APIs WEB Segura

Como aplicar o spring-security corretamente

****

**Antonio Luis Reuter Andrade**

aandrade@avenuecode.com

Antonio Reuter é bacharel em Ciências da Computação pela Faculdade Ruy Barbosa. Trabalha com desenvolvimento de sistemas desde 2000, utilizando Java e JEE. Atualmente é consultor da AvenueCode trabalhando em projetos envolvendo Java/JEE e Ruby on Rails no e-commerce do Walmart.

Cenário

Nesse artigo iremos abordar como tornar uma API WEB segura, através do spring-security. Utilizaremos como base a API de uma loja virtual fictícia, demonstrando passo a passo como blindá-la da maioria dos tipos de ataque, além de apresentar os erros mais comuns ao tentar nesse caminho para torna-la impenetrável.

Introdução

*<< discutir sobre o problema que será tratado no artigo>>*

Neste artigo iremos abordar como tornar a API de uma loja virtual segura, uma vez que o dono da loja deseja expô-la na WEB para que seus parceiros comerciais consigam integrar em suas plataformas. Apresentaremos os erros mais comuns e como evita-los.

*<< descrever brevemente a solução comumente utilizada e que está inadequada >>*

A aplicação do nosso cliente já faz uso do spring-security, uma vez que a API em questão já vinha sendo utilizada internamente para permitir a integração com outros sistemas internos, contudo ao tentar disponibiliza-la na WEB algumas medidas precisam ser adotadas.

*<< descrever brevemente a solução que resolve o problema de uma forma mais adequada >>*

Para garantir que nossa API continue segura, e que os dados dos nossos clientes não fiquem expostos precisamos realizar alguns ajustes simples mas que farão toda a diferença, na tentativa de evitar que usuários maliciosos tentem acessar nossa aplicação.

*<<Cenário para analisar o problema e a solução>>*

Disponibilizando a API da Macaxeira Online na WEB

*<<descrever em detalhes o estudo de caso que será utilizado para analisar a “má prática” e a “boa prática”>>*

Os donos do Macaxeira Online resolveram expor sua API na WEB para alguns de seus parceiros comerciais. Eles reclamavam que desejavam acessar a API da loja virtual a fim de facilitar a integração nas suas plataformas.

O arquiteto do sistema avaliou a demanda e resolveu publicar a API de imediato para os parceiros comerciais, uma vez que a mesma já existia a já vinha sendo utilizada há algum tempo em integrações com outros sistemas internos da companhia.

A equipe de segurança, todavia, ainda precisava avaliar a aplicação, para garantir que a mesma estaria protegida de possíveis ataques. Ao final da análise um relatório deveria ser gerado informando os pontos de não conformidade e sugestões de melhorias.

*<<Estudo de caso considerando a má prática>>*

Relatório de Segurança

*<<descrever em detalhes o desenvolvimento do estudo de caso considerando a forma inadequada>>*

*<<deixar claro os problemas trazidos com o uso da má prática>>*

O time de segurança realizou uma análise detalhada do sistema e encontrou alguns pontos que precisavam ser corrigidos antes de disponibilizar para os usuários externos. O time de desenvolvimento foi chamado para entender melhor a questão.

O arquiteto não entendia como era possível a aplicação ainda apresentar brechas de segurança uma vez que eles estavam utilizando o *spring-security*. A API já vinha sendo consumida por outros sistemas internos havia pelo menos 1 ano.

No relatório foram reportados os seguintes tópicos:

* problemas na autenticação;
* problemas de autorização;
* problemas de configuração

**Problemas na Autenticação**

O primeiro ponto levantado foi referente a forma como a autenticação estava sendo realizada. Eles encontraram um arquivo de texto dentro da aplicação contendo os dados do usuário e senha, conforme a **Listagem 1**. Foi informado para o time de desenvolvimento que qualquer usuário mal intencionado que tivesse acesso ao arquivo obteria as credencias necessárias para se logar na aplicação.

**Listagem 1**. Arquivo de configuração da aplicação: application.properties

...

api.user=macaxeira

api.senha=1234

...

