Java Advanced – lesson 5 – Security

Wim Wiltenburg

(wim.wiltenburg@inholland.nl)

Objectives

- Web Application Security
- HTTPS
- Filters
- Spring Security
- Token based authorization
- User based authorization

Web application security

Web application security is a branch of **information security** that deals specifically with security of websites, web applications and web services.

At a high level, web application security draws on the principles of application security but applies them specifically to **internet and web systems**

The majority of web application attacks [...] are made possible by flawed coding and failure to **sanitize application inputs and outputs**

Top 10 threats to security

- Injection
- **Broken Authentication**
- Sensitive Data Exposure
- XML External Entities (XXE)
- **Broken Access Control**
- **Security Misconfiguration**
- Cross-Site Scripting (XSS)
- Insecure Deserialization
- **Using Components with Known Vulnerabilities**
- Insufficient Logging and Monitoring

Source: https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf (read this!)

Web security tag cloud

HTTPS

The first thing we can do to secure our application is to encrypt our connection using HTTPS.

This will prevent unwanted listening in on our network level transmission of data.

If we wish to encrypt our data, we need to provide a private/public key pair, that we store in a file (keystore), to be presented when a client connects to our server.

The client and our server do a handshake and determine a shared way of communicating securely

Creating a keystore

Keystores come in different formats, e.g. PKCS12, JKS, etc. PKCS12 is very common and these keystores are suffixed by .p12

We create a keystore by using the keytool command:

```
C:\Users\wwilt>keytool -genkey -alias inholland -keyalg RSA -validity 365 -keystore inholland.p12 -storetype PKCS12
Enter keystore password:
Re-enter new password:
What is your first and last name?
  [Unknown]:  Hogeschool Inholland Informatica
What is the name of your organizational unit?
  [Unknown]:  TOI
What is the name of your organization?
  [Unknown]:  Hogeschool Inholland
What is the name of your City or Locality?
  [Unknown]:  Haarlem
What is the name of your State or Province?
  [Unknown]:  Noord-Holland
What is the two-letter country code for this unit?
  [Unknown]:  NL
Is CN=Hogeschool Inholland Informatica, OU=TOI, O=Hogeschool Inholland, L=Haarlem, ST=Noord-Holland, C=NL correct?
  [no]:  yes
```

Configure HTTPS

What we do now is configure the application to accept https, by placing the keystore in the classpath, and configure the application.properties:

```
server.port=8443 # Now the server listens on 8443 instead of 8080
server.ssl.key-store=inholland.p12
server.ssl.key-store-type=PKCS12
server.ssl.key-alias=inholland
server.ssl.key-store-password=inholland
```

Now we use the following url to access our application:

https://localhost:8443/guitars

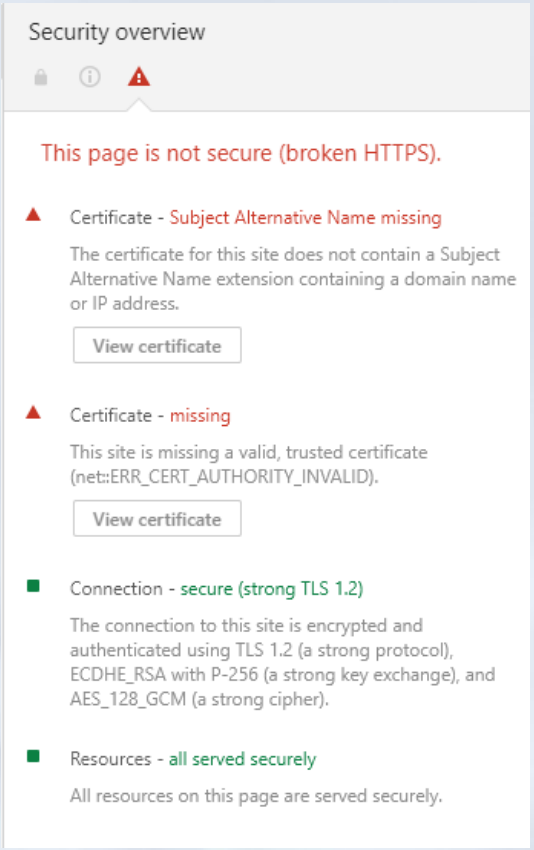
(When using Postman, you need to turn off SSL certificate verification in Settings > General)

Verifying SSL

To verify we are actually using a secure connection, we go into our browser's developers tool window, and check the security tab:

This is not a perfect certificate, but: the connection is secure (using TLS 1.2) with all resources served securely.

The errors can be fixed by using a certificate issued by a Certificate Authority, like VeriSign



HTTP Filter

Spring Boot is basically a web server. It can process requests, and produces a response. We can apply logic to the request and the response before the business logic is activated by applying a filter.

An example would be to intercept a request before the application processes it, and reject the request is the size is too large. It's a common Denial Of Service attack model, by occupying the server by sending a large amount of large requests, e.g. 10M

A filter on size would prevent that.

Interface javax.servlet.Filter


```
package javax.servlet;

import java.io.IOException;

public interface Filter {
    default void init(FilterConfig filterConfig) throws ServletException {
    }

    void doFilter(ServletRequest var1, ServletResponse var2, FilterChain var3) throws
IOException, ServletException;

    default void destroy() {
    }
}
```

A FilterChain object can chain different filters together, if you want more than one filter, and that is usually the case.

