一、XSS

什么是xss漏洞?

SS作为OWASP TOP 10 之一

XSS被称为跨站脚本攻击(Cross-site scripting),本来应该缩写为CSS,但是由于和CSS(Cascading Style Sheets,层叠样式脚本)重名，所以更名为XSS。XSS(跨站脚本攻击) 主要基于javascript (JS) 完成恶意的攻击行为。

JS可以非常灵活的操作html，css和浏览器，这使得XSS攻击的"想象"空间特别大。XSS通过将精心构造的代码(JS)代码注入到网页中，并由浏览器解释运行这段JS代码，以达到恶意攻击的效果。
当用户访问被XSS脚本注入的网页，XSS脚本就会被提取出来，用户浏览器就会解析这段XSS代码，也就是说用户被攻击了。用户最简单的动作就是使用浏览器上网，并且浏览器中有javascrit解释器，可以解析javascript，然而浏览器不会判断代码是否恶意。也就是说，XSS的对象是用户和浏览器。

2XSS漏洞发生在哪里?

服务器
微博、留言板、聊天室等等收集用户输入的地方都有可能被注入XSS代码，都存在遭受XSS的风险，只要没有对用户的输入进行严格过滤，就会被XSS

1.3 XSS的危害

XSS利用JS代码实现攻击，有很多种攻击方法，以下简单列出几种

@ 盗取各种用户账号

@ 窃取用户Cookie资料，冒充用户身份进入网站

@ 劫持用户回话，执行任意操作

@ 刷流量，执行弹窗广告

@ 传播蠕虫病毒

XSS Platform 根据各种需求生成 XSS攻击代码 比如说获得管理员Cookie 等等

获得管理员cookie 就可以直接访问后台了

1.4 XSS的分类

XSS漏洞大概可以分为三个类型:反射型XSS、存储型XSS、DOM型XSS

反射型XSS（用户触发）

反射型XSS是非持久性、参数型的跨站脚本

反射型XSS的JS代码在Web应用的参数(变量)中，如搜索框的反射型XSS

java中反射型XSS

就是通过给别人发送带有恶意脚本代码参数的URL，当URL地址被打开时，特定的代码参数会被HTML解析、执行。

存在 欺骗性，把存在反射型XSS漏洞的地址发给你，你点击了，造成XSS的攻击。

例如：利用EL表达式 ，EL表达式可以通过后台数据取值（这个时候是不能修改的），它地址栏取值 这个时候就可以使用XSS进行攻击

如果参数由后端生成，并且前台改不掉，那就不存在这种漏洞

```
http://localhost:8081/blog/admin/main?
clientds=%3Cscript%3Ealert(111)%3C/script%3E
```

java中存储型XSS（直接写入数据库）

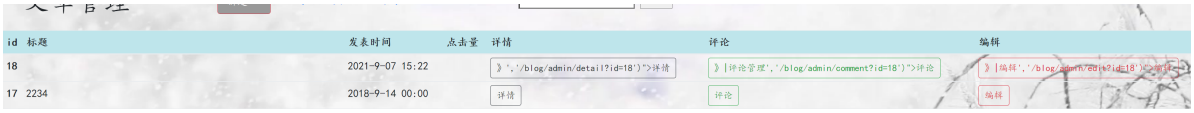
存储型XSS是持久性跨站脚本

持久性体现在XSS代码不是在某个参数(变量)中，而是写进数据库或文件等可以永久保存数据的介质中，存储型XSS通常发生在留言板等地方。我们在留言板位置留言，将恶意代码写入数据库中。

此时，我们只完成了第一步，将恶意代码写入数据库。因为XSS使用的JS代码,JS代码的运行环境是浏览器，所以需要浏览器从服务器载入恶意的XSS代码，才能真正触发XSS