O time de desenvolvimento também utilizava a estratégia de autenticação *BASIC*, dado o fato de que quem acessaria seriam outros sistemas, não havendo portanto um formulário de login. Isso seria um problema quando a API fosse publicada na WEB. Na autenticação *BASIC*, a cada *request* do usuário, em seu *header* vão as credencias de acesso, enviadas então para o servidor. A falha nesse caso é que ela sofra ataques, um *hacker* com o auxílio de alguma ferramenta de *sniffer* pode interceptar as requisições do usuário, caso os dados trafeguem em um canal sem criptografia, e assim capturar seu login e senha.

**Listagem 2**. Configuração da autenticação *BASIC*: SecurityConfig.java

@Configuration

@EnableWebSecurity

**public class** SecurityConfig **extends** WebSecurityConfigurerAdapter {

@Value("api.user")

**private** String user;

@Value("api.password")

**private** String password;

@Autowired

**public void** configureGlobal(AuthenticationManagerBuilder auth) **throws** Exception {

auth.inMemoryAuthentication().withUser(user).password(password);

}

@Override

**protected void** configure(HttpSecurity httpSec) **throws** Exception {

httpSec.authorizeRequests()

.anyRequest().authenticated()

.and()

.httpBasic();

}

}

**Problemas de Autorização (Controle de Acesso)**

Foi questionado quantos parceiros comerciais teriam acesso a API e o nível de iteração. No formato em que a aplicação se encontrava apenas um único usuário era utilizado para se autenticar, vide a **Listagem 2**, no método configureGlobal. Como apenas os sistemas internos utilizavam a API esse problema não era crítico, contudo agora, com mais usuários, era preciso saber se todos teriam os mesmo privilégios. No formato atual isso não seria possível, eles teriam que fazer a aplicação suportar *n* usuários. Além disso era preciso definir os privilégios de cada um e mapear no sistema os pontos em que teriam acesso restrito.

Devido a falta de uma credencial para cada usuário da aplicação, também ficava impossível rastrear quais usuários haviam executado determinada operação. A nível de segurança isso não constituía uma falha em si, porém ajudaria e muito ter esse nível de detalhamento, tanto a fim de oferecer um melhor suporte para os usuários, quanto para ajudar a investigar algum acesso indevido.

**Problemas de configuração**

Foi alertado que embora o *spring-security* fosse um excelente framework algumas configurações estavam pendentes. Quando a configuração é realizada via Java, ou seja, sem ser realizada em arquivos XML, a maioria das configurações já vem habilitadas por padrão. Contudo, ainda sim seria necessário ativar algumas mais especificas, bem como ataques *Cross-site Scripting (XSS)*.

***Nota:*** *Quando configuramos o spring-security via Java, algumas features já vem habilitadas por padrão: Cache-Control, X-Content-Type-Options, X-Frame-Options.*

**Mãos a Obra**

Junto com os pontos a melhorar, o time de segurança definiu também um plano de ação que foi apresentado ao time de desenvolvimento. O plano englobava basicamente 3 tópicos: **autenticação**, **autorização** e **configurações gerais**. Na visão dos especialistas, esses pontos precisavam ser contornados o quanto antes, para que o sistema fosse disponibilizado para o acesso de terceiros.

Quanto a autenticação, era preciso rever a utilização do *BASIC*, devido a sua fragilidade em expor as credencias de acesso, conforme mencionamos acima. Além disso seria preciso suportar um cadastro para cada usuário do sistema, a utilização de arquivos para manter os dados de acesso teria que ser revista.

O problema referente a autorização estava interligado ao fato de haver apenas uma única credencia para toda a aplicação, impossibilitando a criação de perfis de acesso. Uma vez que esse problema fosse resolvido na autenticação, o time deveria efetuar os ajustes necessários no sistema e levantar junto com a área de negócios os papeis dos usuários e quais operações os mesmos estariam aptos a realizar.

Devido a natureza do sistema ser uma API, indicava que a aplicação não seria acessada por um usuário final especificamente, e sim por outros sistemas que consumiriam os serviços expostos. Dessa forma algumas configurações *default* do *spring-security* deveriam ser desabilitados, ao passo que outras deveriam ser habilitadas.

A seguir podemos conferir de forma mais detalhada cada um dos pontos citados e a melhor forma encontrada para tornar o serviço mais seguro.

**Autenticação**

Segundo o estudo realizado, a forma como os usuários se autenticavam para acessar os serviços estava longe do ideal, dado o novo cenário, onde seria preciso expor a aplicação para terceiros. Conforme vimos acima a autenticação *BASIC* apresentava falhas comprometedoras. Para contornar o problema de trafegar os dados da credencial expostos em cada *request*, a solução indicada foi a utilização do protocolo *HTTPS*.

