SEGURANÇA

Protegendo Contra Intenções Maliciosas

Validação de Todos os Dados de Entrada

Nunca confie nas boas intenções do usuário – valide tudo que você utiliza. Por exemplo, ao implementar uma busca, estabeleça um limite de linhas que deve ser retornado. Uma simples solução vem abaixo:

```
def search = {
    def max = params.max?.toInteger() ?: 50
    max = Math.min(max, 500)
    def results = Post.findAllByTag("grails", [ max: max ])
    ...
}
```

As linhas acima fazem com que o usuário não consiga recuperar mais de 500 linhas em um único acesso. Sem a limitação acima, um usuário poderá emitir a seguinte requisição:

```
http://localhost:8080/hubbub/post/search?query=grails&max=1000000000
```

que irá solicitar o envio de um bilhão de linhas.

Implemente o Escape de Todas as Saídas

Agora vamos considerar um outro cenário. Inicie sua aplicação e ao cadastrar um usuário digite no campo referente ao nome o seguinte conteúdo:

```
<script type="text/javascript">alert("Te peguei!")</script>
```

O que acontece? Se estivermos exibindo o campo nome do objeto usuário assim:

```
${usuarioInstance.nome}
```

uma janela de alert aparecerá na sua frente. Por outro lado, se você tiver exibido o campo nome utilizando o tag `${fieldValue(bean: usuarioInstance, field: "nome")}`, a janela de alert não será exibida uma vez que este tag implementa o escape para HTML do conteúdo do campo `nome`.

Para efetuar o escape do campo `nome` (que exibe o alert) podemos chamar manualmente o método `encodeAsHTML()` ou poderemos obter este mesmo resultado automaticamente alterando a entrada abaixo no arquivo `grails-app/conf/Config.groovy`:

```
grails.views.default.codec = "html"
```

Com esta configuração o Grails irá efetuar o escape de todas as expressões nos seus arquivos GSP para HTML.

CUIDADO: Embora a configuração para `grails.views.default.codec` funcione na maioria dos casos, ela não efetua o escape de valores de atributos de tags GSP e nem efetua o escape da saída gerada pelos tags. Os próprios tags devem cuidar disso para funcionarem adequadamente.

Para ver na prática o efeito dessa configuração, considere o fragmento de código GSP:

```
<div>${post.content}</div>
<g:textField name="test" value="${post.content}"/>
```

O conteúdo do `post` sofre o escape automaticamente na primeira linha, pois ele é referenciado por uma expressão GSP. Mas o Grails não fará o mesmo para o valor do atributo na segunda linha. Por este motivo, a implementação do tag `textField` implementa o equivalente a:

```
attrs["value"].encodeAsHTML()
```

Estamos quase terminando, mas ainda precisamos mencionar que o escape ou o encoding dos dados digitados pelos usuários não irá funcionar se você não utilizar o `codec` adequado. Por exemplo, se um dado de entrada for um URL que será utilizado na exibição de um hiperlink, utilize o método `encodeAsURL()`. Se ele for utilizado em qualquer outro lugar de uma página HTML, utilize o `codec` HTML.

Demos apenas dois exemplos de ataques por injeção (SQL e HTML), mas existem outros, tais como a injeção de código (digamos que você permita que os usuários informem código Groovy que você irá executar), a injeção de comandos do SO (qualquer coisa passada para Runtime.exec() ou equivalente), e a injeção de caminho de arquivo. Em todos estes casos, esteja certo de estar lidando com dados de entrada confiáveis ou não, e se o dado não for confiável, faça o escape ou a verificação de alguma outra forma.

SSL, Criptografia e Message Digests

Em vez de utilizar os algoritmos `SHA-1` e `MD5`, considere a possibilidade de utilizar `SHA-256` ou `SHA-512`. No momento, estes algoritmos não possuem fraquezas conhecidas e podem ser utilizados da mesma forma como os outros dois. Se você tiver interesse em conhecer as vulnerabilidades dos algoritmos `SHA-1` e `MD5` veja os artigos sobre eles no Wikipedia.

Antes de seguir em frente, gostaríamos de mencionar o “The Legion of the Bouncy Castle” (`http://www.bouncycastle.org/`), que possui um provedor Open Source de

Criptografia Java com alguns algoritmos, como o Blowfish. Eles inclusive possuem classes para trabalhar com OpenPGP.

