Java 安全之：csrf防护实战分析

上文总结了csrf攻击以及一些常用的防护方式，csrf全称Cross-site request forgery(跨站请求伪造)，是一类利用信任用户已经获取的注册凭证，绕过后台用户验证，向被攻击网站发送未被用户授权的跨站请求以对被攻击网站执行某项操作的一种恶意攻击方式。

上面的定义比较抽象，我们先来举一个简单的例子来详细解释一下csrf攻击，帮助理解。

假设你通过电脑登录银行网站进行转账，一般这类转账页面其实是一个form表单，点击转账其实就是提交表单，向后台发起http请求，请求的格式大概像下面这个样子：

```
POST /transfer HTTP/1.1
Host: xxx.bank.com
Cookie: JSESSIONID=randomid; Domain=xxx.bank.com; Secure; HttpOnly
Content-Type: application/x-www-form-urlencoded

amount=100.00&routingNumber=1234&account=8888
```

好了，现在给自己的账户转完账了，但是这时你一般不会立马退出银行网站的登录，你可能会紧接着去上网浏览别的网页，碰巧你上网的时候看到一些很吸引人眼球的广告(比如在家兼职轻松月入上万。。。之类的)，你点击了一下，但是发现什么也没有，也许你会关掉这个网页，以为什么都没有发生。但是后台可能已经发生了一系列的事情，如果这是个钓鱼网站，并且刚才你点击的页面恰好又包含一个form表单，如下所示：

```
<form action="https://xxx.bank.com/transfer" method="post">
  <input type="hidden"
      name="amount"
      value="100.00"/>
  <input type="hidden"
      name="routingNumber"
      value="evilsRoutingNumber"/>
  <input type="hidden"
      name="account"
      value="evilsAccountNumber"/>
  <input type="submit"
      value="Win Money!"/>
</form>
```

这里只要你点击网页便会自动提交表单，导致你向一个陌生账户转账100元(这些都可通过js实现自动化)，而且是未经过你的授权的情况下，这就是csrf的攻击方式，虽然其不知道你的登录信息，但是其利用浏览器自身的机制来冒充用户绕过后台用户验证从而发起攻击。

csrf是一种常见的web攻击方式，一些现有的安全框架中都对该攻击的防护提供了支持，比如spring security，从4.0开始，默认就会启用CSRF保护，会针对PATCH，POST，PUT和DELETE方法进行防护。本文会结合spring security提供的防护方法，并结合其源码来学习一下其内部防护原理，本文涉及到的Spring Security源码版本为5.1.5。

本文目录如下：

1. 使用Spring Security防护CSRF攻击

通过Spring Security来防护CSRF攻击需要做哪些配置呢，总结如下：

- 使用合适的HTTP请求方式
- 配置CSRF保护
- 使用CSRF Token

1.1 使用合适的HTTP请求方式

第一步是确保要保护的网站暴露的接口使用合适的HTTP请求方式，就是在还未开启Security的CSRF之前需要确保所有的接口都只支持使用PATCH、POST、PUT、DELETE这四种请求方式之一来修改后端数据。

这并不是Spring Security在防护CSRF攻击方面的自身限制，而是合理防护CSRF攻击所必须做的，原因是通过GET的方式传递私有数据容易导致其泄露，使用POST来传递敏感数据更合理。

1.2 配置CSRF保护

下一步就是将Spring Security引入你的后台应用中。有些框架通过让用户session失效来处理无效的CSRF Token，但是这种方式是有问题的，取而代之，Spring Security默认返回一个403的HTTP状态码来拒绝无效访问，可以通过配置AccessDeniedHandler来实现自己的拒绝逻辑。

如果项目中是采用的XML配置，则必须显示的使用<csrf>标签元素来开启CSRF防护，详见<csrf>。

通过Java配置的方式则会默认开启CSRF防护，如果希望禁用这一功能，则需要手动配置，见下面的示例，更详细的配置可以参考csrf()方法的官方文档。

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

  @Override
  protected void configure(HttpSecurity http) throws Exception {
    http
      .csrf().disable();
  }
}
```

1.3 使用CSRF Token

接下来就是在每次请求的时候带上一个CSRF Token，根据请求的方式不同会有不同的方式：

1.3.1 Form表单提交

通过表单提交会将CSRF Token附在Http请求的_csrf属性中，后台接口从请求中获取token，如下是一个示例(JSP)：

