# 《软件安全》实验报告

姓名：许洋　学号：2313721 班级：1070

## 实验名称：

跨站脚本攻击

## 实验要求：

复现课本第十一章实验三，通过$img$和$script$两类方式实现跨站脚本攻击，撰写实验报告。有能力者可以自己撰写更安全的过滤程序。

## 实验过程：

### Script方式

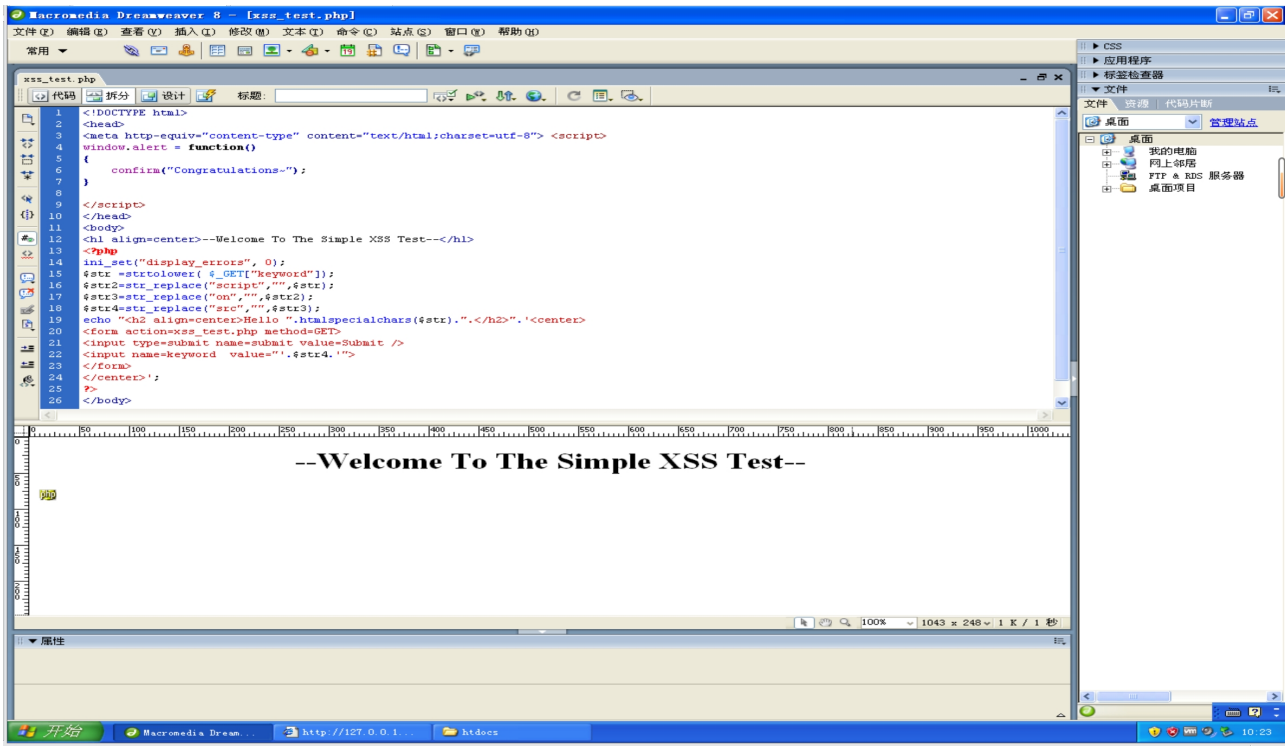
#### 建立Dreamweaver文件

首先我们输入源代码：

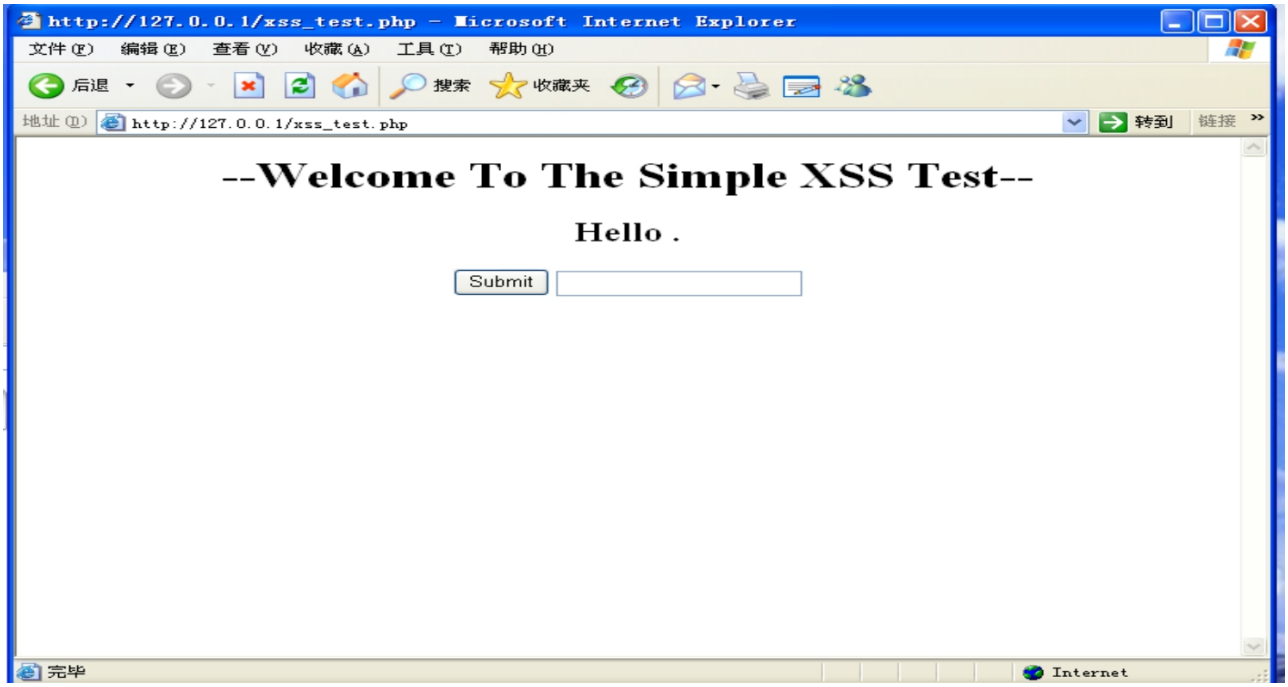
```
<!DOCTYPE html>
<head>
<meta http-equiv="content-type" content="text/html;charset=utf-8">
<script>
window.alert = function()
{
    confirm("Congratulations~");
}

</script>
</head>
<body>
<h1 align=center>--Welcome To The Simple XSS Test--</h1>
```

```php
<?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>
```

