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

A loja virtual Macaxeira online resolveu ampliar os seus negócios. A área de negócios resolveu adotar uma estratégia agressiva a fim de aumentar o catálogo de produtos expostos a venda para seus clientes. A estratégia consiste em permitir que outros vendedores pudessem cadastrar seus itens na loja da Macaxeira online, bem como trocar informações sobre a venda e o *tracking* dos pedidos dos cliente.

Para garantir que essa estratégia seja vencedora, será preciso que os parceiros comerciais tenham acesso a API de serviços da loja, com o intuito de facilitar a integração.

Introdução

A área de negócios da loja de comercio eletrônico Macaxeira Online deseja ampliar os negócios, para isso eles resolveram permitir que terceiros pudessem cadastrar seus produtos no catálogo da loja, afim de aumentar o número de itens disponíveis a venda e por uma participação nos itens vendidos.

Para que a estratégia se torne um sucesso será preciso que os parceiros comerciais tenham acesso a API dos serviços da loja.

Atualmente, a API vem sendo utilizada apenas pelos sistemas internos da companhia. Para liberar o acesso para terceiros precisamos garantir que a aplicação seja segura, para evitar que usuários não autorizados tenham acesso.

O estágio em que a aplicação se encontra atualmente, funciona muito bem para atender os sistemas internos, contudo, para que terceiros tenham acesso, será necessário realizar alguns ajustes. No momento todos os sistemas que consomem a API utilizam as mesmas credenciais, não existindo portanto nenhuma restrição sobre quais operações esse usuário pode vir a executar. Além do mais, todas as informações trocadas com a aplicação trafegam em um canal não seguro.

Para garantir que ao expormos a API para os parceiros comerciais não tenhamos nenhuma problema, precisamos corrigir as falhas já conhecidas. Precisamos então garantir que a aplicação permita que cada usuário tenha sua própria credencial de acesso. A área de negócios precisa ajudar a definir quais operações serão permitidas para cada usuário externo, com isso poderemos mapear as operações que deverão possuir restrição de acesso, além de informar quais *URLs* eles poderão efetuar requisições.

Como a aplicação já utiliza o *spring-security*, nosso caminho será bastante facilitado. Precisamos apenas melhorar o processo de autenticação, bem como definir de forma adequada a autorização e outros ajustes pontuais como veremos a seguir.

Expandindo os negócios

A loja de comercio eletrônico macaxeira online resolveu expandir seus negócios. Para isso os analistas de negócio da empresa resolveram abrir para que outros vendedores pudessem publicar seus produtos no catálogo da companhia.

Para que a estratégia obtivesse sucesso seria preciso melhorar a integração com os sistemas dos novos parceiros comerciais. Um equipe de desenvolvimento foi então chamada para realizar um estudo sobre a viabilidade técnica.

Os analistas propuseram que fosse utilizada a API que já existia atualmente e vinha sendo utilizada por outros sistemas internos. Precisávamos portanto, apenas disponibiliza-la para que os vendedores tivessem acesso e pudessem integrar com seus sistemas.

Uma vez que essa aplicação seria exposta na internet para ser utilizada por usuários externos seria preciso que a equipe de segurança também desse o seu parecer. O time de segurança então entrou em ação e realizou um levantamento sobre as brechas encontradas no sistema, bem como um plano de ação para mitiga-las.

Raio-X na API – Expondo as Falhas

Embora a API da aplicação já vim sendo utilizada há alguns meses por outros sistemas internos, e de utilizar o spring-security para restringir o acesso de usuários não autenticados, o projeto para expor os serviços da loja virtual para terceiros ainda contava com alguns empecilhos. Antes de ser publicada na internet o time de segurança precisava dar o aval final, para garantir que o sistema estaria blindado contra possíveis ataques, bem como garantir o controle de acesso dos novos usuários.

O time de segurança passou um pente fino na aplicação, procurando possíveis brechas de segurança que pudessem vir a deixar a loja da Macaxeira Online suscetível a ataques quando a mesma fosse publicada na internet para seus parceiros comerciais.

