# 常见Web安全漏洞

## XSS攻击

### 什么是XSS攻击手段

XSS攻击使用Javascript脚本注入进行攻击

例如在提交表单后，展示到另一个页面，可能会受到XSS脚本注入，读取本地cookie远程发送给黑客服务器端。

<script>alert('sss')</script>

<script>window.location.href='http://www.itmayiedu.com';</script>

对应html源代码: &lt;script&gt;alert('sss')&lt;/script&gt;

**最好使用火狐浏览器演示效果**

### 如何防御XSS攻击

将脚本特殊字符，转换成html源代码进行展示。

汉子编码http://www.mytju.com/classcode/tools/encode\_gb2312.asp

步骤：编写过滤器拦截所有getParameter参数，重写httpservletwrapp方法

将参数特殊字符转换成html源代码保存.

|  |
| --- |
| // 重写HttpServletRequestWrapper 防止XSS攻击  **public** **class** XssHttpServletRequestWrapper **extends** HttpServletRequestWrapper {  **private** HttpServletRequest request;  /\*\*  \* **@param** request  \*/  **public** XssHttpServletRequestWrapper(HttpServletRequest request) {  **super**(request);  **this**.request = request;  }  @Override  **public** String getParameter(String name) {  // 过滤getParameter参数 检查是否有特殊字符  String value = **super**.getParameter(name);  System.***out***.println("value:" + value);  **if** (!StringUtils.*isEmpty*(value)) {  // 将中文转换为字符编码格式，将特殊字符变为html源代码保存  value = StringEscapeUtils.*escapeHtml*(value);  System.***out***.println("newValue:" + value);  }  **return** value;  }  } |

SpringBoot启动加上@ServletComponentScan

|  |
| --- |
| @SpringBootApplication  @ServletComponentScan  **public** **class** App {  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(App.**class**, args);  }  } |

## SQL注入攻击

### 什么是SQL注入

SQL注入：利用现有应用程序，将（恶意）的SQL命令注入到后台数据库执行一些恶意的

作。

造成SQL注入的原因是因为程序没有有效过滤用户的输入，使攻击者成功的向服务器提交恶意的SQL查询代码，程序在接收后错误的将攻击者的输入作为查询语句的一部分执行，导致原始的查询逻辑被改变，额外的执行了攻击者精心构造的恶意代码

### SQL注入防攻击手段

不要使用拼接SQL语句方式、最好使用预编译方式，在mybatis编写sql语句的时候，最好使用?传参数方式，不要使用#传参数，因为#传参数方式，可能会受到sql语句攻击。

演示案例:

http://127.0.0.1:8080/login?userName='liusi'&password='123'

http://127.0.0.1:8080/login?userName='liusi'&password='123' or 1=1

|  |
| --- |
| @RestController  **public** **class** LoginController {  @Autowired  **private** UserMapper userMapper;  @RequestMapping("/login")  **public** String login(UserEntity userEntity) {  System.***out***.println("账号密码信息:userEntity:" + userEntity.toString());  UserEntity login = userMapper.login(userEntity);  **return** login == **null** ? "登陆失败!" : "登陆成功!";  }  }  **public** **interface** UserMapper {  @Select(" SELECT \* FROM user\_info where userName=${userName} and password=${password}")  **public** UserEntity login(UserEntity userEntity);  } |

### MyBatis #与?区别

#{}: 解析为一个 JDBC 预编译语句（prepared statement）的参数标记符，一个 #{ } 被解析为一个参数占位符,可以防止SQL注入问题。

${}: 仅仅为一个纯碎的 string 替换，在动态 SQL 解析阶段将会进行变量替换。

## Http请求防盗链

### 什么是防盗链

比如A网站有一张图片，被B网站直接通过img标签属性引入，直接盗用A网站图片展示。

### 如何实现防盗链

判断http请求头Referer域中的记录来源的值，如果和当前访问的域名不一致的情况下，说明该图片可能被其他服务器盗用。

#### 使用过滤器判断请求头Referer记录请求来源

|  |
| --- |
| @WebFilter(filterName = "imgFilter", urlPatterns = "/imgs/\*")  **public** **class** ImgFilter **implements** Filter {  @Value("${domain.name}")  **private** String domainName;  **public** **void** init(FilterConfig filterConfig) **throws** ServletException {  }  **public** **void** doFilter(ServletRequest request, ServletResponse response, FilterChain chain)  **throws** IOException, ServletException {  HttpServletRequest req = (HttpServletRequest) request;  String referer = req.getHeader("Referer");  **if** (StringUtils.*isEmpty*(referer)) {  request.getRequestDispatcher("/imgs/error.png").forward(request, response);  **return**;  }  String domain = getDomain(referer);  **if** (!domain.equals(domainName)) {  request.getRequestDispatcher("/imgs/error.png").forward(request, response);  **return**;  }  chain.doFilter(request, response);  }  /\*\*  \* 获取url对应的域名  \*  \* **@param** url  \* **@return**  \*/  **public** String getDomain(String url) {  String result = "";  **int** j = 0, startIndex = 0, endIndex = 0;  **for** (**int** i = 0; i < url.length(); i++) {  **if** (url.charAt(i) == '/') {  j++;  **if** (j == 2)  startIndex = i;  **else** **if** (j == 3)  endIndex = i;  }  }  result = url.substring(startIndex + 1, endIndex);  **return** result;  }  **public** **void** destroy() {  }  } |