Imagine agora que você é um tipo inescrupuloso que deseja invadir aplicações em servidores. O que você faria? Uma abordagem importante seria descobrir a plataforma na qual a aplicação está instalada. Aplicações Grails não mostram facilmente a plataforma na qual foram construídas, mas existem pontos fracos que permitem ao usuário descobrir que se trata de uma aplicação Grails.

No capítulo 5, apresentamos a você uma característica do Grails que ajuda a esconder a plataforma utilizada: o mapeamento de códigos de resposta para actions de controladores. O ideal seria mapear para controladores todos o códigos de erro de respostas que o Grails gera para as visões.

```
class UrlMappings {
    ...
    "403"(controller: "errors", action: "forbidden")
    "404"(controller: "errors", action: "notFound")
    "500"(controller: "errors", action: "internalError")
}
```

Você verá no próximo capítulo que há muitos outros códigos de status HTTP que você pode mapear, mas só vale a pena mapear os códigos que a nossa aplicação utiliza. Mas isso não resolve todos os problemas. Certas partes da infra-estrutura GSP mandam exibir (de forma hard-coded) a página `grails-app/views/error.gsp` sempre que uma exceção é gerada por uma página GSP. Logo, este mecanismo informa a todos que nossa aplicação foi feita em Grails. Para resolver este problema, recomendamos que você modifique a página GSP para enviar um erro 500 se o ambiente estiver designado para produção.

```
<%@ page import="grails.util.Environment" %>
<g:if test="${Environment.current == Environment.PRODUCTION}">
    ${response.sendError(500)}
</g:if>
<g:else>
    ...
</g:else>
```

Esta modificação significa que sua aplicação em produção não irá divulgar informações sobre a plataforma Grails, mas você ainda obtém os benefícios da mensagem de erro e da pilha de execução quando a aplicação estiver sendo executada no ambiente de desenvolvimento e de testes.

Todas as informações que discutimos aqui, assim como algumas outras, podem ser encontradas em "CWE/SANS Top 25 Most Dangerous Programming Errors" (http://cwe.mitre.org/top25/) e o "OWASP Top 10" (http://www.owasp.org/index.php/Top_10_2007). Estes sites contêm importantes informações sobre segurança e devem ser lidos. Lá você também encontrará ferramentas tal como Scarab, uma ferramenta que testa a sua aplicação para as vulnerabilidades mais comuns.

Controle de Acesso

O Controle de acesso possui dois aspectos:

- A pessoa que está se comunicando com você é quem ela diz que é? (Autenticação)
- Ela possui as permissões necessárias para executar uma determinada ação? (Autorização)

Um controle de acesso simples pode ser implementado através de um filtro do Grails e do uso do objeto sessão, mas é bem melhor utilizar um dos plugins de segurança disponíveis:

Authentication plugin: O Authentication plugin possui uma implementação simples, e não tem nenhuma dependência em relação a bibliotecas de terceiros. É adequado a sites que requerem o registro dos usuários, mas não possuem requisitos de autorização complexos.

JSecurity plugin: O JSecurity plugin provê um controle de acesso completo e criptografado utilizando a biblioteca JSecurity do grupo Apache. Embora o JSecurity não seja ainda muito conhecido, há vários anos tem sido utilizado por vários sistemas em produção.

Spring Security (Acegi) plugin: Conhecido anteriormente pelo nome de Acegi Security, o Spring Security é um plugin bem conhecido e um framework mundialmente utilizado por aplicações Java para a web. Em relação aos demais plugins, ele provê a mais ampla suíte de opções de autenticação.

Vamos nos concentrar no plugin Spring Security.

Introdução ao Plugin Spring Security

O primeiro passo é a instalação do Plugin:

```
grails install-plugin acegi
```

Se você estiver criando uma aplicação nova utilizando o Spring Security, o processo é bem simples:

```
grails create-auth-domains modelo.User modelo.Role
➔ modelo.Requestmap
```

Isto irá criar 3 classes do domínio: `User`, `Role`, e `Requestmap`.

Mas se já tivermos uma classe Usuário e quisermos utilizá-la, teremos que executar o seguinte comando:

```
grails create-auth-domains modelo.DummyUser modelo.Role Requestmap
```

Desta forma a nossa classe `Usuario` não será sobrescrita, e o plugin será configurado para obter informações do usuário da classe `DummyUser`.

