gmail.com

http://www.sebastien-arbogast.com