codecentric Blog (https://blog.codecentric.de/)

Overview (https://blog.codecentric.de/en/category/java-en/)

Understanding IoT (Part 1) (https://blog.codecentric.de/en/2018/08/understanding-iot-part-1/)

X.509 client certificates with Spring Security (https://blog.codecentric.de/en/2018/08/x-509-client-certificates-with-spring-security/)

08/21/18 by **Ognjen Mišić (https://blog.codecentric.de/en/author/ognjen-misic/)**

No Comments (https://blog.codecentric.de/en/2018/08/x-509-client-certificates-with-spring-security/#comments)

A disclaimer: this blogpost is a story about the reasons why I ended up securing my API using the X.509 client certificate, in addition to a step-by-step guide on how to implement this yourself. Someone will hopefully find it useful.

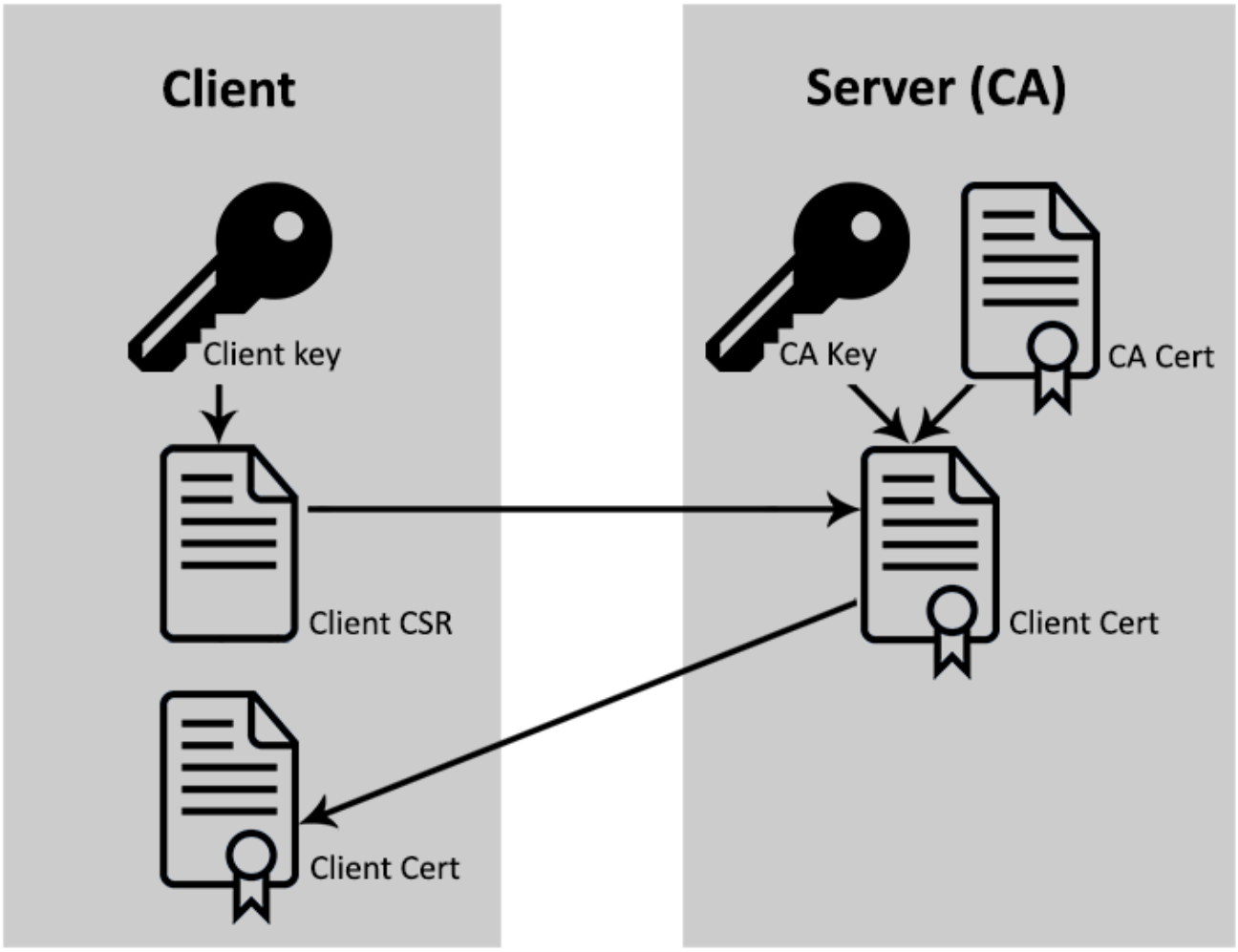
Securing your application or an API is always a challenge, and lack of experience with the topic makes it even more complicated. Deciding on what security approach to take, how to properly implement it, what vectors of attacks you'll be vulnerable to, dealing with the soup of acronyms and nouns such as SSL/TLS, CA, CRT, public/private keys, keystore, truststore – you quickly find yourself with a panicky feeling in your stomach. And this is a pretty common reaction.