O plugin pode utilizar qualquer classe do domínio para armazenar os usuários, contanto que a ela possua os campos nome do usuário, senha, se a conta está habilitada, e uma coleção de autorizações ou perfis.

Exemplo:

```
class User {
    String userId
    String password
    boolean enabled
    static hasMany = [ authorities: modelo.Role ]
    static belongsTo = modelo.Role
    ...
}
```

Também precisamos corrigir a classe `Role`, pois ela possui uma associação com a classe `DummyUser`:

```
class Role {
    static hasMany = [people: User]
    ...
}
```

Como possuímos nossa própria classe `Usuario`, não precisamos mais das classes `DummyUser`, `DummyUserController` e das páginas gsp correspondentes (`grails-app/views/dummyUser`).

Agora temos que acrescentar código no BootStrap para que alguns usuários pré-configurados sejam designados a perfis. No código do BootStrap utilizaremos o objeto `authenticateService` do plugin para codificar senhas antes de armazená-las no banco de dados:

```
authenticateService.encodePassword("password")
```

Quando executamos o comando `create-authdomains` ele adiciona um arquivo extra ao diretório `grails-app/conf`: `SecurityConfig.groovy`. Este arquivo permite que você controle muitos aspectos do plugin, e possui uma página de referência:

http://www.grails.org/AcegiSecurity+Plugin+-+Customizing+with+SecurityConfig.

Exemplo:

```
security {
    active = true
    loginUserDomainClass = "DummyUser"
    authorityDomainClass = "modelo.Role"
    requestMapClass = "Requestmap"
}
```

Agora precisamos trocar o nome da classe "DummyUser" para "modelo.Usuario".

Aviso: Tenha cuidado com a opção de configuração active: se designarmos o valor false para este atributo, estaremos desabilitando o Spring Security. O mesmo irá acontecer se esta opção for removida ou se o arquivo for apagado.

Por default, o plugin espera que a classe Usuario possua campos com os nomes username e password. Mas como os campos da classe Usuario são userId e password, para corrigir este problema temos que adicionar as seguintes configurações ao arquivo SecurityConfig.groovy:

```
userName = "userId"
password = "password"
```

Agora precisamos configurar os URLs que desejamos proteger.

Protegendo URLs

Vamos começar com um modelo de segurança simples: para se acessar qualquer página da nossa aplicação será necessária uma autenticação do usuário, com exceção para a home page que todos podem ver. O plugin nos dá três opções para a especificação desta informação:

Configuração estática — As regras são declaradas no arquivo SecurityConfig.groovy.
Dinâmica — As regras são armazenadas no banco de dados como Requestmaps.
Anotações — As regras são declaradas utilizando anotações em classes controladoras e de serviço.

Agora sabemos para que serve a classe do domínio Requestmap. Ela mapeia URLs a permissões, ou autorizações (usando a terminologia Spring Security). A utilização desta classe do domínio é útil se você quer mudar as configurações de segurança em tempo de execução.

As anotações permitem que os controladores sejam protegidos independentemente dos URLs utilizados para acessá-los. Elas também são convenientes quando desejamos proteger métodos de serviço.

Na nossa aplicação iremos utilizar a abordagem estática, em parte porque ela mantém juntas todas as regras, e em parte porque precisamos falar sobre a ordenação das regras. Como não utilizaremos regras dinâmicas, podemos remover a classe do domínio Requestmap, sua classe controladora e todas as páginas .gsp correspondentes.

Agora veremos a configuração que precisaremos utilizar para implementar o controle de acesso desejado. Segue abaixo o conteúdo do arquivo SecurityConfig.groovy que

contém as regras de acesso aos URLs. Estas regras permitem que todos os usuários tenham acesso à home page e às páginas de login, mas apenas os usuários autenticados podem ver o restante da aplicação. Também configuramos a aplicação de tal forma que todos os arquivos JavaScript, CSS, e de imagens fiquem desprotegidos.

```
security {
    active = true
    cacheUsers = false      ← Desabilita o cache do Hibernate
    loginUserDomainClass = "com.grailsinaction.User"
    authorityDomainClass = "com.grailsinaction.Role"
    userName = "userId"
    password = "password"
    useRequestMapDomainClass = false  ← 1. Desabilita regras
                                         dinâmicas
    requestMapString = """\                          |
    CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON       |
    PATTERN_TYPE_APACHE_ANT                          | ← 2. Configura
    /=IS_AUTHENTICATED_ANONYMOUSLY                   |    regras para
    /login/auth=IS_AUTHENTICATED_ANONYMOUSLY         |    URLs.
    /js/**=IS_AUTHENTICATED_ANONYMOUSLY              |
    /css/**=IS_AUTHENTICATED_ANONYMOUSLY             |
    /images/**=IS_AUTHENTICATED_ANONYMOUSLY          |
    /plugins/**=IS_AUTHENTICATED_ANONYMOUSLY         |
    /**=IS_AUTHENTICATED_FULLY                       |
    """
}
```