此时需要我们模拟网站后台管理员的身份，查看留言

形成的原因：1、输入参数未过滤 2、与数据库交互（该参数存如数据库） 3、输出未过滤



DOM XSS（使用DOM事件才能触发）

DOM XSS比较特殊。owasp关于DOM型号XSS的定义是基于DOM的XSS是一种XSS攻击

其中攻击的payload由于修改受害者浏览器页面的DOM树而执行的

其特殊的地方就是payload比较难以检测

可能触发DOM型XSS的属性

1、document.referer属性

2、location属性

3、innerHTML属性

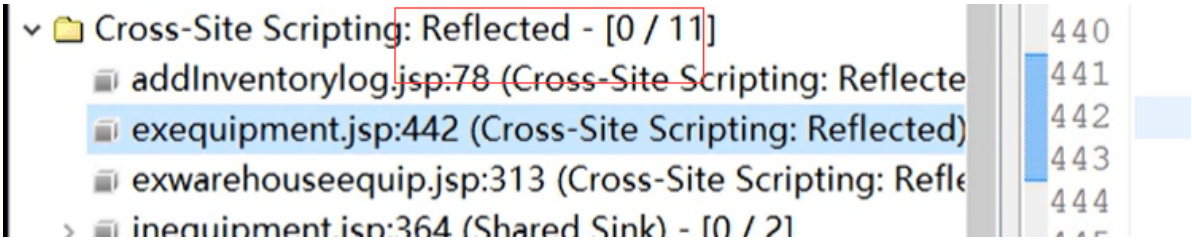
```
    <div id="xssd_main">
        <script>
            function domxss(){
                var str = document.getElementById("text").value;
                document.getElementById("dom").innerHTML = "<a href='"+str+"'>what do you see?</a>";
            }
            //试试：'><img src="#" onmouseover="alert('xss')">
            //试试：' onclick="alert('xss')">,闭合掉就行
        </script>
        <!--<a href="" onclick=('xss')>-->
        <input id="text" name="text" type="text"  value="" />
        <input id="button" type="button" value="click me!" onclick="domxss()" />
        <div id="dom"></div>
    </div>
```

例如：

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title></title>
    </head>
    <body>
        <div class="page-content">

            <div id="xssd_main">

                <!--<a href="" onclick=('xss')>-->
                <input id="text" name="text" type="text"  value="" />
                <input id="button" type="button" value="click me!"
onclick="domxss()" />
                <div id="dom"></div>
            </div>
        </div>
    </body>
              <script>
                   function domxss(){
                       var str = document.getElementById("text").value;
```

```
                        document.getElementById("dom").innerHTML = "<a
href='"+str+"'>what do you see?</a>";
                    }
                    //试试：'><img src="#" onmouseover="alert('xss')">
                    //试试：' onclick="alert('xss')">,闭合掉就行
                </script>
</html>
```

XSS 攻击的防御



XSS防御的总体思路是：**对输入(和URL参数)进行过滤，对输出进行编码。**

也就是对提交的所有内容进行过滤，对url中的参数进行过滤，过滤掉会导致脚本执行的相关内容；然后对动态输出到页面的内容进行html编码，使脚本无法在浏览器中执行。**虽然对输入过滤可以被绕过，但是也还是会拦截很大一部分的XSS攻击 。**

3.1 对输入和URL参数进行过滤(白名单和黑名单)

下面贴出一个常用的XSS filter的实现代码

```
public class XssFilter implements Filter {

    public void init(FilterConfig config) throws ServletException {}

    public void doFilter(ServletRequest request, ServletResponse response,
FilterChain chain)
            throws IOException, ServletException {
        XssHttpServletRequestWrapper xssRequest = new
XssHttpServletRequestWrapper((HttpServletRequest)request);
        chain.doFilter(xssRequest, response);
    }

    public void destroy() {}
}
```

反射型XSS

就是通过给别人发送带有恶意脚本代码参数的URL，当URL地址被打开时，特定的代码参数会被HTML解析、执行。

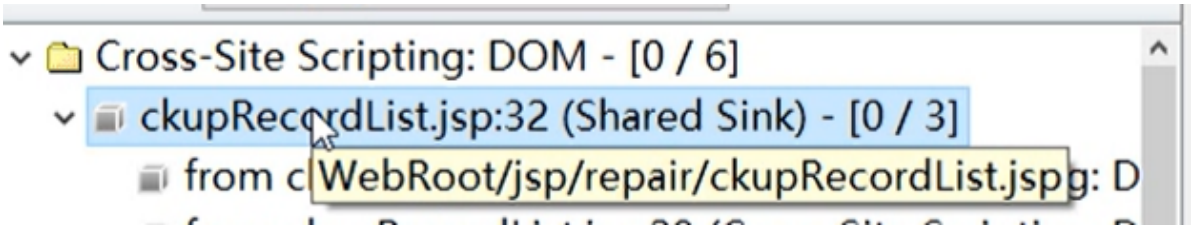
```
public class XssHttpServletRequestWrapper extends HttpServletRequestWrapper {
    HttpServletRequest orgRequest = null;

    public XssHttpServletRequestWrapper(HttpServletRequest request) {
        super(request);
        orgRequest = request;
    }
    /**
     * 覆盖getParameter方法，将参数名和参数值都做xss过滤。<br/>
     * 如果需要获得原始的值，则通过super.getParameterValues(name)来获取<br/>
     * getParameterNames,getParameterValues和getParameterMap也可能需要覆盖
     */
    @Override
    public String getParameter(String name) {
        String value = super.getParameter(xssEncode(name));
        if (value != null) {
            value = xssEncode(value);
        }
        return value;
    }
    /**
     * 覆盖getHeader方法，将参数名和参数值都做xss过滤。<br/>
     * 如果需要获得原始的值，则通过super.getHeaders(name)来获取<br/>
     * getHeaderNames 也可能需要覆盖
     */
    @Override
    public String getHeader(String name) {
        String value = super.getHeader(xssEncode(name));
        if (value != null) {
            value = xssEncode(value);
        }
        return value;
    }
    /**
     * 将容易引起xss漏洞的半角字符直接替换成全角字符
     *
     * @param s
     * @return
     */
    private static String xssEncode(String s) {
        if (s == null || s.isEmpty()) {
            return s;
        }
        StringBuilder sb = new StringBuilder(s.length() + 16);
        for (int i = 0; i < s.length(); i++) {
            char c = s.charAt(i);
            switch (c) {
            case '>':
                sb.append('＞');// 全角大于号
                break;
            case '<':
                sb.append('＜');// 全角小于号
                break;
            case '\'':
                sb.append('‘');// 全角单引号
                break;
            case '\"':
                sb.append('“');// 全角双引号
```

