### 跨站脚本的含义

XSS根据其特征和利用手法的不同，主要分成两大类型：一种是反射式跨站脚本；另一种是持久式跨站脚本。

1. **反射式XSS**

反射式跨站脚本也称作非持久型、参数型跨站脚本。主要用于将恶意脚本附加到URL地址的参数中，下面是一个简单的存在漏洞的php页面：

<?php

if(!array\_key\_exists ("name", $\_GET) || $\_GET['name'] == NULL || $\_GET['name'] == '')

{

$isempty = true;

} else {

echo '<pre>';

echo 'Hello ' . $\_GET['name'];

echo '</pre>';

}

?>

这个php页面将传入的参数name未经过有效性检验而直接写入到响应结果中，所以这个页面容易受到XSS攻击。如果攻击者输入如下脚本，如下图11-11:

<script>alert(‘xss’)</script>

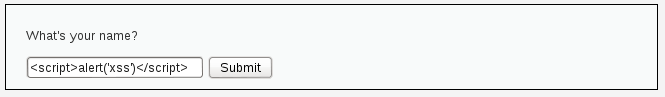


图11-11 输入指定脚本内容

这时就会弹出一个对话框，如下图11-12：



图11-12 执行结果弹出对话框

可以看出，传入的脚本在客户端服务器中得以执行。这个警告框证明此Web应用程序存在反射式XSS漏洞。

1. **存储式XSS**

存储式跨站脚本又称为持久型跨站脚本，比反射式跨站脚本更具有威胁性，并且可能影响到Web服务器自身的安全。

存储式XSS与反射式XSS类似的地方在于，会在Web应用程序的网页中显示未经编码的攻击者脚本。它们的区别在于，存储式XSS中的脚本并非来自于Web应用程序请求；相反，脚本是由Web应用程序进行存储的，并且会将其其作为内容显示给浏览用户。例如，如果论坛或博客网站允许用户上传内容而不进行适当的有效性检查或编码，那么这个网站就容易受到存储式XSS攻击。

下面来看一个存储式XSS漏洞的示例，如下图11-13：

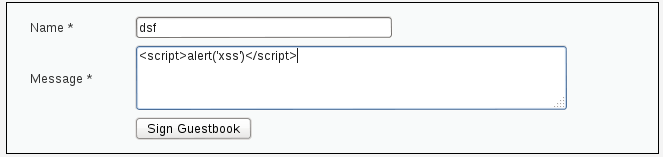


图11-13 向留言板提交攻击脚本实现存储式XSS漏洞

在这个示例中，我们向该留言板提交攻击脚本，该脚本会存储在其后台数据库服务器，每当用户查看留言板时，则会弹出对话框，如下图11-14：

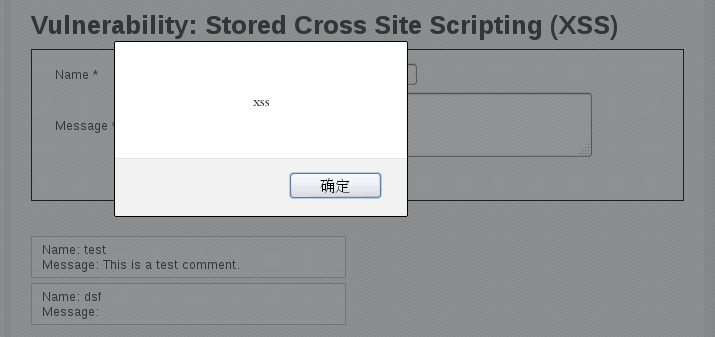


图11-14 当用户查看留言板时，弹出对话框

1. **XSS的攻击途径**

上面演示的XSS攻击只是显示一个警告框，但是在现实的攻击案例中，攻击者有可能进行更具破坏性的攻击。例如。恶意脚本可以将cookie值上传到攻击者的网站，从而有可能让攻击者以该用户的身份登入或恢复正在进行中的会话。脚本还可以改写页面内容，使其看上去已经被涂鸦。JavaScript还可以轻易地实施下面的任何攻击：

（1）通过cookie窃取实现会话劫持

（2）按键记录，将所有输入的文本发送到攻击者网站

（3）网站涂改

（4）向网页中注入链接或广告

（5）立即将网页重定向到恶意网站

（6）窃取用户登录 凭证等等。

### 跨站脚本攻击的危害

跨站脚本攻击(XSS)是当前web应用中最危险和最普遍的漏洞之一。安全研究人员在大部分最受欢迎的网站，包括Google、Facebook、Amazon、PayPal等网站都发现这个漏洞。如果你密切关注bug赏金计划，会发现报道最多的问题属于XSS。

XSS通常用于发动cookie窃取、恶意软件传播(蠕虫攻击)、会话劫持、恶意重定向等。在这种攻击中，攻击者将恶意JavaScript代码注入到网站页面中，这样受害者的浏览器就会执行攻击者编写的恶意脚本。