A primeira metade do arquivo contém informações que já conhecemos. É a segunda parte que nos interessa agora. Para utilizarmos a configuração estática, primeiramente devemos desabilitar a configuração dinâmica (1). Em seguida temos que especificar as regras de acesso aos URLs, que devem ser designadas a um string de várias linhas denominado `requestMapString` (2). Cada configuração deve aparecer em uma linha diferente. As duas primeiras linhas são configurações gerais e são geralmente incluídas em todas as configurações do Spring Security. A primeira regra força o Spring Security a transformar os URLs das requisições para letras minúsculas antes de compará-los às regras. Um URL como `http://localhost:8080/hubbub/starPost/showAll` irá se tornar `http://localhost:8080 /hubbub/starpost/showall` antes da comparação. Isto significa que as regras de URLs em `requestMapString` devem ser especificadas em letras minúsculas.

A segunda configuração geral especifica que o Spring Security irá utilizar o padrão de comportamento Ant para casar os caminhos ao comparar URLs de requisição a regras. Isto significa que para criar uma regra que case com todos os arquivos .js, você pode utilizar /**/*.js, que especifica qualquer arquivo com a extensão .js em qualquer diretório.

Após as configurações gerais, devemos declarar as regras que desejamos em ordem. Como desejamos que todos possam acessar a home page, a primeira regra designa `IS_AUTHENTICATED_ANONYMOUSLY` para o URL raiz `(/)`. Também precisamos permitir

que todos acessem a página de login (`/login/auth`) fornecida pelo plugin. Os recursos estáticos como JavaScript, CSS, e arquivos de imagens (fornecidos pela aplicação e pelos plugins), também devem ser configurados desta forma. E finalmente, restringimos o acesso a todos os demais URLs para os usuários autenticados.

Você tem que ser cuidadoso com a ordem das regras, pois o Spring Security utiliza a primeira que casa com o URL requisitado. Por exemplo, se você colocar a última regra em primeiro lugar `(/**)`, para acessar qualquer página da aplicação seria preciso uma autenticação independentemente da regra que vem em seguida. Quanto mais específico o padrão do URL em uma regra, mas cedo a regra deve aparecer na lista.

Ainda precisamos integrar o login com o resto da aplicação, que espera encontrar uma instância do objeto `Usuario` na sessão. Antes de fazermos isso, vamos dar uma rápida olhada na opção "Remember me" existente na página de login do plugin. Como nossa aplicação não necessita de um alto grau de segurança é interessante permitir que o usuário possa acessar a aplicação sem precisar se logar sempre. Mas para que a função "Remember me" funcione, é preciso alterar a regra `/**=IS_AUTHENTICATED_FULLY` para `/**=IS_AUTHENTICATED_REMEMBERED`.

Obtendo o Usuário Corrente

Por enquanto nosso sistema ainda não está funcionando uma vez que a página que exibe os posts do usuário logado ainda não consegue acessar o usuário.

Temos duas opções para resolver este problema. A primeira seria armazenar o usuário corrente na sessão quando ele se loga, uma vez que podemos modificar o controlador de login que o plugin criou (`grails-app/controllers/LoginController.groovy`). Mas isto seria um desperdício, pois podemos obter o usuário corrente diretamente do Spring Security. Porque armazenar o objeto usuário na sessão se isto não é necessário? Logo, iremos utilizar a segunda abordagem: mudar o código da aplicação para obter o objeto usuário do Spring Security. O usuário corrente pode ser recuperado através do contexto corrente do Spring Security, que pode ser obtido a partir da classe `SecurityContextHolder`. Este contexto contém todas as informações de segurança relacionadas ao usuário corrente, logo ele pode nos fornecer praticamente tudo o que precisamos. Como esse processo não é agradável, felizmente o plugin nos provê um serviço de autenticação que contém diversos métodos muito úteis. A maioria deles consta da tabela abaixo:

Method	Description
`isLoggedIn()`	Returns `true` if the current user is logged in, otherwise `false`
`ifAllGranted(roles)`	Returns `true` if the current user has all the given roles, otherwise `false`; `roles` is a comma-separated list of role names as a string
`ifNotGranted(roles)`	Similar to `ifAllGranted`, but only returns `true` if the user has none of the given roles
`ifAnyGranted(roles)`	Similar to `ifAllGranted`, but returns `true` if the user has at least one of the given roles
`userDomain()`	Returns the domain instance for the current user
`encodePassword(pwd)`	Encodes the given password with the configured hash algorithm and returns the hash; this hash can be stored in the `password` field of the user domain class
`isAjax(request)`	Returns `true` if the given `HttpServletRequest` appears to be an AJAX one

Como você deve supor, utilizaremos o método `userDomain()` para recuperar a instância do usuário corrente. Este método retorna um objeto destacado, isto é, um objeto que não está sendo monitorado pela sessão do Hibernate. Isto significa que não podemos acessar nenhuma de suas coleções ou salvar quaisquer mudanças até que ele seja religado à sessão. Neste caso, como desejamos ter certeza de que temos acesso às informações mais atuais do usuário, iremos efetuar um novo acesso ao banco de dados para recuperar suas informações.

Aqui estão as modificações que precisam ser feitas no controlador de posts:

```
class PostController {
    def authenticateService
    ...
    def list = {
        def user = authenticateService.userDomain()
        user = User.get(user.id)
        ...
    }
    ...
    def add = {
        def content = params.postContent
        if (content) {
            def user = authenticateService.userDomain()
            user = User.get(user.id)
            if (user) {
                ...
            }
        }
        ...
    }
    ...
}
```

O serviço de autenticação do plugin será injetado na propriedade `authenticateService` do controlador, e poderemos utilizá-lo para recuperar o usuário corrente. Isto corrige o controlador, mas se tentarmos acessar a página de posts do usuário ainda veremos um erro. O template que define a barra lateral da página, ainda recupera o usuário diretamente do objeto sessão HTTP. Talvez sua primeira idéia seja recuperar o objeto usuário através do serviço de autenticação (uma solução nada boa) ou adicionar o usuário ao modelo retornado pela action (uma idéia melhor). No entanto, como o template da barra lateral é referenciado por um layout, todas as páginas da visão que utilizam este layout irão necessitar que o modelo possua a instância do usuário.

Felizmente você não precisa se preocupar com este problema, pois o plugin provê uma biblioteca de tags que faz o que precisamos. Em vez de código como este

```
<dd>${session.user.userId}</dd>
```

a página `grails-app/views/post/_sidebar.gsp` pode utilizar um tag especial que nos permite exibir qualquer campo do objeto usuário:

```
<dl>
<dt>User</dt>
<dd><g:loggedInUserInfo field="userId">Guest</g:loggedInUserInfo>
</dl>
```

Podemos inclusive utilizar este tag como um método, como aparece no fragmento de código abaixo:

```
<g:createLink action="following" controller="friend"
id="${loggedInUserInfo(field: 'userId')}"/>
```

Uma vez implementadas estas modificações, a aplicação voltará a funcionar.

E para permitir que um usuário efetue o logout a qualquer momento, basta incluir o código abaixo no layout:

```
<g:isLoggedIn>
    <g:link controller="logout" action="index">Logout</g:link>
</g:isLoggedIn>
```

O tag `isLoggedIn` faz com que o seu conteúdo seja exibido se o usuário tiver efetuado o login com o Spring Security. Neste caso, o conteúdo é um link para o controlador de logout criado pelo plugin.

Agora o controle de acesso está funcionando e apenas usuários autenticados ou (lembrados – remembered) podem acessar as páginas do site.

Utilizando uma Página de Login Customizada

O processo de configuração do plugin que define a nossa home page como a página de login é simples – basta adicionarmos as entradas abaixo no arquivo `SecurityConfig.groovy`:

```
loginFormUrl = "/"
defaultTargetUrl = "/post/timeline"
```

A primeira opção faz com que o Spring Security redirecione para a home page sempre que uma autenticação for necessária, e a segunda diz ao Spring Security para carregar, por default, a página de posts do usuário, após um login ter sido efetuado com sucesso. Se o usuário é redirecionado para a home page para autenticar, o Spring Security irá então redirecionar de volta para a página original.

