Web Security

Saturday, June 29, 2019

4:03 PM

Basic HTTP Authentication

HTTP basic authentication headers, headers allow you to embed the security information (username and password) in the header of the request that you send to the server.

HTTP uses a standard header for carrying this information. The header is appropriately named “Authorization.”

When using this header, the client that is sending the request (for example, a browser) concatenates the username

and the password with a colon between them and then Base64 encodes the resulting string, sending the result of this in the header. For example, if you use the username bart and the password simpson, the client creates the string bart:simpson and encodes it prior to sending it in the header.

Let’s use Basic HTTP authentication in our application. The first and only thing you need to do is replace the <login-form> element in your configuration file applicationContext-security.xml with the following one: <security:http-basic/>. After replacing it, you restart the application and go to the URL <http://localhost:8080/admin/movies> in the browser. A standard HTTP authentication box pops up asking you for your authentication details . Type admin as the username and password, and send the request. You successfully arrive in the

movie x page

Ecurity 
Connecting... 
Authentication Required 
A username and password are being requested by 
http://localhost:8080. The site says: Spring Security 
Application" 
user Name: 
Password: 

When you use the <http-basic/> configuration element, Spring Security's BasicAuthenticationFilter comes into action. A BasicAuthenticationEntryPoint strategy will be configured into the ExceptionTranslationFilter on startup. When you make the first request to the URL /admin/movies, the framework behaves as before, throwing an access-denied exception that is handled by the ExceptionTranslationFilter. This filter delegates to a particular implementation strategy of AuthenticationEntryPoint—in this case, BasicAuthenticationEntryPoint. BasicAuthenticationEntryPoint adds the header “WWW-Authenticate: Basic realm=“Spring Security Application”” to the response and then sends an HTTP status of 401 (Unauthorized) to the client. The client should know how to handle this code and work accordingly. (In the case of a browser, it simply

shows the authentication pop up.)

When you introduce the username and password and submit the request, the request again follows the filter chain until it reaches the BasicAuthenticationFilter. This filter checks the request headers, looking for the “Authorization” header starting with “Basic.” The filter extracts the content of the header and uses Base64.decode to decode the string, and then it extracts the username and password. The filter creates a UsernamePasswordAuthenticationToken object and sends it to the authentication manager for authentication in the

standard way. The authentication manager will ask the authentication provider to retrieve the user and then create an Authentication object with it. This process is standard and independent of using Basic Authentication or form authentication.

Digest Authentication

Digest Authentication is a very close sibling of Basic HTTP Authentication. Its main purpose is to avoid sending clear text passwords on the wire, as Basic Authentication does, by hashing the password prior to sending it to the server.

This makes Digest Authentication more complex than Basic Authentication.

Digest Authentication works with HTTP headers the same way that Basic Authentication does.

Digest Authentication is based in the use of a nonce for hashing the passwords. A nonce is an arbitrary server-generated number that is used in the authentication process and that is used only once. It is passed through the digest computation together with the username, password, nonce, URI being requested, and so on. In the authentication process, both the server and client do the digest computation and they should match.

The main processing lies in two classes:

* + DigestAuthenticationFilter
  + DigestAuthenticationEntryPoint

**DigestAuthenticationFilter** queries the request’s headers looking for the Authorization header, and then it checks that the header’s value starts with “Digest.” If this is the case, the request is carrying the security credentials that will be used for authentication.

**DigestAuthenticationEntryPoint** is the class that is invoked to generate a response that demands that a digest security authentication process begin. This class sets the header “WWW-Authenticate” with the correct values (including the nonce) so that the client agent (the browser) knows it has to start the digest authentication process.

including the nonce) so that the client agent (the browser) knows it has to start the digest authentication process. To configure it, let’s add the filter to the filter chain. In this case, there is no custom XML element to define it, so you need to create a <custom-filter> element and then create <bean>s for both the filter and the entry point.