```
<c:url var="logoutUrl" value="/logout"/>
<form action="${logoutUrl}"
    method="post">
  <input type="submit"
    value="Log out" />
  <input type="hidden"
    name="${_csrf.parameterName}"
    value="${_csrf.token}"/>
</form>
```

其实就是后台在渲染页面时先生成一个CSRF Token，放到表单中；然后在用户提交表单时就会附带上这个CSRF Token，后台将其取出来并进行校验，不一致则拒绝这次请求。这里因为这Token是后台生成的，这对于第三方网站是获取不到的，通过这种方式实现防护。

1.3.2 Ajax和JSON请求

如果是使用的JSON，则不需要将CSRF Token以HTTP参数的形式提交，而是放在HTTP请求头中。典型的做法是将CSRF Token包含在在页面的元标签中。如下是一个JSP的例子：

```
<html>
  <head>
    <meta name="_csrf" content="${_csrf.token}"/>
    <!-- default header name is X-CSRF-TOKEN -->
    <meta name="_csrf_header" content="${_csrf.headerName}"/>
    <!-- ... -->
  </head>
  <!-- ... -->
```

然后在所有的Ajax请求中需要带上CSRF Token，如下是jQuery中的实现：

```
$(function () {
  var token = $("meta[name='_csrf']").attr("content");
  var header = $("meta[name='_csrf_header']").attr("content");
  $(document).ajaxSend(function(e, xhr, options) {
    xhr.setRequestHeader(header, token);
  });
});
```

到这里所有的配置都已经好了，包括接口调用方式的设计、框架的配置、前端页面的配置，前文中讲了一系列的防护方式，Spring Security又是采用的什么方式呢，最直接的方式就是看源码了。

2. Spring Security的CSRF防护原理

Spring Security是基于Filter(过滤器)来实现其安全功能的，关于CSRF防护的主要逻辑是在CsrfFilter这个过滤器中的，继承自OncePerRequestFilter，并且重写了doFilterInternal方法：

```
    protected void doFilterInternal(HttpServletRequest request,
            HttpServletResponse response, FilterChain filterChain)
                    throws ServletException, IOException {
        request.setAttribute(HttpServletResponse.class.getName(), response);
        // 通过tokenRepository从request中获取csrf token
        CsrfToken csrfToken = this.tokenRepository.loadToken(request);
        final boolean missingToken = csrfToken == null;
        // 如果未获取到token则新生成token并保存
        if (missingToken) {
            csrfToken = this.tokenRepository.generateToken(request);
            this.tokenRepository.saveToken(csrfToken, request, response);
        }
        request.setAttribute(CsrfToken.class.getName(), csrfToken);
        request.setAttribute(csrfToken.getParameterName(), csrfToken);
        // 判断是否需要进行csrf token校验
        if (!this.requireCsrfProtectionMatcher.matches(request)) {
            filterChain.doFilter(request, response);
            return;
        }
        // 获取前端传过来的实际token
        String actualToken = request.getHeader(csrfToken.getHeaderName());
        if (actualToken == null) {
            actualToken = request.getParameter(csrfToken.getParameterName());
        }
        // 校验两个token是否相等
        if (!csrfToken.getToken().equals(actualToken)) {
            if (this.logger.isDebugEnabled()) {
                this.logger.debug("Invalid CSRF token found for "
                        + UrlUtils.buildFullRequestUrl(request));
            }
            // 如果是token缺失导致，则抛出MissingCsrfTokenException异常
            if (missingToken) {
                this.accessDeniedHandler.handle(request, response,
                        new MissingCsrfTokenException(actualToken));
            }
            // 如果不是同一个token则抛出InvalidCsrfTokenException异常
            else {
                this.accessDeniedHandler.handle(request, response,
                        new InvalidCsrfTokenException(csrfToken, actualToken));
            }
            return;
        }
        // 执行下一个过滤器
        filterChain.doFilter(request, response);
    }
```

整个流程还是很清晰的，我们总结一下：

- 先通过tokenRepository从request中获取csrf token；
- 如果未获取到token则新生成token并保存；
- 判断是否需要进行csrf token校验，不需要则直接执行下一个过滤器；
- 调用request的getHeader()方法或者getParameter()方法获取前端传过来的实际token；
- 校验两个token是否相等，不相等则抛出异常，相等则校验通过，执行下一个过滤器；

可以知道，Spring Security是借助CSRF Token来实现防护的，上文我们讲到，通过token的方式可以选择cookie来存储也可以选择session的方式，那Spring Security提供了什么方式呢，答案就在获取token的tokenRepository中，我们看一下，这个tokenRepository类型是CsrfTokenRepository(这是一个接口)，Spring Security提供了三种实现，分别是HttpSessionCsrfTokenRepository、CookieCsrfTokenRepository、LazyCsrfTokenRepository，我们着重看一下前两者，顾名思义，一个是通过session，而另一个则是通过cookie，我们再分别看一下其各自实现的loadToken()方法，验证一下。

