Massimiliano Sciacco
personal website

Spring Security JWT Authentication

Submitted on Fri, 04/24/2015 - 21:37

So you've heard about this fabulous JWT and want to implement it into your Spring application? Good, let's see how can be done.

If instead you're wondering what this fancy term refers to, check this article from Jesus Rodriguez Json Web Tokens: Introduction or a more in-depth documentation on the JSON Web Token.

Just to be sure you fully understand how it work, let's summarize it. JWT is a self contained token (literally a string) composed of three blocks: one containing the information of the applicant (the user itself usually) in a JSON format, one containing a signature computed by the server on the content of the JWT and one containing the information on how the signature was computed (the algorithm and encryption method for example). Given that, it's clear that the token must be generated by the server, because it's the only one that knows the secret used to compute the signature. This means that the server must first authenticate the client with another mechanism (for example a basic authentication over secure channel), then generate and send the json web token to the client. I cannot stress it enough: the JWT is not meant to authenticate an user with standard methods (like username and password), but a mechanism to ensure an identity once this has been validated by another mechanism.

What is good for then? For example the fact that the token is JSON based, made it extremely shareable with different applications. Moreover the token is self-contained, so the client just need to resend to the server for each request, and the server just have to check the signature to ensure its validity. No more useless call to database or LDAP.

For more information about it, read this great article from Chris Sevilleja The Ins and Outs of Token Based Authentication.

Libraries

So let's start with some libraries. There are a lot of them out there, but I choose the one from Nimbus for its completeness. Their site is full of examples to brutally copy-paste into our project.

Here the maven dependency

```
<dependency>
    <groupId>com.nimbusds</groupId>
    <artifactId>nimbus-jose-jwt</artifactId>
    <version>3.8</version>
</dependency>
```

For my project the last version was the 3.8. Other than that we don't need anything in particular (except for Spring Security libraries of course).

Filter

We must create a filter to intercept all requests, extract the token and send it to the authentication manager to validate it. As a good practice, the JWT should be stored in an Authorization header usually prefixed with the *Bearer* or *JWT* scheme. For example:

```
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ
```

The name for the schema it's up to you, but for the header I strongly suggest to stick to the standard Authorization.

Having established how we expect to receive our token, let's create a suitable filter to extract it:

```
public class JWTFilter extends GenericFilterBean {

    private AuthenticationEntryPoint entryPoint;
    private AuthenticationManager authenticationManager;
```

```
    public JWTFilter(AuthenticationManager authenticationManager, AuthenticationEntryPoint entryPoint) {
        this.authenticationManager = authenticationManager;
        this.entryPoint = entryPoint;
    }

    @Override
    public void afterPropertiesSet() throws ServletException {
        Assert.notNull(authenticationManager);
    }

    @Override
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException,
            ServletException {
        HttpServletRequest req = (HttpServletRequest) request;
        HttpServletResponse res = (HttpServletResponse) response;

        try {
            String stringToken = req.getHeader("Authorization");
            if (stringToken == null) {
                throw new InsufficientAuthenticationException("Authorization header not found");
            }

            // remove schema from token
            String authorizationSchema = "Bearer";
            if (stringToken.indexOf(authorizationSchema) == -1) {
                throw new InsufficientAuthenticationException("Authorization schema not found");
            }
            stringToken = stringToken.substring(authorizationSchema.length()).trim();

            try {
                JWT jwt = JWTParser.parse(stringToken);
                JWTToken token = new JWTToken(jwt);

                Authentication auth = authenticationManager.authenticate(token);
                SecurityContextHolder.getContext().setAuthentication(auth);
                chain.doFilter(request, response);
            } catch (ParseException e) {
                throw new InvalidTokenException("Invalid token");
            }
        } catch (AuthenticationException e) {
            SecurityContextHolder.clearContext();
            if (entryPoint != null) {
                entryPoint.commence(req, res, e);
            }
        }
    }
}
```