Para garantir que todos os dados fossem trafegados de forma encriptada, seria necessário habilitar o servidor de aplicação para suportar o protocolo, além de configurar no código para que toda requisição realizada utilizasse o canal seguro.

Na classe *SecurityConfig*, na **Listagem 3**, podemos ver o método *configure*, onde definimos que será utilizada a autenticação *BASIC*, contudo, mais a diante também informamos que toda requisição precisa de um canal seguro. Com isso além dos dados da credencial que vão no *header* em cada *request*, toda a mensagem passará a trafegar encriptada. Garantindo que dados sensíveis também estarão protegidos.

**Listagem 3**. Autenticação no banco de dados: SecurityConfig.java

@Configuration

@EnableWebSecurity

**public** **class** SecurityConfig **extends** WebSecurityConfigurerAdapter {

@Autowired

**private** PasswordEncoder encoder;

@Autowired

**private** DataSource dataSource;

@Autowired

**public** **void** configureGlobal(AuthenticationManagerBuilder auth) **throws** Exception {

auth.jdbcAuthentication()

.dataSource(dataSource)

.usersByUsernameQuery("select login, password, enabled from users where login = ?")

.authoritiesByUsernameQuery("select u.login, r.role from roles r, users u where

u.user\_id = r.user\_id and u.login = ?")

.passwordEncoder(encoder);

}

@Override

**protected** **void** configure(HttpSecurity httpSec) **throws** Exception {

httpSec

.csrf().disable()

.authorizeRequests()

.antMatchers("/healthcheck").hasRole(RoleEnum.USER.name()) .antMatchers("/api/users").hasRole(RoleEnum.ADMIN.name())

.antMatchers("/api/\*\*").hasAnyRole( RoleEnum.USER.name(), RoleEnum.SELLER.name(),

RoleEnum.COMMERCIAL\_PARTNER.name(),

RoleEnum.ADMIN.name())

.and()

.httpBasic()

.and()

.requiresChannel()

.anyRequest().requiresSecure();

}

@Bean

**public** PasswordEncoder passwordEncoder() {

if (encoder == null) {

encoder = new BCryptPasswordEncoder(12);

}

**return** encoder;

}

}

A seguir podemos conferir como habilitar o **SSL** *(secured socket layer)* no *tomcat*. O *spring-security* já traz alguns certificados gerados no diretório *sample/certificates*. A seguir iremos utilizar o arquivo **server.jks**, para isso precisamos copia-lo para o diretório *conf* do *tomcat* e reiniciar o servidor. Para entender melhor podemos conferir na pagina do *tomcat: <<URL>>*

**Listagem 4**. Arquivo: server.xml

...

<Connector port="8443" protocol="HTTP/1.1" SSLEnabled="true" scheme="https" secure="true"

clientAuth="want" sslProtocol="TLS"

keystoreFile="${catalina.home}/conf/server.jks"

keystoreType="JKS" keystorePass="password"

truststoreFile="${catalina.home}/conf/server.jks"

truststoreType="JKS" truststorePass="password"

/>

...

Nesse cenário temos dois problemas para resolver: um único usuário para toda a aplicação e *password* sem criptografia. A melhor estratégia nessa situação seria registrarmos os usuários em um banco de dados. Um modelo para suportar nossas esse cadastro seria nos termos no mínimo duas tabelas *User* e *Role*. Podemos ver as seguintes entidades nas listagens: **Listagem 5** e **Listagem 6**. Com isso também preparamos o ambiente para que possamos suportar o controle de acesso.

**Listagem 5**. Classe User.java

@Entity

@Table(name="users")

**public class** User **implements** Serializable {

...

@Id

@GeneratedValue(strategy=GenerationType.AUTO)

@Column(name="user\_id")

**private** Long id;

@Column(unique=true, nullable=false, name="user\_name")

**private** String name;

@Column(unique=true, nullable=false)

**private** String login;

private String password;

@Column(unique=true, nullable=false)

**private** String email;

**private** boolean enabled = true;

@OneToMany(mappedBy = "user")

**private** Set<Role> roles;

*//getters and setters*

}

**Listagem 6**. Classe Role.java

@Entity

@Table(name="roles")

**public class** Role **implements** Serializable {

...