我们将其输入到软件中，显示如下：

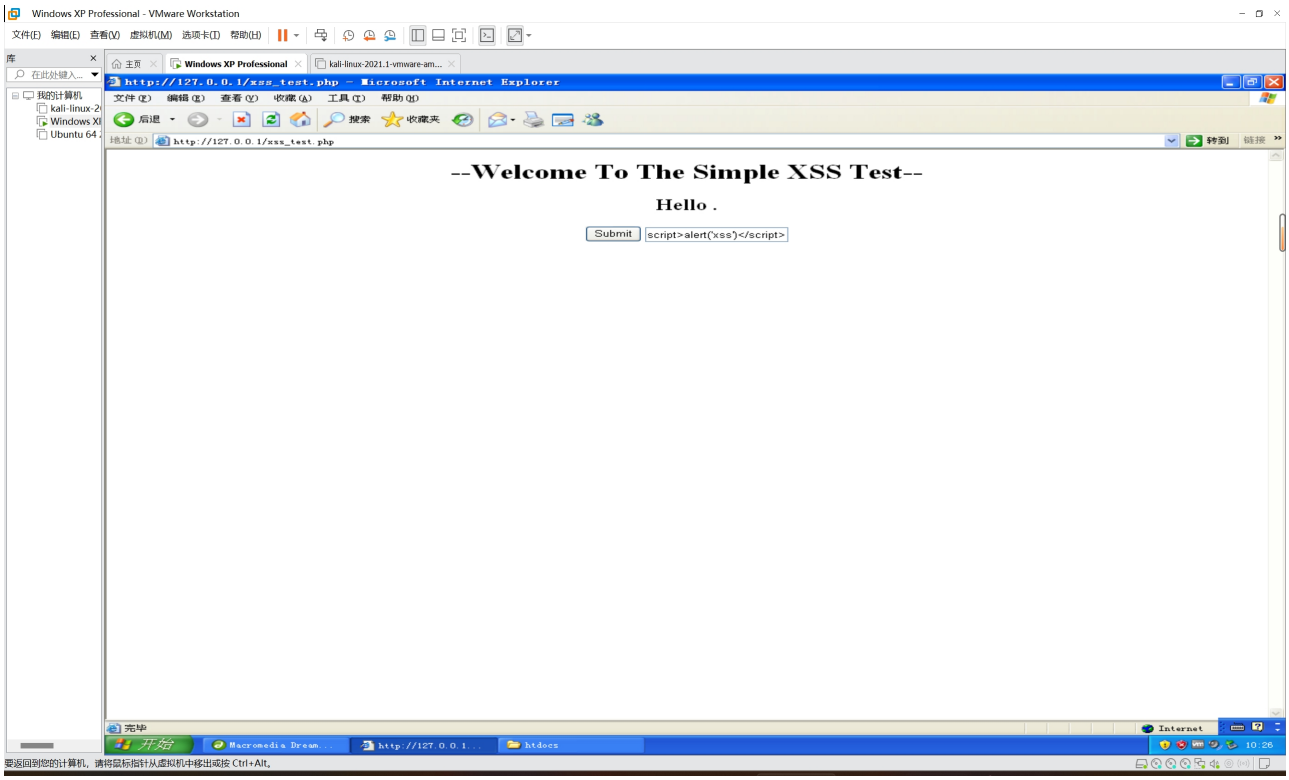


我们打开网页，输入地址：http://127.0.0.1/xss_test.php ，看到如下界面，证明代码运行无误，可以开始测试。




## 黑盒测试

首先从黑盒测试的角度来进行实验，访问URL：http://127.0.0.1/xss_test.php，在界面中，我们可以看到一个 `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。我们点击 `Submit` 按钮以后，效果如下：

