通过上篇文章的介绍，我们知道了同源策略可以隔离各个站点之间的DOM交互、页面数据和网络通信，虽然严格的同源策略会带来更多的安全，但是也束缚了Web。这就需要在安全和自由之间找到一个平衡点，所以我们默认页面中可以引用任意第三方资源，然后又引入CSP策略来加以限制；默认XMLHttpRequest和Fetch不能跨站请求资源，然后又通过CORS策略来支持其跨域。

不过支持页面中的第三方资源引用和CORS也带来了很多安全问题，其中最典型的就是XSS攻击。

什么是XSS攻击

XSS全称是Cross Site Scripting，为了与"CSS"区分开来，故简称XSS，翻译过来就是"跨站脚本"。XSS攻击是指黑客往HTML文件中或者DOM中注入恶意脚本，从而在用户浏览页面时利用注入的恶意脚本对用户实施攻击的一种手段。

最开始的时候，这种攻击是通过跨域来实现的，所以叫"跨域脚本"。但是发展到现在，往HTML文件中注入恶意代码的方式越来越多了，所以是否跨域注入脚本已经不是唯一的注入手段了，但是XSS这个名字却一直保留至今。

当页面被注入了恶意JavaScript脚本时，浏览器无法区分这些脚本是被恶意注入的还是正常的页面内容，所以恶意注入JavaScript脚本也拥有所有的脚本权限。下面我们就来看看，如果页面被注入了恶意JavaScript脚本，恶意脚本都能做哪些事情。

- 可以**窃取Cookie信息**。恶意JavaScript可以通过"document.cookie"获取Cookie信息，然后通过XMLHttpRequest或者Fetch加上CORS功能将数据发送给恶意服务器；恶意服务器拿到用户的Cookie信息之后，就可以在其他电脑上模拟用户的登录，然后进行转账等操作。
- 可以**监听用户行为**。恶意JavaScript可以使用"addEventListener"接口来监听键盘事件，比如可以获取用户输入的信用卡等信息，将其发送到恶意服务器。黑客掌握了这些信息之后，又可以做很多违法的事情。
- 可以通过**修改DOM**伪造假的登录窗口，用来欺骗用户输入用户名和密码等信息。
- 还可以**在页面内生成浮窗广告**，这些广告会严重地影响用户体验。

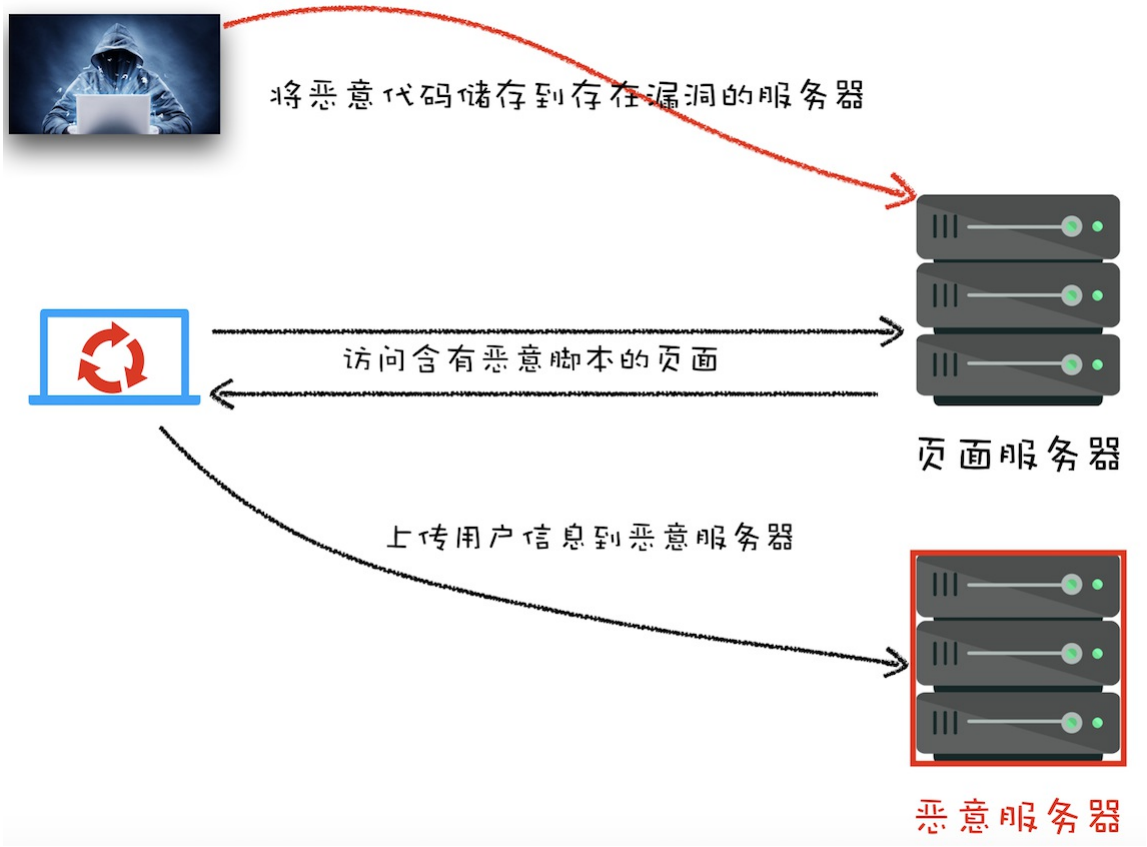
除了以上几种情况外，恶意脚本还能做很多其他的事情，这里就不一一介绍了。总之，如果让页面插入了恶意脚本，那么就相当于把我们页面的隐私数据和行为完全暴露给黑客了。

恶意脚本是怎么注入的

现在我们知道了页面中被注入恶意的JavaScript脚本是一件非常危险的事情，所以网站开发者会尽可能地避免页面中被注入恶意脚本。要想避免站点被注入恶意脚本，就要知道有哪些常见的注入方式。通常情况下，主要有**存储型XSS攻击**、**反射型XSS攻击**和**基于DOM的XSS攻击**三种方式来注入恶意脚本。

1. 存储型XSS攻击

我们先来看看存储型XSS攻击是怎么向HTML文件中注入恶意脚本的，你可以参考下图：



存储型XSS攻击

通过上图，我们可以看出存储型XSS攻击大致需要经过如下步骤：

- 首先黑客利用站点漏洞将一段恶意JavaScript代码提交到网站的数据库中；
- 然后用户向网站请求包含了恶意JavaScript脚本的页面；
- 当用户浏览该页面的时候，恶意脚本就会将用户的Cookie信息等数据上传到服务器。

下面我们来看个例子，2015年喜马拉雅就被曝出了存储型XSS漏洞。起因是在用户设置专辑名称时，服务器对关键字过滤不严格，比如可以将专辑名称设置为一段JavaScript，如下图所示：

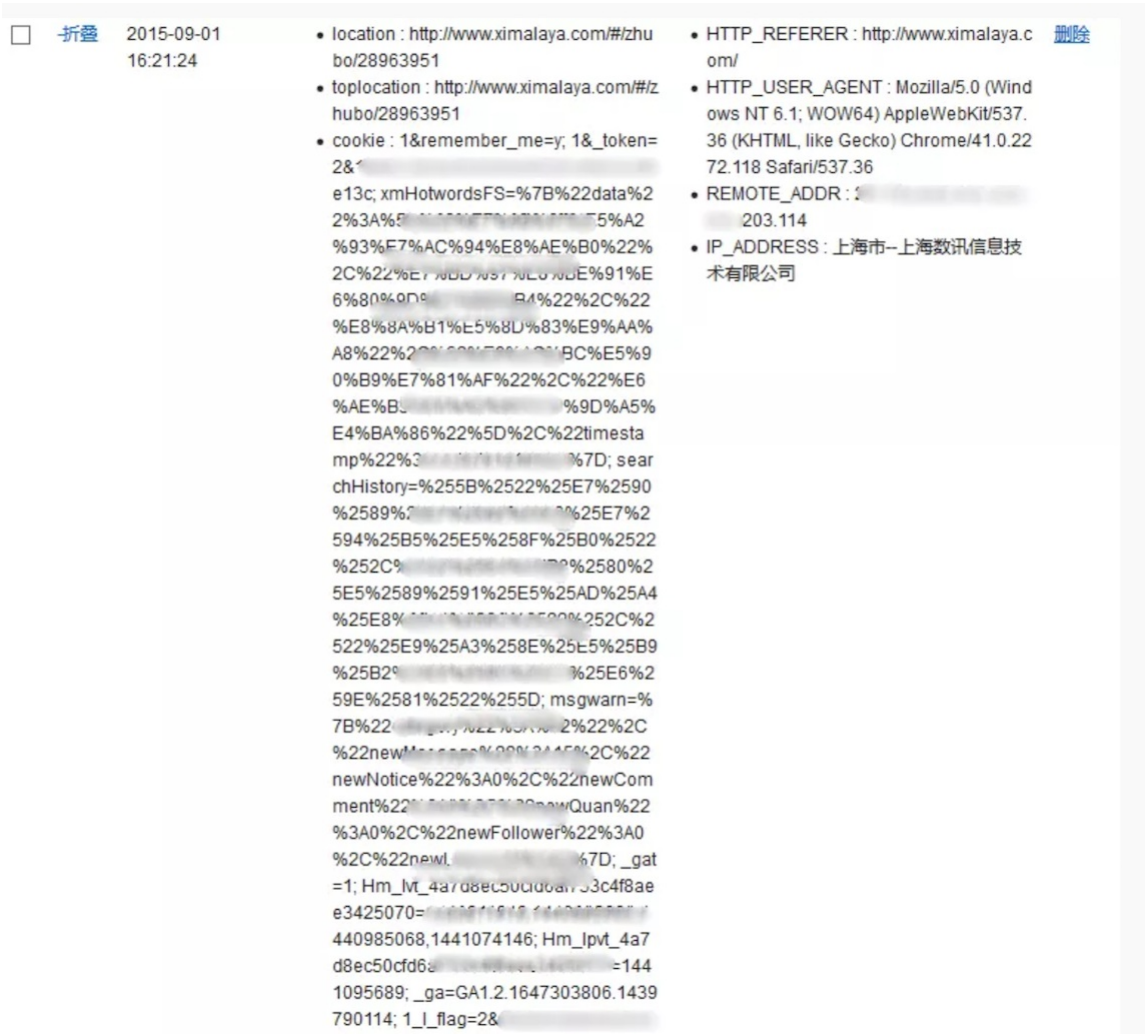
黑客将恶意代码存储到漏洞服务器上

当黑客将专辑名称设置为一段JavaScript代码并提交时，喜马拉雅的服务器会保存该段JavaScript代码到数据库中。然后当用户打开黑客设置的专辑时，这段代码就会在用户的页面里执行（如下图），这样就可以获取用户的Cookie等数据信息。

用户打开了含有恶意脚本的页面

当用户打开黑客设置的专辑页面时，服务器也会将这段恶意JavaScript代码返回给用户，因此这段恶意脚本就在用户的页面中执行了。

恶意脚本可以通过XMLHttpRequest或者Fetch将用户的Cookie数据上传到黑客的服务器，如下图所示：



将Cookie等数据上传到黑客服务器

黑客拿到了用户Cookie信息之后，就可以利用Cookie信息在其他机器上登录该用户的账号（如下图），并利用用户账号进行一些恶意操作。

黑客利用Cookie信息登录用户账户

以上就是存储型XSS攻击的一个典型案例，这是乌云网在2015年曝出来的，虽然乌云网由于某些原因被关停了，但是你依然可以通过这个站点来查看乌云网的一些备份信息。

2. 反射型XSS攻击

在一个反射型XSS攻击过程中，恶意JavaScript脚本属于用户发送给网站请求中的一部分，随后网站又把恶意JavaScript脚本返回给用户。当恶意JavaScript脚本在用户页面中被执行时，黑客就可以利用该脚本做一些恶意操作。

这样讲有点抽象，下面我们结合一个简单的Node服务程序来看看什么是反射型XSS。首先我们使用Node来搭建一个简单的页面环境，搭建好的服务代码如下所示：

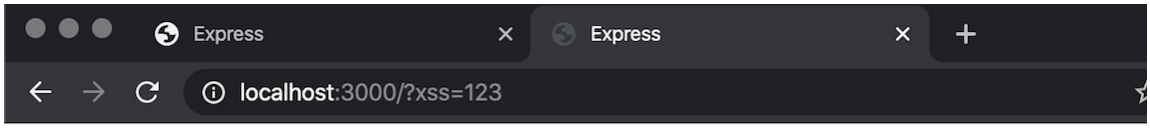
```
var express = require('express');
var router = express.Router();
```

```
/* GET home page. */
router.get('/', function(req, res, next) {
  res.render('index', { title: 'Express',xss:req.query.xss });
});


module.exports = router;
```

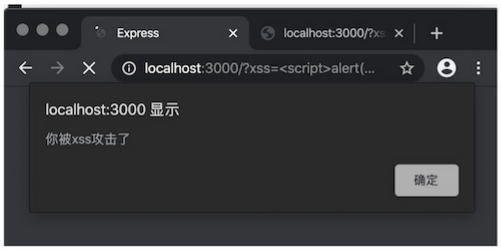
```
<!DOCTYPE html>
<html>
<head>
  <title><%= title %></title>
  <link rel='stylesheet' href='/stylesheets/style.css' />
</head>
<body>
  <h1><%= title %></h1>
  <p>Welcome to <%= title %></p>
  <div>
      <%- xss %>
  </div>
</body>
</html>
```

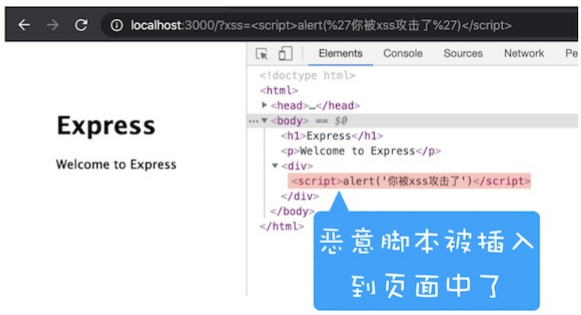
上面这两段代码，第一段是路由，第二段是视图，作用是将URL中xss参数的内容显示在页面。我们可以在本地演示下，比如打开`http://localhost:3000/?xss=123`这个链接，这样在页面中展示就是“123”了（如下图），是正常的，没有问题的。



正常打开页面

但当打开`http://localhost:3000/?xss=<script>alert('你被xss攻击了')</script>`这段URL时，其结果如下图所示：





反射型XSS攻击

通过这个操作，我们会发现用户将一段含有恶意代码的请求提交给Web服务器，Web服务器接收到请求时，又将恶意代码反射给了浏览器端，这就是反射型XSS攻击。在现实生活中，黑客经常会通过QQ群或者邮件等渠道诱导用户去点击这些恶意链接，所以对于一些链接我们一定要慎之又慎。

另外需要注意的是，**Web服务器不会存储反射型XSS攻击的恶意脚本，这是和存储型XSS攻击不同的地方**。

3. 基于DOM的XSS攻击

基于DOM的XSS攻击是不牵涉到页面Web服务器的。具体来讲，黑客通过各种手段将恶意脚本注入用户的页面中，比如通过网络劫持在页面传输过程中修改HTML页面的内容，这种劫持类型很多，有通过WiFi路由器劫持的，有通过本地恶意软件来劫持的，它们的共同点是在Web资源传输过程或者在用户使用页面的过程中修改Web页面的数据。

如何阻止XSS攻击

我们知道存储型XSS攻击和反射型XSS攻击都是需要经过Web服务器来处理的，因此可以认为这两种类型的漏洞是服务端的安全漏洞。而基于DOM的XSS攻击全部都是在浏览器端完成的，因此基于DOM的XSS攻击是属于前端的安全漏洞。

但无论是何种类型的XSS攻击，它们都有一个共同点，那就是首先往浏览器中注入恶意脚本，然后再通过恶意脚本将用户信息发送至黑客部署的恶意服务器上。

所以要阻止XSS攻击，我们可以通过阻止恶意JavaScript脚本的注入和恶意消息的发送来实现。

接下来我们就来看看一些常用的阻止XSS攻击的策略。

1. 服务器对输入脚本进行过滤或转码

不管是反射型还是存储型XSS攻击，我们都可以在服务器端将一些关键的字符进行转码，比如最典型的：

```
code:<script>alert('你被xss攻击了')</script>
```

这段代码过滤后，只留下了：

```
code:
```

这样，当用户再次请求该页面时，由于<script>标签的内容都被过滤了，所以这段脚本在客户端是不可能被执行的。

除了过滤之外，服务器还可以对这些内容进行转码，还是上面那段代码，经过转码之后，效果如下所示：

```
code:&lt;script&gt;alert(&#39;你被xss攻击了&#39;)&lt;/script&gt;
```

经过转码之后的内容，如<script>标签被转换为<script>，因此即使这段脚本返回给页面，页面也不会执行这段脚本。

2. 充分利用CSP

虽然在服务器端执行过滤或者转码可以阻止 XSS 攻击的发生，但完全依靠服务器端依然是不够的，我们还需要把CSP等策略充分地利用起来，以降低 XSS攻击带来的风险和后果。

实施严格的CSP可以有效地防范XSS攻击，具体来讲CSP有如下几个功能：

- 限制加载其他域下的资源文件，这样即使黑客插入了一个JavaScript文件，这个JavaScript文件也是无法被加载的；
- 禁止向第三方域提交数据，这样用户数据也不会外泄；
- 禁止执行内联脚本和未授权的脚本；
- 还提供了上报机制，这样可以帮助我们尽快发现有哪些XSS攻击，以便尽快修复问题。

因此，利用好CSP能够有效降低XSS攻击的概率。

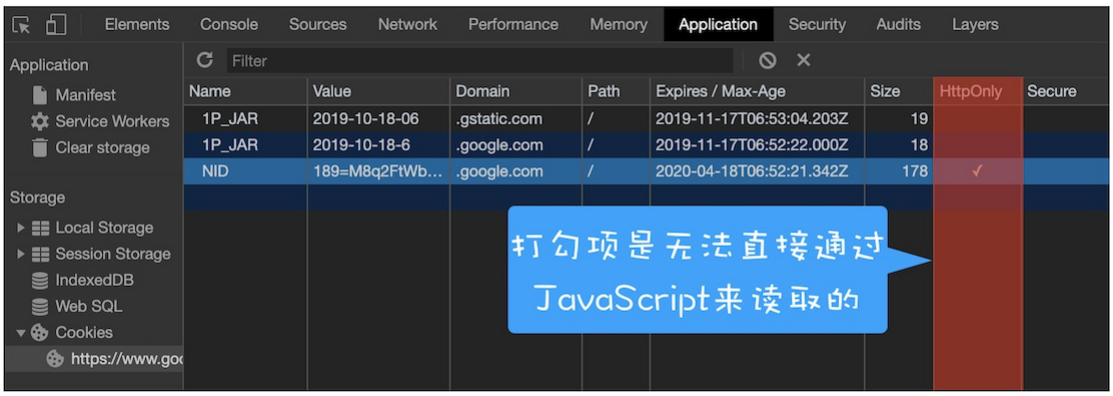
3. 使用HttpOnly属性

由于很多XSS攻击都是来盗用Cookie的，因此还可以通过使用HttpOnly属性来保护我们Cookie的安全。

通常服务器可以将某些Cookie设置为HttpOnly标志，HttpOnly是服务器通过HTTP响应头来设置的，下面是打开Google时，HTTP响应头中的一段：

```
set-cookie: NID=189=M8q2FtWbsR8RlcldPVt7qkrqR38LmFY9jUxkKo3-4Bi6Qu_ocNOat7nkYZUTzolHjFnwBw0izgsATSI7TZyiiiaV94qGh-BzEYsNVa7TZmjAYTxYTOM9L_-0CN9ipL6cXi8l6-z41asXtm2uEwcOC5oh9djkff(
```

我们可以看到，set-cookie属性值最后使用了HttpOnly来标记该Cookie。顾名思义，使用HttpOnly标记的Cookie只能使用在HTTP请求过程中，所以无法通过JavaScript来读取这段Cookie。我们还可以通过Chrome开发者工具来查看哪些Cookie被标记了HttpOnly，如下图：



HttpOnly演示

从图中可以看出，NID这个Cookie的HttpOlny属性是被勾选上的，所以NID的内容是无法通过document.cookie是来读取的。

由于JavaScript无法读取设置了HttpOnly的Cookie数据，所以即使页面被注入了恶意JavaScript脚本，也是无法获取到设置了HttpOnly的数据。因此一些比较重要的数据我们建议设置HttpOnly标志。

总结

好了，今天我们就介绍到这里，下面我来总结下本文的主要内容。

XSS攻击就是黑客往页面中注入恶意脚本，然后将页面的一些重要数据上传到恶意服务器。常见的三种XSS攻击模式是存储型XSS攻击、反射型XSS攻击和基于DOM的XSS攻击。

这三种攻击方式的共同点是都需要往用户的页面中注入恶意脚本，然后再通过恶意脚本将用户数据上传到黑客的恶意服务器上。而三者的不同点在于注入的方式不一样，有通过服务器漏洞来进行注入的，还有在客户端直接注入的。

针对这些XSS攻击，主要有三种防范策略，第一种是通过服务器对输入的内容进行过滤或者转码，第二种是充分利用好CSP，第三种是使用HttpOnly来保护重要的Cookie信息。

当然除了以上策略之外，我们还可以通过添加验证码防止脚本冒充用户提交危险操作。而对于一些不受信任的输入，还可以限制其输入长度，这样可以增大XSS攻击的难度。

思考时间

今天留给你的思考题是：你认为前端开发者对XSS攻击应该负多大责任？

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

通过上篇文章的介绍，我们知道了同源策略可以隔离各个站点之间的DOM交互、页面数据和网络通信，虽然严格的同源策略会带来更多的安全，但是也束缚了Web。这就需要在安全和自由之间找到一个平衡点，所以我们默认页面中可以引用任意第三方资源，然后又引入CSP策略来加以限制；默认XMLHttpRequest和Fetch不能跨站请求资源，然后又通过CORS策略来支持其跨域。