```
// CookieCsrfTokenRepository中的实现
public CsrfToken loadToken(HttpServletRequest request) {
    Cookie cookie = WebUtils.getCookie(request, this.cookieName);
    if (cookie == null) {
        return null;
    }
    String token = cookie.getValue();
    if (!StringUtils.hasLength(token)) {
        return null;
    }
    return new DefaultCsrfToken(this.headerName, this.parameterName, token);
}

// HttpSessionCsrfTokenRepository中的实现
public CsrfToken loadToken(HttpServletRequest request) {
    HttpSession session = request.getSession(false);
    if (session == null) {
        return null;
    }
    return (CsrfToken) session.getAttribute(this.sessionAttributeName);
}
```

到这里我们已经很清楚了，Spring Security提供多种保存token的策略，既可以保存在cookie中，也可以保存在session中，这个可以手动指定。所以前文说到的两个关于token的防护方式，Spring Security都支持。既然到这里了，我们就再看一下Spring Security是如何生成和保存token的，这里仅以CookieCsrfTokenRepository的实现为例：

```
// 生成token
public CsrfToken generateToken(HttpServletRequest request) {
    return new DefaultCsrfToken(this.headerName, this.parameterName,
            createNewToken());
}

private String createNewToken() {
    return UUID.randomUUID().toString();
}

// 保存token
public void saveToken(CsrfToken token, HttpServletRequest request,
        HttpServletResponse response) {
    String tokenValue = token == null ? "" : token.getToken();
    Cookie cookie = new Cookie(this.cookieName, tokenValue);
    cookie.setSecure(request.isSecure());
    if (this.cookiePath != null && !this.cookiePath.isEmpty()) {
            cookie.setPath(this.cookiePath);
    } else {
            cookie.setPath(this.getRequestContext(request));
    }
    if (token == null) {
        cookie.setMaxAge(0);
    }
    else {
        cookie.setMaxAge(-1);
```

```
        }
        if (cookieHttpOnly && setHttpOnlyMethod != null) {
            ReflectionUtils.invokeMethod(setHttpOnlyMethod, cookie, Boolean.TRUE);
        }

        response.addCookie(cookie);
    }
```

可以看到，生成的token其实本质就是一个uuid，而保存则是保存在cookie中，涉及到cookie操作，其中有很多细节，本文就不详述了。

3. 总结

本文先解释了一个csrf攻击的基本例子，然后介绍了使用Spring Security来防护csrf攻击所需要的配置，最后再从Spring Security源码的角度学习了一下其是如何实现csrf防护的，基本原理还是通过token来实现，具体可以借助于cookie和session的方式来实现。

注：本文涉及到的源码均来自Spring Security 5.1.5。

参考文献：

Cross Site Request Forgery (CSRF)

Spring Security Architecture

相关阅读:

打开Fiddle，提示"Machine-wide Progress Telerik Fiddler installation has been found at ...Please, use that one or uninstall it ...

Type Target runtime Apache Tomcat v8.0 is not defined.

Loadrunner的安装注意事项

11.java并发编程的艺术-java并发编程实践

10.java并发编程的艺术-Executor框架

9.java并发编程的艺术-java中的线程池

8.java并发编程的艺术-java中的并发工具类

Java并发编程的艺术-java中的13个原子操作类

6.java并发容器和框架——Fork/Join框架

6.java并发容器和框架——Java中的阻塞队列

原文地址：https://www.cnblogs.com/volcano-liu/p/11301057.html

最新文章

单点登录(SSO)与用户业务介绍

MySql的锁机制

计算一个字符串种每个字符出现的次数

记录下线程问题

spring种的事务传播属性

一个SQL查询出每门课程的成绩都大于80的学生姓名

面试总结

HashMap原理的理解

摘取栈和堆的理解

Arraylist与LinkedList的比较

热门文章

接口和抽象类的区别
栈与队列的区别
springmvc入门案例
TortoiseSVN的使用
TortoiseSVN安装
VisualSVN的使用
Windows下的subversion（SVN）下载安装
解决 Android Studio logcat 无法显示日志，一条命令搞定
【转载】数据挖掘系列（2）--关联规则FpGrowth算法
【转载】数据挖掘系列（1）关联规则挖掘基本概念与Aprior算法