```
                break;
            case '&':
                sb.append('＆');// 全角
                break;
            case '\\':
                sb.append('＼');// 全角斜线
                break;
            case '#':
                sb.append('＃');// 全角井号
                break;
            case '%':    // < 字符的 URL 编码形式表示的 ASCII 字符（十六进制格式） 是:
%3c
                processUrlEncoder(sb, s, i);
                break;
            default:
                sb.append(c);
                break;
            }
        }
        return sb.toString();
    }
    public static void processUrlEncoder(StringBuilder sb, String s, int index){
        if(s.length() >= index + 2){
            if(s.charAt(index+1) == '3' && (s.charAt(index+2) == 'c' ||
s.charAt(index+2) == 'C')){    // %3c, %3C
                sb.append('＜');
                return;
            }
            if(s.charAt(index+1) == '6' && s.charAt(index+2) == '0'){    // %3c
(0x3c=60)
                sb.append('＜');
                return;
            }
            if(s.charAt(index+1) == '3' && (s.charAt(index+2) == 'e' ||
s.charAt(index+2) == 'E')){    // %3e, %3E
                sb.append('＞');
                return;
            }
            if(s.charAt(index+1) == '6' && s.charAt(index+2) == '2'){    // %3e
(0x3e=62)
                sb.append('＞');
                return;
            }
        }
        sb.append(s.charAt(index));
    }
    /**
     * 获取最原始的request
     *
     * @return
     */
    public HttpServletRequest getOrgRequest() {
        return orgRequest;
    }
    /**
     * 获取最原始的request的静态方法
     *
     * @return
```

```
     */
    public static HttpServletRequest getOrgRequest(HttpServletRequest req) {
        if (req instanceof XssHttpServletRequestWrapper) {
            return ((XssHttpServletRequestWrapper) req).getOrgRequest();
        }
        return req;
    }
}
```

然后在web.xml中配置该filter:

```
<filter>
        <filter-name>xssFilter</filter-name>
        <filter-class>com.xxxxxx.filter.XssFilter</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>xssFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
```



3.2 对输出进行编码

在输出数据之前对潜在的威胁的字符进行编码、转义是防御XSS攻击十分有效的措施。如果使用好的话，理论上是可以防御住所有的XSS攻击的。

对所有要动态输出到页面的内容，通通进行相关的编码和转义。当然转义是按照其输出的上下文环境来决定如何转义的。

1> 作为body文本输出，作为html标签的属性输出：

比如：`<span>`${username},

<c:out value="$

```
<input type="text" value="${username}" />
```

此时的转义规则如下：

2> javascript事件

除了上面的那些转义之外，还要附加上下面的转义：

\ 转成 \

/ 转成 \/

; 转成 ； (全角;)

3> URL属性

如果 `<script>`, `<style>`, `<imt>` 等标签的 src 和 href 属性值为动态内容，那么要确保这些url没有执行恶意连接。

确保：href 和 src 的值必须以 http://开头，白名单方式；不能有10进制和16进制编码字符。

4. HttpOnly 与 XSS防御

XSS 一般利用js脚步读取用户浏览器中的Cookie，而如果在服务器端对 Cookie 设置了HttpOnly 属性，那么js脚本就不能读取到cookie，但是浏览器还是能够正常使用cookie。

一般的Cookie都是从document对象中获得的，现在浏览器在设置 Cookie的时候一般都接受一个叫做HttpOnly的参数，跟domain等其他参数一样，一旦这个HttpOnly被设置，**你在浏览器的 document对象中就看不到Cookie了，而浏览器在浏览的时候不受任何影响**，因为Cookie会被放在浏览器头中发送出去(包括ajax的时 候)，应用程序也一般不会在js里操作这些敏感Cookie的，对于一些敏感的Cookie我们采用HttpOnly，对于一些需要在应用程序中用js操作的cookie我们就不予设置，这样就保障了Cookie信息的安全也保证了应用。如果你正在使用的是兼容 Java EE 6.0 的容器，如 Tomcat 7，那么 Cookie 类已经有了 setHttpOnly 的方法来使用 HttpOnly 的 Cookie 属性了。

```
cookie.setHttpOnly(true );
```

```
    <div id="xssd_main">
        <script>
            function domxss(){
                var str = document.getElementById("text").value;
                document.getElementById("dom").innerHTML = "<a href='"+str+"'>what do you see?</a>";
            }
            //试试：'><img src="#" onmouseover="alert('xss')">
            //试试：' onclick="alert('xss')">,闭合掉就行
        </script>
        <!--<a href="" onclick=('xss')>-->
        <input id="text" name="text" type="text"  value="" />
        <input id="button" type="button" value="click me!" onclick="domxss()" />
        <div id="dom"></div>
    </div>
```