不过支持页面中的第三方资源引用和CORS也带来了很多安全问题，其中最典型的就是XSS攻击。

什么是XSS攻击

XSS全称是Cross Site Scripting，为了与“CSS”区分开来，故简称XSS，翻译过来就是“跨站脚本”。XSS攻击是指黑客往HTML文件中或者DOM中注入恶意脚本，从而在用户浏览页面时利用注入的恶意脚本对用户实施攻击的一种手段。

最开始的时候，这种攻击是通过跨域来实现的，所以叫“跨域脚本”。但是发展到现在，往HTML文件中注入恶意代码的方式越来越多了，所以是否跨域注入脚本已经不是唯一的注入手段了，但是XSS这个名字却一直保留至今。

当页面被注入了恶意JavaScript脚本时，浏览器无法区分这些脚本是被恶意注入的还是正常的页面内容，所以恶意注入JavaScript脚本也拥有所有的脚本权限。下面我们就来看看，如果页面被注入了恶意JavaScript脚本，恶意脚本都能做哪些事情。

- 可以**窃取Cookie信息**。恶意JavaScript可以通过“document.cookie”获取Cookie信息，然后通过XMLHttpRequest或者Fetch加上CORS功能将数据发送给恶意服务器；恶意服务器拿到用户的Cookie信息之后，就可以在其他电脑上模拟用户的登录，然后进行转账等操作。
- 可以**监听用户行为**。恶意JavaScript可以使用“addEventListener”接口来监听键盘事件，比如可以获取用户输入的信用卡等信息，将其发送到恶意服务器。黑客掌握了这些信息之后，又可以做很多违法的事情。
- 可以通过**修改DOM**伪造假的登录窗口，用来欺骗用户输入用户名和密码等信息。
- 还可以**在页面内生成浮窗广告**，这些广告会严重地影响用户体验。

除了以上几种情况外，恶意脚本还能做很多其他的事情，这里就不一一介绍了。总之，如果让页面插入了恶意脚本，那么就相当于把我们页面的隐私数据和行为完全暴露给黑客了。

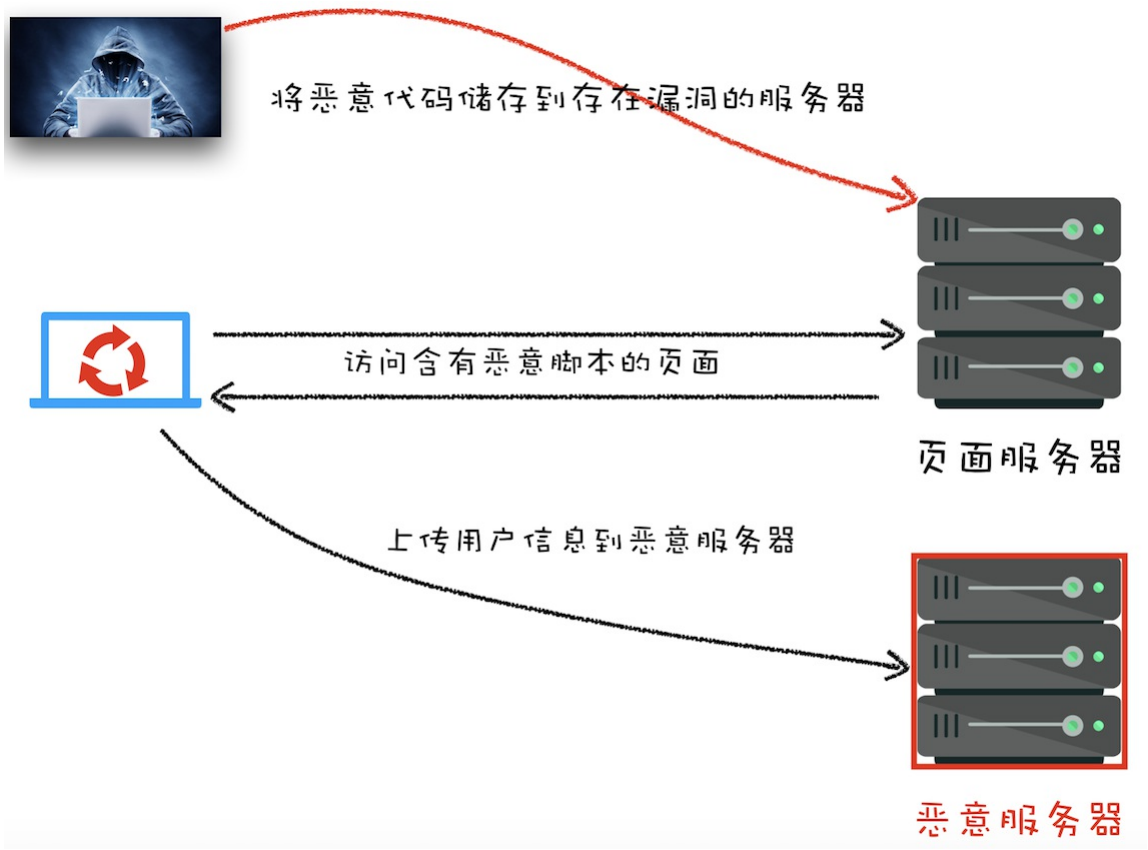
恶意脚本是怎么注入的

现在我们知道了页面中被注入恶意的JavaScript脚本是一件非常危险的事情，所以网站开发者会尽可能地避免页面中被注入恶意脚本。要想避免站点被注入恶意脚本，就要知道有哪些常见的注入方式。通

常情况下，主要有**存储型XSS攻击**、**反射型XSS攻击**和**基于DOM的XSS攻击**三种方式来注入恶意脚本。

1. 存储型XSS攻击

我们先来看看存储型XSS攻击是怎么向HTML文件中注入恶意脚本的，你可以参考下图：



存储型XSS攻击

通过上图，我们可以看出存储型XSS攻击大致需要经过如下步骤：

- 首先黑客利用站点漏洞将一段恶意JavaScript代码提交到网站的数据库中；
- 然后用户向网站请求包含了恶意JavaScript脚本的页面；
- 当用户浏览该页面的时候，恶意脚本就会将用户的Cookie信息等数据上传到服务器。

下面我们来看个例子，2015年喜马拉雅就被曝出了存储型XSS漏洞。起因是在用户设置专辑名称时，服务器对关键字过滤不严格，比如可以将专辑名称设置为一段JavaScript，如下图所示：

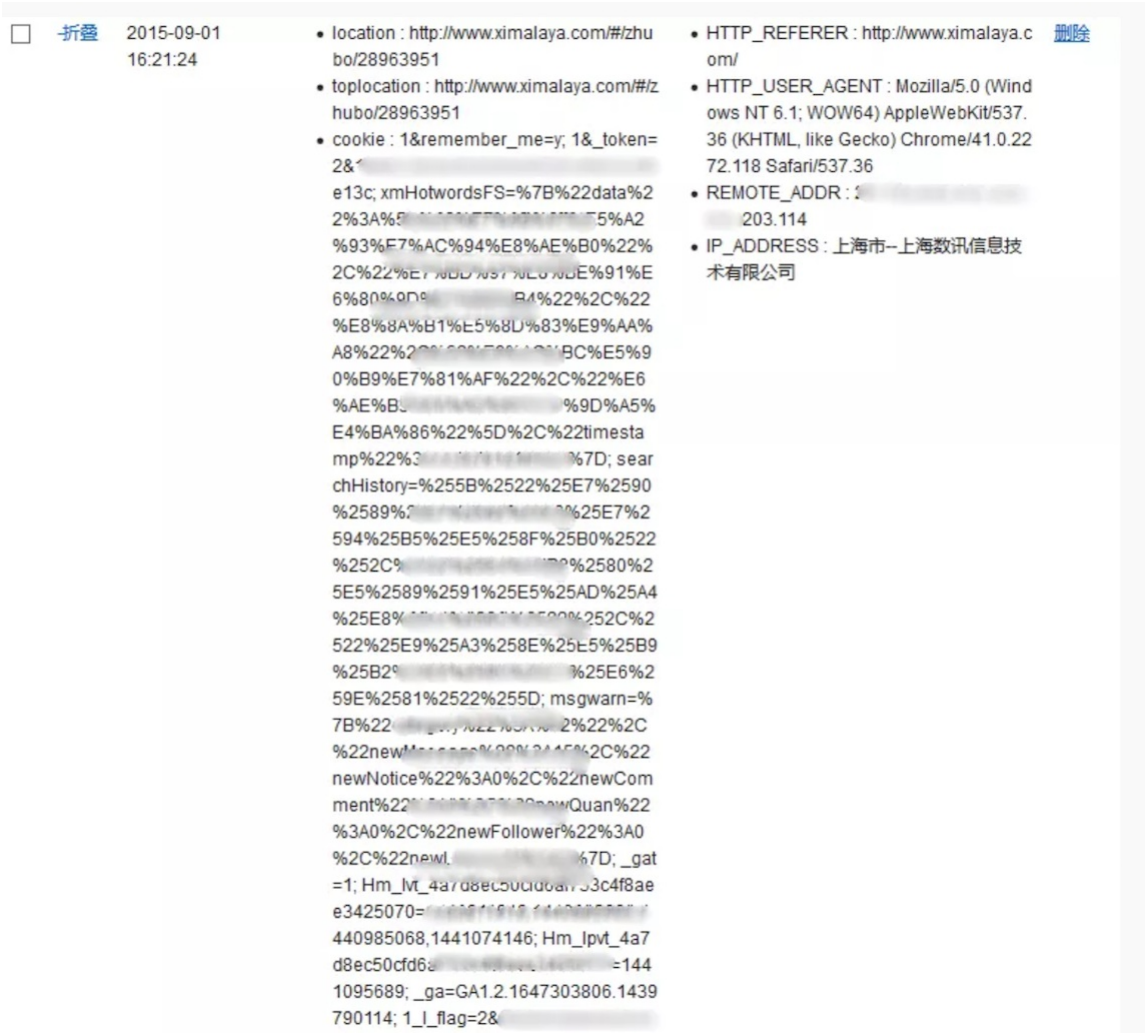
黑客将恶意代码存储到漏洞服务器上

当黑客将专辑名称设置为一段JavaScript代码并提交时，喜马拉雅的服务器会保存该段JavaScript代码到数据库中。然后当用户打开黑客设置的专辑时，这段代码就会在用户的页面里执行（如下图），这样就可以获取用户的Cookie等数据信息。

用户打开了含有恶意脚本的页面

当用户打开黑客设置的专辑页面时，服务器也会将这段恶意JavaScript代码返回给用户，因此这段恶意脚本就在用户的页面中执行了。

恶意脚本可以通过XMLHttpRequest或者Fetch将用户的Cookie数据上传到黑客的服务器，如下图所示：



将Cookie等数据上传到黑客服务器

黑客拿到了用户Cookie信息之后，就可以利用Cookie信息在其他机器上登录该用户的账号（如下图），并利用用户账号进行一些恶意操作。

黑客利用Cookie信息登录用户账户

以上就是存储型XSS攻击的一个典型案例，这是乌云网在2015年曝出来的，虽然乌云网由于某些原因被关停了，但是你依然可以通过这个站点来查看乌云网的一些备份信息。

2. 反射型XSS攻击

在一个反射型XSS攻击过程中，恶意JavaScript脚本属于用户发送给网站请求中的一部分，随后网站又把恶意JavaScript脚本返回给用户。当恶意JavaScript脚本在用户页面中被执行时，黑客就可以利用该脚本做一些恶意操作。

这样讲有点抽象，下面我们结合一个简单的Node服务程序来看看什么是反射型XSS。首先我们使用Node来搭建一个简单的页面环境，搭建好的服务代码如下所示：

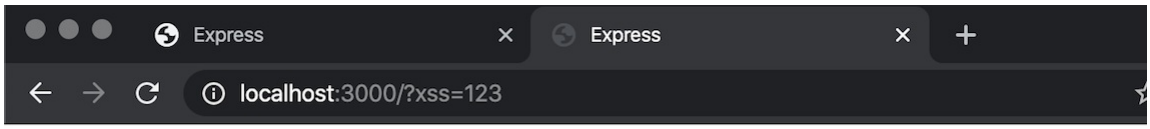
```
var express = require('express');
var router = express.Router();


/* GET home page. */
router.get('/', function(req, res, next) {
  res.render('index', { title: 'Express',xss:req.query.xss });
});
```

```
module.exports = router;
```

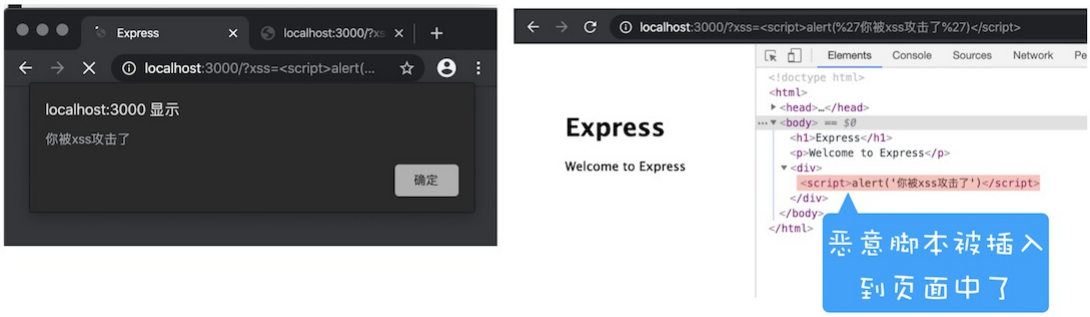
```
<!DOCTYPE html>
<html>
<head>
  <title><%= title %></title>
  <link rel='stylesheet' href='/stylesheets/style.css' />
</head>
<body>
  <h1><%= title %></h1>
  <p>Welcome to <%= title %></p>
  <div>
      <%- xss %>
  </div>
</body>
</html>
```

上面这两段代码，第一段是路由，第二段是视图，作用是将URL中xss参数的内容显示在页面。我们可以在本地演示下，比如打开`http://localhost:3000/?xss=123`这个链接，这样在页面中展示就是“123”了（如下图），是正常的，没有问题的。



正常打开页面

但当打开`http://localhost:3000/?xss=<script>alert('你被xss攻击了')</script>`这段URL时，其结果如下图所示：



反射型XSS攻击

通过这个操作，我们会发现用户将一段含有恶意代码的请求提交给Web服务器，Web服务器接收到请求时，又将恶意代码反射给了浏览器端，这就是反射型XSS攻击。在现实生活中，黑客经常会通过QQ群或者邮件等渠道诱导用户去点击这些恶意链接，所以对于一些链接我们一定要慎之又慎。

另外需要注意的是，**Web服务器不会存储反射型XSS攻击的恶意脚本，这是和存储型XSS攻击不同的地方**。

3. 基于DOM的XSS攻击

基于DOM的XSS攻击是不牵涉到页面Web服务器的。具体来讲，黑客通过各种手段将恶意脚本注入用户的页面中，比如通过网络劫持在页面传输过程中修改HTML页面的内容，这种劫持类型很多，有通过WiFi路由器劫持的，有通过本地恶意软件来劫持的，它们的共同点是在Web资源传输过程或者在用户使用页面的过程中修改Web页面的数据。

如何阻止XSS攻击

我们知道存储型XSS攻击和反射型XSS攻击都是需要经过Web服务器来处理的，因此可以认为这两种类型的漏洞是服务端的安全漏洞。而基于DOM的XSS攻击全部都是在浏览器端完成的，因此基于DOM的XSS攻击是属于前端的安全漏洞。

但无论是何种类型的XSS攻击，它们都有一个共同点，那就是首先往浏览器中注入恶意脚本，然后再通过恶意脚本将用户信息发送至黑客部署的恶意服务器上。

所以要阻止XSS攻击，我们可以通过阻止恶意JavaScript脚本的注入和恶意消息的发送来实现。

接下来我们就来看看一些常用的阻止XSS攻击的策略。

1. 服务器对输入脚本进行过滤或转码

不管是反射型还是存储型XSS攻击，我们都可以在服务器端将一些关键的字符进行转码，比如最典型的：

```
code:<script>alert('你被xss攻击了')</script>
```

这段代码过滤后，只留下了：

```
code:
```

这样，当用户再次请求该页面时，由于`<script>`标签的内容都被过滤了，所以这段脚本在客户端是不可能被执行的。

除了过滤之外，服务器还可以对这些内容进行转码，还是上面那段代码，经过转码之后，效果如下所示：

```
code:&lt;script&gt;alert(&#39;你被xss攻击了&#39;)&lt;/script&gt;
```

经过转码之后的内容，如`<script>`标签被转换为`<script>`，因此即使这段脚本返回给页面，页面也不会执行这段脚本。

2. 充分利用CSP

虽然在服务器端执行过滤或者转码可以阻止 XSS 攻击的发生，但完全依靠服务器端依然是不够的，我们还需要把CSP等策略充分地利用起来，以降低 XSS攻击带来的风险和后果。

实施严格的CSP可以有效地防范XSS攻击，具体来讲CSP有如下几个功能：

- 限制加载其他域下的资源文件，这样即使黑客插入了一个JavaScript文件，这个JavaScript文件也是无法被加载的；
- 禁止向第三方域提交数据，这样用户数据也不会外泄；
- 禁止执行内联脚本和未授权的脚本；
- 还提供了上报机制，这样可以帮助我们尽快发现有哪些XSS攻击，以便尽快修复问题。

因此，利用好CSP能够有效降低XSS攻击的概率。

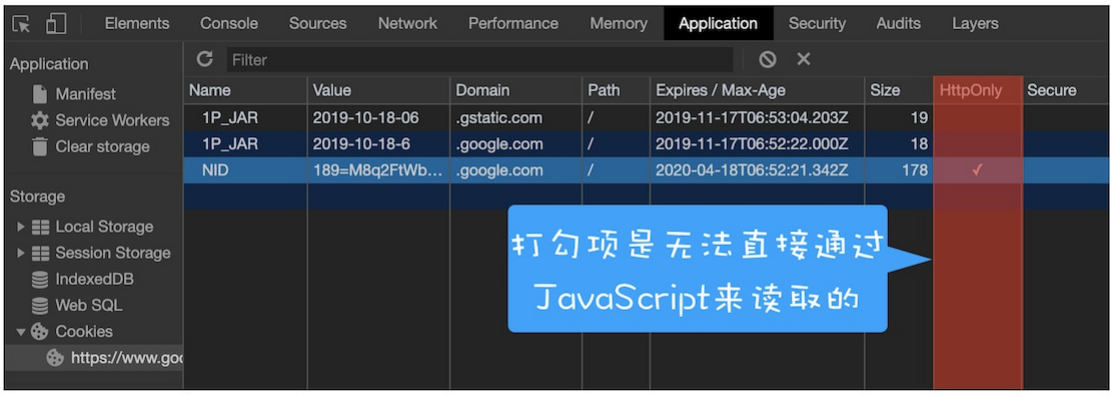
3. 使用HttpOnly属性

由于很多XSS攻击都是来盗用Cookie的，因此还可以通过使用HttpOnly属性来保护我们Cookie的安全。

通常服务器可以将某些Cookie设置为HttpOnly标志，HttpOnly是服务器通过HTTP响应头来设置的，下面是打开Google时，HTTP响应头中的一段：

```
set-cookie: NID=189=M8q2FtWbsR8RlcldPVt7qkrqR38LmFY9jUxkKo3-4Bi6Qu_ocNOat7nkYZUTzolHjFnwBw0izgsATSI7TZyiiiaV94qGh-BzEYsNVa7TZmjAYTxYTOM9L_-0CN9ipL6cXi8l6-z41asXtm2uEwcOC5oh9djkff(
```

我们可以看到，set-cookie属性值最后使用了HttpOnly来标记该Cookie。顾名思义，使用HttpOnly标记的Cookie只能使用在HTTP请求过程中，所以无法通过JavaScript来读取这段Cookie。我们还可以通过Chrome开发者工具来查看哪些Cookie被标记了HttpOnly，如下图：



HttpOnly演示

从图中可以看出，NID这个Cookie的HttpOlny属性是被勾选上的，所以NID的内容是无法通过document.cookie是来读取的。

由于JavaScript无法读取设置了HttpOnly的Cookie数据，所以即使页面被注入了恶意JavaScript脚本，也是无法获取到设置了HttpOnly的数据。因此一些比较重要的数据我们建议设置HttpOnly标志。

总结

好了，今天我们就介绍到这里，下面我来总结下本文的主要内容。

XSS攻击就是黑客往页面中注入恶意脚本，然后将页面的一些重要数据上传到恶意服务器。常见的三种XSS攻击模式是存储型XSS攻击、反射型XSS攻击和基于DOM的XSS攻击。

这三种攻击方式的共同点是都需要往用户的页面中注入恶意脚本，然后再通过恶意脚本将用户数据上传到黑客的恶意服务器上。而三者的不同点在于注入的方式不一样，有通过服务器漏洞来进行注入的，还有在客户端直接注入的。

针对这些XSS攻击，主要有三种防范策略，第一种是通过服务器对输入的内容进行过滤或者转码，第二种是充分利用好CSP，第三种是使用HttpOnly来保护重要的Cookie信息。

当然除了以上策略之外，我们还可以通过添加验证码防止脚本冒充用户提交危险操作。而对于一些不受信任的输入，还可以限制其输入长度，这样可以增大XSS攻击的难度。

思考时间

今天留给你的思考题是：你认为前端开发者对XSS攻击应该负多大责任？

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

通过上篇文章的介绍，我们知道了同源策略可以隔离各个站点之间的DOM交互、页面数据和网络通信，虽然严格的同源策略会带来更多的安全，但是也束缚了Web。这就需要在安全和自由之间找到一个平衡点，所以我们默认页面中可以引用任意第三方资源，然后又引入CSP策略来加以限制；默认XMLHttpRequest和Fetch不能跨站请求资源，然后又通过CORS策略来支持其跨域。

