Grails在应用安全方面自动完成的工作有：

1. 通过GROM访问数据库的所有sql语句都经过了SQL转义用来抵御sql注入攻击
2. grails生成的所有HTML页面对显式的数据项都做了转义
3. grails生成链接的Tags（link、form、createLink、createLinkTo等）都使用了适合的转义机制防止代码注入
4. grails还提供了codecs给开发人员手工对需要展现的HTML、JavaScript和URLs做转义，防止代码注入攻击。

# 防止攻击

## SQL注入攻击

尽管GORM已经对写入数据库的数据做了转义，但是还是有可能在HQL代码中对转入的请求参数没有做检查导致发生HQL注入的风险。例如：

|  |
| --- |
| def vulnerable() {  def books = Book.find("from Book as b where b.title ='" + params.title + "'")  } |

或者使用GString

|  |
| --- |
| def vulnerable() {  def books = Book.find("from Book as b where b.title ='${params.title}'")  } |

这样的代码就有发生HQL注入的风险。应该使用如下的方式避免：

|  |
| --- |
| def safe() {  def books = Book.find("from Book as b where b.title = ?",  [params.title])  } |
| def safe() {  def books = Book.find("from Book as b where b.title = :title",  [title: params.title])  } |

## 跨站脚本攻击

需要明确的确定请求是来自自己的应用而非其他站点。需要明确所有显示在页面上的数据都经过正确的转义。例如，对于HTML或者XHTML页面，需要确定人们不能注入恶意的JavaScript或者HTML代码到其他人查看的数据或标签上。

另外也需要避免使用用户请求参数或者提交的数据值来作为需要重定向的URL。例如有个请求参数sucessURL用来确定用户成功登陆后要重定向的下一个URL，攻击者就可以在自己的网站上模拟一个登陆界面，然后把用户重定向到自己的站点上去，从而获取用户的账户信息。

## 跨站请求伪造攻击

跨站请求伪造是指一个当前站点信任的用户从另外一个站点发送一个未经授权的请求到当前站点。例如一个用户登录A网站后，就可以从B网站通过潜入的链接发送未授权请求到A网站。防止攻击的最好办法是给每个form附加一个userToken属性。可以查看[防止form重复提交](http://grails.org/doc/latest/guide/single.html" \l "formtokens)。另外不要启用remember-me cookies。

## HTML/URL注入攻击

Grails提供的Tag libraries对URL做了转义，并且提供了[codecs](http://grails.org/doc/latest/guide/single.html" \l "codecs)用来对显示的内容作转义。

## 拒绝服务攻击

攻击者可能会通过请求一个能返回大量数据的查询从何引起应用服务宕机或者相应变慢，一个好的建议是对所有的查询请求都设定一个最大结果集。

|  |
| --- |
| def safeMax = Math.max(params.max?.toInteger(), 100) // limit to 100 results  return Book.list(max:safeMax) |

# 防止跨站脚本攻击

跨站脚本攻击是指攻击者通过在form中提交HTML或者JavaScript代码，当这些提交的代码在页面上显示的时候引起浏览器发生崩溃等反应。可以通过对所有要显示的内容作转义来防止攻击。

例如：

|  |
| --- |
| <script>alert('Got ya!');</script> |

转义成：

|  |
| --- |
| &lt;script&gt;alert('Got ya!');&lt;/script&gt; |

这样这段代码在页面上显示的时候就不会跳出alert窗口了。

Grails默认对于所有在GSP中通过${}显示的内容都做了转义，如果在某些情况下，不需要转义，则可以这样使用。

|  |
| --- |
| <section>${raw(page.content)}</section> |

## 配置

对于防止跨站脚本攻击的默认配置都在Config.groovy中，内容如下：

|  |
| --- |
| grails {  views {  gsp {  encoding = 'UTF-8'  htmlcodec = 'xml' // use xml escaping instead of HTML4 escaping  codecs {  expression = 'html' // escapes values inside ${}  scriptlet = 'html' // escapes output from scriptlets in GSPs  taglib = 'none' // escapes output from taglibs  staticparts = 'none' // escapes output from static template parts  }  }  // escapes all not-encoded output at final stage of outputting  filteringCodecForContentType {  //'text/html' = 'html'  }  }  } |