@Id

@GeneratedValue(strategy=GenerationType.AUTO)

**private** Long id;

@JsonIgnore

@ManyToOne(fetch = FetchType.LAZY)

@JoinColumn(name = "user\_id", nullable = false)

**private** User user;

@Column(nullable=false)

**private** String role;

*// getters and setters*

}

Para tornarmos a senha mais segura, o ideal é adotarmos um dos dois algoritmos de criptografia de *password* já disponível no spring-security: **BCryptPasswordEncoder** ou **StandardPasswordEncoder**. Ambos utilizam um *salt*, para ajudar a tornar o processo de criptografia ainda mais seguro. Um salt nada mais é do que uma chave *hash*, que é concatenada a senha original do usuário, que depois é encriptada. O algoritmo sugerido aqui é o **BCryptPasswordEncoder**, que gera uma chave *salt* dinamicamente. Ele é um algoritmo mais lento, de forma proposital, com o intuito de ajudar na proteção contra ataques de força-bruta.

**Autorização**

Uma das restrições impostas ao liberar a API para terceiros, foi a necessidade de se definir perfis de acesso para os usuários. A equipe de segurança informou ao time de desenvolvimento que eles precisariam realizar um levantamento junto com a área de negócio a fim de mapear as *roles* e as operações que cada uma poderia executar.

Os desenvolvedores levantaram as seguintes *roles*: *USER, SELLER, CUSTOMER SERVICE e ADMIN*. A *role*  *ADMIN*, como o próprio nome já sugere deveria possuir acesso irrestrito ao sistema. A *role SELLER*, deverá ter acesso completo para poder cadastrar produtos e interagir com pedidos referente aos produtos vendidos por eles, contudo deveria ter uma visão limitada do cadastro dos clientes. A *role CUSTOMER SERVICE*, não deveria ter restrição alguma ao cadastro dos clientes, além de ter permissão para cancelar pedidos, e trocas e estorno de produtos. Por fim, a *role USER* deverá apenas servir para consultas básicas, sem maiores privilégios.

Por sorte, ao cadastrarmos os usuários no banco de dados grande parte do problema já encontrava-se resolvida, restava agora definir no código o controle de acesso para as operações. Para ajudar nesse processo de restringir as operações permitidas, o *spring-security* disponibiliza uma anotação específica: *@Secured.* Onde definimos quais *roles* estão aptas a executar aquele trecho de código.

Podemos ver na **Listagem 7**, na classe *CustomerController*, um método *delete*, onde é possível excluir um cadastro de cliente, porém com execução restrito para usuários com o perfil *ADMIN* ou *CUSTOMER SERVICE*, caso contrário o usuário que não atenda a essas condições que tentar executar irá receber o erro HTTP 403, que significa *forbidden*, ou proibido.

A anotação *@Secured*, pode ser empregada tanto a nível de classe, onde todos os métodos públicos responderão sob a mesma *constraint* de acesso, quanto a nível de método, uma opção mais granular, portanto, permitindo um controle mais fino.

**Listagem 7**. Classe CustomerController.java

**package** oms.controller.api;

...

@RestController

@RequestMapping("/api/customers")

**public** **class** CustomerController {

...

@Autowired

**private** CustomerService customerService;

@Autowired

**private** CustomerResourceAssembler customerResourceAssembler;

@RequestMapping(value="/{id}", method= RequestMethod.GET)

@ResponseBody

**public** CustomerResource customer(@PathVariable("id") Long customerId) {

Customer customer = customerService.get(customerId);

return customerResourceAssembler.toResource(customer);

}

@Secured(value={"ROLE\_ADMIN", "ROLE\_CUSTOMER\_SERVICE"})

@RequestMapping(value="/{id}", method=RequestMethod.DELETE)

**public** **void** delete(@PathVariable("customer\_id") Long customerId) {

System.out.println("Deletando o customer "+customerId);

}

...

}

**Configuração e Prevenção**

<<Falar aqui sobre as outras configurações e o motivo que nos levou a escolhe-las>>

**Conclusão**

*<<comparar: boa prática VS má prática>>*

**Links**

* descrever em detalhes o desenvolvimento do estudo de caso considerando a forma adequada
* deixar claro as vantagens trazidas com o uso da boa prática

**Entendendo o porquê desta ser a melhor solução**

* comparar: boa prática VS má prática