不过支持页面中的第三方资源引用和CORS也带来了很多安全问题，其中最典型的就是XSS攻击。

什么是XSS攻击

XSS全称是Cross Site Scripting，为了与“CSS”区分开来，故简称XSS，翻译过来就是“跨站脚本”。XSS攻击是指黑客往HTML文件中或者DOM中注入恶意脚本，从而在用户浏览页面时利用注入的恶意脚本对用户实施攻击的一种手段。

最开始的时候，这种攻击是通过跨域来实现的，所以叫“跨域脚本”。但是发展到现在，往HTML文件中注入恶意代码的方式越来越多了，所以是否跨域注入脚本已经不是唯一的注入手段了，但是XSS这个名字却一直保留至今。

当页面被注入了恶意JavaScript脚本时，浏览器无法区分这些脚本是被恶意注入的还是正常的页面内容，所以恶意注入JavaScript脚本也拥有所有的脚本权限。下面我们就来看看，如果页面被注入了恶意JavaScript脚本，恶意脚本都能做哪些事情。

- 可以**窃取Cookie信息**。恶意JavaScript可以通过“document.cookie”获取Cookie信息，然后通过XMLHttpRequest或者Fetch加上CORS功能将数据发送给恶意服务器；恶意服务器拿到用户的Cookie信息之后，就可以在其他电脑上模拟用户的登录，然后进行转账等操作。
- 可以**监听用户行为**。恶意JavaScript可以使用“addEventListener”接口来监听键盘事件，比如可以获取用户输入的信用卡等信息，将其发送到恶意服务器。黑客掌握了这些信息之后，又可以做很多违法的事情。
- 可以通过**修改DOM**伪造假的登录窗口，用来欺骗用户输入用户名和密码等信息。
- 还可以**在页面内生成浮窗广告**，这些广告会严重地影响用户体验。

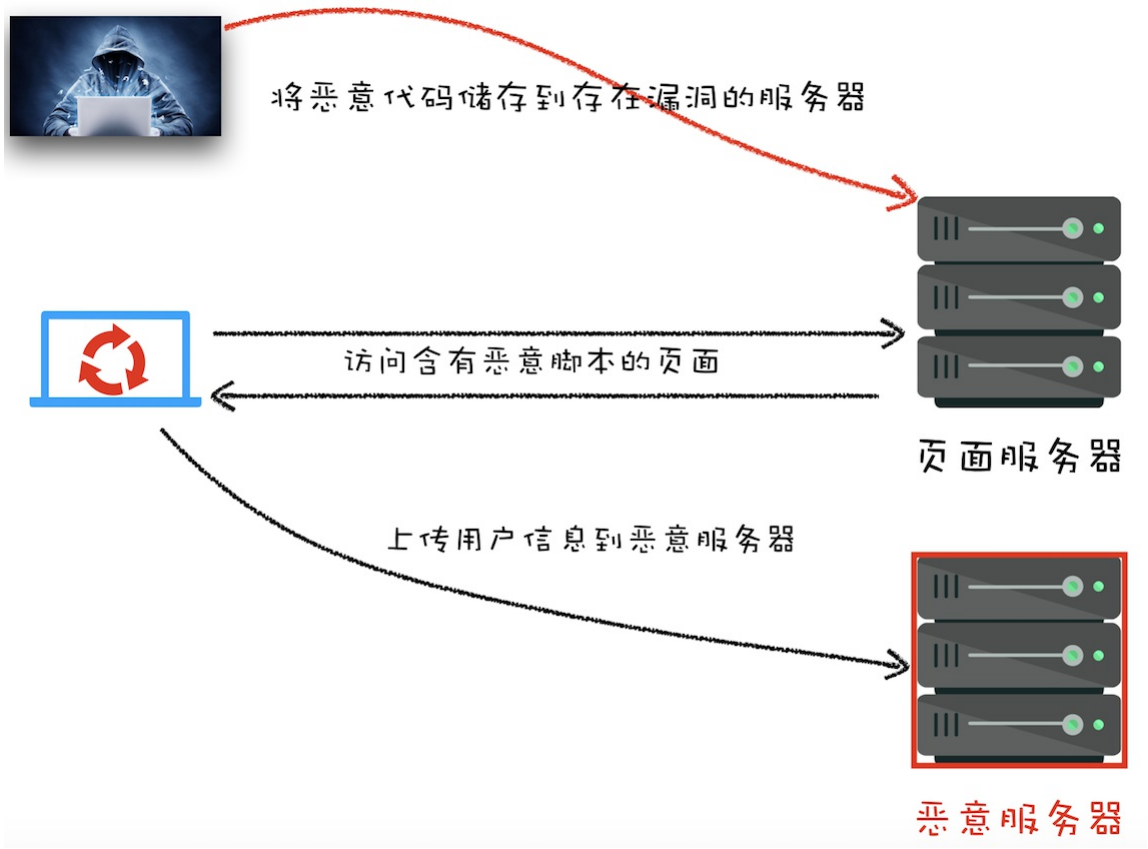
除了以上几种情况外，恶意脚本还能做很多其他的事情，这里就不一一介绍了。总之，如果让页面插入了恶意脚本，那么就相当于把我们页面的隐私数据和行为完全暴露给黑客了。

恶意脚本是怎么注入的

现在我们知道了页面中被注入恶意的JavaScript脚本是一件非常危险的事情，所以网站开发者会尽可能地避免页面中被注入恶意脚本。要想避免站点被注入恶意脚本，就要知道有哪些常见的注入方式。通常情况下，主要有**存储型XSS攻击**、**反射型XSS攻击**和**基于DOM的XSS攻击**三种方式来注入恶意脚本。

1. 存储型XSS攻击

我们先来看看存储型XSS攻击是怎么向HTML文件中注入恶意脚本的，你可以参考下图：



存储型XSS攻击

通过上图，我们可以看出存储型XSS攻击大致需要经过如下步骤：

- 首先黑客利用站点漏洞将一段恶意JavaScript代码提交到网站的数据库中；
- 然后用户向网站请求包含了恶意JavaScript脚本的页面；
- 当用户浏览该页面的时候，恶意脚本就会将用户的Cookie信息等数据上传到服务器。

下面我们来看个例子，2015年喜马拉雅就被曝出了存储型XSS漏洞。起因是在用户设置专辑名称时，服务器对关键字过滤不严格，比如可以将专辑名称设置为一段JavaScript，如下图所示：

黑客将恶意代码存储到漏洞服务器上

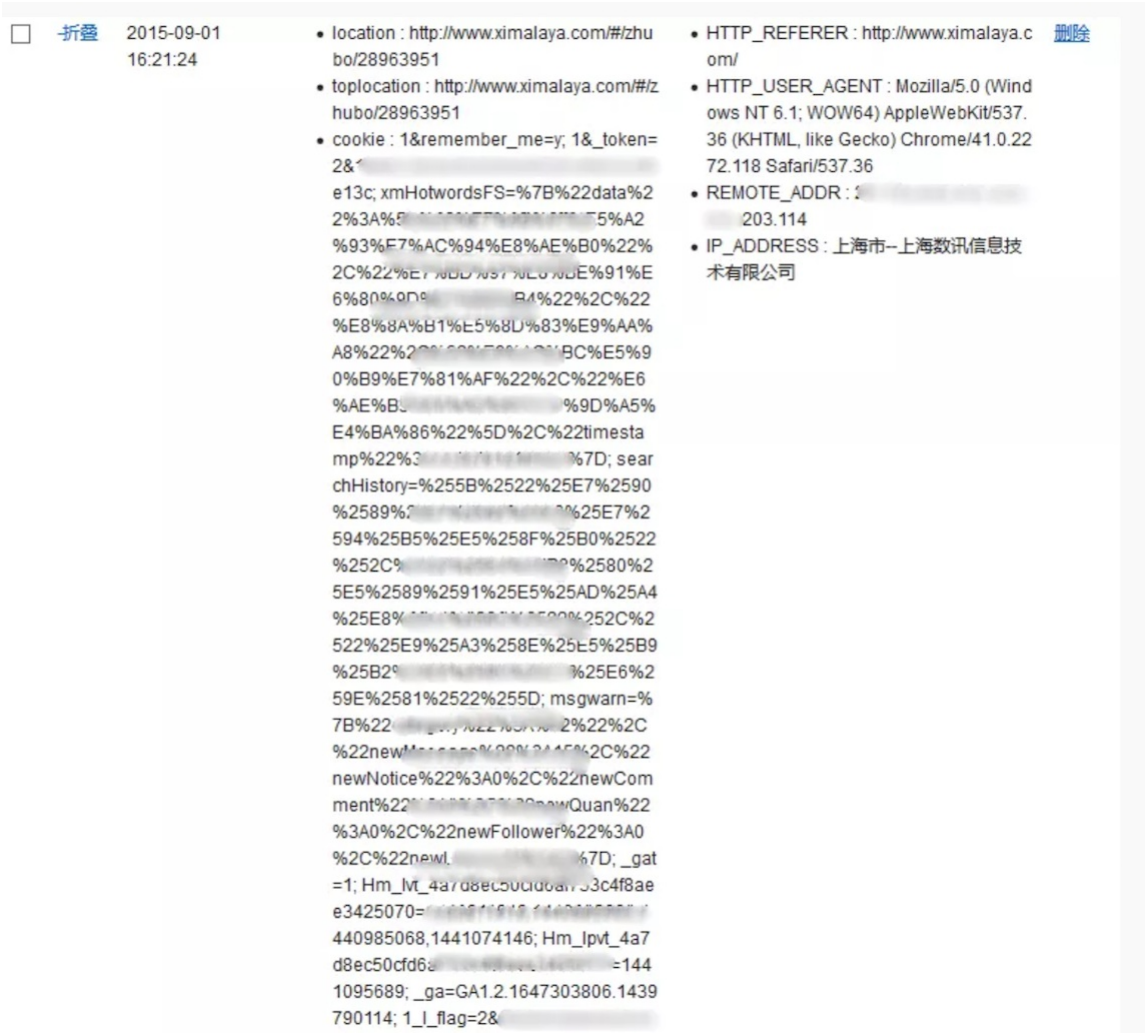
当黑客将专辑名称设置为一段JavaScript代码并提交时，喜马拉雅的服务器会保存该段JavaScript代码到数据库中。然后当用户打开黑客设置的专辑时，这段代码就会在用户的页面里执行（如下图），这样

就可以获取用户的Cookie等数据信息。

用户打开了含有恶意脚本的页面

当用户打开黑客设置的专辑页面时，服务器也会将这段恶意JavaScript代码返回给用户，因此这段恶意脚本就在用户的页面中执行了。

恶意脚本可以通过XMLHttpRequest或者Fetch将用户的Cookie数据上传到黑客的服务器，如下图所示：



将Cookie等数据上传到黑客服务器

黑客拿到了用户Cookie信息之后，就可以利用Cookie信息在其他机器上登录该用户的账号（如下图），并利用用户账号进行一些恶意操作。

黑客利用Cookie信息登录用户账户

以上就是存储型XSS攻击的一个典型案例，这是乌云网在2015年曝出来的，虽然乌云网由于某些原因被关停了，但是你依然可以通过这个站点来查看乌云网的一些备份信息。

2. 反射型XSS攻击

在一个反射型XSS攻击过程中，恶意JavaScript脚本属于用户发送给网站请求中的一部分，随后网站又把恶意JavaScript脚本返回给用户。当恶意JavaScript脚本在用户页面中被执行时，黑客就可以利用该脚本做一些恶意操作。

这样讲有点抽象，下面我们结合一个简单的Node服务程序来看看什么是反射型XSS。首先我们使用Node来搭建一个简单的页面环境，搭建好的服务代码如下所示：

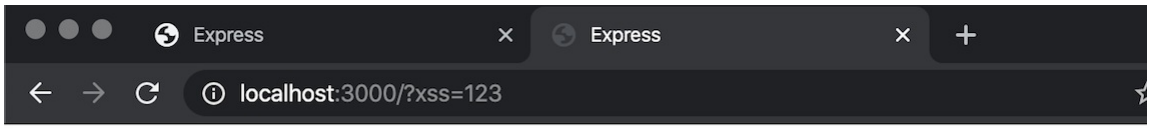
```
var express = require('express');
var router = express.Router();


/* GET home page. */
router.get('/', function(req, res, next) {
  res.render('index', { title: 'Express',xss:req.query.xss });
});
```

```
module.exports = router;
```

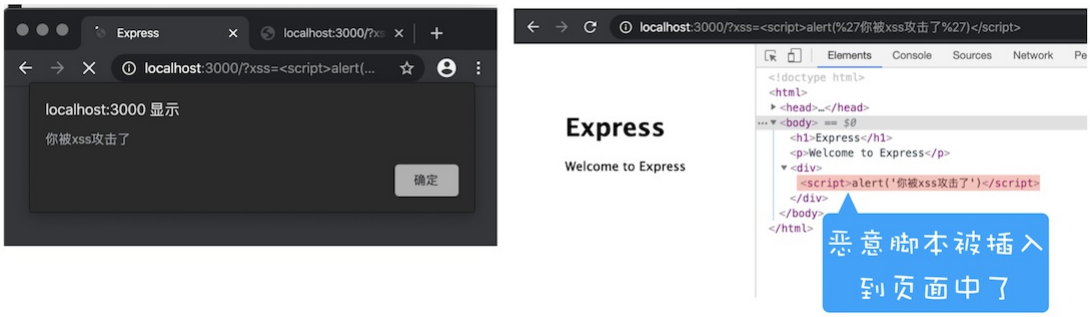
```
<!DOCTYPE html>
<html>
<head>
  <title><%= title %></title>
  <link rel='stylesheet' href='/stylesheets/style.css' />
</head>
<body>
  <h1><%= title %></h1>
  <p>Welcome to <%= title %></p>
  <div>
      <%- xss %>
  </div>
</body>
</html>
```

上面这两段代码，第一段是路由，第二段是视图，作用是将URL中xss参数的内容显示在页面。我们可以在本地演示下，比如打开`http://localhost:3000/?xss=123`这个链接，这样在页面中展示就是“123”了（如下图），是正常的，没有问题的。



正常打开页面

但当打开`http://localhost:3000/?xss=<script>alert('你被xss攻击了')</script>`这段URL时，其结果如下图所示：



反射型XSS攻击

通过这个操作，我们会发现用户将一段含有恶意代码的请求提交给Web服务器，Web服务器接收到请求时，又将恶意代码反射给了浏览器端，这就是反射型XSS攻击。在现实生活中，黑客经常会通过QQ群或者邮件等渠道诱导用户去点击这些恶意链接，所以对于一些链接我们一定要慎之又慎。

另外需要注意的是，**Web服务器不会存储反射型XSS攻击的恶意脚本，这是和存储型XSS攻击不同的地方**。

3. 基于DOM的XSS攻击

基于DOM的XSS攻击是不牵涉到页面Web服务器的。具体来讲，黑客通过各种手段将恶意脚本注入用户的页面中，比如通过网络劫持在页面传输过程中修改HTML页面的内容，这种劫持类型很多，有通过WiFi路由器劫持的，有通过本地恶意软件来劫持的，它们的共同点是在Web资源传输过程或者在用户使用页面的过程中修改Web页面的数据。

如何阻止XSS攻击

我们知道存储型XSS攻击和反射型XSS攻击都是需要经过Web服务器来处理的，因此可以认为这两种类型的漏洞是服务端的安全漏洞。而基于DOM的XSS攻击全部都是在浏览器端完成的，因此基于DOM的XSS攻击是属于前端的安全漏洞。

但无论是何种类型的XSS攻击，它们都有一个共同点，那就是首先往浏览器中注入恶意脚本，然后再通过恶意脚本将用户信息发送至黑客部署的恶意服务器上。

所以要阻止XSS攻击，我们可以通过阻止恶意JavaScript脚本的注入和恶意消息的发送来实现。

接下来我们就来看看一些常用的阻止XSS攻击的策略。

1. 服务器对输入脚本进行过滤或转码

不管是反射型还是存储型XSS攻击，我们都可以在服务器端将一些关键的字符进行转码，比如最典型的：

```
code:<script>alert('你被xss攻击了')</script>
```

这段代码过滤后，只留下了：

```
code:
```

这样，当用户再次请求该页面时，由于`<script>`标签的内容都被过滤了，所以这段脚本在客户端是不可能被执行的。

除了过滤之外，服务器还可以对这些内容进行转码，还是上面那段代码，经过转码之后，效果如下所示：

```
code:&lt;script&gt;alert(&#39;你被xss攻击了&#39;)&lt;/script&gt;
```

经过转码之后的内容，如`<script>`标签被转换为`<script>`，因此即使这段脚本返回给页面，页面也不会执行这段脚本。

2. 充分利用CSP

虽然在服务器端执行过滤或者转码可以阻止 XSS 攻击的发生，但完全依靠服务器端依然是不够的，我们还需要把CSP等策略充分地利用起来，以降低 XSS攻击带来的风险和后果。

实施严格的CSP可以有效地防范XSS攻击，具体来讲CSP有如下几个功能：

- 限制加载其他域下的资源文件，这样即使黑客插入了一个JavaScript文件，这个JavaScript文件也是无法被加载的；
- 禁止向第三方域提交数据，这样用户数据也不会外泄；
- 禁止执行内联脚本和未授权的脚本；
- 还提供了上报机制，这样可以帮助我们尽快发现有哪些XSS攻击，以便尽快修复问题。

因此，利用好CSP能够有效降低XSS攻击的概率。

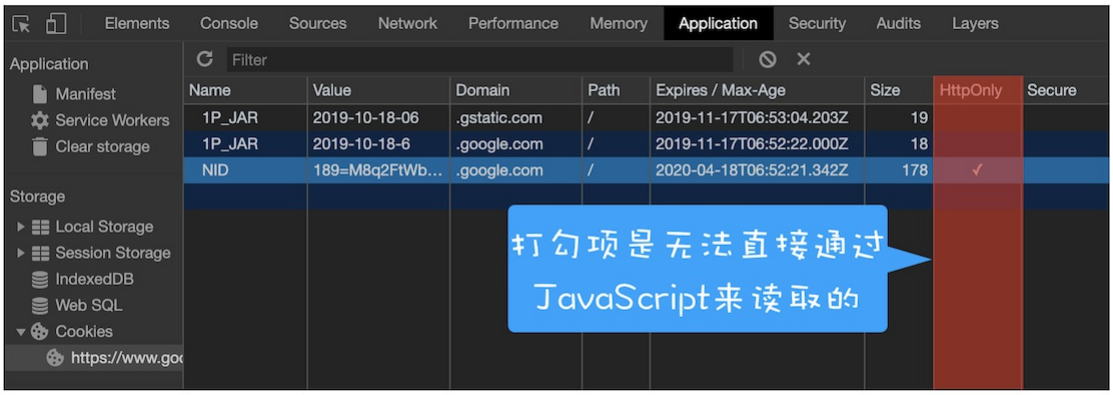
3. 使用HttpOnly属性

由于很多XSS攻击都是来盗用Cookie的，因此还可以通过使用HttpOnly属性来保护我们Cookie的安全。

通常服务器可以将某些Cookie设置为HttpOnly标志，HttpOnly是服务器通过HTTP响应头来设置的，下面是打开Google时，HTTP响应头中的一段：

```
set-cookie: NID=189=M8q2FtWbsR8RlcldPVt7qkrqR38LmFY9jUxkKo3-4Bi6Qu_ocNOat7nkYZUTzolHjFnwBw0izgsATSI7TZyiiiaV94qGh-BzEYsNVa7TZmjAYTxYTOM9L_-0CN9ipL6cXi8l6-z41asXtm2uEwcOC5oh9djkff(
```

我们可以看到，set-cookie属性值最后使用了HttpOnly来标记该Cookie。顾名思义，使用HttpOnly标记的Cookie只能使用在HTTP请求过程中，所以无法通过JavaScript来读取这段Cookie。我们还可以通过Chrome开发者工具来查看哪些Cookie被标记了HttpOnly，如下图：



HttpOnly演示

从图中可以看出，NID这个Cookie的HttpOlny属性是被勾选上的，所以NID的内容是无法通过document.cookie是来读取的。

由于JavaScript无法读取设置了HttpOnly的Cookie数据，所以即使页面被注入了恶意JavaScript脚本，也是无法获取到设置了HttpOnly的数据。因此一些比较重要的数据我们建议设置HttpOnly标志。

总结

好了，今天我们就介绍到这里，下面我来总结下本文的主要内容。

XSS攻击就是黑客往页面中注入恶意脚本，然后将页面的一些重要数据上传到恶意服务器。常见的三种XSS攻击模式是存储型XSS攻击、反射型XSS攻击和基于DOM的XSS攻击。

这三种攻击方式的共同点是都需要往用户的页面中注入恶意脚本，然后再通过恶意脚本将用户数据上传到黑客的恶意服务器上。而三者的不同点在于注入的方式不一样，有通过服务器漏洞来进行注入的，还有在客户端直接注入的。

针对这些XSS攻击，主要有三种防范策略，第一种是通过服务器对输入的内容进行过滤或者转码，第二种是充分利用好CSP，第三种是使用HttpOnly来保护重要的Cookie信息。

当然除了以上策略之外，我们还可以通过添加验证码防止脚本冒充用户提交危险操作。而对于一些不受信任的输入，还可以限制其输入长度，这样可以增大XSS攻击的难度。

思考时间

今天留给你的思考题是：你认为前端开发者对XSS攻击应该负多大责任？

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

通过上篇文章的介绍，我们知道了同源策略可以隔离各个站点之间的DOM交互、页面数据和网络通信，虽然严格的同源策略会带来更多的安全，但是也束缚了Web。这就需要在安全和自由之间找到一个平衡点，所以我们默认页面中可以引用任意第三方资源，然后又引入CSP策略来加以限制；默认XMLHttpRequest和Fetch不能跨站请求资源，然后又通过CORS策略来支持其跨域。