Uma série de pontos foram levantados e colocados no relatório. Os principais pontos a serem atacados seriam referentes a **autenticação** e **autorização**. O time de desenvolvimento foi chamado para entender melhor os pontos abordados e o plano de ação que deveria ser executado antes do sistema ser liberado.

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.*

**Fechando a porta – Blindando a API**

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 6**, 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.

A seguir podemos conferir na **Listagem 3**, 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 página do projeto do *tomcat:* [*http://tomcat.apache.org/tomcat-6.0-doc/ssl-howto.html*](http://tomcat.apache.org/tomcat-6.0-doc/ssl-howto.html)

**Listagem 3**. 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"

/>

...

O próximo passo a ser atacado é referente ao suporte do cadastro individual para cada usuário da aplicação. Permitir que os dados da credencia fiquem expostos em um arquivo texto já vimos que não é uma boa prática.

Foi proposto para os desenvolvedores que os usuários fossem cadastrados em um banco de dados. A seguir foi apresentado uma modelagem a fim de cobrir tanto os usuários quanto os perfis de acesso, conforme veremos a seguir na seção de **autorização**.

A classe *User*, como podemos ver na **Listagem 4**, define os campos de nome, login, email, senha e roles, onde temos um relacionamento com outra entidade, *Role*, representada na **Listagem 5**, responsável por mapear os perfis. Na classe *User* podemos encontrar ainda o atributo *enabled*, onde é possível habilitar e desabilitar o cadastro de um usuário.

**Listagem 4**. 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 5**. 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*

}

Nosso problema de autenticação esta quase resolvido, resta ainda mais um ponto em aberto: *senha exposta*. As credencias de acesso estão salvas agora em um banco de dados, o acesso as informações dos usuários estão restritas aos *DBAs* e a usuários com acesso ao banco, contudo não podemos deixar as senhas expostas, para o caso de um funcionário descontente com a Macaxeira Online tivesse conhecimento de dados sigilosos.

Por fim, mais uma vez podemos encontrar a resposta no *spring-security*. Os analistas de segurança sugeriram que antes de salvarmos os dados do usuário no banco de dados a senha fosse encriptada. Dentre os algoritmos de criptografia disponíveis foi indicado o uso do ***Bcrypt*.**

O *spring-security* traz consigo uma implementação do *Bcrypt*, o ***BCryptPasswordEncoder***, uma implementação do ***PasswordEncoder***. A vantagem do *BCryptPasswordEncoder* é que além de ser um algoritmo de criptografia baseado em *salt*, ele o gera de forma dinâmica, tornando ainda mais difícil quebrar sigilo. Outra vantagem dessa implementação se deve ao fato de ser um algoritmo mais lento, característica essa realizada de forma proposital, com isso ele ajuda na proteção quanto a ataques de força bruta que tentem decifrar a senha dos usuários.

Abaixo, podemos conferir na **Listagem 6**, a classe *UserServiceImpl*, onde no método *createUserDefault*, vemos a implementação da lógica para tornar a senha do usuário mais segura. Na **Listagem 7**, nos definimos um *bean*, no método *passwordEncoder*, referente ao algoritmo de criptografia que utilizaremos ao longo da aplicação.

**Listagem 6**. UserServiceImpl.java

**package** oms.service.impl;

...

@Service

**public** **class** UserServiceImpl **implements** UserService {

@Autowired

**private** PasswordEncoder passwordEncoder;

@Autowired

**private** UserRepository userRepository;

...

@Override

**public** User createUser(User user) {

createUserDefault(user);

user = userRepository.save(user);

**return** user;

}

...

**private** **void** createUserDefault(User user) {

...

user.setPassword(passwordEncoder.encode(user.getPassword()));

Set<Role> userRoles = new HashSet<Role>();

userRoles.add(new Role(RoleEnum.USER.toString(), user));

user.setRoles(userRoles);

}

}

Na **Listagem 7**, podemos ver então na classe *SecurityConfig*, como ficou a configuração no método *configureGlobal*. Aqui nos precisamos definir o *provider* que iremos utilizar para realizar a autenticação, nesse caso optamos por um especifico para *JDBC*.