First of all, X.509 is a digital certificate which uses the X.509 public key infrastructure standard to verify that a public key, which belongs to a user, service or a server, is contained within the certificate, as well as the identity of said user, service, or server.
The certificate can be signed by a *trusted certificate authority*, or *self-signed*.
SSL and TLS are most widely known protocols which use the X.509 format. They are routinely used to verify the identity of servers each time you open your browser and visit a webpage via HTTPS.

The goal in mind is to secure communication from a known server to my service. The decision ultimately came down to use the client certificate approach since authenticating users is not my concern – users do not interact with me directly. This means that there are no username/passwords being sent back and forth, no cookies and no sessions – which means that we maintain statelessness of our REST API. Also, as I am the certificate authority, I'm always going to stay in control of who gets a valid certificate, meaning I only trust myself to manage and maintain who can talk to my service.

The general workflow

In order to secure and authenticate communication between client and the se cates. When you send a browser request to an HTTPS website, your browser would just ve authority. In this case, not only the server's identity is verified, but also the server gets to



(https://blog.codecentric.de/files/2018/08/client-crt.png)

The first thing the client has to do in order to communicate with the secured service is to generate a private key and a certificate signing request (CSR). This CSR is then sent to a Certificate Authority (CA) to be signed. In my use case, I represent both the server and the CA, since I want to be in charge of managing who gets to talk to my service. Signing the CSR produces the client certificate which is then sent back to the client.
In order to send a valid and authenticated HTTPS request, the client also needs to provide the signed certificate (unlocked with the client's private key), which is then validated during the SSL handshake with the trusted CA certificate in the Java truststore on the server side.

Enough theory, let's see what the implementation looks like.

Spring Security Configuration

My REST service is a regular spring-boot 2.0.2 app using the spring-boot-starter-security dependency:

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```


The configuration class:

```
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

        /*
         * Enables x509 client authentication.
         */
        @Override
        protected void configure(HttpSecurity http) throws Exception {
                // @formatter:off
        http
                .authorizeRequests()
                    .anyRequest()
                        .authenticated()
                .and()
                    .x509()
                .and()
                    .sessionManagement()
                        .sessionCreationPolicy(SessionCreationPolicy.NEVER)
                .and()
                    .csrf()
                        .disable();
        // @formatter:on
        }

        /*
         * Create an in-memory authentication manager. We create 1 user (localhost which
         * is the CN of the client certificate) which has a role of USER.
         */
        @Override
        protected void configure(AuthenticationManagerBuilder auth) throws Exception {
                auth.inMemoryAuthentication().withUser("localhost").password("none").roles("USER");
        }
}
```

Usually known to be cumbersome, in this case the *SpringSecurityConfig* class is pretty lightweight, since we want to authenticate all requests coming into the service, and we want to do so using x509 authentication.
SessionCreationPolicy.NEVER tells Spring to not bother creating sessions since all requests must have a certificate.
We can also disable cross-site request forgery protection since we aren't using HTML forms, but only send REST calls back and forth. You must do so if you're going to follow this blog to the end, because CURL requests won't pass through Spring's csrf filter.

Enabling HTTPS on the REST service itself is just a manner of setting a couple of properties in our *application.properties* file:

```
server.port=8443
server.ssl.key-store=classpath:keystore.p12
server.ssl.key-store-password=changeit
server.ssl.trust-store=classpath:truststore.jks
server.ssl.trust-store-password=changeit
server.ssl.client-auth=need
```

And this is pretty much it, you can go on and create your `@RestControllers` with endpoints fully secured behind a x509 certificate.