不过支持页面中的第三方资源引用和CORS也带来了很多安全问题，其中最典型的就是XSS攻击。

什么是XSS攻击

XSS全称是Cross Site Scripting，为了与“CSS”区分开来，故简称XSS，翻译过来就是“跨站脚本”。XSS攻击是指黑客往HTML文件中或者DOM中注入恶意脚本，从而在用户浏览页面时利用注入的恶意脚本对用户实施攻击的一种手段。

最开始的时候，这种攻击是通过跨域来实现的，所以叫“跨域脚本”。但是发展到现在，往HTML文件中注入恶意代码的方式越来越多了，所以是否跨域注入脚本已经不是唯一的注入手段了，但是XSS这个名字却一直保留至今。

当页面被注入了恶意JavaScript脚本时，浏览器无法区分这些脚本是被恶意注入的还是正常的页面内容，所以恶意注入JavaScript脚本也拥有所有的脚本权限。下面我们就来看看，如果页面被注入了恶意JavaScript脚本，恶意脚本都能做哪些事情。

- 可以**窃取Cookie信息**。恶意JavaScript可以通过“document.cookie”获取Cookie信息，然后通过XMLHttpRequest或者Fetch加上CORS功能将数据发送给恶意服务器；恶意服务器拿到用户的Cookie信息之后，就可以在其他电脑上模拟用户的登录，然后进行转账等操作。
- 可以**监听用户行为**。恶意JavaScript可以使用“addEventListener”接口来监听键盘事件，比如可以获取用户输入的信用卡等信息，将其发送到恶意服务器。黑客掌握了这些信息之后，又可以做很多违法的事情。
- 可以通过**修改DOM**伪造假的登录窗口，用来欺骗用户输入用户名和密码等信息。
- 还可以**在页面内生成浮窗广告**，这些广告会严重地影响用户体验。

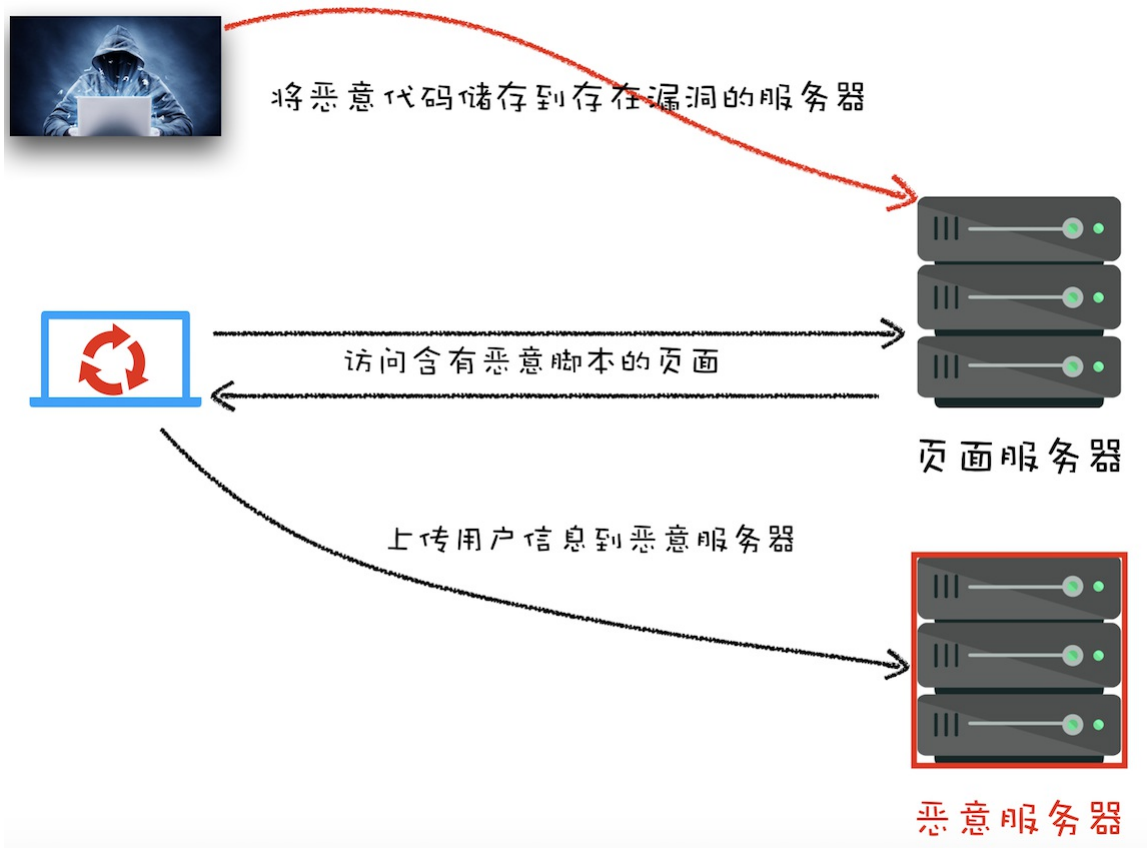
除了以上几种情况外，恶意脚本还能做很多其他的事情，这里就不一一介绍了。总之，如果让页面插入了恶意脚本，那么就相当于把我们页面的隐私数据和行为完全暴露给黑客了。

恶意脚本是怎么注入的

现在我们知道了页面中被注入恶意的JavaScript脚本是一件非常危险的事情，所以网站开发者会尽可能地避免页面中被注入恶意脚本。要想避免站点被注入恶意脚本，就要知道有哪些常见的注入方式。通常情况下，主要有**存储型XSS攻击**、**反射型XSS攻击**和**基于DOM的XSS攻击**三种方式来注入恶意脚本。

1. 存储型XSS攻击

我们先来看看存储型XSS攻击是怎么向HTML文件中注入恶意脚本的，你可以参考下图：



存储型XSS攻击

通过上图，我们可以看出存储型XSS攻击大致需要经过如下步骤：

- 首先黑客利用站点漏洞将一段恶意JavaScript代码提交到网站的数据库中；
- 然后用户向网站请求包含了恶意JavaScript脚本的页面；
- 当用户浏览该页面的时候，恶意脚本就会将用户的Cookie信息等数据上传到服务器。

下面我们来看个例子，2015年喜马拉雅就被曝出了存储型XSS漏洞。起因是在用户设置专辑名称时，服务器对关键字过滤不严格，比如可以将专辑名称设置为一段JavaScript，如下图所示：

黑客将恶意代码存储到漏洞服务器上

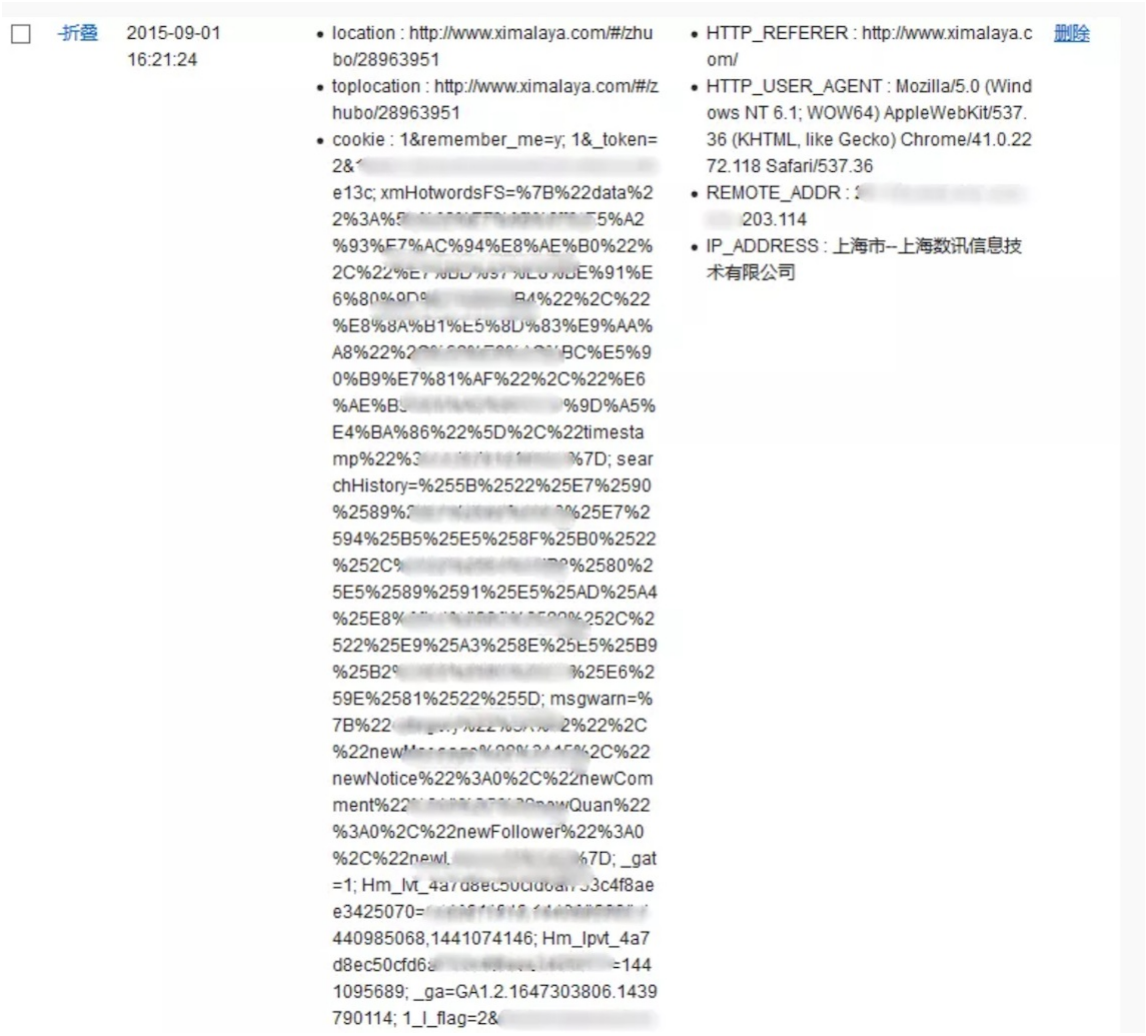
当黑客将专辑名称设置为一段JavaScript代码并提交时，喜马拉雅的服务器会保存该段JavaScript代码到数据库中。然后当用户打开黑客设置的专辑时，这段代码就会在用户的页面里执行（如下图），这样

就可以获取用户的Cookie等数据信息。

用户打开了含有恶意脚本的页面

当用户打开黑客设置的专辑页面时，服务器也会将这段恶意JavaScript代码返回给用户，因此这段恶意脚本就在用户的页面中执行了。

恶意脚本可以通过XMLHttpRequest或者Fetch将用户的Cookie数据上传到黑客的服务器，如下图所示：



将Cookie等数据上传到黑客服务器

黑客拿到了用户Cookie信息之后，就可以利用Cookie信息在其他机器上登录该用户的账号（如下图），并利用用户账号进行一些恶意操作。

黑客利用Cookie信息登录用户账户

以上就是存储型XSS攻击的一个典型案例，这是乌云网在2015年曝出来的，虽然乌云网由于某些原因被关停了，但是你依然可以通过这个站点来查看乌云网的一些备份信息。

2. 反射型XSS攻击

在一个反射型XSS攻击过程中，恶意JavaScript脚本属于用户发送给网站请求中的一部分，随后网站又把恶意JavaScript脚本返回给用户。当恶意JavaScript脚本在用户页面中被执行时，黑客就可以利用该脚本做一些恶意操作。

这样讲有点抽象，下面我们结合一个简单的Node服务程序来看看什么是反射型XSS。首先我们使用Node来搭建一个简单的页面环境，搭建好的服务代码如下所示：

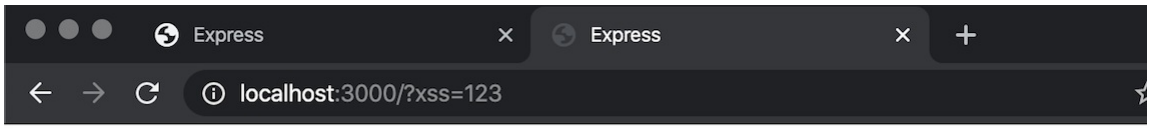
```
var express = require('express');
var router = express.Router();


/* GET home page. */
router.get('/', function(req, res, next) {
  res.render('index', { title: 'Express',xss:req.query.xss });
});
```

```
module.exports = router;
```

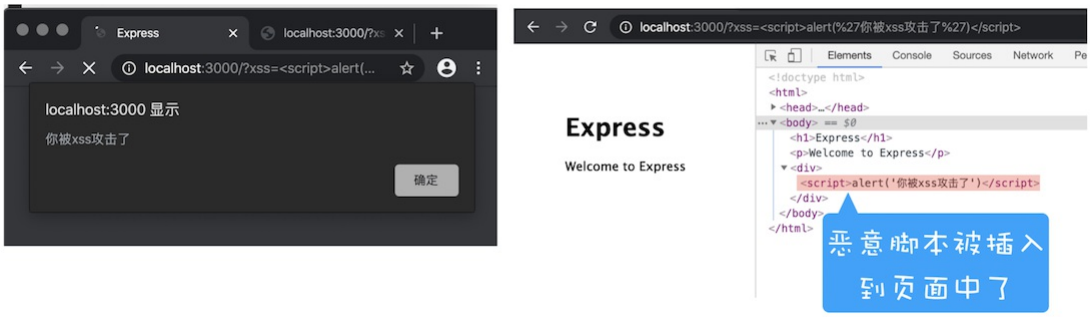
```
<!DOCTYPE html>
<html>
<head>
  <title><%= title %></title>
  <link rel='stylesheet' href='/stylesheets/style.css' />
</head>
<body>
  <h1><%= title %></h1>
  <p>Welcome to <%= title %></p>
  <div>
      <%- xss %>
  </div>
</body>
</html>
```

上面这两段代码，第一段是路由，第二段是视图，作用是将URL中xss参数的内容显示在页面。我们可以在本地演示下，比如打开`http://localhost:3000/?xss=123`这个链接，这样在页面中展示就是“123”了（如下图），是正常的，没有问题的。



正常打开页面

但当打开`http://localhost:3000/?xss=<script>alert('你被xss攻击了')</script>`这段URL时，其结果如下图所示：



反射型XSS攻击

通过这个操作，我们会发现用户将一段含有恶意代码的请求提交给Web服务器，Web服务器接收到请求时，又将恶意代码反射给了浏览器端，这就是反射型XSS攻击。在现实生活中，黑客经常会通过QQ群或者邮件等渠道诱导用户去点击这些恶意链接，所以对于一些链接我们一定要慎之又慎。

另外需要注意的是，**Web服务器不会存储反射型XSS攻击的恶意脚本，这是和存储型XSS攻击不同的地方**。

3. 基于DOM的XSS攻击

基于DOM的XSS攻击是不牵涉到页面Web服务器的。具体来讲，黑客通过各种手段将恶意脚本注入用户的页面中，比如通过网络劫持在页面传输过程中修改HTML页面的内容，这种劫持类型很多，有通过WiFi路由器劫持的，有通过本地恶意软件来劫持的，它们的共同点是在Web资源传输过程或者在用户使用页面的过程中修改Web页面的数据。

如何阻止XSS攻击

我们知道存储型XSS攻击和反射型XSS攻击都是需要经过Web服务器来处理的，因此可以认为这两种类型的漏洞是服务端的安全漏洞。而基于DOM的XSS攻击全部都是在浏览器端完成的，因此基于DOM的XSS攻击是属于前端的安全漏洞。

但无论是何种类型的XSS攻击，它们都有一个共同点，那就是首先往浏览器中注入恶意脚本，然后再通过恶意脚本将用户信息发送至黑客部署的恶意服务器上。

所以要阻止XSS攻击，我们可以通过阻止恶意JavaScript脚本的注入和恶意消息的发送来实现。

接下来我们就来看看一些常用的阻止XSS攻击的策略。

1. 服务器对输入脚本进行过滤或转码

不管是反射型还是存储型XSS攻击，我们都可以在服务器端将一些关键的字符进行转码，比如最典型的：

```
code:<script>alert('你被xss攻击了')</script>
```

这段代码过滤后，只留下了：

```
code:
```

这样，当用户再次请求该页面时，由于`<script>`标签的内容都被过滤了，所以这段脚本在客户端是不可能被执行的。

除了过滤之外，服务器还可以对这些内容进行转码，还是上面那段代码，经过转码之后，效果如下所示：

```
code:&lt;script&gt;alert(&#39;你被xss攻击了&#39;)&lt;/script&gt;
```

经过转码之后的内容，如`<script>`标签被转换为`<script>`，因此即使这段脚本返回给页面，页面也不会执行这段脚本。

2. 充分利用CSP

虽然在服务器端执行过滤或者转码可以阻止 XSS 攻击的发生，但完全依靠服务器端依然是不够的，我们还需要把CSP等策略充分地利用起来，以降低 XSS攻击带来的风险和后果。

实施严格的CSP可以有效地防范XSS攻击，具体来讲CSP有如下几个功能：

- 限制加载其他域下的资源文件，这样即使黑客插入了一个JavaScript文件，这个JavaScript文件也是无法被加载的；
- 禁止向第三方域提交数据，这样用户数据也不会外泄；
- 禁止执行内联脚本和未授权的脚本；
- 还提供了上报机制，这样可以帮助我们尽快发现有哪些XSS攻击，以便尽快修复问题。

因此，利用好CSP能够有效降低XSS攻击的概率。

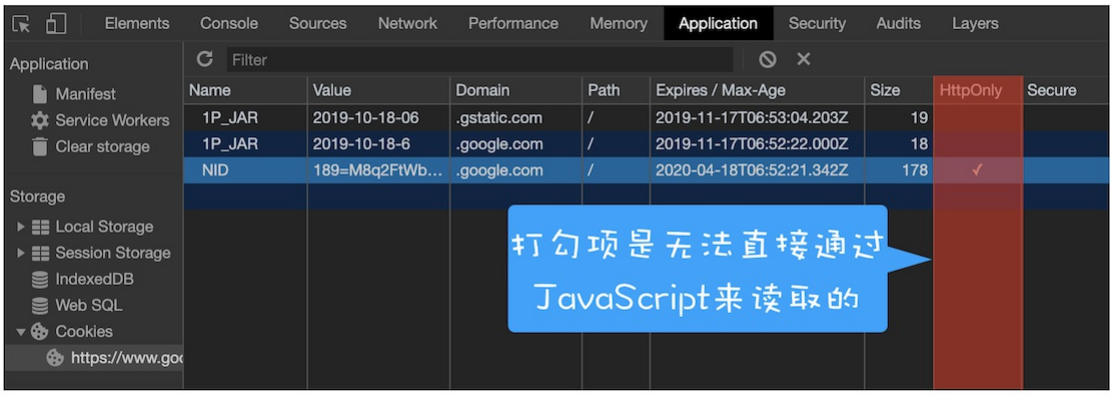
3. 使用HttpOnly属性

由于很多XSS攻击都是来盗用Cookie的，因此还可以通过使用HttpOnly属性来保护我们Cookie的安全。

通常服务器可以将某些Cookie设置为HttpOnly标志，HttpOnly是服务器通过HTTP响应头来设置的，下面是打开Google时，HTTP响应头中的一段：

```
set-cookie: NID=189=M8q2FtWbsR8RlcldPVt7qkrqR38LmFY9jUxkKo3-4Bi6Qu_ocNOat7nkYZUTzolHjFnwBw0izgsATSI7TZyiiiaV94qGh-BzEYsNVa7TZmjAYTxYTOM9L_-0CN9ipL6cXi8l6-z41asXtm2uEwcOC5oh9djkff(
```

我们可以看到，set-cookie属性值最后使用了HttpOnly来标记该Cookie。顾名思义，使用HttpOnly标记的Cookie只能使用在HTTP请求过程中，所以无法通过JavaScript来读取这段Cookie。我们还可以通过Chrome开发者工具来查看哪些Cookie被标记了HttpOnly，如下图：



HttpOnly演示

从图中可以看出，NID这个Cookie的HttpOlny属性是被勾选上的，所以NID的内容是无法通过document.cookie是来读取的。

由于JavaScript无法读取设置了HttpOnly的Cookie数据，所以即使页面被注入了恶意JavaScript脚本，也是无法获取到设置了HttpOnly的数据。因此一些比较重要的数据我们建议设置HttpOnly标志。

总结

好了，今天我们就介绍到这里，下面我来总结下本文的主要内容。

XSS攻击就是黑客往页面中注入恶意脚本，然后将页面的一些重要数据上传到恶意服务器。常见的三种XSS攻击模式是存储型XSS攻击、反射型XSS攻击和基于DOM的XSS攻击。

这三种攻击方式的共同点是都需要往用户的页面中注入恶意脚本，然后再通过恶意脚本将用户数据上传到黑客的恶意服务器上。而三者的不同点在于注入的方式不一样，有通过服务器漏洞来进行注入的，还有在客户端直接注入的。

针对这些XSS攻击，主要有三种防范策略，第一种是通过服务器对输入的内容进行过滤或者转码，第二种是充分利用好CSP，第三种是使用HttpOnly来保护重要的Cookie信息。

当然除了以上策略之外，我们还可以通过添加验证码防止脚本冒充用户提交危险操作。而对于一些不受信任的输入，还可以限制其输入长度，这样可以增大XSS攻击的难度。

思考时间

今天留给你的思考题是：你认为前端开发者对XSS攻击应该负多大责任？

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

通过上篇文章的介绍，我们知道了同源策略可以隔离各个站点之间的DOM交互、页面数据和网络通信，虽然严格的同源策略会带来更多的安全，但是也束缚了Web。这就需要在安全和自由之间找到一个平衡点，所以我们默认页面中可以引用任意第三方资源，然后又引入CSP策略来加以限制；默认XMLHttpRequest和Fetch不能跨站请求资源，然后又通过CORS策略来支持其跨域。

