Spring framework 5 - Security

Tipos de chicos malos

Como desarrollador de software, debes proteger tu sitio web, existen 3 tipos de chicos malos:

- **Impersonators**(Imitadores)
- **Upgraders**(Mejoradores)
- **Eavesdropper**(Espía)

Imitadores

Los imitadores(**Impersonators**) son personas finjen ser alguien que no son para poder acceder a algún recurso.

En seguridad evitamos este tipo de ataques a través de **Autenticación**, este es el proceso de validar que eres quien dices ser. Para hacerlo debes de contar con:

- **Ser alguien**: Authenticación a través de huella o retina
- **Tener algo**: Authenticación a través de un certificado
- **Saber algo**: Authenticación a través de usuario y contraseña

Muchos sistemas realizan una doble authenticación, a esto se le conoce como **MFA (Multi-Factor Authentication)**. De este modo se combina el saber algo con el tener algo.

Mejoradores

Los mejoradores (**Upgraders**) son personas que acceden a un sistema como un usuario regular y buscan la forma de obtener privilegios de miembros "**premium**".

En seguridad evitamos este tipo de ataques a través de **Autorización**, este es el proceso que valida que tienes permisos suficientes para acceder al recurso solicitado. Esto se consigue a través de la definición de **ROLES**.

Un **ROL** es un grupo de usuarios que tiene acceso a un conjunto de recursos especificados. Un usuario puede tener múltiples roles.

Espías

Un espía(**Eavesdropper**) es una persona que no trata de acceder al sistema pero trata de intervenir los canales de comunicación para analizar la información que viaja sobre el.

En seguridad evitamos este tipo de ataques a través de **Confidencialidad**, este es el proceso a través del cual se asegura la privacidad de los usuarios. El cifrado de los datos es un método común para asegurar la confidencialidad de una aplicación.

Configuración

Para habilitar Spring security se debe incluir la siguiente dependencia en nuestra aplicación:

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

Si la dependencia anterior se encuentra en el *classpath*, la aplicación será segura por *default* a través de **basic authentication** y **form login**, con las siguientes credenciales:

- **User**: user
- **Passowrd**: Al iniciar la aplicación se imprimirá un *log* como el siguiente indicando el *password*:

```
Using generated security password:
e60bf7e9-542f-4154-8a97-51f3167e267f
```

Cambio de *password*

Para cambiar el *password* por defecto es posible incluir las siguientes líneas en el archivo application.properties:

```
spring.security.user.name
spring.security.user.password
```

In memory authentication

Para habilitar autenticación en memoria, se incluirá la siguiente clase de configuración:

```
@Configuration
@EnableWebSecurity
public class SecurityJavaConfig extends
WebSecurityConfigurerAdapter {

    @Override
    protected void
configure(AuthenticationManagerBuilder auth) throws
Exception {
        auth.inMemoryAuthentication().
        withUser("admin").
        password(encoder().encode("password")).
        roles("ADMIN");
    }
    @Bean
    public PasswordEncoder encoder() {
        return new BCryptPasswordEncoder();
    }
}
```

La clase anterior define un nuevo **user**, **password** y **rol**, el *bean* **PasswordEncoder** nos permitirá cifrar en SHA-1.

Definición de recursos a proteger

Ahora incluiremos el siguiente método para definir los recursos que se deberán proteger:

```
@Override
protected void configure(HttpSecurity http) throws
Exception {
    http
    .csrf().disable()
    .authorizeRequests()
    .antMatchers("/roles/**").hasRole("ADMIN")
    .and()
    .httpBasic() ;
}
```

La definición anterior indica lo siguiente:

- Se deshabilita el soporte para **csrf**
- **authorizeRequests**() Permite restringir el acceso a recursos
- **antMatchers**(...) Define los recursos a proteger
- **hasRole**(...) Indica el rol que debe tener el usuario para acceder al recurso especificado
- **httpBasic**() Configura ***Basic Authentication***

Uso de multiples roles para diferentes recursos

En el ejemplo anterior definimos un recurso con un solo rol, ahora definiremos múltiples recursos con diferentes roles:

```
@Override
protected void configure (HttpSecurity http) throws
Exception {
    http.csrf().disable().authorizeRequests()
    .antMatchers("/users/**")
    .hasRole("ADMIN")
    .antMatchers("/roles/**")
    .authenticated()
    .and().httpBasic();
}
```

DEVS4J
Hecho por Devs para Devs

Autenticación contra una base de datos

En el ejemplo anterior se mostró como autenticar contra credenciales definidas en memoria, en caso de que se desee realizar contra una base de datos se deberá implementar la interfaz **UserDetails** como se muestra a continuación:

```
@Service
public class Devs4jUserDetailsService implements
UserDetailsService {

    @Autowired
    private UserRepository userRepository;

    @Autowired
    private UserInRoleRepository userInRoleRepository;

    @Override
    public UserDetails loadUserByUsername(String
username) throws UsernameNotFoundException {
        Optional<User> optional =
     userRepository.findByUsername(username);
        if (optional.isPresent()) {
            User user = optional.get();
            List<UserInRole> userInRoles =
        userInRoleRepository.findByUser(user);
             String[] roles =
            userInRoles.stream()
            .map(r->r.getRole().getName())
            .toArray(String[]::new);
            return
org.springframework.security.core.userdetails.
            User.withUsername(user.getUsername())
            .password(passwordEncoder()
            .encode(user.getPassword()))
            .roles(roles).build();
        } else {
            throw new
UsernameNotFoundException("Username " + username +
" not found");
        }
 }
    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```

De la definición anterior podemos analizar lo siguiente:

- El desarrollador no es reponsable de validar el *password*, solo de obtenerlo de la base de datos y devolverlo
- El bean **PasswordEncoder** se utiliza para cifrar los *passwords*.
- El método roles recibe un var args de String, por esto se debe transformar de UserInRole a String[].
- En caso de que no se encuentre el usuario se generará una **UsernameNotFoundException**.