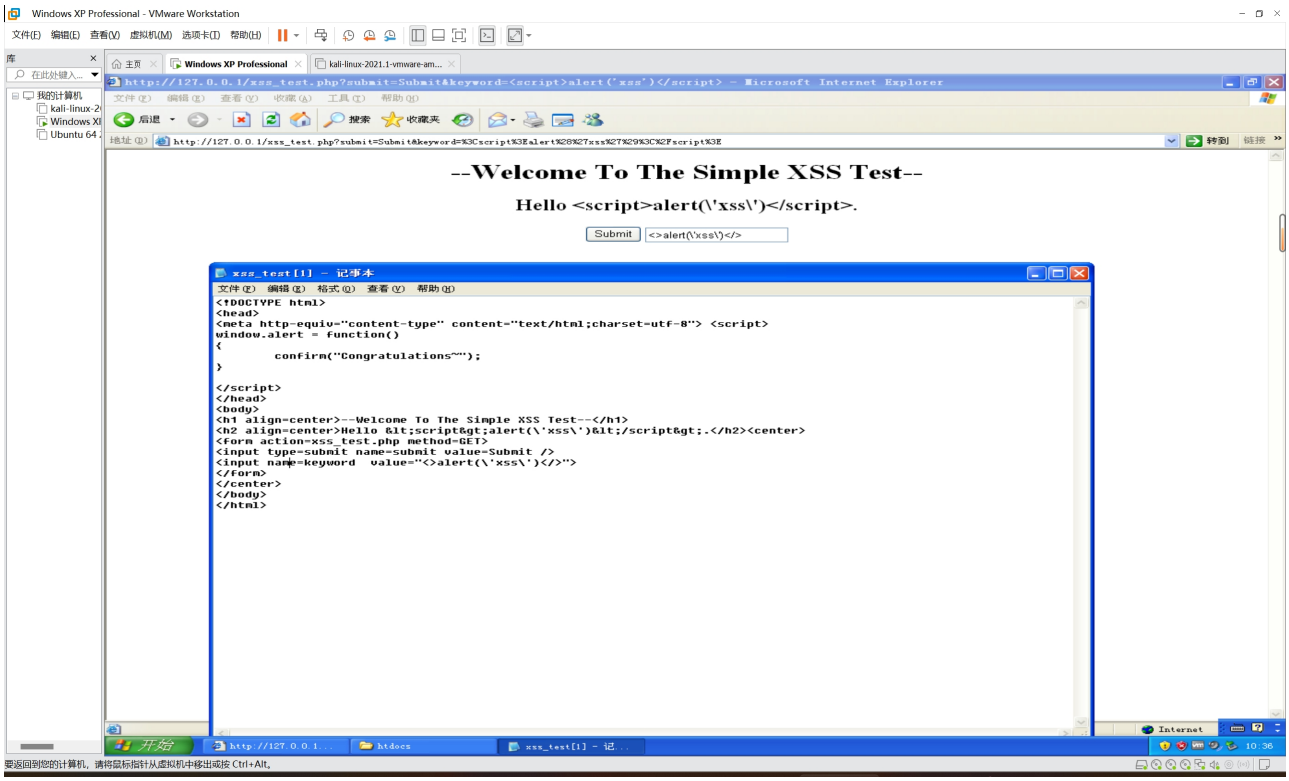


我们发现，发现`Hello`后面出现了我们输入的内容，并且输入框中的回显过滤了`script`关键字，这个时候考虑后台只是最简单的一次过滤。

我们再利用双写关键字来重新构造脚本：`<scrscriptipt>alert('xss')</scscriptript>`，并再次进行测试，还是没有预期输出。

至此，虽然输入框中显示出来的代码确为我们想要运行的攻击脚本，但是这句代码并没有被执行（如果成功执行，会弹窗输出“`congratulations`”）。