不过支持页面中的第三方资源引用和CORS也带来了很多安全问题，其中最典型的就是XSS攻击。

什么是XSS攻击

XSS全称是Cross Site Scripting，为了与“CSS”区分开来，故简称XSS，翻译过来就是“跨站脚本”。XSS攻击是指黑客往HTML文件中或者DOM中注入恶意脚本，从而在用户浏览页面时利用注入的恶意脚本对用户实施攻击的一种手段。

最开始的时候，这种攻击是通过跨域来实现的，所以叫“跨域脚本”。但是发展到现在，往HTML文件中注入恶意代码的方式越来越多了，所以是否跨域注入脚本已经不是唯一的注入手段了，但是XSS这个名字却一直保留至今。

当页面被注入了恶意JavaScript脚本时，浏览器无法区分这些脚本是被恶意注入的还是正常的页面内容，所以恶意注入JavaScript脚本也拥有所有的脚本权限。下面我们就来看看，如果页面被注入了恶意JavaScript脚本，恶意脚本都能做哪些事情。

- 可以**窃取Cookie信息**。恶意JavaScript可以通过“document.cookie”获取Cookie信息，然后通过XMLHttpRequest或者Fetch加上CORS功能将数据发送给恶意服务器；恶意服务器拿到用户的Cookie信息之后，就可以在其他电脑上模拟用户的登录，然后进行转账等操作。
- 可以**监听用户行为**。恶意JavaScript可以使用“addEventListener”接口来监听键盘事件，比如可以获取用户输入的信用卡等信息，将其发送到恶意服务器。黑客掌握了这些信息之后，又可以做很多违法的事情。
- 可以通过**修改DOM**伪造假的登录窗口，用来欺骗用户输入用户名和密码等信息。
- 还可以**在页面内生成浮窗广告**，这些广告会严重地影响用户体验。

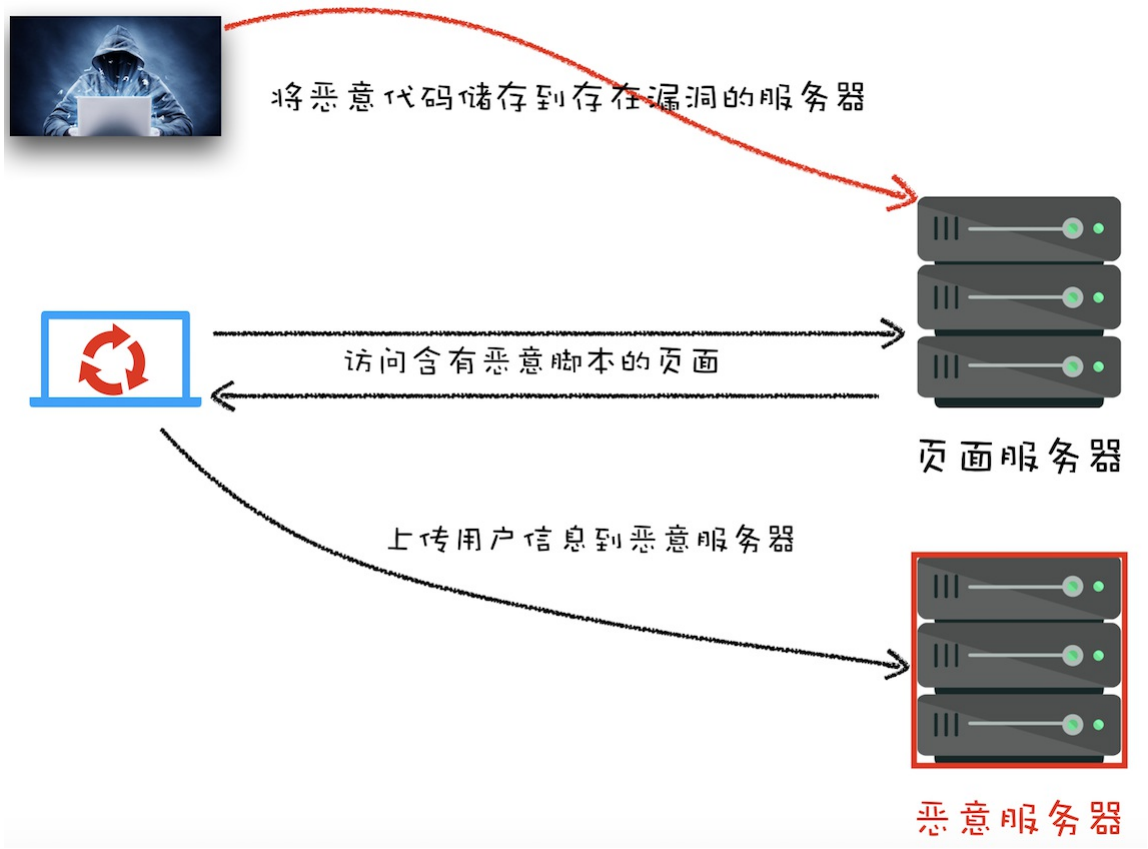
除了以上几种情况外，恶意脚本还能做很多其他的事情，这里就不一一介绍了。总之，如果让页面插入了恶意脚本，那么就相当于把我们页面的隐私数据和行为完全暴露给黑客了。

恶意脚本是怎么注入的

现在我们知道了页面中被注入恶意的JavaScript脚本是一件非常危险的事情，所以网站开发者会尽可能地避免页面中被注入恶意脚本。要想避免站点被注入恶意脚本，就要知道有哪些常见的注入方式。通常情况下，主要有**存储型XSS攻击**、**反射型XSS攻击**和**基于DOM的XSS攻击**三种方式来注入恶意脚本。

1. 存储型XSS攻击

我们先来看看存储型XSS攻击是怎么向HTML文件中注入恶意脚本的，你可以参考下图：



存储型XSS攻击

通过上图，我们可以看出存储型XSS攻击大致需要经过如下步骤：

- 首先黑客利用站点漏洞将一段恶意JavaScript代码提交到网站的数据库中；
- 然后用户向网站请求包含了恶意JavaScript脚本的页面；
- 当用户浏览该页面的时候，恶意脚本就会将用户的Cookie信息等数据上传到服务器。

下面我们来看个例子，2015年喜马拉雅就被曝出了存储型XSS漏洞。起因是在用户设置专辑名称时，服务器对关键字过滤不严格，比如可以将专辑名称设置为一段JavaScript，如下图所示：

黑客将恶意代码存储到漏洞服务器上

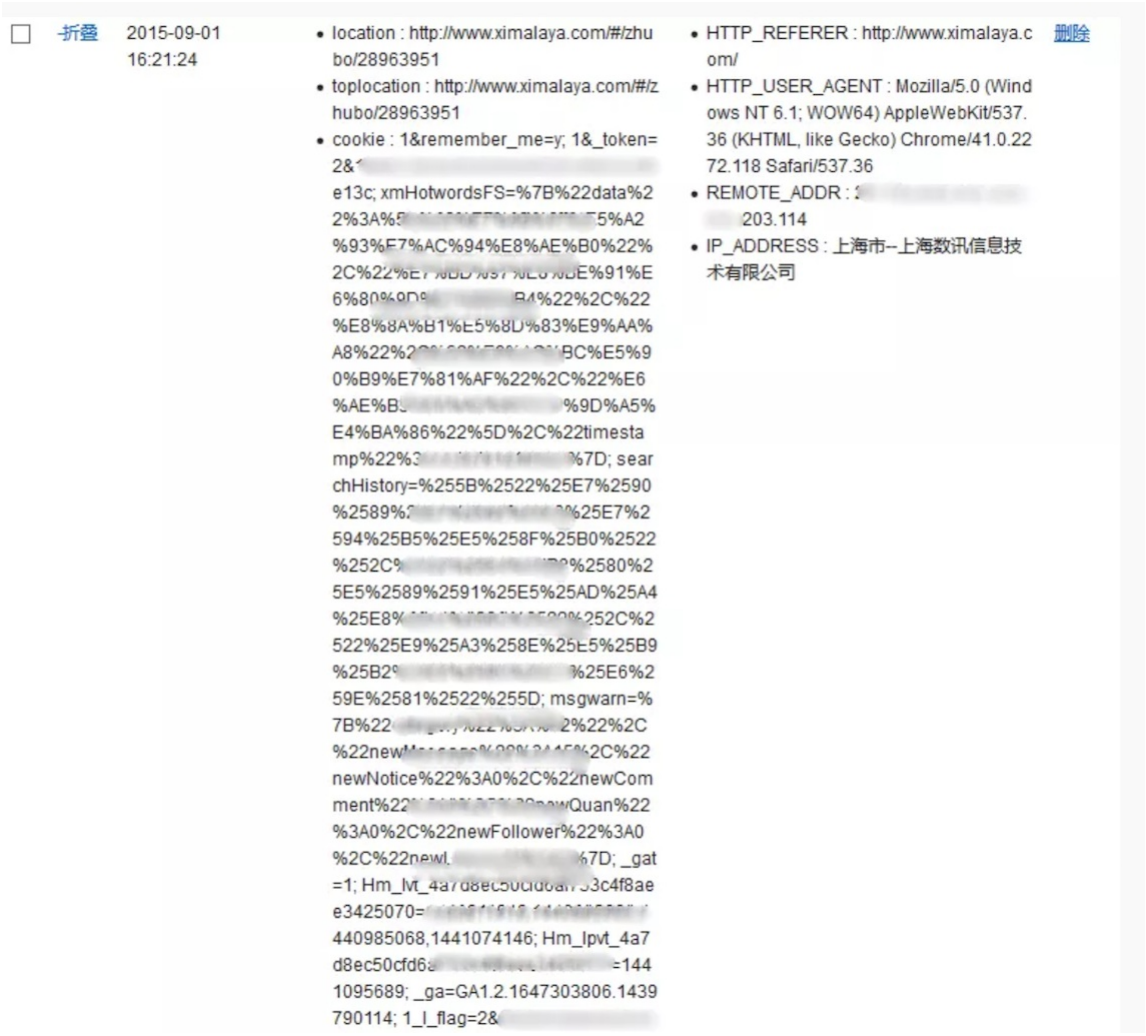
当黑客将专辑名称设置为一段JavaScript代码并提交时，喜马拉雅的服务器会保存该段JavaScript代码到数据库中。然后当用户打开黑客设置的专辑时，这段代码就会在用户的页面里执行（如下图），这样

就可以获取用户的Cookie等数据信息。

用户打开了含有恶意脚本的页面

当用户打开黑客设置的专辑页面时，服务器也会将这段恶意JavaScript代码返回给用户，因此这段恶意脚本就在用户的页面中执行了。

恶意脚本可以通过XMLHttpRequest或者Fetch将用户的Cookie数据上传到黑客的服务器，如下图所示：



将Cookie等数据上传到黑客服务器

黑客拿到了用户Cookie信息之后，就可以利用Cookie信息在其他机器上登录该用户的账号（如下图），并利用用户账号进行一些恶意操作。

黑客利用Cookie信息登录用户账户

以上就是存储型XSS攻击的一个典型案例，这是乌云网在2015年曝出来的，虽然乌云网由于某些原因被关停了，但是你依然可以通过这个站点来查看乌云网的一些备份信息。

2. 反射型XSS攻击

在一个反射型XSS攻击过程中，恶意JavaScript脚本属于用户发送给网站请求中的一部分，随后网站又把恶意JavaScript脚本返回给用户。当恶意JavaScript脚本在用户页面中被执行时，黑客就可以利用该脚本做一些恶意操作。

这样讲有点抽象，下面我们结合一个简单的Node服务程序来看看什么是反射型XSS。首先我们使用Node来搭建一个简单的页面环境，搭建好的服务代码如下所示：

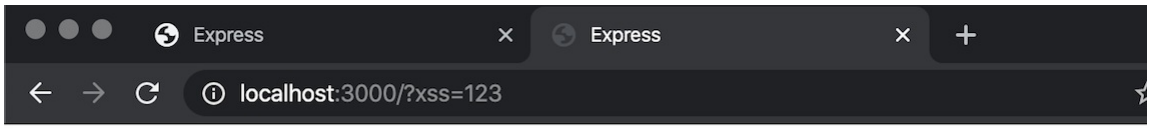
```
var express = require('express');
var router = express.Router();


/* GET home page. */
router.get('/', function(req, res, next) {
  res.render('index', { title: 'Express',xss:req.query.xss });
});
```

```
module.exports = router;
```

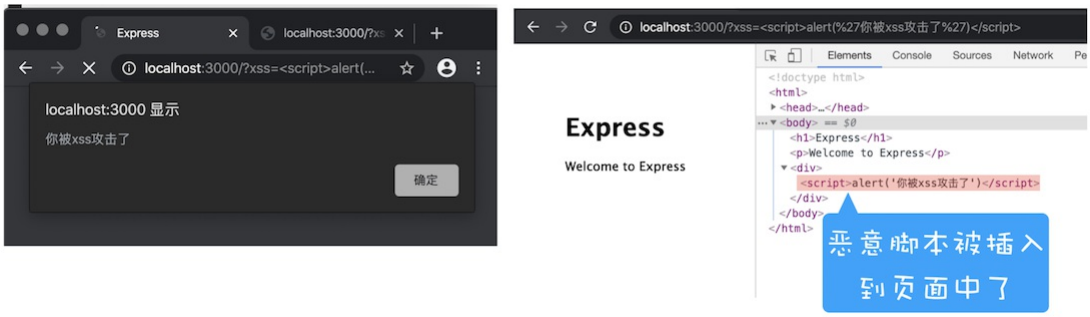
```
<!DOCTYPE html>
<html>
<head>
  <title><%= title %></title>
  <link rel='stylesheet' href='/stylesheets/style.css' />
</head>
<body>
  <h1><%= title %></h1>
  <p>Welcome to <%= title %></p>
  <div>
      <%- xss %>
  </div>
</body>
</html>
```

上面这两段代码，第一段是路由，第二段是视图，作用是将URL中xss参数的内容显示在页面。我们可以在本地演示下，比如打开`http://localhost:3000/?xss=123`这个链接，这样在页面中展示就是“123”了（如下图），是正常的，没有问题的。



正常打开页面

但当打开`http://localhost:3000/?xss=<script>alert('你被xss攻击了')</script>`这段URL时，其结果如下图所示：



反射型XSS攻击

通过这个操作，我们会发现用户将一段含有恶意代码的请求提交给Web服务器，Web服务器接收到请求时，又将恶意代码反射给了浏览器端，这就是反射型XSS攻击。在现实生活中，黑客经常会通过QQ群或者邮件等渠道诱导用户去点击这些恶意链接，所以对于一些链接我们一定要慎之又慎。

另外需要注意的是，**Web服务器不会存储反射型XSS攻击的恶意脚本，这是和存储型XSS攻击不同的地方**。

3. 基于DOM的XSS攻击

基于DOM的XSS攻击是不牵涉到页面Web服务器的。具体来讲，黑客通过各种手段将恶意脚本注入用户的页面中，比如通过网络劫持在页面传输过程中修改HTML页面的内容，这种劫持类型很多，有通过WiFi路由器劫持的，有通过本地恶意软件来劫持的，它们的共同点是在Web资源传输过程或者在用户使用页面的过程中修改Web页面的数据。

如何阻止XSS攻击

我们知道存储型XSS攻击和反射型XSS攻击都是需要经过Web服务器来处理的，因此可以认为这两种类型的漏洞是服务端的安全漏洞。而基于DOM的XSS攻击全部都是在浏览器端完成的，因此基于DOM的XSS攻击是属于前端的安全漏洞。

但无论是何种类型的XSS攻击，它们都有一个共同点，那就是首先往浏览器中注入恶意脚本，然后再通过恶意脚本将用户信息发送至黑客部署的恶意服务器上。

所以要阻止XSS攻击，我们可以通过阻止恶意JavaScript脚本的注入和恶意消息的发送来实现。

接下来我们就来看看一些常用的阻止XSS攻击的策略。

1. 服务器对输入脚本进行过滤或转码

不管是反射型还是存储型XSS攻击，我们都可以在服务器端将一些关键的字符进行转码，比如最典型的：

```
code:<script>alert('你被xss攻击了')</script>
```

这段代码过滤后，只留下了：

```
code:
```

这样，当用户再次请求该页面时，由于`<script>`标签的内容都被过滤了，所以这段脚本在客户端是不可能被执行的。

除了过滤之外，服务器还可以对这些内容进行转码，还是上面那段代码，经过转码之后，效果如下所示：

```
code:&lt;script&gt;alert(&#39;你被xss攻击了&#39;)&lt;/script&gt;
```

经过转码之后的内容，如`<script>`标签被转换为`<script>`，因此即使这段脚本返回给页面，页面也不会执行这段脚本。

2. 充分利用CSP

虽然在服务器端执行过滤或者转码可以阻止 XSS 攻击的发生，但完全依靠服务器端依然是不够的，我们还需要把CSP等策略充分地利用起来，以降低 XSS攻击带来的风险和后果。

实施严格的CSP可以有效地防范XSS攻击，具体来讲CSP有如下几个功能：

- 限制加载其他域下的资源文件，这样即使黑客插入了一个JavaScript文件，这个JavaScript文件也是无法被加载的；
- 禁止向第三方域提交数据，这样用户数据也不会外泄；
- 禁止执行内联脚本和未授权的脚本；
- 还提供了上报机制，这样可以帮助我们尽快发现有哪些XSS攻击，以便尽快修复问题。

因此，利用好CSP能够有效降低XSS攻击的概率。

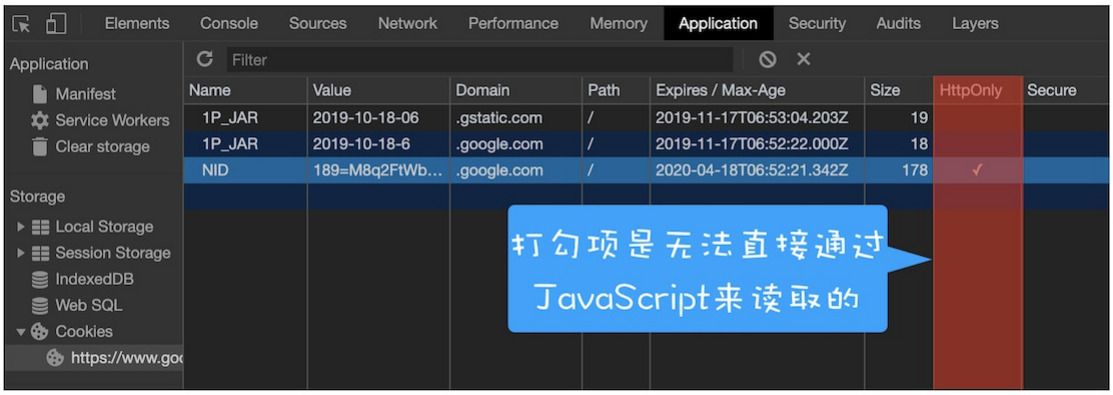
3. 使用HttpOnly属性

由于很多XSS攻击都是来盗用Cookie的，因此还可以通过使用HttpOnly属性来保护我们Cookie的安全。

通常服务器可以将某些Cookie设置为HttpOnly标志，HttpOnly是服务器通过HTTP响应头来设置的，下面是打开Google时，HTTP响应头中的一段：

```
set-cookie: NID=189=M8q2FtWbsR8RlcldPVt7qkrqR38LmFY9jUxkKo3-4Bi6Qu_ocNOat7nkYZUTzolHjFnwBw0izgsATSI7TZyiiiaV94qGh-BzEYsNVa7TZmjAYTxYTOM9L_-0CN9ipL6cXi8l6-z41asXtm2uEwcOC5oh9djkff(
```

我们可以看到，set-cookie属性值最后使用了HttpOnly来标记该Cookie。顾名思义，使用HttpOnly标记的Cookie只能使用在HTTP请求过程中，所以无法通过JavaScript来读取这段Cookie。我们还可以通过Chrome开发者工具来查看哪些Cookie被标记了HttpOnly，如下图：



HttpOnly演示

从图中可以看出，NID这个Cookie的HttpOlny属性是被勾选上的，所以NID的内容是无法通过document.cookie是来读取的。

由于JavaScript无法读取设置了HttpOnly的Cookie数据，所以即使页面被注入了恶意JavaScript脚本，也是无法获取到设置了HttpOnly的数据。因此一些比较重要的数据我们建议设置HttpOnly标志。

总结

好了，今天我们就介绍到这里，下面我来总结下本文的主要内容。

XSS攻击就是黑客往页面中注入恶意脚本，然后将页面的一些重要数据上传到恶意服务器。常见的三种XSS攻击模式是存储型XSS攻击、反射型XSS攻击和基于DOM的XSS攻击。

这三种攻击方式的共同点是都需要往用户的页面中注入恶意脚本，然后再通过恶意脚本将用户数据上传到黑客的恶意服务器上。而三者的不同点在于注入的方式不一样，有通过服务器漏洞来进行注入的，还有在客户端直接注入的。

针对这些XSS攻击，主要有三种防范策略，第一种是通过服务器对输入的内容进行过滤或者转码，第二种是充分利用好CSP，第三种是使用HttpOnly来保护重要的Cookie信息。