and a token to hold the credentials used by the authentication manager throught the *Authentication* interface:

```
public class JWTToken implements Authentication {

    private static final long serialVersionUID = 1L;

    private JWT jwt;
    private final Collection<GrantedAuthority> authorities;
    private boolean authenticated;
    private ReadOnlyJWTClaimsSet claims;

    public JWTToken(JWT jwt) throws ParseException {
        this.jwt = jwt;
        List<String> roles;
```

```
        try {
            roles = jwt.getJWTClaimsSet().getStringListClaim("roles");
        } catch (ParseException e) {
            roles = new ArrayList<>();
        }
        List<GrantedAuthority> tmp = new ArrayList<>();
        if (roles != null) {
            for (String role : roles) {
                tmp.add(new SimpleGrantedAuthority(role));
            }
        }
        this.authorities = Collections.unmodifiableList(tmp);
        this.claims = jwt.getJWTClaimsSet();
        authenticated = false;
    }

    public JWT getJwt() {
        return jwt;
    }

    public ReadOnlyJWTClaimsSet getClaims() {
        return claims;
    }

    @Override
    public Object getCredentials() {
        return "";
    }

    @Override
    public Object getPrincipal() {
        return claims.getSubject();
    }

    @Override
    public String getName() {
        return claims.getSubject();
    }

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return authorities;
    }

    @Override
    public Object getDetails() {
        return claims.toJSONObject();
    }

    @Override
    public boolean isAuthenticated() {
        return authenticated;
    }

    @Override
    public void setAuthenticated(boolean isAuthenticated) throws IllegalArgumentException {
        this.authenticated = isAuthenticated;
    }

}
```

The filter is pretty straightforward:

1. We create it injecting the authentication manager and an implementation of entry point (optional). The entry point trigger a 401 response with the *WWW-Authenticate* header in case of authentication failure.
2. The filter search for an Authorization header. If found, extract the token removing the schema from it. If the header or the schema is not present, an authentication exception is thrown.
3. We pass the JWT token to the Nimbus JWT parser. If the parsing fails, it means that the token was malformed.
4. If the parse is successfull, we create our JWT Authentication token used by the authentication manager. The token is simply an implementation of the Authentication interface with a parsing of the roles in the constructor. Notes that the "role" claim is not a standard one expected by the JWT specification, rather a custom one used by us to create the authorities that Spring needs to set roles. If you don't need the roles, you can simply return an empty list as authorities. Feel free to adapt this token as you wish. Remember that this token is what Spring will considered the authenticated user when searching for the "logged in" user in the security context holder.
5. The previous token is then passed to the authentication manager to check its validity. If everything goes well, and no authentication exception is thrown, the object returned by the authentication manager is then inserted as an authenticated token into the security context holder and the request proceed towards the desidered controller.

Let's see how the provider verify the validity of the passed token.

Provider

The provider is really simple in fact, as it should only check that the signature matches the content, and the Nimbus library already did this for us. YAI! Here the code:

```
public class JWTAuthenticationProvider implements AuthenticationProvider {

    private JWSVerifier verifier;

    public JWTAuthenticationProvider() {
        this.verifier = new MACVerifier("superSecretKey");
    }

    @Override
    public Authentication authenticate(Authentication authentication) throws AuthenticationException {
        JWTToken jwtToken = (JWTToken) authentication;
        JWT jwt = jwtToken.getJwt();

        // Check type of the parsed JOSE object
        if (jwt instanceof PlainJWT) {
            handlePlainToken((PlainJWT) jwt);
        } else if (jwt instanceof SignedJWT) {
            handleSignedToken((SignedJWT) jwt);
        } else if (jwt instanceof EncryptedJWT) {
            handleEncryptedToken((EncryptedJWT) jwt);
        }

        Date referenceTime = new Date();
        ReadOnlyJWTClaimsSet claims = jwtToken.getClaims();

        Date expirationTime = claims.getExpirationTime();
        if (expirationTime == null || expirationTime.before(referenceTime)) {
            throw new TokenExpiredException("The token is expired");
        }

        Date notBeforeTime = claims.getNotBeforeTime();
        if (notBeforeTime == null || notBeforeTime.after(referenceTime)) {
            throw new InvalidTokenException("Not before is after sysdate");
        }

        String issuerReference = "my.site.com";
        String issuer = claims.getIssuer();
        if (!issuerReference.equals(issuer)) {
            throw new InvalidTokenException("Invalid issuer");
        }

        jwtToken.setAuthenticated(true);
        return jwtToken;
    }

    @Override
```

```
    public boolean supports(Class<?> authentication) {
        return JWTToken.class.isAssignableFrom(authentication);
    }

    private void handlePlainToken(PlainJWT jwt) {
        throw new InvalidTokenException("Unsecured plain tokens are not supported");
    }

    private void handleSignedToken(SignedJWT jwt) {
        try {
            if (!jwt.verify(verifier)) {
                throw new InvalidSignatureException("Signature validation failed");
            }
        } catch (JOSEException e) {
            throw new InvalidSignatureException("Signature validation failed");
        }
    }

    private void handleEncryptedToken(EncryptedJWT jwt) {
        throw new UnsupportedOperationException("Unsupported token type");
    }

}
```

The provider too is very simple:

