

SSLWithFormFallbackAuthenticator

The following code was tested with Tomcat 5.5.17 and 5.5.20. See the page [SSLWithFORMFallback](#) for instructions.

You will need the Tomcat JARs from a tomcat installation to compile this class.

```
package at.telekom.tomcat.security;

import java.io.IOException;
import java.security.Principal;
import java.security.cert.X509Certificate;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Locale;

import javax.servlet.ServletException;

import org.apache.catalina.Globals;
import org.apache.catalina.Realm;
import org.apache.catalina.Session;
import org.apache.catalina.authenticator.Constants;
import org.apache.catalina.authenticator.FormAuthenticator;
import org.apache.catalina.connector.Request;
import org.apache.catalina.connector.Response;
import org.apache.catalina.deploy.LoginConfig;
import org.apache.catalina.deploy.SecurityConstraint;
import org.apache.catalina.util.DateTool;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.apache.coyote.ActionCode;

/**
 * <p>
 * Authenticator Valve for Tomcat
 * </p>
 * <p>
 * This Valve implements a special login method - SSL Certificate based, with FORM fallback.
 * It is intended to be used on an SSL protected channel. The Authenticator first tries to authenticate
 * based on a client certificate included in the SSL Request. If this is not possible, or the request
 * does not contain a client certificate, the Authenticator attempts to authenticate using the
 * FORM login method.
 * </p>
 * <p>
 * To use this authenticator, include a <login-config> in your web.xml, but do not include a
 * <auth-method> tag. Do include the <form-login-config> tag to configure the pages used if
 * FORM login occurs.
 * </p>
 * <p>
 * Instead of the <auth-method> tag, manually configure a Valve in your tomcat configuration
 * (server.xml or context.xml). Configure the Valve on the corresponding context, and use this
 * class as the Valve type.
 * </p>
 * @author Richard Unger
 */
public class SSLWithFormFallbackAuthenticator extends FormAuthenticator {

    private static Log log = LogFactory.getLog(SSLWithFormFallbackAuthenticator.class);

    /**
     * "Expires" header always set to Date(1), so generate once only
     */
    private static final String DATE_ONE =
        (new SimpleDateFormat(DateTool.HTTP_RESPONSE_DATE_HEADER,
            Locale.US)).format(new Date(1));

    /**
     *
     */
}
```

```

public SSLWithFormFallbackAuthenticator() {
    super();
}

/**
 * @see org.apache.catalina.authenticator.AuthenticatorBase#invoke(org.apache.catalina.connector.
Request, org.apache.catalina.connector.Response)
 */
@Override
public void invoke(Request request, Response response)
    throws IOException, ServletException {

    if (log.isDebugEnabled())
        log.debug("Security checking request " +
            request.getMethod() + " " + request.getRequestURI());
    LoginConfig config = this.context.getLoginConfig();

    // Have we got a cached authenticated Principal to record?
    if (cache) {
        Principal principal = request.getUserPrincipal();
        if (principal == null) {
            Session session = request.getSessionInternal(false);
            if (session != null) {
                principal = session.getPrincipal();
                if (principal != null) {
                    if (log.isDebugEnabled())
                        log.debug("We have cached auth type " +
                            session.getAuthType() +
                            " for principal " +
                            session.getPrincipal());
                    request.setAuthType(session.getAuthType());
                    request.setUserPrincipal(principal);
                }
            }
        }
    }

    // Special handling for form-based logins to deal with the case
    // where the login form (and therefore the "j_security_check" URI
    // to which it submits) might be outside the secured area
    String contextPath = this.context.getPath();
    String requestURI = request.getDecodedRequestURI();
    if (requestURI.startsWith(contextPath) &&
        requestURI.endsWith(Constants.FORM_ACTION)) {
        if (!authenticate(request, response, config)) { // RU - this is a call to FORM based auth!
            if (log.isDebugEnabled())
                log.debug(" Failed authenticate() test ??" + requestURI );
            return;
        }
    }

    Realm realm = this.context.getRealm();
    // Is this request URI subject to a security constraint?
    SecurityConstraint [] constraints
        = realm.findSecurityConstraints(request, this.context);

    if ((constraints == null) /* &&
        (!Constants.FORM_METHOD.equals(config.getAuthMethod())) */ ) {
        if (log.isDebugEnabled())
            log.debug(" Not subject to any constraint");
        getNext().invoke(request, response);
        return;
    }

    // Make sure that constrained resources are not cached by web proxies
    // or browsers as caching can provide a security hole
    if (disableProxyCaching &&
        // FIX-ME: Disabled for Mozilla FORM support over SSL
        // (improper caching issue)
        !request.isSecure() &&
        !"POST".equalsIgnoreCase(request.getMethod())) {