当然除了以上策略之外，我们还可以通过添加验证码防止脚本冒充用户提交危险操作。而对于一些不受信任的输入，还可以限制其输入长度，这样可以增大XSS攻击的难度。

思考时间

今天留给你的思考题是：你认为前端开发者对XSS攻击应该负多大责任？

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

通过上篇文章的介绍，我们知道了同源策略可以隔离各个站点之间的DOM交互、页面数据和网络通信，虽然严格的同源策略会带来更多的安全，但是也束缚了Web。这就需要在安全和自由之间找到一个平衡点，所以我们默认页面中可以引用任意第三方资源，然后又引入CSP策略来加以限制；默认XMLHttpRequest和Fetch不能跨站请求资源，然后又通过CORS策略来支持其跨域。

不过支持页面中的第三方资源引用和CORS也带来了很多安全问题，其中最典型的就是XSS攻击。

什么是XSS攻击

XSS全称是Cross Site Scripting，为了与“CSS”区分开来，故简称XSS，翻译过来就是“跨站脚本”。XSS攻击是指黑客往HTML文件中或者DOM中注入恶意脚本，从而在用户浏览页面时利用注入的恶意脚本对用户实施攻击的一种手段。

最开始的时候，这种攻击是通过跨域来实现的，所以叫“跨域脚本”。但是发展到现在，往HTML文件中注入恶意代码的方式越来越多了，所以是否跨域注入脚本已经不是唯一的注入手段了，但是XSS这个名字却一直保留至今。

当页面被注入了恶意JavaScript脚本时，浏览器无法区分这些脚本是被恶意注入的还是正常的页面内容，所以恶意注入JavaScript脚本也拥有所有的脚本权限。下面我们就来看看，如果页面被注入了恶意JavaScript脚本，恶意脚本都能做哪些事情。

- 可以**窃取Cookie信息**。恶意JavaScript可以通过“document.cookie”获取Cookie信息，然后通过XMLHttpRequest或者Fetch加上CORS功能将数据发送给恶意服务器；恶意服务器拿到用户的Cookie信息之后，就可以在其他电脑上模拟用户的登录，然后进行转账等操作。
- 可以**监听用户行为**。恶意JavaScript可以使用“addEventListener”接口来监听键盘事件，比如可以获取用户输入的信用卡等信息，将其发送到恶意服务器。黑客掌握了这些信息之后，又可以做很多违法的事情。
- 可以通过**修改DOM**伪造假的登录窗口，用来欺骗用户输入用户名和密码等信息。
- 还可以**在页面内生成浮窗广告**，这些广告会严重地影响用户体验。

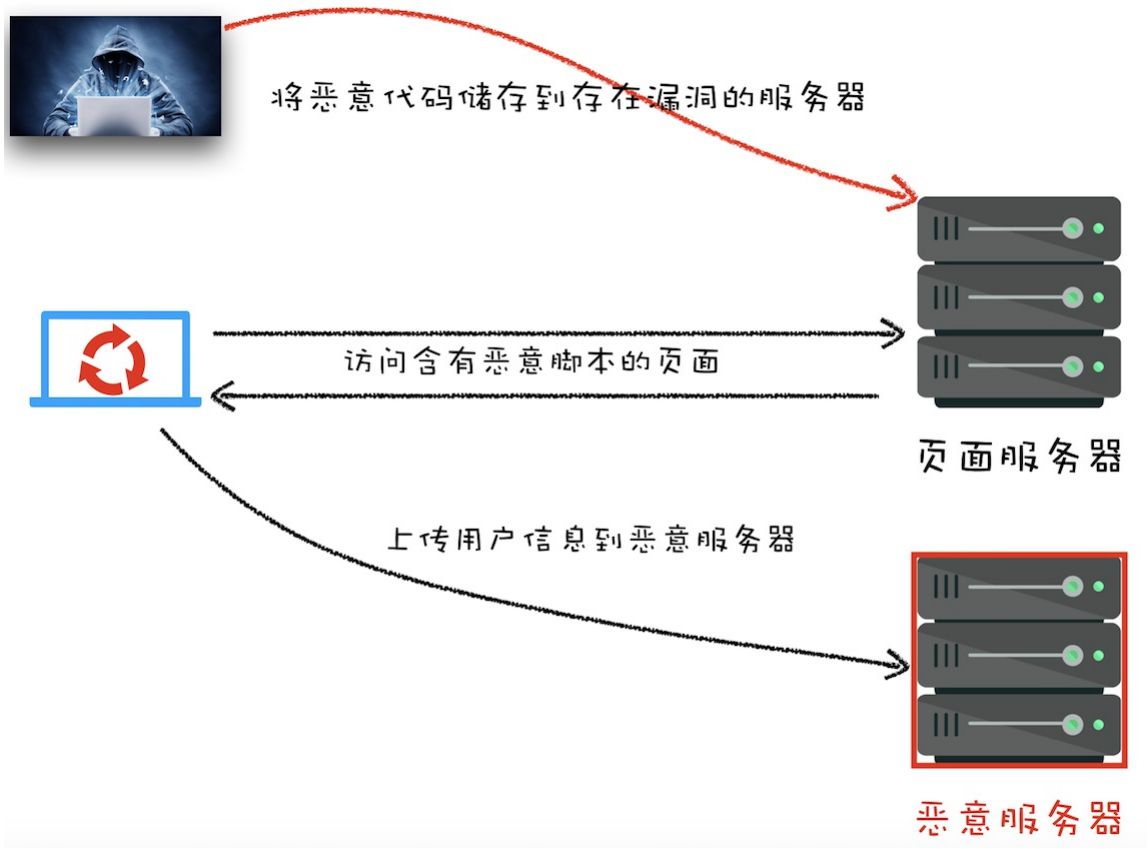
除了以上几种情况外，恶意脚本还能做很多其他的事情，这里就不一一介绍了。总之，如果让页面插入了恶意脚本，那么就相当于把我们页面的隐私数据和行为完全暴露给黑客了。

恶意脚本是怎么注入的

现在我们知道了页面中被注入恶意的JavaScript脚本是一件非常危险的事情，所以网站开发者会尽可能地避免页面中被注入恶意脚本。要想避免站点被注入恶意脚本，就要知道有哪些常见的注入方式。通常情况下，主要有**存储型XSS攻击**、**反射型XSS攻击**和**基于DOM的XSS攻击**三种方式来注入恶意脚本。

1. 存储型XSS攻击

我们先来看看存储型XSS攻击是怎么向HTML文件中注入恶意脚本的，你可以参考下图：



存储型XSS攻击

通过上图，我们可以看出存储型XSS攻击大致需要经过如下步骤：

- 首先黑客利用站点漏洞将一段恶意JavaScript代码提交到网站的数据库中；
- 然后用户向网站请求包含了恶意JavaScript脚本的页面；
- 当用户浏览该页面的时候，恶意脚本就会将用户的Cookie信息等数据上传到服务器。

下面我们来看个例子，2015年喜马拉雅就被曝出了存储型XSS漏洞。起因是在用户设置专辑名称时，服务器对关键字过滤不严格，比如可以将专辑名称设置为一段JavaScript，如下图所示：

黑客将恶意代码存储到漏洞服务器上

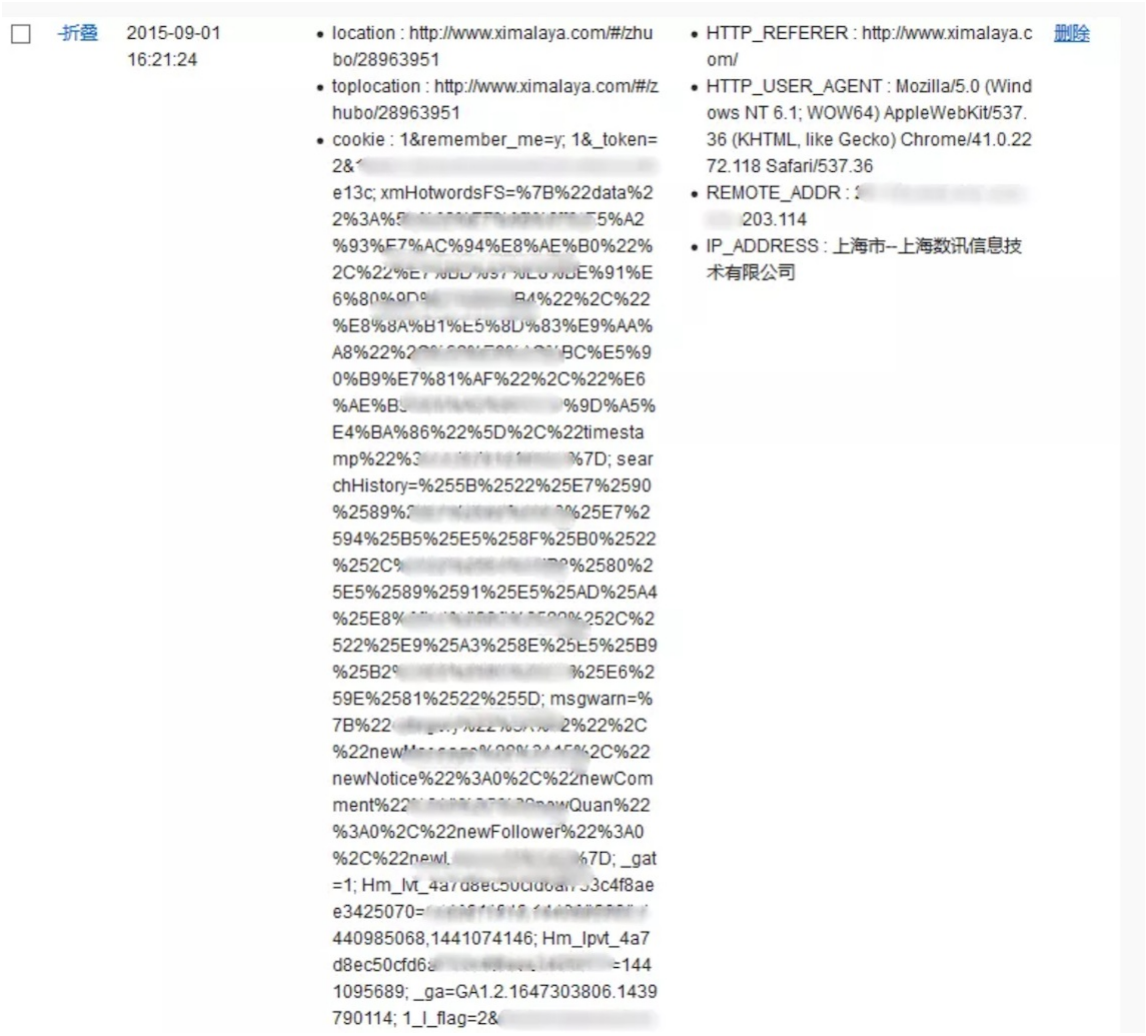
当黑客将专辑名称设置为一段JavaScript代码并提交时，喜马拉雅的服务器会保存该段JavaScript代码到数据库中。然后当用户打开黑客设置的专辑时，这段代码就会在用户的页面里执行（如下图），这样

就可以获取用户的Cookie等数据信息。

用户打开了含有恶意脚本的页面

当用户打开黑客设置的专辑页面时，服务器也会将这段恶意JavaScript代码返回给用户，因此这段恶意脚本就在用户的页面中执行了。

恶意脚本可以通过XMLHttpRequest或者Fetch将用户的Cookie数据上传到黑客的服务器，如下图所示：



将Cookie等数据上传到黑客服务器

黑客拿到了用户Cookie信息之后，就可以利用Cookie信息在其他机器上登录该用户的账号（如下图），并利用用户账号进行一些恶意操作。

黑客利用Cookie信息登录用户账户

以上就是存储型XSS攻击的一个典型案例，这是乌云网在2015年曝出来的，虽然乌云网由于某些原因被关停了，但是你依然可以通过这个站点来查看乌云网的一些备份信息。

2. 反射型XSS攻击

在一个反射型XSS攻击过程中，恶意JavaScript脚本属于用户发送给网站请求中的一部分，随后网站又把恶意JavaScript脚本返回给用户。当恶意JavaScript脚本在用户页面中被执行时，黑客就可以利用该脚本做一些恶意操作。

这样讲有点抽象，下面我们结合一个简单的Node服务程序来看看什么是反射型XSS。首先我们使用Node来搭建一个简单的页面环境，搭建好的服务代码如下所示：

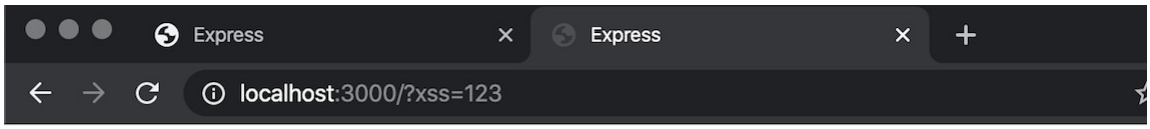
```
var express = require('express');
var router = express.Router();


/* GET home page. */
router.get('/', function(req, res, next) {
  res.render('index', { title: 'Express',xss:req.query.xss });
});
```

```
module.exports = router;
```

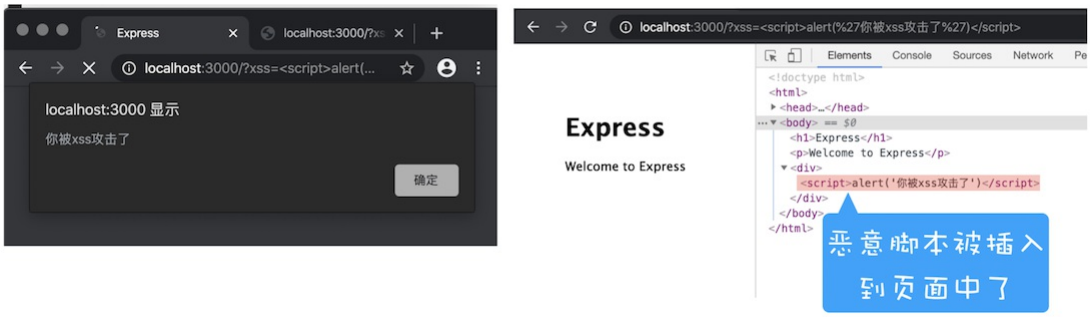
```
<!DOCTYPE html>
<html>
<head>
  <title><%= title %></title>
  <link rel='stylesheet' href='/stylesheets/style.css' />
</head>
<body>
  <h1><%= title %></h1>
  <p>Welcome to <%= title %></p>
  <div>
      <%- xss %>
  </div>
</body>
</html>
```

上面这两段代码，第一段是路由，第二段是视图，作用是将URL中xss参数的内容显示在页面。我们可以在本地演示下，比如打开`http://localhost:3000/?xss=123`这个链接，这样在页面中展示就是“123”了（如下图），是正常的，没有问题的。



正常打开页面

但当打开`http://localhost:3000/?xss=<script>alert('你被xss攻击了')</script>`这段URL时，其结果如下图所示：



反射型XSS攻击

通过这个操作，我们会发现用户将一段含有恶意代码的请求提交给Web服务器，Web服务器接收到请求时，又将恶意代码反射给了浏览器端，这就是反射型XSS攻击。在现实生活中，黑客经常会通过QQ群或者邮件等渠道诱导用户去点击这些恶意链接，所以对于一些链接我们一定要慎之又慎。

另外需要注意的是，**Web服务器不会存储反射型XSS攻击的恶意脚本，这是和存储型XSS攻击不同的地方**。

3. 基于DOM的XSS攻击

基于DOM的XSS攻击是不牵涉到页面Web服务器的。具体来讲，黑客通过各种手段将恶意脚本注入用户的页面中，比如通过网络劫持在页面传输过程中修改HTML页面的内容，这种劫持类型很多，有通过WiFi路由器劫持的，有通过本地恶意软件来劫持的，它们的共同点是在Web资源传输过程或者在用户使用页面的过程中修改Web页面的数据。

如何阻止XSS攻击

我们知道存储型XSS攻击和反射型XSS攻击都是需要经过Web服务器来处理的，因此可以认为这两种类型的漏洞是服务端的安全漏洞。而基于DOM的XSS攻击全部都是在浏览器端完成的，因此基于DOM的XSS攻击是属于前端的安全漏洞。

但无论是何种类型的XSS攻击，它们都有一个共同点，那就是首先往浏览器中注入恶意脚本，然后再通过恶意脚本将用户信息发送至黑客部署的恶意服务器上。

所以要阻止XSS攻击，我们可以通过阻止恶意JavaScript脚本的注入和恶意消息的发送来实现。

接下来我们就来看看一些常用的阻止XSS攻击的策略。

1. 服务器对输入脚本进行过滤或转码

不管是反射型还是存储型XSS攻击，我们都可以在服务器端将一些关键的字符进行转码，比如最典型的：

```
code:<script>alert('你被xss攻击了')</script>
```

这段代码过滤后，只留下了：

```
code:
```

这样，当用户再次请求该页面时，由于`<script>`标签的内容都被过滤了，所以这段脚本在客户端是不可能被执行的。

除了过滤之外，服务器还可以对这些内容进行转码，还是上面那段代码，经过转码之后，效果如下所示：

```
code:&lt;script&gt;alert(&#39;你被xss攻击了&#39;)&lt;/script&gt;
```

经过转码之后的内容，如`<script>`标签被转换为`<script>`，因此即使这段脚本返回给页面，页面也不会执行这段脚本。

2. 充分利用CSP

虽然在服务器端执行过滤或者转码可以阻止 XSS 攻击的发生，但完全依靠服务器端依然是不够的，我们还需要把CSP等策略充分地利用起来，以降低 XSS攻击带来的风险和后果。

实施严格的CSP可以有效地防范XSS攻击，具体来讲CSP有如下几个功能：

- 限制加载其他域下的资源文件，这样即使黑客插入了一个JavaScript文件，这个JavaScript文件也是无法被加载的；
- 禁止向第三方域提交数据，这样用户数据也不会外泄；
- 禁止执行内联脚本和未授权的脚本；
- 还提供了上报机制，这样可以帮助我们尽快发现有哪些XSS攻击，以便尽快修复问题。

因此，利用好CSP能够有效降低XSS攻击的概率。

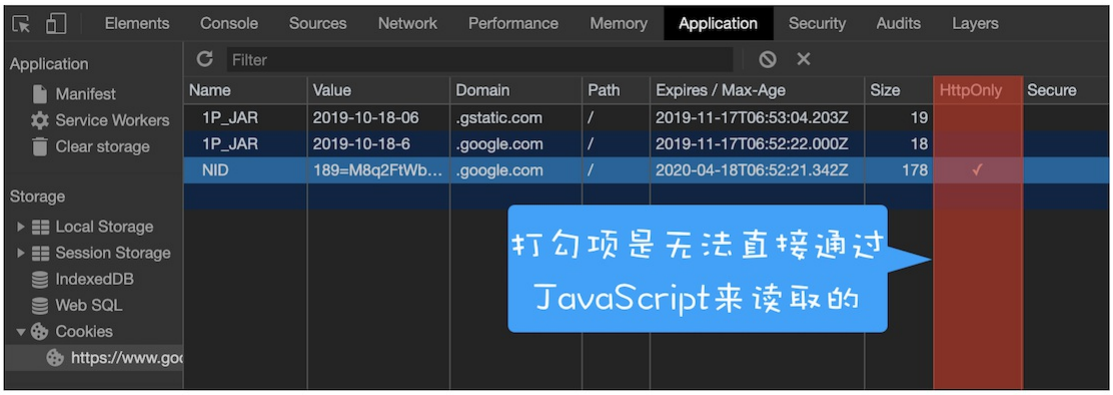
3. 使用HttpOnly属性

由于很多XSS攻击都是来盗用Cookie的，因此还可以通过使用HttpOnly属性来保护我们Cookie的安全。

通常服务器可以将某些Cookie设置为HttpOnly标志，HttpOnly是服务器通过HTTP响应头来设置的，下面是打开Google时，HTTP响应头中的一段：

```
set-cookie: NID=189=M8q2FtWbsR8RlcldPVt7qkrqR38LmFY9jUxkKo3-4Bi6Qu_ocNOat7nkYZUTzolHjFnwBw0izgsATSI7TZyiiiaV94qGh-BzEYsNVa7TZmjAYTxYTOM9L_-0CN9ipL6cXi8l6-z41asXtm2uEwcOC5oh9djkff(
```

我们可以看到，set-cookie属性值最后使用了HttpOnly来标记该Cookie。顾名思义，使用HttpOnly标记的Cookie只能使用在HTTP请求过程中，所以无法通过JavaScript来读取这段Cookie。我们还可以通过Chrome开发者工具来查看哪些Cookie被标记了HttpOnly，如下图：



HttpOnly演示

从图中可以看出，NID这个Cookie的HttpOlny属性是被勾选上的，所以NID的内容是无法通过document.cookie是来读取的。

由于JavaScript无法读取设置了HttpOnly的Cookie数据，所以即使页面被注入了恶意JavaScript脚本，也是无法获取到设置了HttpOnly的数据。因此一些比较重要的数据我们建议设置HttpOnly标志。

总结

好了，今天我们就介绍到这里，下面我来总结下本文的主要内容。

XSS攻击就是黑客往页面中注入恶意脚本，然后将页面的一些重要数据上传到恶意服务器。常见的三种XSS攻击模式是存储型XSS攻击、反射型XSS攻击和基于DOM的XSS攻击。

这三种攻击方式的共同点是都需要往用户的页面中注入恶意脚本，然后再通过恶意脚本将用户数据上传到黑客的恶意服务器上。而三者的不同点在于注入的方式不一样，有通过服务器漏洞来进行注入的，还有在客户端直接注入的。

针对这些XSS攻击，主要有三种防范策略，第一种是通过服务器对输入的内容进行过滤或者转码，第二种是充分利用好CSP，第三种是使用HttpOnly来保护重要的Cookie信息。