The new <custom-filter> element should look like this: <security:custom-filter ref="digestFilter"

before="BASIC\_AUTH\_FILTER"/>. Also, it needs to be added as a child of the <http> element. In the <http> element, you add the attribute entry-point-ref="digestEntryPoint", which will allow the ExceptionTranslationFilter to use this entry point when an AccessDeniedException is encountered. Listing 4-14 shows the two new beans you need to define in the file applicationContext-security.xml. You also need to give the ID “userService” to the configured

<user-service> in the same file.

<bean id="digestFilter" class=

"org.springframework.security.web.authentication.www.DigestAuthenticationFilter">

<property name="userDetailsService" ref="userService" />

<property name="authenticationEntryPoint" ref="digestEntryPoint" />

</bean>

<bean id="digestEntryPoint" class=

"org.springframework.security.web.authentication.www.DigestAuthenticationEntryPoint">

<property name="realmName" value="terrormovies-realm"/>

<property name="key" value="terrorific" />

</bean>

Remember-Me Authentication

The remember-me authentication functionality is used for allowing returning users of the application to use it without

needing to log in every time. Basically, the application will remember certain visitors, allowing them to just open the

application and be greeted with their personalized version of the application, as if they were logged in.

*How does remember-me functionality work in Spring Security?*

Remember-me functionality in Spring Security is supported mainly by two components: the

RememberMeServices interface and the RememberMeAuthenticationFilter class. Let’s see how they work in the context of a request.

* + When the application starts up, the RememberMeAuthenticationFilter will be in the filter chain of the server. Also, a TokenBasedRememberMeServices will be instantiated and injected into the AbstractAuthenticationProcessingFilter replacing the no-op NullRememberMeServices. Go and visit the URL <http://localhost:8080/admin/movies>, and log in with admin as the username and password.
  + When the request gets into the application, UsernamePasswordAuthenticationFilter (a subclass of AbstractAuthenticationProcessingFilter) will handle the authentication process in the standard way already explained.
  + After the authentication is successful, UsernamePasswordAuthenticationFilter invokes the configured TokenBasedRememberMeServices's loginSuccess method. This method looks to see if the request contains the parameter \_spring\_security\_remember\_me in order to apply the remember-me functionality. (If the property alwaysRemember is set to true in the service, it will also apply the remember-me functionality.) Because you didn’t

send this request, nothing will happen.

* + So let’s add the parameter to the login form you have. Open the file login.jsp and somewhere inside the <form> paste the following element: <input type="checkbox" name="\_spring\_security\_ remember\_me" value="yes"/>. (As a value attribute, you can use any of the following values: “yes”, “on”, “1”, or “true”.)
  + Restart the application, and visit the URL <http://localhost:8080/admin/movies>. You should now see a check box along with the username and password fields. Select the check box, and log in with admin/admin.
  + This time, the request carries the required parameter and **TokenBasedRememberMeServices** does its work. It extracts the username and password from the Authentication object and creates a token with this information and

a time to expire. It basically concatenates these three values and the remember-me key specified in the XML element (terror-key). And it creates an MD5 encoding out of the resulting string. This value will then be Base64-encoded again, together with the username, and added to the response as a cookie with the name SPRING\_SECURITY\_REMEMBER\_ME\_COOKIE that will be returned to the browser.

Name: 
Content: 
Domain: 
Path: 
Send for: 
Accessible to script: 
Created: 
Expires: 
Remove 
SPRING SECURITY REMEMBER ME COOKIE 
YWRtaW46MTMINJg4NjEwOTEzNzozYWZmZTY3YjVjMTlzY 
TYyYTdmZmNhZTNIZTVjZDA2Mw 
localhost 
Any kind of connection 
No (HttpOnly) 
Sunday, December 16, 2012 PM 
Sunday, December 30, 2012 PM 

* 1. Restart the application. Visit the URL <http://localhost:8080/admin/movies>. You should be able to access the page without logging in.

* 1. When this request gets in the system, it is intercepted by the RememberMeAuthenticationFilter, which gets into action. The first thing the filter does is check that there is no current Authentication in the SecurityContext.Because this means there is no user logged in, the filter calls the RememberMeServices’s autoLogin method.