```

```

/*      if (securePagesWithPragma) {
        // FIX-ME: These cause problems with downloading office docs
        // from IE under SSL and may not be needed for newer Mozilla
        // clients.
        response.setHeader("Pragma", "No-cache");
        response.setHeader("Cache-Control", "no-cache");
    } else { */
        response.setHeader("Cache-Control", "private");
    // }
    response.setHeader("Expires", DATE_ONE);
}

int i;
// Enforce any user data constraint for this security constraint
if (log.isDebugEnabled()) {
    log.debug(" Calling hasUserDataPermission()");
}
if (!realm.hasUserDataPermission(request, response,
                                constraints)) {
    if (log.isDebugEnabled()) {
        log.debug(" Failed hasUserDataPermission() test");
    }
    /*
    * ASSERT: Authenticator already set the appropriate
    * HTTP status code, so we do not have to do anything special
    */
    return;
}

// Since authenticate modifies the response on failure,
// we have to check for allow-from-all first.
boolean authRequired = true;
for(i=0; i < constraints.length && authRequired; i++) {
    if(!constraints[i].getAuthConstraint()) {
        authRequired = false;
    } else if(!constraints[i].getAllRoles()) {
        String [] roles = constraints[i].findAuthRoles();
        if(roles == null || roles.length == 0) {
            authRequired = false;
        }
    }
}

if(authRequired) {
    if (log.isDebugEnabled()) {
        log.debug(" Calling authenticate()");
    }
    if (!authenticateSSL(request, response, config)){ // RU - this is the SSL auth
        if (!authenticate(request, response, config)) { // RU - and this is the FORM fallback!
            if (log.isDebugEnabled()) {
                log.debug(" Failed authenticate() test for both SSL and form");
            }
            /*
            * ASSERT: Authenticator already set the appropriate
            * HTTP status code, so we do not have to do anything
            * special
            */
            return;
        }
    }
}

if (log.isDebugEnabled()) {
    log.debug(" Calling accessControl()");
}
if (!realm.hasResourcePermission(request, response,
                                constraints,
                                this.context)) {
    if (log.isDebugEnabled()) {
        log.debug(" Failed accessControl() test");
    }
}

```

```

        /*
        * ASSERT: AccessControl method has already set the
        * appropriate HTTP status code, so we do not have to do
        * anything special
        */
        return;
    }

    // Any and all specified constraints have been satisfied
    if (log.isDebugEnabled()) {
        log.debug(" Successfully passed all security constraints");
    }
    getNext().invoke(request, response);
}

/**
 * Authenticate the user by checking for the existence of a certificate
 * chain, and optionally asking a trust manager to validate that we trust
 * this user.
 *
 * This method was modified from the original SSLAuthenticator not to throw exceptions
 * or modify the response, since there will be FORM authentication fallback.
 *
 * @param request Request we are processing
 * @param response Response we are creating
 * @param config Login configuration describing how authentication
 *               should be performed
 *
 * @exception IOException if an input/output error occurs
 */
public boolean authenticateSSL(Request request,
                               Response response,
                               LoginConfig config)
    throws IOException {

    // Have we already authenticated someone?
    Principal principal = request.getUserPrincipal();
    //String ssoId = (String) request.getNote(Constants.REQ_SSOID_NOTE);
    if (principal != null) {
        if (log.isDebugEnabled())
            log.debug("Already authenticated " + principal.getName() + "");
        // Associate the session with any existing SSO session in order
        // to get coordinated session invalidation at logout
        String ssoId = (String) request.getNote(Constants.REQ_SSOID_NOTE);
        if (ssoId != null)
            associate(ssoId, request.getSessionInternal(true));
        return (true);
    }

    // NOTE: We don't try to reauthenticate using any existing SSO session,
    // because that will only work if the original authentication was
    // BASIC or FORM, which are less secure than the CLIENT-CERT auth-type
    // specified for this webapp
    //
    // Uncomment below to allow previous FORM or BASIC authentications
    // to authenticate users for this webapp
    // TO-DO make this a configurable attribute (in SingleSignOn??)
    /*
    // Is there an SSO session against which we can try to reauthenticate?
    if (ssoId != null) {
        if (log.isDebugEnabled())
            log.debug("SSO Id " + ssoId + " set; attempting " +
                      "reauthentication");
        // Try to reauthenticate using data cached by SSO. If this fails,
        // either the original SSO logon was of DIGEST or SSL (which
        // we can't reauthenticate ourselves because there is no
        // cached username and password), or the realm denied
    }
    */

```

```

        // the user's reauthentication for some reason.
        // In either case we have to prompt the user for a logon
        if (reauthenticateFromSSO(ssoId, request))
            return true;
    }
    */

    // Retrieve the certificate chain for this client
    if (log.isDebugEnabled())
        log.debug(" Looking up certificates");

    X509Certificate certs[] = (X509Certificate[])
        request.getAttribute(Globals.CERTIFICATES_ATTR);
    if ((certs == null) || (certs.length < 1)) {
        log.debug(" No certificates found in HttpRequest.");
        request.getCoyoteRequest().action
            (ActionCode.ACTION_REQ_SSL_CERTIFICATE, null);
        certs = (X509Certificate[])
            request.getAttribute(Globals.CERTIFICATES_ATTR);
        if (certs != null && (certs.length >= 1))
            log.debug("Certificates found in CoyoteRequest");
    }
    else
        log.debug("Certificates found in HttpRequest");
    if ((certs == null) || (certs.length < 1)) {
        if (log.isDebugEnabled())
            log.debug(" No certificates included with this request");
        /* response.sendError(HttpServletResponse.SC_BAD_REQUEST,
            sm.getString("authenticator.certificates")); */
        return (false);
    }

    log.debug("Found certificates, authenticating using them!");

    // Authenticate the specified certificate chain
    principal = context.getRealm().authenticate(certs);
    if (principal == null) {
        if (log.isDebugEnabled())
            log.debug(" Realm.authenticate() returned false");
        /* response.sendError(HttpServletResponse.SC_UNAUTHORIZED,
            sm.getString("authenticator.unauthorized")); */
        return (false);
    }

    // Cache the principal (if requested) and record this authentication
    register(request, response, principal, Constants.CERT_METHOD,
        null, null);
    return (true);
}

}

```