**一般来说，存储式XSS的风险会高于反射式XSS**。因为存储式XSS会保存在服务器上，有可能会跨页面存在。它不改变页面URL的原有结构，因此有时候还能逃过一些IDS的检测。比如IE8的XSS Filter和Firefox的Noscript Extension，都会检查地址栏中的地址是否包含XSS脚本。而跨页面的存储式XSS可能会绕过这些检测工具。

从攻击过程来说，反射式XSS一般要求攻击者诱使用户点击一个包含XSS代码的URL链接；而存储式XSS则只需让用户查看一个正常的URL链接，而这个链接中存储了一段脚本。比如一个Web邮箱的邮件正文页面存在一个存储式XSS漏洞，当用户打开一封新邮件时，XSS Payload会被执行。这样的漏洞极其隐蔽，且埋伏在用户的正常业务中，风险颇高。

**实验三：对如下示例代码的php网页进行XSS攻击，实现简单的弹窗效果即可。**

<!DOCTYPE html>

<head>

<meta http-equiv="content-type" content="text/html;charset=utf-8">

<script>

window.alert = function()

{

confirm("Congratulations~");

}

</script>

</head>

<body>

<h1 align=center>--Welcome To The Simple XSS Test--</h1>

<?php

ini\_set("display\_errors", 0);

$str =strtolower( $\_GET["keyword"]);

$str2=str\_replace("script","",$str);

$str3=str\_replace("on","",$str2);

$str4=str\_replace("src","",$str3);

echo "<h2 align=center>Hello ".htmlspecialchars($str).".</h2>".'<center>

<form action=xss\_test.php method=GET>

<input type=submit name=submit value=Submit />

<input name=keyword value="'.$str4.'">

</form>

</center>';

?>

</body>

</html>

为了读者方便实验，编者已经将页面全部的源码给出。但是编者将从黑盒和白盒两个角度来进行实验的讲述。

首先从黑盒测试的角度来进行实验，访问URL：<http://192.168.19.131/xss_test.php>

页面显示效果如下图11-15：



图11-15 访问URL的页面效果

如图11-15可以看到一个Submit按钮和输入框，并且还有标题提示XSS。于是输入上面学过最简单的XSS脚本：<script>alert('xss')</script>来进行测试。点击Submit按钮以后，效果如下图11-16：



图11-16 输入测试XSS脚本

结果发现Hello后面出现了我们输入的内容，并且输入框中的回显过滤了script关键字，这个时候考虑后台只是最简单的一次过滤。于是可以利用双写关键字绕过，构造脚本：

<scrscriptipt>alert('xss')</scscriptript>测试。执行效果如下图11-17：



图11-17 输入测试XSS脚本

发现虽然输入框中的回显确实是我们想要攻击的脚本，但是代码并没有执行。因为在黑盒测试情况下，我们并不能看到全部代码的整个逻辑，所以无法判断问题到底出在哪里。这个时候我们可以在页面点击右键查看源码，尝试从源码片段中分析问题。右键源码如下图11-18：

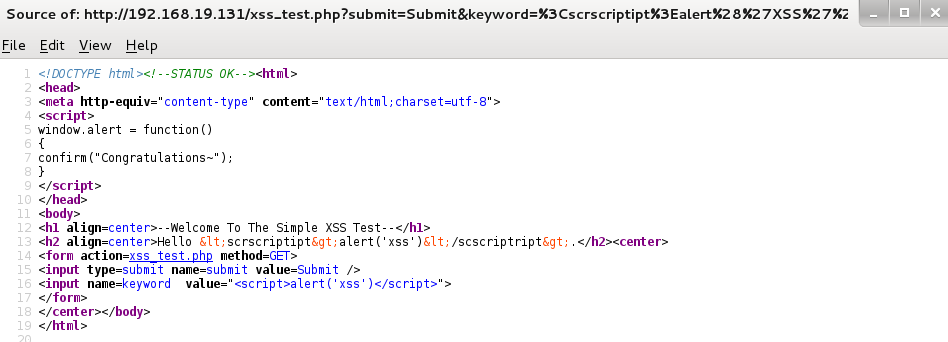


图11-18 右键查看源码

刚开始就会看到第5行重写的alert函数。如果可以成功执行alert函数的话，页面将会跳出一个确认框，显示Congratulations~。这应该是我们XSS成功攻击的~~的~~标志。

接着往下查看16行的<input>标签，我们唯一能输入且有可能控制的地方。

<input name=keyword value="<script>alert('xss')</script>">

分析这行代码知道，虽然我们成功的插入了<script></script>标签组,但是我们并没有跳出input的标签，使得我们的脚本仅仅可以回显而不能利用。这个时候的思路就是想办法将前面的<input>标签闭合，于是构造如下脚本：