1. When it's created, we create also a MACVerifier (Nimbus library) used to check the validity of the signature. The verifier requires the secret key with wich the signature was calculated.
2. Note the method *support()*. This is fundamental to allow the authentication manager to trigger this provider when a JWTToken object is passed as Authentication implementation.
3. We extract the Nimbus JWT interface implementation from the token and check it's type with *instanceof* operators. In fact JWT can be plain, signed or encrypted. The first one is obviously discouraged. In the provided code only the signed JWT is implemented. The Nimbus site provides example of encrypted JWT validation.
4. The signature validation is completely handled by the MACVerifier through the method *verify()* of the JWT interface itself.
5. After having validated the signature we can check the claims like *expirationTime, notBeforeTime, issuer*, etc. Adapt this checks on your likings.
6. If all checks pass, set the token to authenticated and return it. The filter will take it, set into the security context holder and forward the request to the controller.

NB: the exceptions are custom made. They are simple extension of the AuthenticationException defined by Spring. You can use whatever exception you like, even the Spring default ones.

Test

To test you can use the Spring Test framework that allow to mock filter behaviour:

```
public class JWTFilterTest {

    @Autowired
    private JWTFilter filter;

    public MockHttpServletResponse doFilter(MockHttpServletRequest request) throws IOException, ServletException {
        MockHttpServletResponse response = new MockHttpServletResponse();
        MockFilterChain chain = new MockFilterChain();
        filter.doFilter(request, response, chain);
        return response;
    }

    @Test
    public void jwtFilterValidTest() throws IOException, ServletException {
        MockHttpServletRequest request = new MockHttpServletRequest("GET", "http://somesecuredomain.com");

        Date now = new Date();
        JWTClaimsSet claimsSet = new JWTClaimsSet();
        claimsSet.setSubject("alice");
        claimsSet.setIssueTime(now);
        claimsSet.setIssuer("my.site.com");
        claimsSet.setExpirationTime(new DateTime().plusHours(1).toDate());
```

```
        claimsSet.setNotBeforeTime(now);

        String token = "Bearer " + this.signAndSerializeJWT(claimsSet, "superSecretKey");
        request.addHeader("Authorization", token);
        MockHttpServletResponse response = doFilter(request);
        Assert.assertEquals(200, response.getStatus());
    }

    private String signAndSerializeJWT(JWTClaimsSet claimsSet, String secret) {
        JWSSigner signer = new MACSigner(secret);
        SignedJWT signedJWT = new SignedJWT(new JWSHeader(JWSAlgorithm.HS512), claimsSet);
        try {
            signedJWT.sign(signer);
            return signedJWT.serialize();
        } catch(JOSEException e) {
            e.printStackTrace();
            return null;
        }
    }
}
```

I've use the JodaTime library to handle the date manipulation for the claim *notBeforeTime*.

Tags: Spring Security JWT
Log in to post comments

Comments

JWTFilter constructor arguments

Permalink Submitted by James (not verified) on Mon, 10/19/2015 - 10:36.

Can you advise how to autowire the constructor arguments for the JWTFilter filter.

Log in to post comments

Hi James,

Permalink Submitted by Max on Mon, 10/19/2015 - 19:25.

Hi James,

if your beans (the authentication entry point and authentication manager) are already defined as Spring bean (for example with the @Component annotation over their class name), you can simply inject them in the Filter with the @Autowired annotation over the constructor.

Alternatively you can inject them with the xml configuration file. Something like this:

```
<bean id="entryPoint" class="it.massimilianosciacco.security.DefaultEntryPoint">
    <property name="realmName" value="Secure realm" />
</bean>

<bean id="filter" class="it.massimilianosciacco.security.JWTFilter">
    <constructor-arg name="authenticationManager" ref="authenticationManager" />
    <constructor-arg name="entryPoint" ref="entryPoint" />
</bean>

<bean id="provider" class="it.massimilianosciacco.security.JWTAuthenticationProvider" />

<sec:authentication-manager alias="authenticationManager">
    <sec:authentication-provider ref="provider" />
</sec:authentication-manager>
```

You can check my other tutorial for an example: http://www.massimilianosciacco.com/implementing-hmac-authentication-rest-api-spring-security

Log in to post comments

Thanks

Permalink Submitted by James (not verified) on Tue, 10/20/2015 - 13:04.

Hey, forgot to mention that your article helped a lot, I have now a demo which work at https://github.com/tekbird/springjwt

Log in to post comments

Nice work

Permalink Submitted by Max on Wed, 10/21/2015 - 19:33.

Nice work! Glad it helps.

Keep up the good work.

Log in to post comments

Question about exceptions

Permalink Submitted by Jacek (not verified) on Sun, 11/15/2015 - 12:56.

Hello. Thank you for the article. Do TokenExpiredException, InvalidTokenException and InvalidSignatureException are your custom exception classes or are they from some kind of a library? I don't see such classes in Nimbus Jose + JWT.