Além do provider, precisamos especificar a *query* que será utilizada para validar o login e senha do usuário, bem como a *query* para realizar o carregamento dos perfis, que serão utilizados na autorização, como veremos a seguir. Ainda é necessário informar o algoritmo de criptografia utilizado, reparem que devemos utilizar o mesmo algoritmo empregado na criptografia das senhas no momento do cadastro dos usuários.

**Listagem 7**. 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.ADMIN.name())

.and()

.httpBasic()

.and()

.requiresChannel()

.anyRequest().requiresSecure();

}

@Bean

**public** PasswordEncoder passwordEncoder() {

if (encoder == null) {

encoder = new BCryptPasswordEncoder(12);

}

**return** encoder;

}

}

***Nota****: Um salt nada mais é do que uma chave. Ela é comumente utilizada por algoritmos de criptografia que se utilizam de função hash, para tornar o processo de encriptação ainda mais seguro. Essa chave é concatenada então ao valor de origem que se deseja encriptar, gerando uma senha hash.*

**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 8**, 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 8**. 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);

}

...

}

**Conclusão**

Embora a primeira solução apresentada no artigo seja considerada como não sendo a solução ideal, ainda assim podemos encontra-la rodando em muitos sistemas em produção. O principal fator se dá por conta da falta de conhecimento do *framework* e a respeito de segurança; segundo, pela facilidade da sua implementação. Desde que a aplicação seja executada em um ambiente restrito, a vulnerabilidade é menor.

No segundo cenário, após a aplicação passar pela auditoria da equipe de segurança, uma série de ajustes foram realizados a fim de blindar ainda mais a API. Foi habilitado o *HTTPS* para garantir que a troca das informações se desse em um canal seguro, melhoramos o mecanismo de autenticação, adotando uma solução mais *enterprise*, permitindo múltiplos usuários utilizando suas próprias credenciais de acesso, encriptamos a senha dos usuários utilizando o algoritmo Bcrypt, além de definirmos quais operações cada usuário estaria apto a executar.

Sem que essas modificações fossem realizadas, nossa aplicação estaria completamente exposta, uma vez que fosse pulicada a API na internet. Nossos usuários teriam todas suas informações expostas a cada requisição, tornando-os presa fácil para qualquer ataque, desde os mais básicos. A Macaxeira Online por sua vez poderia correr o risco de comprometer a sua operação, caso sofresse um ataque direto.

Atualmente o *spring-security* já vem por padrão com a maioria das suas *features* habilitadas, deixando a cargo do desenvolvedor decisões a respeito do *provider* de autenticação que será utilizado, se uma determinada URL está disponível apenas para usuários autenticado, etc.

O *spring-security* desponta como uma excelente ferramenta para tornar aplicações java mais segura, abrangendo todas as funcionalidades do JAAS, além de outras funcionalidades. Sua capacidade de customização e extensibilidade também são pontos fortes do *framework*. Uma vez que sua aplicação já utilize o *spring-framework*, o tempo para implantação é extremamente reduzido, facilitando e muito a sua adoção.

**Links**

Macaxeira Online – Primeira versão (código fonte)

***https://github.com/antonioreuter/hateoas-oms-basic***

Macaxeira Online – Solução Ideal - (código fonte)

***https://github.com/antonioreuter/hateoas-oms-sec***

Spring Security - Referência

[***http://docs.spring.io/spring-security/site/docs/3.2.5.RELEASE/reference/htmlsingle/***](http://docs.spring.io/spring-security/site/docs/3.2.5.RELEASE/reference/htmlsingle/)

Tomcat – Como habilitar SSL

<http://tomcat.apache.org/tomcat-6.0-doc/ssl-howto.html>

Spring Security – Security Headers

http://spring.io/blog/2013/08/23/spring-security-3-2-0-rc1-highlights-security-headers/#hsts