注意测试的时候,最好开启两个不同的浏览器测试，避免图片缓存的原因

## CSRF攻击

### CSRF攻击产生的原因

（Cross Site Request Forgery, 跨站域请求伪造）是一种网络的攻击方式，它在 2007 年曾被列为互联网 20 大安全隐患之一,也被称为“One Click Attack”或者Session Riding，通常缩写为CSRF或者XSRF，是一种对网站的恶意利用也就是人们所知道的钓鱼网站。尽管听起来像跨站脚本（XSS），但它与XSS非常不同，并且攻击方式几乎相左。XSS利用站点内的信任用户，而CSRF则通过伪装来自受信任用户的请求来利用受信任的网站。与XSS攻击相比，CSRF攻击往往不大流行（因此对其进行防范的资源也相当稀少）和难以防范，所以被认为比XSS更具危险性。

### API接口幂等性设计

#### API接口幂等性设计方案

##### MVCC方案

　多版本并发控制，该策略主要使用 update with condition（更新带条件来防止）来保证多次外部请求调用对系统的影响是一致的。在系统设计的过程中，合理的使用乐观锁，通过 version 或者 updateTime（timestamp）等其他条件，来做乐观锁的判断条件，这样保证更新操作即使在并发的情况下，也不会有太大的问题。例如

select \* from tablename where condition=#condition# // 取出要跟新的对象，带有版本 versoin

update tableName set name=#name#,version=version+1 where version=#version#

在更新的过程中利用 version 来防止，其他操作对对象的并发更新，导致更新丢失。为了避免失败，通常需要一定的重试机制。

##### 去重表

在插入数据的时候，插入去重表，利用数据库的唯一索引特性，保证唯一的逻辑。

##### 悲观锁

select for update，整个执行过程中锁定该订单对应的记录。注意：这种在 DB 读大于写的情况下尽量少用。

##### Token机制,防止页面重复提交

　业务要求：页面的数据只能被点击提交一次

　　发生原因：由于重复点击或者网络重发，或者 nginx 重发等情况会导致数据被重复提交

解决办法：

集群环境：采用 token 加 redis（redis 单线程的，处理需要排队）

单 JVM 环境：采用 token 加 redis 或 token 加 jvm 内存

处理流程：

数据提交前要向服务的申请 token，token 放到 redis 或 jvm 内存，token 有效时间

提交后后台校验 token，同时删除 token，生成新的 token 返回

token 特点:要申请，一次有效性，可以限流

#### 基于Token方式防止API接口幂等

客户端每次在调用接口的时候,需要在请求头中，传递令牌参数，每次令牌只能用一次。

一旦使用之后，就会被删除，这样可以有效防止重复提交。

步骤:

1.生成令牌接口

2. 接口中获取令牌验证

##### 生成令牌接口

|  |
| --- |
| **public** **class** TokenUtils {  **private** **static** Map<String, Object> *tokenMap* = **new** ConcurrentHashMap<String, Object>();  // 获取token  **public** **static** **synchronized** String getToken() {  // 1.生成令牌  String token = "token-" + System.*currentTimeMillis*();  // 2.存入tokenMap  *tokenMap*.put(token, token);  **return** token;  }  // 验证token,并且删除对应的token  **public** **static** Boolean exisToken(String token) {  // 1.从集合中获取token  Object result = *tokenMap*.get(token);  **if** (result == **null**) {  **return** **false**;  }  // 2.删除对应的token  *tokenMap*.remove(token);  **return** **true**;  }  } |

##### 接口中获取令牌验证

|  |
| --- |
| @RestController  **public** **class** OrderController {  @Autowired  **private** OrderMapper orderMapper;  // 获取Token  @RequestMapping("/getToken")  **public** String getToken() {  **return** TokenUtils.*getToken*();  }  // 验证Token  @RequestMapping(value = "/addOrder", produces = "application/json; charset=utf-8")  **public** String addOrder(@RequestBody OrderEntity orderEntity, HttpServletRequest request) {  String token = request.getHeader("token");  **if** (StringUtils.*isEmpty*(token)) {  **return** "参数错误!";  }  **if** (!TokenUtils.*exisToken*(token)) {  **return** "请勿重复提交!";  }  **int** result = orderMapper.addOrder(orderEntity);  **return** result > 0 ? "添加成功" : "添加失败" + "";  }  } |

### 防御CSRF攻击手段

使用图形验证码防止机器模拟接口请求攻击，在调用核心业务接口时，比如支付、下单、等接口，最好使用手机短信验证验证或者是人脸识别，防止其他用户使用Token伪造请求。

## 其他攻击和漏洞

## 网站安全漏洞扫描

# 信息加密与密钥管理

# API安全接口安全设计

# 电子商务风控系统

# 黑名单与白名单系统

# 基于多种方式实现防御DDOS攻击