Log in to post comments

Hi. Yes, the exception are

Permalink Submitted by Max on Mon, 11/16/2015 - 19:27.

Hi. Yes, the exceptions are custom created. You could use the Spring built-in or create your own exception. Maybe I'll specify better in the tutorial.

Log in to post comments

Problem with intercepting urls

Permalink Submitted by Jacek (not verified) on Sun, 11/15/2015 - 19:34.

Hello again. I have configured my app just like you but with one difference. I wanted to have an url which is accessible with no authentication. The problem is that even though I marked it as access=permitAll(), it is still processed by the filter and 401 is thrown. Can you check? BTW. these urls are mapped by Spring Controller. This is my http tag:

```
<security:http entry-point-ref="mobileJWTAuthenticationEntryPoint"
authentication-manager-ref="mobileJWTAuthenticationManager"
create-session="stateless"
use-expressions="true">
<security:custom-filter ref="mobileJWTAuthenticationFilter" position="FORM_LOGIN_FILTER" />
<security:intercept-url pattern="/services/public/mobileFBAuth" access="permitAll()"/>
<security:intercept-url pattern="/services/restAPI/**" access="isAuthenticated()" />
</security:http>
```

Log in to post comments

Try with another http section block

Permalink Submitted by Max on Mon, 11/16/2015 - 19:39.

Have you tried to create another http block with security="none" before the secure one? Like this:

```
<security:http pattern="/services/public/mobileFBAuth" security="none"/>
```

Log in to post comments

Thank you for your attention,

Permalink Submitted by Jacek (not verified) on Sat, 11/21/2015 - 16:49.

Thank you for your attention, Actually it's not the case. I've solved it myself - to permit all requests the authentication filter has to be modified - instead of lines in which you throw an exception there have to be these lines:

chain.doFilter(request, response);
return;

Log in to post comments

A public endpoint should never hit the JWT filter

Permalink Submitted by Max on Sun, 11/22/2015 - 18:02.

That's strange, because the "public" endpoint should never hit the filter in the first place. Well, if it works for you, is good enough :)

Log in to post comments

Source code

Permalink Submitted by Alex (not verified) on Tue, 01/05/2016 - 12:13.

Hello. Thank you for this awesome guide! Could you please also post a working code(project) on GitHub ?

Log in to post comments

Great!

Permalink Submitted by Luis (not verified) on Sat, 02/27/2016 - 12:38.

Well done,thank you for the clear explanation and congratulations for the article!

Log in to post comments

ClassCastException for JWTAuthenticationProvider

Permalink Submitted by Dian (not verified) on Wed, 05/11/2016 - 05:48.

I followed this tutorial to implement JWT authentication in Spring Boot. However I got "ClassCastException: org.springframework.security.authentication.UsernamePasswordAuthenticationToken cannot be cast to org.mdacc.rists.ristore.ws.security.jwt.JWTToken" for "JWTToken jwtToken = (JWTToken) authentication;" in the authenticate method of JWTAuthenticationProvider. What did I do wrong? My code is at https://github.com/dnjiao/ristore-web-services/tree/master/src/main/java...

Log in to post comments

check the support function

Permalink Submitted by Max on Wed, 05/11/2016 - 22:39.

Hi Dian,

the problem is in the support function.

You wrote

```
public boolean supports(Class<?> authentication) {
    return true;
}
```

when it actually must contains

```
public boolean supports(Class<?> authentication) {
    return JWTToken.class.isAssignableFrom(authentication);
}
```

Why is that? Because Spring use the authentication token created by the filter and pass it to all the registered authentication providers. How did he knows what provider can understand (and thus authenticate) the passed token? By calling the support function. If it returns true, then the provider can handle the token, otherwise is skipped and the next provider in the configuration is tried.

This workflow is described in the code here at the lines 156-159.

Since your provider implementation always returns true, it means that should be able to handle every kind of tokens, included the UsernamePasswordAuthenticationToken (which is usually used on basic and form authentication), which is not true. It can handle only JWTToken instances of tokens.

As of why an UsernamePasswordAuthenticationToken is passed, I cannot tell, because the sources you provide don't include it. Maybe you have a form that took the username, password couple and send them to another filter and then after, that a JWT is created. The problem is, that when the first form triggers, since your provider always returns true at the support function, it tries to handle the login form data too.

Max

Log in to post comments

Wire Up

Permalink Submitted by Ilja (not verified) on Thu, 10/06/2016 - 20:32.

Hello
Thanks for the great explanation.
How to wire up these classes with a spring boot application?

Thank you!
Regards

Log in to post comments