Para que uma página de login customizada possa ser utilizada é preciso que os campos do formulário tenham nomes especiais – nomes que são reconhecidos pelo Spring Security. E isto nos dá a oportunidade de adicionar uma caixa de verificação "Remeber me". Segue abaixo o novo formulário de login que deverá estar definido na página `grails-app/views/_sidebar.gsp`:

```
<form action="${resource(file: 'j_spring_security_check')}"
      method="POST">
    <table>
        <tr>
            <td>User Id:</td>
            <td><g:textField name="j_username"/></td>
        </tr>
        <tr>
            <td>Password:</td>
            <td><input name="j_password" type="password"/></td>
        </tr>
        <tr>
            <td>Remember:</td>
            <td><input type="checkbox"
                       name="_spring_security_remember_me"/></td>
        </tr>
        <tr>
            <td colspan="2">
                <g:submitButton name="login" value="Login"/>
            </td>
        </tr>
    </table>
</form>
```

A parte crítica deste código é o URL que consta do atributo action do form, e os campos `j_username`, `j_password` e `_spring_security_remember_me`.

Registrando um Novo Usuário

Uma possibilidade para o registro de usuários seria a construção de código de interface de gerenciamento de usuários em torno das classes `Usuario` e `Role`. O plugin inclusive nos ajuda a implementar estas funcionalidades com o comando:

```
generate-manager command
```

Mas e se o nosso tipo de aplicação necessita que os próprios usuários efetuem seus registros? Novamente o plugin pode nos ajudar através do comando:

```
grails generate-registration
```

Este comando irá configurar os controladores e as visões necessários, e irá inclusive efetuar o download da API JavaMail e Activation JARs para que o plugin possa enviar e-mails de confirmação. Além de tudo isso, a página de registro irá incluir um captcha para imperir que robôs geradores de spam se registrem. Isso é quase tudo o que precisamos fazer, mas esta característica (de registro de usuários) do plugin requer a criação de alguns novos campos na classe `Usuario`.

```
class Usuario {
    ...
    String email
    String userRealName
    boolean emailShow = false
    ...
    static constraints = {
        ...
        email(nullable: true, blank: true)
        userRealName(nullable: true, blank: true)
    }
    ...
}
```

Adicionamos novas constraints, pois o `BootStrap` não está fornecendo aos usuários que ele cria, valores para os novos campos, logo precisamos torná-los opcionais. Note que a classe `Usuario` gerada pelo plugin inclui estes campos extras por default.

Uma outra coisa que não podemos esquecer é que o registro de usuários adiciona alguns URLs extras que precisam ser acessados pelos usuários que não estão autenticados. Logo, precisamos adicionar estas duas regas na configuração de segurança.

```
/captcha/**=IS_AUTHENTICATED_ANONYMOUSLY
/register/**=IS_AUTHENTICATED_ANONYMOUSLY
```

Lembre-se que estas regras devem vir antes de `/**;` .

Após estas modificações poderemos acessar a tela de registro de usuários em `http://localhost:8080/hubbub/register/`. A interface gerada é bem básica, mas você pode registrar um novo ID do usuário e se logar com ele em seguida. Como o plugin gera os controladores e as páginas da visão, podemos customizar todos eles. Uma outra característica do registro de usuários que ainda não mencionamos é a confirmação de e-mails. Você pode configurar o plugin para que ele envie um e-mail para todos os usuários que se registrarem com sucesso. Esta característica encontra-se desabilitada por default, mas você pode habilitá-la assim:

```
useMail = true
```

Para o plugin conseguir enviar e-mails você precisará configurar um servidor SMTP. A lista completa das configurações (para SecurityConfig) de e-mail pode ser encontrada em: `http://www.grails.org/AcegiSecurity+Plugin+-+Customizing+with+SecurityConfig`).

Note que esta característica é bem básica e apenas envia uma mensagem de notificação. Ela não envia um e-mail contendo um código ou link para ativar a conta do usuário correspondente. Logo, se você quiser este comportamento terá que implementá-lo.

Estreitando as Restrições de Acesso

O Spring Security utiliza o conceito de autorizações para designar e determinar direitos, isto é, quem pode fazer o que.