"><scrscriptipt>alert('XSS')</scscriptript><!—

弹处确认框，XSS攻击成功。执行效果如下图11-19：

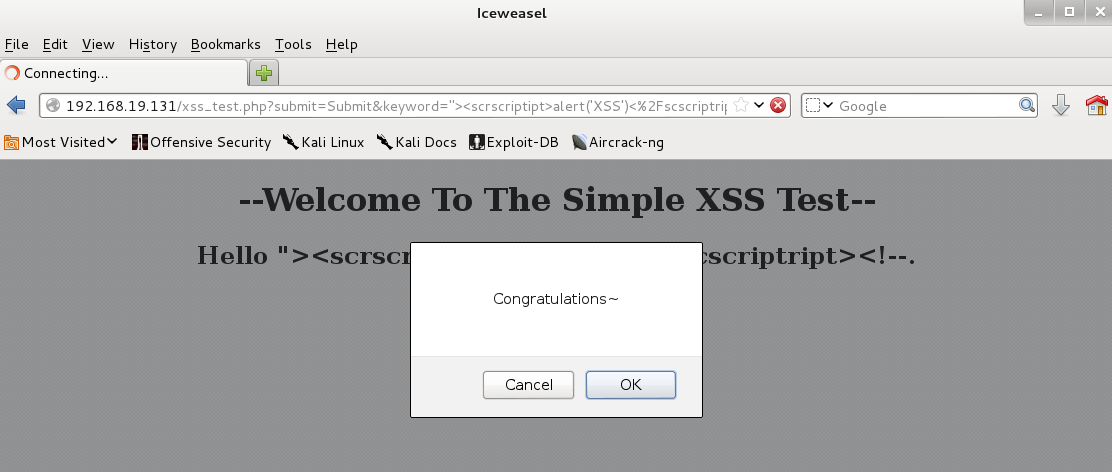


图11-19 XSS攻击成功

**重要提醒**：如果实践过程出现错误，通常表现为输入的双引号不能正常被处理，是因为php服务器自动会对输入的双引号等进行转义，以预防用户构造特殊输入进行攻击，比如本实验所进行的攻击。为了确保实验可以成功运行，请在phpnow安装目录下搜索文件php-apache2handler.ini，并将“magic\_quotes\_gpc = On”设置为“magic\_quotes\_gpc = Off”。

这个时候我们再来查看一下页面源码，如图11-20，仔细查看一下16行代码执行的逻辑：

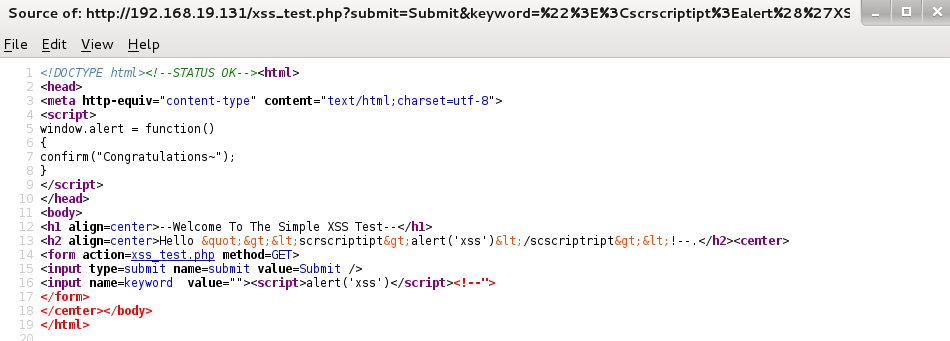


图11-20 右键查看源码

<input name=keyword value="**"><script>alert('xss')</script><!--**">

其实很简单，如上所示，其中粗体是我们成功构造的脚本，**">** 用来闭合前面的**<input>**标签。而 **<!--** 其实是为了美观，用来注释掉后面不需要的 **">** ,否则页面就会在输入框后面回显 **">** ，这里读者可以自行测试。

接下来，我们从源码的角度来看一下页面的核心逻辑。

<?php

ini\_set("display\_errors", 0);

$str =strtolower( $\_GET["keyword"]);

$str2=str\_replace("script","",$str);

$str3=str\_replace("on","",$str2);

$str4=str\_replace("src","",$str3);

echo "<h2 align=center>Hello ".htmlspecialchars($str).".</h2>".'<center>

<form action=xss\_test.php method=GET>

<input type=submit name=submit value=Submit />

<input name=keyword value="'.$str4.'">

</form>

</center>';

?>

发现跟我们上面黑盒测试的情况差不多，但是也有没测试到的地方。比如，Hello 后面显示的值是经过小写转换的。输入框中回显值的过滤方法是将script、on、src等关键字都替换成了空，其实过滤的内容并不是很多。这也会导致攻击脚本的构造方法多种多样。

这里就再为大家提供一种使用<img>标签的脚本构造方法：

<img src=ops! onerror="alert('XSS')">

<img>标签是用来定义HTML中的图像，src一般是图像的来源。而onerror 事件会在文档或图像加载过程中发生错误时被触发。所以上面这个攻击脚本的逻辑是，当img加载一个错误的图像来源ops!时，会触发onerror事件，从而执行alert函数。

读者可以根据本实验源码中过滤的内容将上述Payload加工一番，就可以成功弹窗了。其他的Payload也无非就是利用一些标签和事件组合构造的，本质是不变的，有兴趣的同学可以自行搜集资料进行测试。