所以我们接着寻找问题。我们跳转到该`php`文件的源码，就可以发现：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。但现在我们并没有看到该弹窗，说明代码确实没有被执行。

我们继续查看源代码，我们发现相比于最先的`php`代码，第十六行发生了改变。

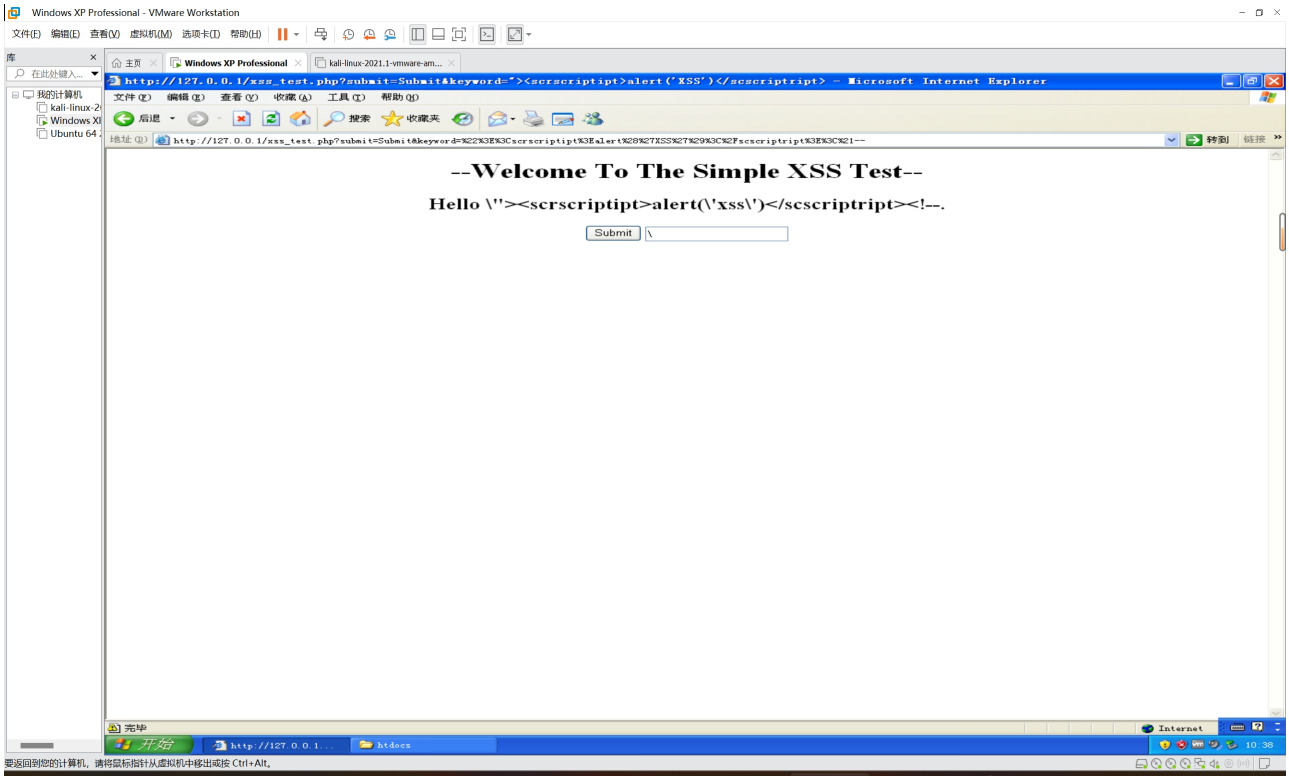



```
<input name=keyword value="<script>alert('xss')</script>">
```