O plugin resolve este problema utilizando autorizações denominadas perfis (roles). Todo usuário possui alguns perfis que dão a ele direitos de acessar determinados recursos ou URLs. A tabela abaixo mostra como regras podem ser combinadas com perfis para determinar se um usuário pode acessar um determinado URL.

Rule requires	User has	Access granted?
ROLE_USER	ROLE_USER	Yes
ROLE_ADMIN	ROLE_USER	No
ROLE_USER,ROLE_ADMIN	ROLE_USER	Yes

Se qualquer perfil que o usuário possui casar com um dos perfis necessários, o acesso é permitido. Observe que nos exemplos acima, `ROLE_USER` e `ROLE_ADMIN` são meros strings. E também que, se o usuário não possui nenhum perfil, não consegue nem se logar.

E como os perfis são designados? Ligando uma instância do tipo `Role` a um objeto `Usuario`, programaticamente, como fizemos no `BootStrap`, ou através de uma interface.

Uma instância `Role` típica possui um nome como `ROLE_USER` e uma descrição. Você pode tornar este perfil um requisito de acesso para um determinado URL adicionando-o à regra correspondente, conforme vem abaixo:

```
/profile/**=ROLE_USER
```

Qual a diferença entre `IS_AUTHENTICATED_REMEMBERED`, `IS_AUTHENTICATED_FULLY`, e `ROLE_USER`?

- `IS_AUTHENTICATED_REMEMBERED` – Permite o acesso de qualquer usuário que tiver sido autenticado ou reconhecido (remembered).
- `IS_AUTHENTICATED_FULLY` - Permite o acesso de qualquer usuário que tiver sido autenticado. Não permite o acesso de usuários reconhecidos.
- `ROLE_USER` – Um perfil definido pelo usuário que somente permite o acesso a usuários que possuem este perfil. É utilizado por usuários autenticados ou reconhecidos.

Vamos considerar um exemplo concreto. Digamos que você criou um "user-management UI" utilizando o comando `generate-manager`. Certamente você não quer que qualquer um tenha acesso a este subsistema de segurança, pois a modificação de informações de usuários assim como a adição, deleção e desabilitação de contas são tarefas que não devem ser executadas por qualquer usuário. O que fazer nesses casos?

Primeiramente, crie um novo perfil, no `BootStrap`, por exemplo, com o nome `ROLE_ADMIN`, e o designe a um usuário:

```
def role = new Role(authority: "ROLE_ADMIN", description: "A super user.")
def admin = new User(userId: "dilbert", ...).save()
role.addToPeople(admin)
role.save()
```

Em seguida, você deve restringir o acesso a este subsistema de segurança a apenas este perfil:

```
/user/**=ROLE_ADMIN
/role/**=ROLE_ADMIN
/requestmap/**=ROLE_ADMIN
```

É isto! O subsistema de segurança está restrito apenas a usuários administradores.

Mas ainda há um problema. Imagine que todos os usuários possam alterar seus profiles. O que impede um usuário de alterar o profile de outro? Neste caso, o Spring Security não consegue resolver o problema. A solução está na utilização de um filtro do Grails.

Tudo o que precisamos fazer é a configuração de um interceptador do tipo before para a classe `ProfileController`. Este interceptador deve verificar se o usuário corrente é o dono do profile requisitado.

Segue abaixo o código do filtro que deve ser definido em `grails-app/conf/SecurityFilters.groovy`. Se o `id` do usuário não casar com o usuário requisitado no URL, o acesso é bloqueado.

```
package com.grailsinaction
class SecurityFilters {
    def authenticateService          ← 1. Serviço de Autenticação
    def filters = {
        profileChanges(controller: "profile", action: "edit,update") {
            before = {
                def currUserId = authenticateService.userDomain().userId
                if (currUserId != params.userId) {          ← 2
                    redirect(controller: "login", action: "denied")
                    return false
                }
                return true
            }
        }
    }
}
```

O filtro é surpreendentemente simples. O primeiro ponto a chamar atenção é o uso do `authenticateService` (1) para recuperar as informações do usuário corrente. Dentro do filtro, comparamos o `id` do usuário corrente com o valor do parâmetro de requisição `userId` (2). Se eles não casarem, redirecionamos para a página padrão de acesso negado do plugin e retornamos `false` para evitar que o Grails siga em frente com a requisição.