Implementing the Filter

- If you want to implement your own filter you create a class that implements Filter. In the doFilter(…) method you can implement your own logic, e.g. filter on a request size that is over 60 bytes.
- A filter must be ready to be applied as the application starts, so it must be a @Component. And if you want more than one filter, you need to specify which filter comes first. Therefore you annotate the class with @Order(int), e.g. @Order(1)
- This class will apply logic to each http request and each http response, as specified in the arguments in doFilter(…)

LargeRequestFilter


```
@Component
@Order(1)
public class LargeRequestFilter implements Filter {

    private final Logger logger = Logger.getLogger(this.getClass().getName());

    @Override
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws
IOException, ServletException {

        int size = request.getContentLength();
        logger.info("Request size: " + size);
         if (size >  100) {
             logger.severe( "request with size " + size + " was rejected");
             throw new ServletException("Request too large");
         } else {
             chain.doFilter(request, response); // pass on to the next filter
         }
    }
}
```

Result: 2019-05-18 14:35:17,497 ERROR [https-jsse-nio-8443-exec-5] n.i.filter.LargeRequestFilter: request with size 62 was rejected

Spring Security

The library spring-web-starter-security works on the level of filters. Each request gets evaluated if the proper authentication and/or authorization is in place.

- Authentication: confirmation of identity, e.g. by
  - Username/password
  - LDAP
  - Certificate
  - Custom authentication mechanism
- Authorization: which functions can be performed, depending on the role, e.g. user or administrator

Case: use header token

We can secure our application by sending a http header with a specific token, or API key. Many APIs on the internet use this mechanism, e.g. Google Cloud Message (nowadays: Google Firebase Messaging) for sending push messages to mobile phones.

When you register as a user on the platform, this key is generated and used *every time the users want to call the API as a registered user*. Non registered users get denied.

It must not come as a surprise that this key must remain secret (keep it out of your code base)

Step one: the token

We need to generate a token, and store it, so that when the API gets called with that token, the request is let through.

For now we create a simple class that serves as an entity and a repository

```
@Entity
@Data
@NoArgsConstructor
public class ApiKey {
    @Id
    private String key;

    public ApiKey(String key) {
        this.key = key;
    }
}
```

```
@Repository
public interface ApiKeyRepository extends CrudRepository<ApiKey, String> {
}
```

Step two: the filter

In this scenario, the application assumes that you have already been authenticated in another way, and are therefore using "pre-authentication". The filter we provide extends AbstractPreAuthenticatedProcessingFilter.

```
public class APIKeyAuthFilter extends AbstractPreAuthenticatedProcessingFilter {

    private String headerName;

    public APIKeyAuthFilter(String headerName) {
        this.headerName = headerName;
    }

    @Override
    protected Object getPreAuthenticatedPrincipal(HttpServletRequest httpServletRequest) {
        return httpServletRequest.getHeader(headerName);
    }

    @Override
    protected Object getPreAuthenticatedCredentials(HttpServletRequest httpServletRequest) {
        return "N/A";
    }
}
```

Step three: Security Config

The Security Configuration class should extend WebSecurityConfigurerAdapter and override the configure method.

This class needs:

- A repository for getting API keys
- The name of the header that keeps the key
- The filter that takes the value from the header
- An AuthenticationManager object
- An Authentication object
- HttpSecurity object

Step three: Security Config (2)


```
@Configuration
@EnableWebSecurity
@Order(2)
public class APISecurityConfig extends WebSecurityConfigurerAdapter {


    private ApiKeyRepository apiKeyRepository;

    public APISecurityConfig(ApiKeyRepository apiKeyRepository) {
        this.apiKeyRepository = apiKeyRepository;
    }

    @Value("X-AUTHTOKEN")
    private String headerName;

    @Override
    protected void configure(HttpSecurity httpSecurity) throws Exception {
        APIKeyAuthFilter filter = new APIKeyAuthFilter(headerName);
        filter.setAuthenticationManager(authentication -> {
            String principal = (String) authentication.getPrincipal();

            if (!apiKeyRepository.findById(principal).isPresent()) {
                throw new BadCredentialsException("API Key was not found on the system");
            }
            authentication.setAuthenticated(true);
            return authentication;
        });
```

This is just the setup. The actual authorization comes in HttpSecurity

Step four: HttpSecurity

HttpSecurity basically an authorization builder. It allows you to chain methods and constructors, so it becomes readable, e.g.

```
httpSecurity
        .antMatcher("/guitars/**")
        .csrf().disable()    // disable X-site request forgery
        .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS) // If Stateless every
requests needs authentication
        .and()
        .addFilter(filter).authorizeRequests() // authorize all requests that has a correct header value
        .anyRequest().authenticated();  // all requests are authenticated
```

User authentication

When authenticating a user *we create a session, that will last as long as the user is logged in*, or until the browser closes

For this we need a different Authentication Manager, that can create users in memory, and assigns a role to them: InMemoryAuthenticationManager:

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.inMemoryAuthentication()
            .withUser("user")
            .password("{noop}password") // {noop} means password is being sent without encoding
            .roles("USER") // user has role USER
            .and()
            .withUser("admin")
            .password("{noop}password")
            .roles("ADMIN"); // admin has role ADMIN
}
```

HttpSecurity

Now when a user logs in a session will be created in which the role is set, and now we can configure the security on the endpoints:

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.csrf().disable();
    http.authorizeRequests()
            .antMatchers("/").permitAll()// Everybody can see root
            .antMatchers(HttpMethod.POST, "/guitars/**").hasRole("ADMIN") // Only Admin can POST
            .antMatchers(HttpMethod.GET,"/guitars/**").permitAll() // All users can GET
            .anyRequest().authenticated()
            .and()
            .formLogin().permitAll(); // The login page can be seen by everybody
}
```

Homework

Secure your API with API Key authentication