* 1. In the standard configuration, the TokenBasedRememberMeServices is the concrete class that implements RememberMeServices. This implementation’s autoLogin method tries to parse the incoming cookie into its composing elements, which are the username, the hashed value of the combined elements (username + “:” + tokenExpiryTime

+ “:” + password + “:” + key), and the expiry time of the token. Then it retrieves the UserDetails from the UserDetailsService with the username, recomputes the hashed value with the retrieved user, and compares it with the arriving one. If they don’t match, an InvalidCookieException is thrown. If they do match, the UserDetails is

checked and an Authentication object is created and returned to the caller.

* 1. The autoLogin method extracts the remember-me cookie out of the request, decodes it, does some validation and then calls the configured UserDetailsService's loadUserByUsername method with the username extracted from

the cookie. It then creates a RememberMeAuthenticationToken object (an implementation of Authentication).

The RememberMeAuthenticationFilter then tries to authenticate this new Authentication object against the

* 1. AuthenticationProvider’s implementation of RememberMeAuthenticationProvider, which simply returns the same Authentication object after making sure that the hash from the incoming request matches the stored one for the

remember-me key.

Note:

Remember-me is not enabled by default. To enable it, include the following element inside the <http> element in the configuration file <security:remember-me key="terror-key"/>.

Allowing Remember-Me Access to Selected Parts of the Application.

Remember-me authentication can be easily configured so that certain URLs require a fully authenticated user (meaning the user is explicitly logged in) to access them. To do this in your application, replace the

<security:intercept-url/> element you have in the file applicationContext-security.xml with the following one:

<security:intercept-url pattern="/admin/\*" access="ROLE\_ADMIN,IS\_AUTHENTICATED\_FULLY" />

You have added the access rule IS\_AUTHENTICATED\_FULLY to the access attribute.

You still need more configuration to make this work. By default, Spring Security’s <http> element configures an AffirmativeBased access-decision manager.

Let’s add the bean definition to your applicationContext-security.xml. And make the

<http> element’s opening tag look like the following:

<bean id="accessDecisionManager" class="org.springframework.security.access.vote.UnanimousBased">

<constructor-arg>

<list>

<bean class="org.springframework.security.access.vote.RoleVoter"/>

<bean class="org.springframework.security.access.vote.AuthenticatedVoter"/>

</list>

</constructor-arg>

</bean>

This time, the access-decision manager calls the AuthenticatedVoter. The AuthenticatedVoter will see that the URL requires a fully authenticated user to access it, so it will query the requesting Authentication to see if it is fully authenticated. The way this check takes place is straightforward: the voter simply checks if the Authentication object implementation is neither an AnonymousAuthenticationToken nor a RememberMeAuthenticationToken instance, assuming it is then a fully authenticated Authentication object and allowing access. In this case, access will be denied because the Authentication object is a remember-me implementation. This means that the login form will be shown even if a remember-me option was used previously.

*Note:*

*Remember-me authentication supports the use of persistent storage, so the token is kept in a*

*datastore and survives application restarts. The main class supporting persistent remember-me storage is*

*PersistentTokenBasedRememberMeServices, which extends from AbstractRememberMeServices the same way that TokenBasedRememberMeServices does. To activate the use of the persistent functionality, you need to add the attribute data-source-ref="someDataSource" to the element <security:remember-me> that you defined in the configuration file. This way, you can reference a data source bean in the application context.*

*Persistent remember-me tokens:*

Persistent tokens is to define the attribute token-repository-ref="someTokenRepo"

in the element <security:remember-me>. In this attribute you need to put a reference to a bean of type

PersistentTokenRepository, or more exactly, an implementation of PersistentTokenRepository as that

is an interface. If you were to use the implementation JdbcTokenRepositoryImpl, it would be as if you were

defining the data-source-ref attribute that I explained before, because this is what such an attribute does

internally. However, using the attribute token-repository-ref, you also have available the implementation

InMemoryTokenRepositoryImpl, which is backed by a simple in-memory map and is recommended for testing purposes only. You could also create your own implementation based on some other kind of datastore and inject it into your <remember-me> element.