Generating a server CA certificate

Let's see what has to be done on the server's side with regards to creating the certificate:

```
openssl genrsa -aes256 -out serverprivate.key 2048
```


First of all, we have to generate an rsa key encrypted by aes256 encryption w ld be more secure, but the handshake would be slowed down quite significantly. 1024 is is obviously less secure. Used **server** as pass phrase here.

```
openssl req -x509 -new -nodes -key serverprivate.key -sha256 -days 1024 -out serverCA.crt
```

Now, we use the generated key in order to create a x509 certificate and sign it with our key. A form must be filled out which will map the certificate to an identity. Most of the fields can be filled out subjectively, except the **CN** (common name) which must match the domain we are securing (in this case, it's localhost).

```
keytool -import -file serverCA.crt -alias serverCA -keystore truststore.jks
```

imports our server CA certificate to our Java truststore. The stored password in this case is **changeit**.

```
openssl pkcs12 -export -in serverCA.crt -inkey serverprivate.key -certfile serverCA.crt -out keystore.p12
```

exports the server CA certificate to our keystore. The stored password is again **changeit**.

Note: you could use *.jks* as the format of the keystore instead of .p12, you can easily convert it with:

```
keytool -importkeystore -srckeystore keystore.p12 -srcstoretype pkcs12 -destkeystore keystore.jks -deststoretype JKS
```

Generating a client certificate

The client has to go through a similar process:

```
openssl genrsa -aes256 -out clientprivate.key 2048
```

Again, the first thing we have to do is to create the private key. Interactively asks for a passphrase, I'm using **client** here.

```
openssl req -new -key clientprivate.key -out client.csr
```

Now we create the certificate signing request and sign it with the client's private key. We are asked to fill a form to map the identity to the output certificate. Similar to the step 2 when generating the Server CA, the **CN** is the most important field and must match the domain.

Client sends the CSR to the CA

```
openssl x509 -req -in client.csr -CA serverCA.crt -CAkey serverprivate.key -CAcreateserial -out client.crt -days 365 -sha256
```

CA does this step, not the client. We sign the certificate signing request using the server's private key and the CA.crt. *client.crt* is produced, and it has to be securely sent back to the client.

Certificates in action

Now that we have everything configured and signed, it's time to see if it all ties in properly.
First thing, we can send a request without the certificate:

```
curl -ik "https://localhost:8443/foo/"
```

and this will produce an error, just as we would have hoped:

```
curl: (35) error:14094412:SSL routines:SSL3_READ_BYTES:sslv3 alert bad certificate
```

This time we create a request with the certificate (using the client's private key):

```
curl -ik --cert client.crt --key clientprivate.key "https://localhost:8443/foo/"
```

*at this point we are asked for the key's passphrase, type in **client***
produces a nice "200 OK" response!

```
HTTP/1.1 200
X-Content-Type-Options: nosniff
X-XSS-Protection: 1; mode=block
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
Strict-Transport-Security: max-age=31536000 ; includeSubDomains
X-Frame-Options: DENY
Content-Type: text/plain;charset=UTF-8
Content-Length: 12
Date: Fri, 10 Aug 2018 11:39:51 GMT

hello there!%
```

Example POST request:

```
curl -ik --cert client.crt --key clientprivate.key -X POST -d '{"greeting": "Hello there"}' "https://localhost:8443/foo/"
```

*type in **client** as before*

```
HTTP/1.1 201
X-Content-Type-Options: nosniff
X-XSS-Protection: 1; mode=block
Cache-Control: no-cache, no-store, max-age=0, must-revalidate
Pragma: no-cache
Expires: 0
Strict-Transport-Security: max-age=31536000 ; includeSubDomains
X-Frame-Options: DENY
Content-Type: text/plain;charset=UTF-8
Content-Length: 15
Date: Fri, 10 Aug 2018 12:02:33 GMT

Hello there GENERAL KENOBI!%
```