当然除了以上策略之外，我们还可以通过添加验证码防止脚本冒充用户提交危险操作。而对于一些不受信任的输入，还可以限制其输入长度，这样可以增大XSS攻击的难度。

思考时间

今天留给你的思考题是：你认为前端开发者对XSS攻击应该负多大责任？

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

通过上篇文章的介绍，我们知道了同源策略可以隔离各个站点之间的DOM交互、页面数据和网络通信，虽然严格的同源策略会带来更多的安全，但是也束缚了Web。这就需要在安全和自由之间找到一个平衡点，所以我们默认页面中可以引用任意第三方资源，然后又引入CSP策略来加以限制；默认XMLHttpRequest和Fetch不能跨站请求资源，然后又通过CORS策略来支持其跨域。

不过支持页面中的第三方资源引用和CORS也带来了很多安全问题，其中最典型的就是XSS攻击。

什么是XSS攻击

XSS全称是Cross Site Scripting，为了与“CSS”区分开来，故简称XSS，翻译过来就是“跨站脚本”。XSS攻击是指黑客往HTML文件中或者DOM中注入恶意脚本，从而在用户浏览页面时利用注入的恶意脚本对用户实施攻击的一种手段。

最开始的时候，这种攻击是通过跨域来实现的，所以叫“跨域脚本”。但是发展到现在，往HTML文件中注入恶意代码的方式越来越多了，所以是否跨域注入脚本已经不是唯一的注入手段了，但是XSS这个名字却一直保留至今。

当页面被注入了恶意JavaScript脚本时，浏览器无法区分这些脚本是被恶意注入的还是正常的页面内容，所以恶意注入JavaScript脚本也拥有所有的脚本权限。下面我们就来看看，如果页面被注入了恶意JavaScript脚本，恶意脚本都能做哪些事情。

- 可以**窃取Cookie信息**。恶意JavaScript可以通过“document.cookie”获取Cookie信息，然后通过XMLHttpRequest或者Fetch加上CORS功能将数据发送给恶意服务器；恶意服务器拿到用户的Cookie信息之后，就可以在其他电脑上模拟用户的登录，然后进行转账等操作。
- 可以**监听用户行为**。恶意JavaScript可以使用“addEventListener”接口来监听键盘事件，比如可以获取用户输入的信用卡等信息，将其发送到恶意服务器。黑客掌握了这些信息之后，又可以做很多违法的事情。
- 可以通过**修改DOM**伪造假的登录窗口，用来欺骗用户输入用户名和密码等信息。
- 还可以**在页面内生成浮窗广告**，这些广告会严重地影响用户体验。

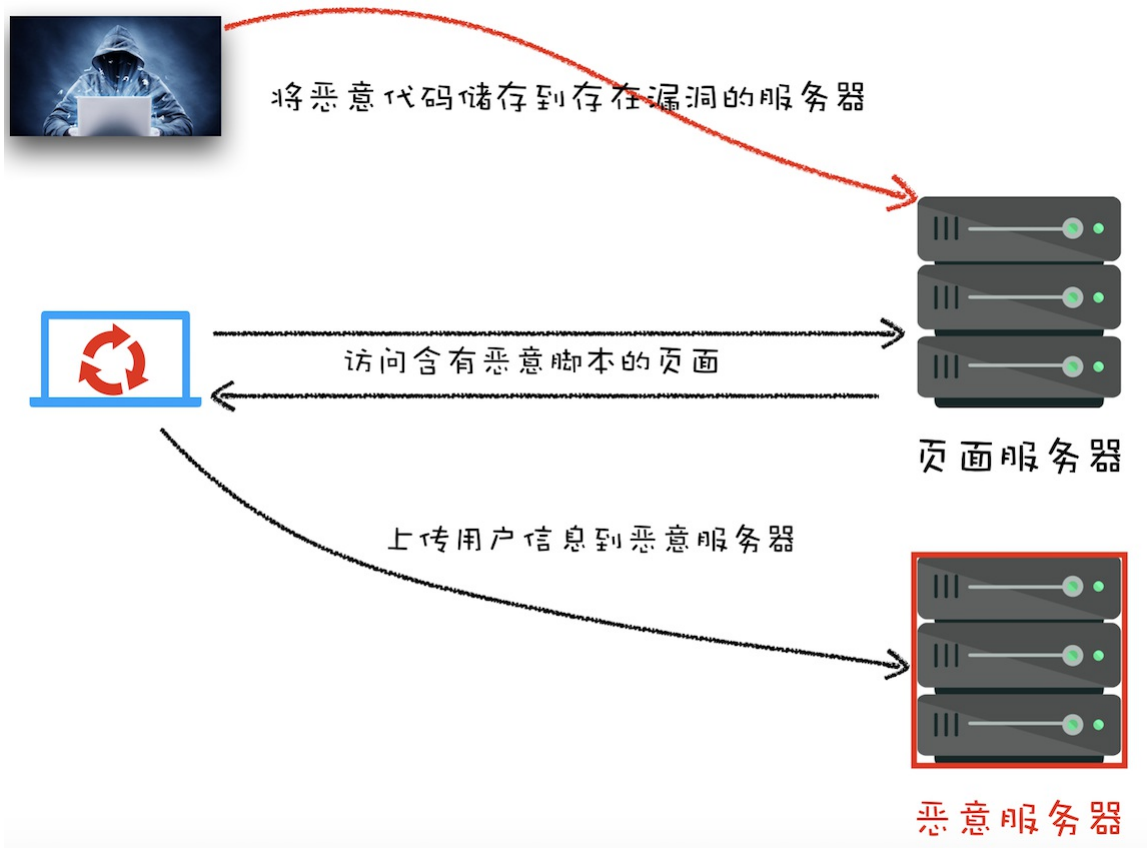
除了以上几种情况外，恶意脚本还能做很多其他的事情，这里就不一一介绍了。总之，如果让页面插入了恶意脚本，那么就相当于把我们页面的隐私数据和行为完全暴露给黑客了。

恶意脚本是怎么注入的

现在我们知道了页面中被注入恶意的JavaScript脚本是一件非常危险的事情，所以网站开发者会尽可能地避免页面中被注入恶意脚本。要想避免站点被注入恶意脚本，就要知道有哪些常见的注入方式。通常情况下，主要有**存储型XSS攻击**、**反射型XSS攻击**和**基于DOM的XSS攻击**三种方式来注入恶意脚本。

1. 存储型XSS攻击

我们先来看看存储型XSS攻击是怎么向HTML文件中注入恶意脚本的，你可以参考下图：



存储型XSS攻击

通过上图，我们可以看出存储型XSS攻击大致需要经过如下步骤：

- 首先黑客利用站点漏洞将一段恶意JavaScript代码提交到网站的数据库中；
- 然后用户向网站请求包含了恶意JavaScript脚本的页面；
- 当用户浏览该页面的时候，恶意脚本就会将用户的Cookie信息等数据上传到服务器。

下面我们来看个例子，2015年喜马拉雅就被曝出了存储型XSS漏洞。起因是在用户设置专辑名称时，服务器对关键字过滤不严格，比如可以将专辑名称设置为一段JavaScript，如下图所示：

黑客将恶意代码存储到漏洞服务器上

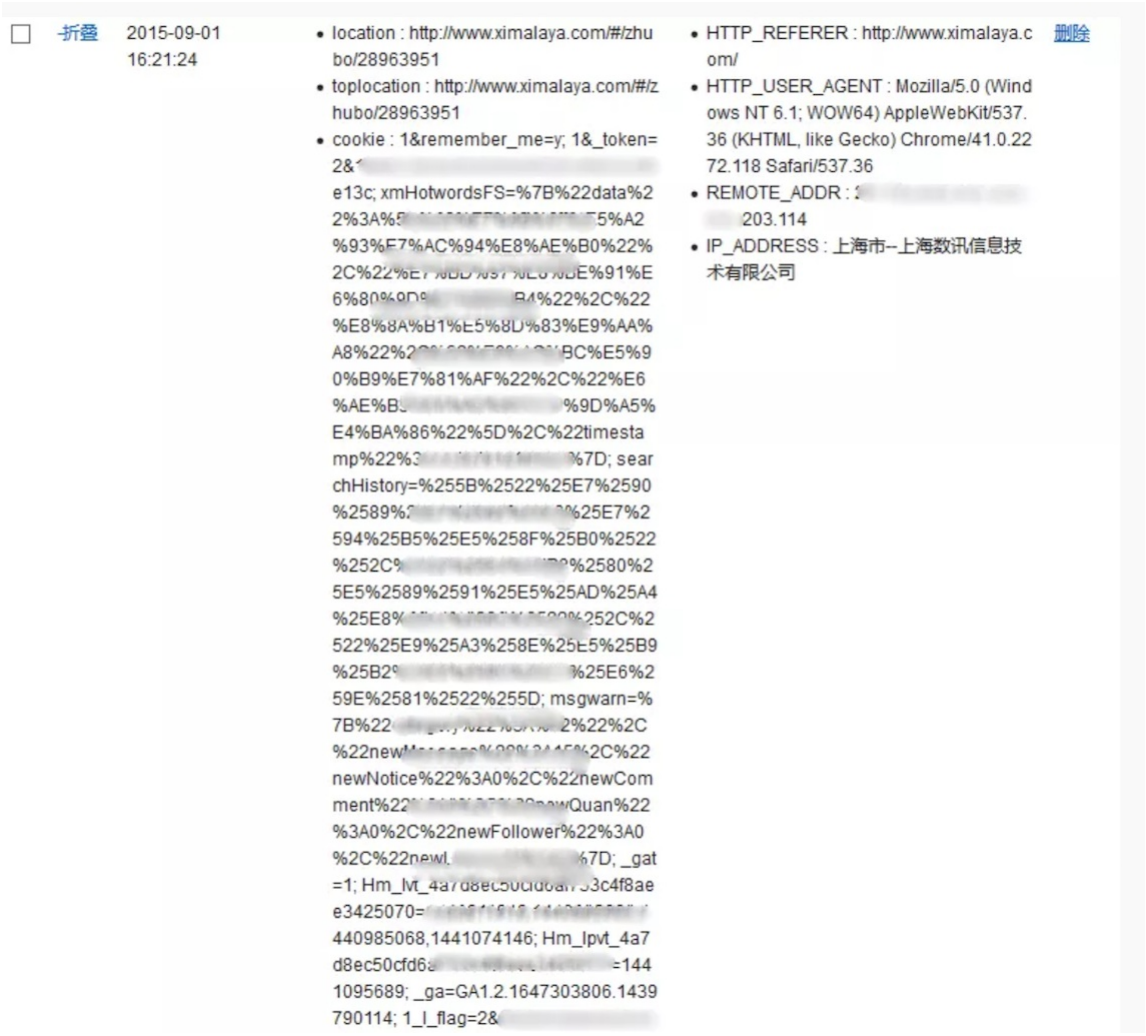
当黑客将专辑名称设置为一段JavaScript代码并提交时，喜马拉雅的服务器会保存该段JavaScript代码到数据库中。然后当用户打开黑客设置的专辑时，这段代码就会在用户的页面里执行（如下图），这样

就可以获取用户的Cookie等数据信息。

用户打开了含有恶意脚本的页面

当用户打开黑客设置的专辑页面时，服务器也会将这段恶意JavaScript代码返回给用户，因此这段恶意脚本就在用户的页面中执行了。

恶意脚本可以通过XMLHttpRequest或者Fetch将用户的Cookie数据上传到黑客的服务器，如下图所示：



将Cookie等数据上传到黑客服务器

黑客拿到了用户Cookie信息之后，就可以利用Cookie信息在其他机器上登录该用户的账号（如下图），并利用用户账号进行一些恶意操作。

黑客利用Cookie信息登录用户账户

以上就是存储型XSS攻击的一个典型案例，这是乌云网在2015年曝出来的，虽然乌云网由于某些原因被关停了，但是你依然可以通过这个站点来查看乌云网的一些备份信息。

2. 反射型XSS攻击

在一个反射型XSS攻击过程中，恶意JavaScript脚本属于用户发送给网站请求中的一部分，随后网站又把恶意JavaScript脚本返回给用户。当恶意JavaScript脚本在用户页面中被执行时，黑客就可以利用该脚本做一些恶意操作。

这样讲有点抽象，下面我们结合一个简单的Node服务程序来看看什么是反射型XSS。首先我们使用Node来搭建一个简单的页面环境，搭建好的服务代码如下所示：

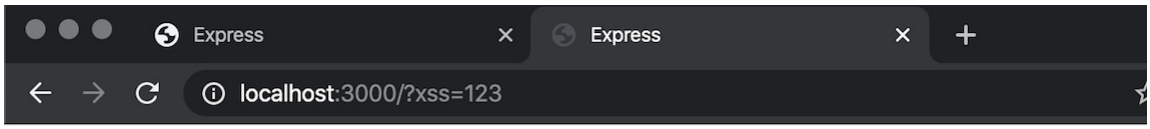
```
var express = require('express');
var router = express.Router();


/* GET home page. */
router.get('/', function(req, res, next) {
  res.render('index', { title: 'Express',xss:req.query.xss });
});
```

```
module.exports = router;
```

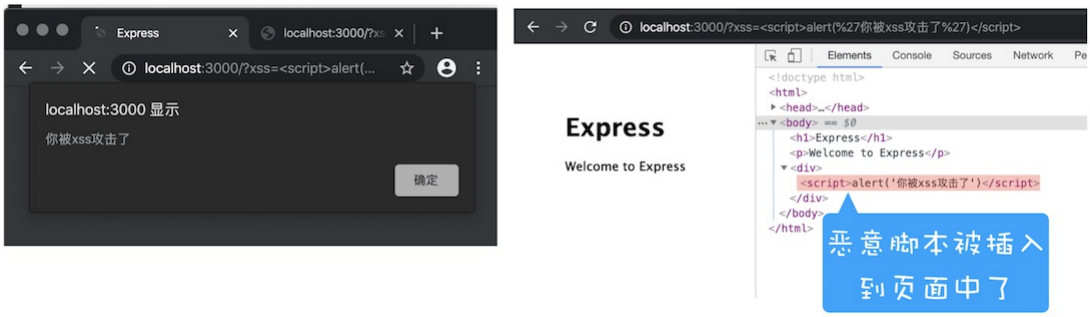
```
<!DOCTYPE html>
<html>
<head>
  <title><%= title %></title>
  <link rel='stylesheet' href='/stylesheets/style.css' />
</head>
<body>
  <h1><%= title %></h1>
  <p>Welcome to <%= title %></p>
  <div>
      <%- xss %>
  </div>
</body>
</html>
```

上面这两段代码，第一段是路由，第二段是视图，作用是将URL中xss参数的内容显示在页面。我们可以在本地演示下，比如打开`http://localhost:3000/?xss=123`这个链接，这样在页面中展示就是“123”了（如下图），是正常的，没有问题的。



正常打开页面

但当打开`http://localhost:3000/?xss=<script>alert('你被xss攻击了')</script>`这段URL时，其结果如下图所示：



反射型XSS攻击

通过这个操作，我们会发现用户将一段含有恶意代码的请求提交给Web服务器，Web服务器接收到请求时，又将恶意代码反射给了浏览器端，这就是反射型XSS攻击。在现实生活中，黑客经常会通过QQ群或者邮件等渠道诱导用户去点击这些恶意链接，所以对于一些链接我们一定要慎之又慎。

另外需要注意的是，**Web服务器不会存储反射型XSS攻击的恶意脚本，这是和存储型XSS攻击不同的地方**。

3. 基于DOM的XSS攻击

基于DOM的XSS攻击是不牵涉到页面Web服务器的。具体来讲，黑客通过各种手段将恶意脚本注入用户的页面中，比如通过网络劫持在页面传输过程中修改HTML页面的内容，这种劫持类型很多，有通过WiFi路由器劫持的，有通过本地恶意软件来劫持的，它们的共同点是在Web资源传输过程或者在用户使用页面的过程中修改Web页面的数据。

如何阻止XSS攻击

我们知道存储型XSS攻击和反射型XSS攻击都是需要经过Web服务器来处理的，因此可以认为这两种类型的漏洞是服务端的安全漏洞。而基于DOM的XSS攻击全部都是在浏览器端完成的，因此基于DOM的XSS攻击是属于前端的安全漏洞。

但无论是何种类型的XSS攻击，它们都有一个共同点，那就是首先往浏览器中注入恶意脚本，然后再通过恶意脚本将用户信息发送至黑客部署的恶意服务器上。

所以要阻止XSS攻击，我们可以通过阻止恶意JavaScript脚本的注入和恶意消息的发送来实现。

接下来我们就来看看一些常用的阻止XSS攻击的策略。

1. 服务器对输入脚本进行过滤或转码

不管是反射型还是存储型XSS攻击，我们都可以在服务器端将一些关键的字符进行转码，比如最典型的：

```
code:<script>alert('你被xss攻击了')</script>
```

这段代码过滤后，只留下了：

```
code:
```

这样，当用户再次请求该页面时，由于`<script>`标签的内容都被过滤了，所以这段脚本在客户端是不可能被执行的。

除了过滤之外，服务器还可以对这些内容进行转码，还是上面那段代码，经过转码之后，效果如下所示：

```
code:&lt;script&gt;alert(&#39;你被xss攻击了&#39;)&lt;/script&gt;
```

经过转码之后的内容，如`<script>`标签被转换为`<script>`，因此即使这段脚本返回给页面，页面也不会执行这段脚本。

2. 充分利用CSP

虽然在服务器端执行过滤或者转码可以阻止 XSS 攻击的发生，但完全依靠服务器端依然是不够的，我们还需要把CSP等策略充分地利用起来，以降低 XSS攻击带来的风险和后果。

实施严格的CSP可以有效地防范XSS攻击，具体来讲CSP有如下几个功能：

- 限制加载其他域下的资源文件，这样即使黑客插入了一个JavaScript文件，这个JavaScript文件也是无法被加载的；
- 禁止向第三方域提交数据，这样用户数据也不会外泄；
- 禁止执行内联脚本和未授权的脚本；
- 还提供了上报机制，这样可以帮助我们尽快发现有哪些XSS攻击，以便尽快修复问题。

因此，利用好CSP能够有效降低XSS攻击的概率。

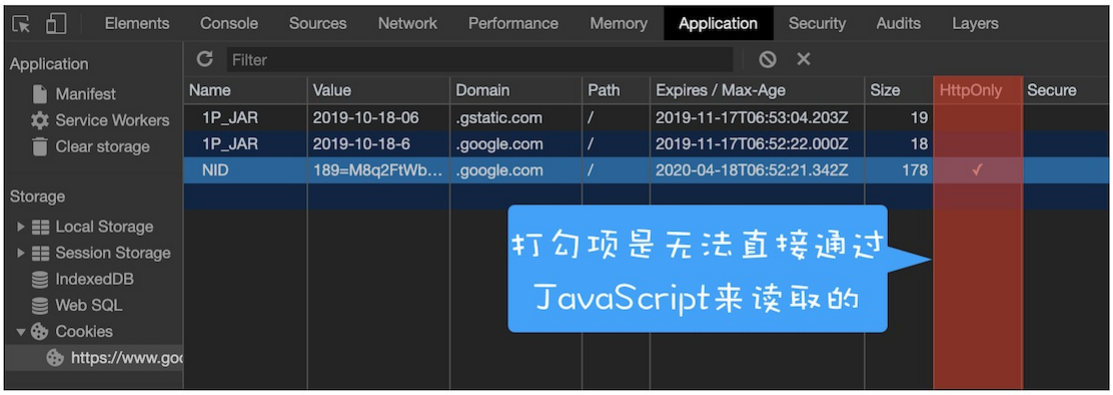
3. 使用HttpOnly属性

由于很多XSS攻击都是来盗用Cookie的，因此还可以通过使用HttpOnly属性来保护我们Cookie的安全。

通常服务器可以将某些Cookie设置为HttpOnly标志，HttpOnly是服务器通过HTTP响应头来设置的，下面是打开Google时，HTTP响应头中的一段：

```
set-cookie: NID=189=M8q2FtWbsR8RlcldPVt7qkrqR38LmFY9jUxkKo3-4Bi6Qu_ocNOat7nkYZUTzolHjFnwBw0izgsATSI7TZyiiiaV94qGh-BzEYsNVa7TZmjAYTxYTOM9L_-0CN9ipL6cXi8l6-z41asXtm2uEwcOC5oh9djkff(
```

我们可以看到，set-cookie属性值最后使用了HttpOnly来标记该Cookie。顾名思义，使用HttpOnly标记的Cookie只能使用在HTTP请求过程中，所以无法通过JavaScript来读取这段Cookie。我们还可以通过Chrome开发者工具来查看哪些Cookie被标记了HttpOnly，如下图：



HttpOnly演示

从图中可以看出，NID这个Cookie的HttpOlny属性是被勾选上的，所以NID的内容是无法通过document.cookie是来读取的。

由于JavaScript无法读取设置了HttpOnly的Cookie数据，所以即使页面被注入了恶意JavaScript脚本，也是无法获取到设置了HttpOnly的数据。因此一些比较重要的数据我们建议设置HttpOnly标志。

总结

好了，今天我们就介绍到这里，下面我来总结下本文的主要内容。

XSS攻击就是黑客往页面中注入恶意脚本，然后将页面的一些重要数据上传到恶意服务器。常见的三种XSS攻击模式是存储型XSS攻击、反射型XSS攻击和基于DOM的XSS攻击。

这三种攻击方式的共同点是都需要往用户的页面中注入恶意脚本，然后再通过恶意脚本将用户数据上传到黑客的恶意服务器上。而三者的不同点在于注入的方式不一样，有通过服务器漏洞来进行注入的，还有在客户端直接注入的。

针对这些XSS攻击，主要有三种防范策略，第一种是通过服务器对输入的内容进行过滤或者转码，第二种是充分利用好CSP，第三种是使用HttpOnly来保护重要的Cookie信息。