codecs配置在codecs配置块中，主要有：

* expression - The expression codec is used to encode any code found within ${..} expressions. The default for newly created application is htmlencoding.
* scriptlet - Used for output from GSP scriplets (<% %>, <%= %> blocks). The default for newly created applications is html encoding
* taglib - Used to encode output from GSP tag libraries. The default is none for new applications, as typically it is the responsibility of the tag author to define the encoding of a given tag and by specifying none Grails remains backwards compatible with older tag libraries.
* staticparts - Used to encode the raw markup output by a GSP page. The default is none.

### Per Plugin Encoding

针对每一个插件也可以配置是否需要对输出encoding，例如有一个名家foo的插件可以在Config.groovy中配置该插件不对输出encoding：

|  |
| --- |
| foo.grails.views.gsp.codecs.expression = "none" |

### Per Page Encoding

针对每一个页面也可以配置是否需要对输出做encoding，使用page directive：

|  |
| --- |
| <%@page expressionCodec="none" %> |

### Per Tag Library Encoding

创建TagLib时可以指定tag输出的encoding

|  |
| --- |
| static defaultEncodeAs = 'html' |

## Encoding and Decoding Objects

Grails支持动态Encode/decode方法，详见官方文档。

# 认证

Grails并不包含认证模块，通过插件的方式，Grails能支持各种类型的认证机制。

## Spring Security插件

Grails的Spring Security插件是Grails的官方插件。该插件是由一组模块组成，可以选择特定的模块支持特定的认证方式。

主要模块有[Core plugin](http://grails.org/plugin/spring-security-core)：支持简单的Form-based认证，Http basic认证，密码加密解密等。

[OpenID](http://grails.org/plugin/spring-security-openid)，[ACL](http://grails.org/plugin/spring-security-acl)，[single sign-on with Jasig CAS](http://grails.org/plugin/spring-security-cas)，[LDAP](http://grails.org/plugin/spring-security-ldap)，[Kerberos](http://grails.org/plugin/spring-security-kerberos)。

并且Spring Security还提供了一个[UI](http://grails.org/plugin/spring-security-ui)插件，支持用户注册、找回密码、登录等界面，并提供了一套关于安全的Tag标签。

### 安装security-core插件

安装完成后运行s2-quickstart命令进行初始化工作：

|  |
| --- |
| grails s2-quickstart com.yourapp User Role Requestmap |

### 安装security-cas插件

### 配置cas插件

在Config.groovy中增加如下配置：

|  |
| --- |
| grails.plugin.springsecurity.cas. loginUri = '/login'  grails.plugin.springsecurity.cas. serviceUrl = 'http://localhost:8080/your-app-name /j\_spring\_cas\_security\_check'  grails.plugin.springsecurity.cas. serverUrlPrefix = 'https://your-cas-server/cas'  grails.plugin.springsecurity.cas. proxyCallbackUrl = 'http://localhost:8080/your-app-name /secure/receptor'  grails.plugin.springsecurity.cas. proxyReceptorUrl = '/secure/receptor' |

## Shiro插件

[Shiro插件](http://grails.org/plugin/shiro)提供了基于用户和角色对controller和action进行访问控制的机制。controller需要继承JsecAuthBase，并实现一个accessControl块，例如：

|  |
| --- |
| class ExampleController extends JsecAuthBase {  static accessControl = {  // All actions require the 'Observer' role.  role(name: 'Observer')  // The 'edit' action requires the 'Administrator' role.  role(name: 'Administrator', action: 'edit')  // Alternatively, several actions can be specified.  role(name: 'Administrator', only: [ 'create', 'edit', 'save', 'update' ])  }  …  } |