You can set

```
logging.level.org.springframework.security=DEBUG
```

in your application.properties to trace the handshake.

```
2018-08-16 16:24:40.190 DEBUG 7206 --- [nio-8443-exec-3] o.s.s.w.a.p.x.X509AuthenticationFilter   : X.509 client authentic
[
  Version: V1
  Subject: EMAILADDRESS=ognjen.misic@client.com, CN=localhost, O=DS, L=Be                                    C=DE
  Signature Algorithm: SHA256withRSA, OID = 1.2.840.113549.1.1.11

  Key:  Sun RSA public key, 2048 bits
  modulus: 23780269493490771497396618182382760925123234234245678323529966                                  )7002479861
  public exponent: 65537
  Validity: [From: Fri Aug 10 13:35:10 CEST 2018,
               To: Sat Aug 10 13:35:10 CEST 2019]
  Issuer: EMAILADDRESS=ognjen.misic@codecentric.de, CN=localhost, OU=Banja Luka office, O=cc, L=Solingen, ST=Whatever, C=D
  SerialNumber: [    aecc9b1c 2b56df2d]

]
  Algorithm: [SHA256withRSA]
  Signature:
0000: 69 97 0A EF 5C F8 64 58   50 C8 A4 A5 33 86 0B 6A  i...\.dXP...3..j
0010: 64 24 D9 90 BF CF FB EC   7B AC E9 3C 23 88 81 7E  d$.........<#...
0020: 66 11 77 87 A8 AF 52 49   C9 8F F4 7B 2D 9F F2 50  f.w...RI....-..P
0030: FF 76 38 C1 89 2B 56 A8   26 21 DA 7B C1 A7 D1 13  .v8..+V.&!......
0040: 2B 84 5D 14 2C FD F6 B1   23 28 A3 DB A6 35 BB 97  +.].,...#(...5..
0050: 11 60 E5 58 24 42 68 91   43 21 BD E3 75 34 A8 14  .`.X$Bh.C!..u4..
0060: F7 E1 95 01 E6 E0 79 9E   86 E8 8D D4 64 DD 77 CF  ......y.....d.w.
0070: 27 1B A4 H4 25 8E AF 36   49 C9 2C 7D 0F 2A 6C 11  '...%..6I.,..*l.
0080: C6 3A DE 02 7F 06 91 CF   73 3B 4F E8 81 E5 54 E1  .:......s;O...T.
0090: 2B CB D8 DD FE EB 64 8B   A3 5A 15 EB 86 D4 11 9D  +.....d..Z......
00A0: B1 F8 57 FF FA A1 2E B0   AF B5 D9 71 21 25 9F 0F  ..W........q!%..
00B0: 18 33 A4 M9 CA E5 C4 83   A8 28 00 81 DF 81 20 E9  .w.......w.... .
00C0: 45 FA 37 F3 20 07 19 51   1F AE BA FD 79 A8 C9 6D  E.7. ..Q....y..m
00D0: 82 7D 1A C8 B5 7A 40 19   38 76 0E AF 52 F3 AB 87  .....z@.8v..R...
00E0: 01 05 B9 94 79 EA 4B 20   19 74 6B 4B 84 E6 6F CE  ....y.K .tkK..o.
00F0: E8 BB F3 F3 A5 54 DF EB   5D 6B A6 8F 15 5E 36 28  .....T..]k...^6(

]
2018-08-16 16:24:40.190 DEBUG 7206 --- [nio-8443-exec-3] .w.a.p.x.SubjectDnX509PrincipalExtractor : Subject DN is 'EMAILAD
2018-08-16 16:24:40.190 DEBUG 7206 --- [nio-8443-exec-3] .w.a.p.x.SubjectDnX509PrincipalExtractor : Extracted Principal na
2018-08-16 16:24:40.192 DEBUG 7206 --- [nio-8443-exec-3] o.s.s.w.a.p.x.X509AuthenticationFilter   : preAuthenticatedPrinci
```

We can see that the received certificate is signed by our own trusted serverCA.crt (Issuer: EMAILADDRESS being ognjen.misic@codecentric.de – the email was set in the second step when generating the serverCA.crt, and the Subject: EMAILADDRESS is ognjen.misic@client.com, the value that was set when the client was generating the CSR).

The security principal:

```
o.s.s.w.a.p.x.X509AuthenticationFilter   : Authentication success: org.springframework.security.web.authentication.preauth
```

And that would be it!

Special thanks to Jonas Hecht, whose example helped me quite a bit to grasp the workflow of this topic (you can find it here: https://github.com/jonashackt/spring-boot-rest-clientcertificate (https://github.com/jonashackt/spring-boot-rest-clientcertificate)) and to Daniel Marks (https://blog.codecentric.de/en/author/daniel-marks/), for helping me fill out the missing pieces of the puzzle.

Tags

X509 (HTTPS://BLOG.CODECENTRIC.DE/EN/TAG/X509/)

Ognjen Mišić (https://blog.codecentric.de/en/author/ognjen-misic/)

Software developer at codecentric Bosnia since April 2016.

Post by **Ognjen Mišić**

SECURITY

A good password (https://blog.codecentric.de/en/2019/06/good-password/)

More content about **Java**

JAVA

RFC-7807 problem details with Spring Boot and JAX-RS (https://blog.codecentric.de/en/2020/01/rfc-7807-problem-details-with-spring-boot-and-jax-rs/)

JAVA

DON'T make an ASS out of U and ME when dealing with Hibernate caching! (https://blog.codecentric.de/en/2019/07/hibernate-caching/)

Comment

Nachricht

Name

E-Mail

Save my name, email, and website in this browser for the next time I comment.

SUBMIT