当然除了以上策略之外，我们还可以通过添加验证码防止脚本冒充用户提交危险操作。而对于一些不受信任的输入，还可以限制其输入长度，这样可以增大XSS攻击的难度。

思考时间

今天留给你的思考题是：你认为前端开发者对XSS攻击应该负多大责任？

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

通过上篇文章的介绍，我们知道了同源策略可以隔离各个站点之间的DOM交互、页面数据和网络通信，虽然严格的同源策略会带来更多的安全，但是也束缚了Web。这就需要在安全和自由之间找到一个平衡点，所以我们默认页面中可以引用任意第三方资源，然后又引入CSP策略来加以限制；默认XMLHttpRequest和Fetch不能跨站请求资源，然后又通过CORS策略来支持其跨域。

不过支持页面中的第三方资源引用和CORS也带来了很多安全问题，其中最典型的就是XSS攻击。

什么是XSS攻击

XSS全称是Cross Site Scripting，为了与“CSS”区分开来，故简称XSS，翻译过来就是“跨站脚本”。XSS攻击是指黑客往HTML文件中或者DOM中注入恶意脚本，从而在用户浏览页面时利用注入的恶意脚本对用户实施攻击的一种手段。

最开始的时候，这种攻击是通过跨域来实现的，所以叫“跨域脚本”。但是发展到现在，往HTML文件中注入恶意代码的方式越来越多了，所以是否跨域注入脚本已经不是唯一的注入手段了，但是XSS这个名字却一直保留至今。

当页面被注入了恶意JavaScript脚本时，浏览器无法区分这些脚本是被恶意注入的还是正常的页面内容，所以恶意注入JavaScript脚本也拥有所有的脚本权限。下面我们就来看看，如果页面被注入了恶意JavaScript脚本，恶意脚本都能做哪些事情。

- 可以**窃取Cookie信息**。恶意JavaScript可以通过“document.cookie”获取Cookie信息，然后通过XMLHttpRequest或者Fetch加上CORS功能将数据发送给恶意服务器；恶意服务器拿到用户的Cookie信息之后，就可以在其他电脑上模拟用户的登录，然后进行转账等操作。
- 可以**监听用户行为**。恶意JavaScript可以使用“addEventListener”接口来监听键盘事件，比如可以获取用户输入的信用卡等信息，将其发送到恶意服务器。黑客掌握了这些信息之后，又可以做很多违法的事情。
- 可以通过**修改DOM**伪造假的登录窗口，用来欺骗用户输入用户名和密码等信息。
- 还可以**在页面内生成浮窗广告**，这些广告会严重地影响用户体验。

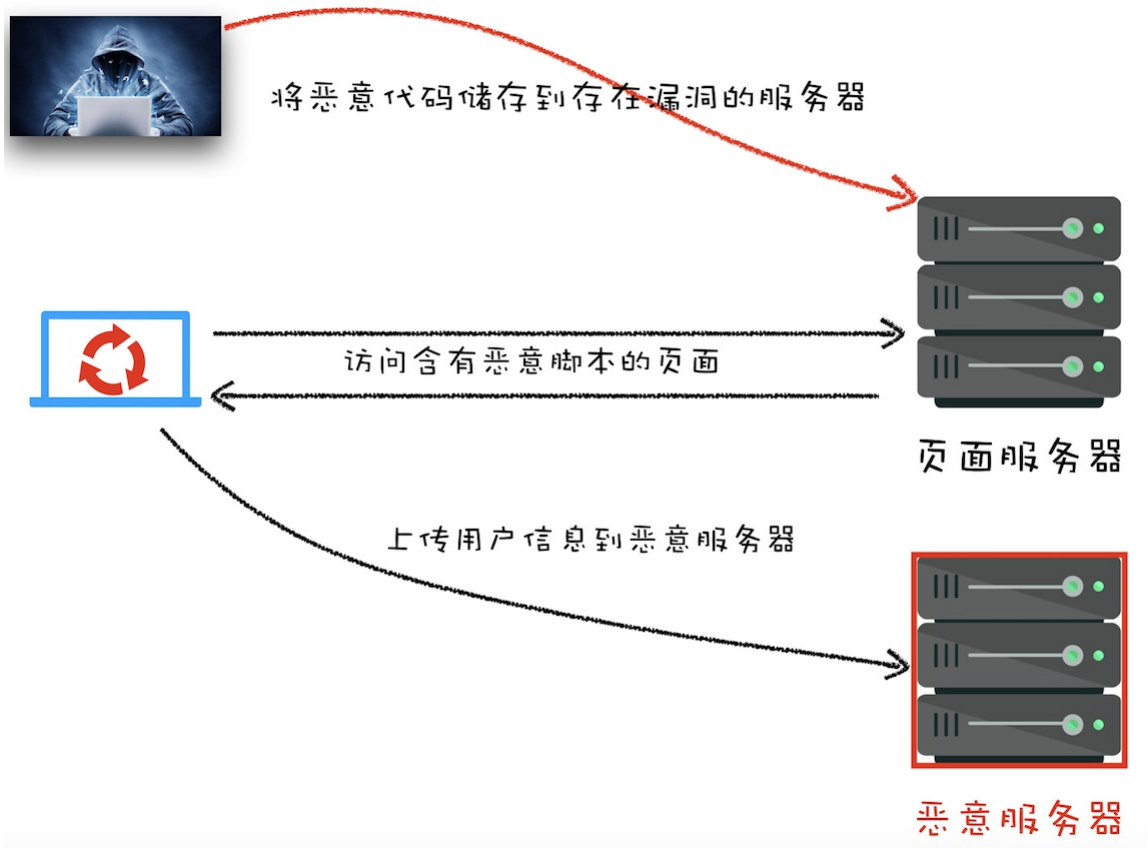
除了以上几种情况外，恶意脚本还能做很多其他的事情，这里就不一一介绍了。总之，如果让页面插入了恶意脚本，那么就相当于把我们页面的隐私数据和行为完全暴露给黑客了。

恶意脚本是怎么注入的

现在我们知道了页面中被注入恶意的JavaScript脚本是一件非常危险的事情，所以网站开发者会尽可能地避免页面中被注入恶意脚本。要想避免站点被注入恶意脚本，就要知道有哪些常见的注入方式。通常情况下，主要有**存储型XSS攻击**、**反射型XSS攻击**和**基于DOM的XSS攻击**三种方式来注入恶意脚本。

1. 存储型XSS攻击

我们先来看看存储型XSS攻击是怎么向HTML文件中注入恶意脚本的，你可以参考下图：



存储型XSS攻击

通过上图，我们可以看出存储型XSS攻击大致需要经过如下步骤：

- 首先黑客利用站点漏洞将一段恶意JavaScript代码提交到网站的数据库中；
- 然后用户向网站请求包含了恶意JavaScript脚本的页面；
- 当用户浏览该页面的时候，恶意脚本就会将用户的Cookie信息等数据上传到服务器。

下面我们来看个例子，2015年喜马拉雅就被曝出了存储型XSS漏洞。起因是在用户设置专辑名称时，服务器对关键字过滤不严格，比如可以将专辑名称设置为一段JavaScript，如下图所示：

黑客将恶意代码存储到漏洞服务器上

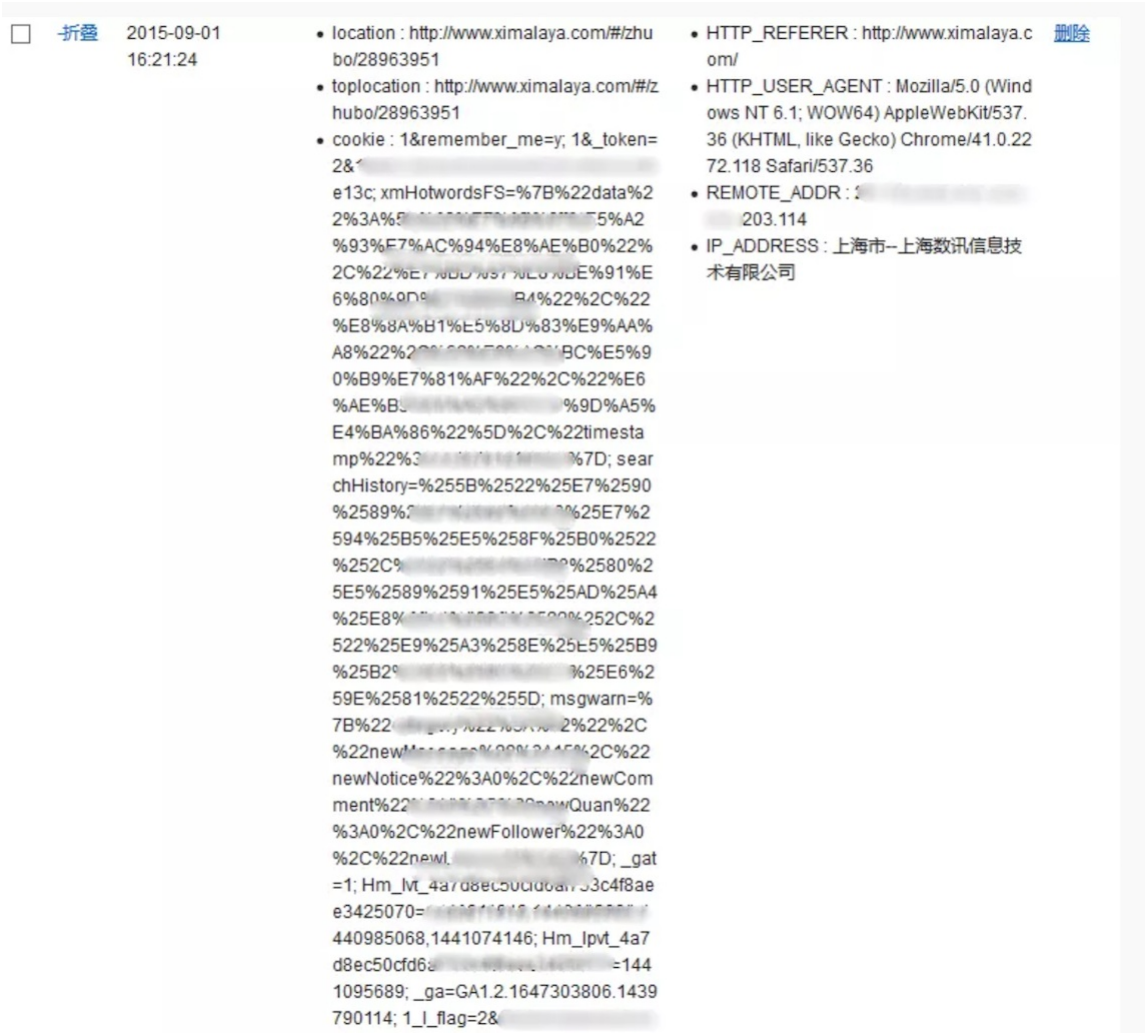
当黑客将专辑名称设置为一段JavaScript代码并提交时，喜马拉雅的服务器会保存该段JavaScript代码到数据库中。然后当用户打开黑客设置的专辑时，这段代码就会在用户的页面里执行（如下图），这样

就可以获取用户的Cookie等数据信息。

用户打开了含有恶意脚本的页面

当用户打开黑客设置的专辑页面时，服务器也会将这段恶意JavaScript代码返回给用户，因此这段恶意脚本就在用户的页面中执行了。

恶意脚本可以通过XMLHttpRequest或者Fetch将用户的Cookie数据上传到黑客的服务器，如下图所示：



将Cookie等数据上传到黑客服务器

黑客拿到了用户Cookie信息之后，就可以利用Cookie信息在其他机器上登录该用户的账号（如下图），并利用用户账号进行一些恶意操作。

黑客利用Cookie信息登录用户账户

以上就是存储型XSS攻击的一个典型案例，这是乌云网在2015年曝出来的，虽然乌云网由于某些原因被关停了，但是你依然可以通过这个站点来查看乌云网的一些备份信息。

2. 反射型XSS攻击

在一个反射型XSS攻击过程中，恶意JavaScript脚本属于用户发送给网站请求中的一部分，随后网站又把恶意JavaScript脚本返回给用户。当恶意JavaScript脚本在用户页面中被执行时，黑客就可以利用该脚本做一些恶意操作。

这样讲有点抽象，下面我们结合一个简单的Node服务程序来看看什么是反射型XSS。首先我们使用Node来搭建一个简单的页面环境，搭建好的服务代码如下所示：

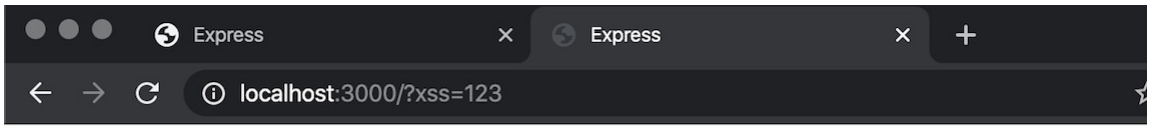
```
var express = require('express');
var router = express.Router();


/* GET home page. */
router.get('/', function(req, res, next) {
  res.render('index', { title: 'Express',xss:req.query.xss });
});
```

```
module.exports = router;
```

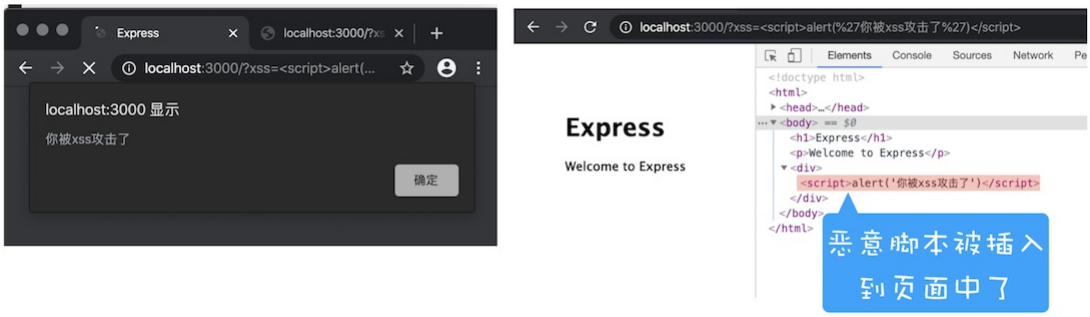
```
<!DOCTYPE html>
<html>
<head>
  <title><%= title %></title>
  <link rel='stylesheet' href='/stylesheets/style.css' />
</head>
<body>
  <h1><%= title %></h1>
  <p>Welcome to <%= title %></p>
  <div>
      <%- xss %>
  </div>
</body>
</html>
```

上面这两段代码，第一段是路由，第二段是视图，作用是将URL中xss参数的内容显示在页面。我们可以在本地演示下，比如打开`http://localhost:3000/?xss=123`这个链接，这样在页面中展示就是“123”了（如下图），是正常的，没有问题的。



正常打开页面

但当打开`http://localhost:3000/?xss=<script>alert('你被xss攻击了')</script>`这段URL时，其结果如下图所示：



反射型XSS攻击

通过这个操作，我们会发现用户将一段含有恶意代码的请求提交给Web服务器，Web服务器接收到请求时，又将恶意代码反射给了浏览器端，这就是反射型XSS攻击。在现实生活中，黑客经常会通过QQ群或者邮件等渠道诱导用户去点击这些恶意链接，所以对于一些链接我们一定要慎之又慎。

另外需要注意的是，**Web服务器不会存储反射型XSS攻击的恶意脚本，这是和存储型XSS攻击不同的地方**。

3. 基于DOM的XSS攻击

基于DOM的XSS攻击是不牵涉到页面Web服务器的。具体来讲，黑客通过各种手段将恶意脚本注入用户的页面中，比如通过网络劫持在页面传输过程中修改HTML页面的内容，这种劫持类型很多，有通过WiFi路由器劫持的，有通过本地恶意软件来劫持的，它们的共同点是在Web资源传输过程或者在用户使用页面的过程中修改Web页面的数据。

如何阻止XSS攻击

我们知道存储型XSS攻击和反射型XSS攻击都是需要经过Web服务器来处理的，因此可以认为这两种类型的漏洞是服务端的安全漏洞。而基于DOM的XSS攻击全部都是在浏览器端完成的，因此基于DOM的XSS攻击是属于前端的安全漏洞。

但无论是何种类型的XSS攻击，它们都有一个共同点，那就是首先往浏览器中注入恶意脚本，然后再通过恶意脚本将用户信息发送至黑客部署的恶意服务器上。

所以要阻止XSS攻击，我们可以通过阻止恶意JavaScript脚本的注入和恶意消息的发送来实现。

接下来我们就来看看一些常用的阻止XSS攻击的策略。

1. 服务器对输入脚本进行过滤或转码

不管是反射型还是存储型XSS攻击，我们都可以在服务器端将一些关键的字符进行转码，比如最典型的：

```
code:<script>alert('你被xss攻击了')</script>
```

这段代码过滤后，只留下了：

```
code:
```

这样，当用户再次请求该页面时，由于`<script>`标签的内容都被过滤了，所以这段脚本在客户端是不可能被执行的。

除了过滤之外，服务器还可以对这些内容进行转码，还是上面那段代码，经过转码之后，效果如下所示：

```
code:&lt;script&gt;alert(&#39;你被xss攻击了&#39;)&lt;/script&gt;
```

经过转码之后的内容，如`<script>`标签被转换为`&lt;script&gt;`，因此即使这段脚本返回给页面，页面也不会执行这段脚本。

2. 充分利用CSP

虽然在服务器端执行过滤或者转码可以阻止 XSS 攻击的发生，但完全依靠服务器端依然是不够的，我们还需要把CSP等策略充分地利用起来，以降低 XSS攻击带来的风险和后果。

实施严格的CSP可以有效地防范XSS攻击，具体来讲CSP有如下几个功能：

- 限制加载其他域下的资源文件，这样即使黑客插入了一个JavaScript文件，这个JavaScript文件也是无法被加载的；
- 禁止向第三方域提交数据，这样用户数据也不会外泄；
- 禁止执行内联脚本和未授权的脚本；
- 还提供了上报机制，这样可以帮助我们尽快发现有哪些XSS攻击，以便尽快修复问题。

因此，利用好CSP能够有效降低XSS攻击的概率。

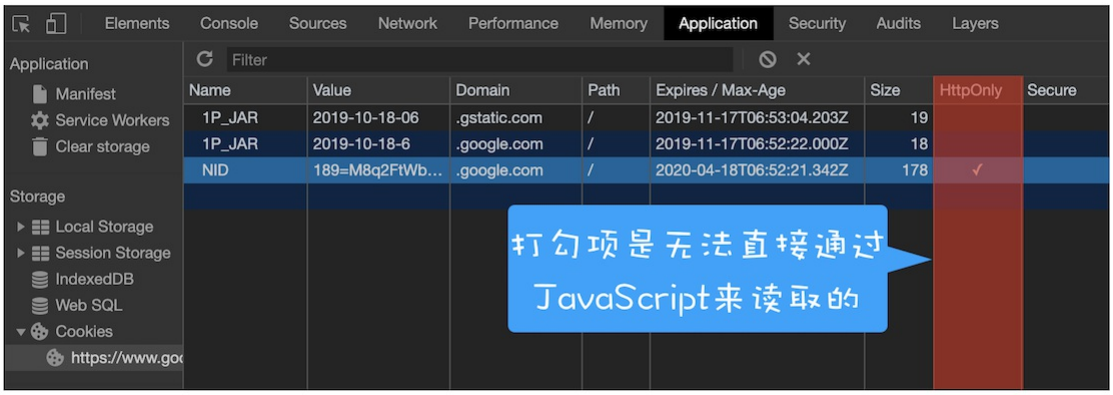
3. 使用HttpOnly属性

由于很多XSS攻击都是来盗用Cookie的，因此还可以通过使用HttpOnly属性来保护我们Cookie的安全。

通常服务器可以将某些Cookie设置为HttpOnly标志，HttpOnly是服务器通过HTTP响应头来设置的，下面是打开Google时，HTTP响应头中的一段：

```
set-cookie: NID=189=M8q2FtWbsR8RlcldPVt7qkrqR38LmFY9jUxkKo3-4Bi6Qu_ocNOat7nkYZUTzolHjFnwBw0izgsATSI7TZyiiiaV94qGh-BzEYsNVa7TZmjAYTxYTOM9L_-0CN9ipL6cXi8l6-z41asXtm2uEwcOC5oh9djkff(
```

我们可以看到，set-cookie属性值最后使用了HttpOnly来标记该Cookie。顾名思义，使用HttpOnly标记的Cookie只能使用在HTTP请求过程中，所以无法通过JavaScript来读取这段Cookie。我们还可以通过Chrome开发者工具来查看哪些Cookie被标记了HttpOnly，如下图：



HttpOnly演示

从图中可以看出，NID这个Cookie的HttpOlny属性是被勾选上的，所以NID的内容是无法通过document.cookie是来读取的。

由于JavaScript无法读取设置了HttpOnly的Cookie数据，所以即使页面被注入了恶意JavaScript脚本，也是无法获取到设置了HttpOnly的数据。因此一些比较重要的数据我们建议设置HttpOnly标志。

总结

好了，今天我们就介绍到这里，下面我来总结下本文的主要内容。

XSS攻击就是黑客往页面中注入恶意脚本，然后将页面的一些重要数据上传到恶意服务器。常见的三种XSS攻击模式是存储型XSS攻击、反射型XSS攻击和基于DOM的XSS攻击。

这三种攻击方式的共同点是都需要往用户的页面中注入恶意脚本，然后再通过恶意脚本将用户数据上传到黑客的恶意服务器上。而三者的不同点在于注入的方式不一样，有通过服务器漏洞来进行注入的，还有在客户端直接注入的。

针对这些XSS攻击，主要有三种防范策略，第一种是通过服务器对输入的内容进行过滤或者转码，第二种是充分利用好CSP，第三种是使用HttpOnly来保护重要的Cookie信息。

当然除了以上策略之外，我们还可以通过添加验证码防止脚本冒充用户提交危险操作。而对于一些不受信任的输入，还可以限制其输入长度，这样可以增大XSS攻击的难度。

思考时间

今天留给你的思考题是：你认为前端开发者对XSS攻击应该负多大责任？

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。