In the persistent model, tokens are stored against a series-id. A series-id is simply a random Base64 string

that is generated whenever a successful login is done in the system. This series-id will be part of the remember-me cookie (SPRING\_SECURITY\_REMEMBER\_ME\_COOKIE) that is sent back to the browser together with the token when this successful login happens.

Every time a new autoLogin request comes to the PersistentTokenBasedRememberMeServices, the cookie

values (series-id and token) are extracted and compared with the stored ones. If they both match, another token is generated for the same serial-id and the datastore is updated with this new token value. If the series-id matches but the token doesn’t match, it is assumed that a cookie theft has happened (or basically two people have the same cookie). This is assumed because, as I just explained, the series-id is a quasi-unique random number that is generated on successful login and maintained for that user in the cookie. That means it is virtually impossible that some other user in a different browser will have the same series-id, unless he had the same cookie. This is because every time a remember-me autologin is performed, the series-id is used to retrieve the token from the store and then the token is updated.

On subsequent requests, if a user’s token doesn’t match the stored one, this is because someone else accessed the autologin functionality from somewhere else causing the token to be updated for the user’s series-id, so that the old legitimate user token doesn’t match the one that is now stored.

When the system detects this, it throws a CookieTheftException and removes all the user tokens from the

datastore.

Logging Out

Logging out is pretty simple. When you log out of an application, you want the application to end your current session, but also to remove any information it might have stored on the client for you.

In Spring Security, logging out is very easy. The only thing you need to do by default is to visit the URL

/j\_spring\_security\_logout. Let’s try that. Remove the latest added IS\_AUTHENTICATED\_FULLY from the

configuration XML file and restart the application.

TokenBasedRememberMeServices simply removes the remember-me cookie by setting its age to 0.

Spring Security 3.1 includes a new LogoutHandler called CookieClearingLogoutHandler. This handler, as its

name implies, removes cookies as specified in its constructor. This handler, however, is not configured by default.

To enable the CookieClearingLogoutHandler handler, add the following XML element as a child of the <http>

element in the applicationContext-security.xml file: <security:logout delete-cookies="JSESSIONID"/>.

The <logout> element in the applicationContext-security.xml file with the following one:

<security:logout delete-cookies="JSESSIONID" success-handler-ref="logoutRedirectToAny"/>

And somewhere else in the same file, define the following bean:

<bean id="logoutRedirectToAny" class="org.springframework.security.web.authentication.logout.

SimpleUrlLogoutSuccessHandler">

<property name="targetUrlParameter" value="redirectTo"/>

</bean>

After this is done, go and restart the application, go to http:localhost:8080/admin/movies, log in, and then

request the following URL: /j\_spring\_security\_logout?redirectTo=http://www.google.com

You will be logged out and redirected to Google.

WebSecurityExpressionRoot:

request Direct public access to the request object.

boolean hasIpAddress(String) Evaluates whether the request matches a particular

IP address or IP mask.

Authentication getAuthentication() Access to the Authentication object.

boolean hasAuthority(String authority) Alias for hasRole.

boolean hasAnyAuthority(String. . . authorities) Alias for hasAnyRole.

boolean hasRole(String role) Determines whether the Authentication object has the

specified role.

boolean hasAnyRole(String. . . roles) Determines whether the Authentication object

has any of the specified roles.

boolean permitAll() Returns true all the time.

boolean denyAll() Returns false all the time.

boolean isAnonymous() Determines whether the current Authentication is an

anonymous one.

boolean isAuthenticated() Determines whether the current Authentication is not an

anonymous one.

boolean isRememberMe() Determines whether the current Authentication

is a remember-me one.

boolean isFullyAuthenticated() Determines whether the current Authentication is

neither a remember-me nor an anonymous one.

boolean hasPermission(Object target, Object permission) Evaluates whether the

current Authentication has certain permissions on the specified domain object. I’ll say more

about this in Chapter 7.

boolean hasPermission(Object targetId, String targetType, Object

permission) Evaluates whether the current Authentication has certain permissions on the

specified domain object. I’ll say more about this in Chapter 7.

Object getPrincipal() Returns the principal from the authentication