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><!--
```

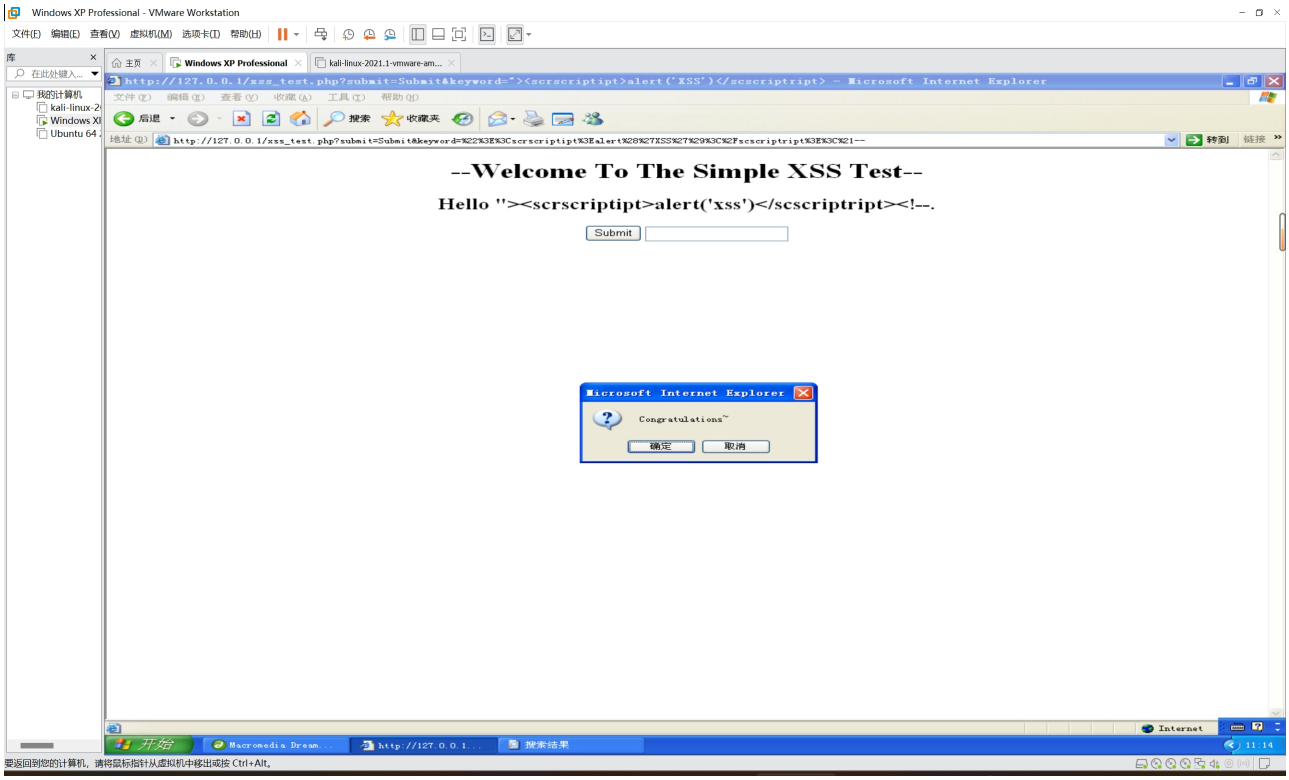
分析一下这行代码：`">` 用来闭合前面的`<input>`标签。而 `<!--` 其实是为了美观，用来注释掉后面不需要的 `">`,否则页面就会在输入框后面回显 `">`。我们输入该代码，进行测试，得到以下结果：




说明我们的攻击还是没有成功。这是因为 `php` 服务器为了避免一些用户特殊构造的攻击，将双引号等符号转义，于是修改 php-`apache2handler.ini`，将“`magic_quotes_gpc = On`”设置为“`magic_quotes_gpc = Off`”。

修改之后网页并没有立即完成同步服务，所以需要进行重启，才能正常显示。

我们发现经过调试，我们成功的完成了这次黑盒测试，输出我们想要的结果“Congratulations~”！




## 白盒测试

我们前往`xss_test.php`文件中查看页面的核心逻辑。

```
<?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>';
 ?>
```

分析上述代码可知，这些代码的逻辑与我们进行的黑盒测试所总结出的逻辑基本相符。但是也有黑盒测试中没测试到的地方。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。所以我们如果修改`php`代码，将这些过滤全部取消的话，我们也可以实施攻击。

## img方式

### 用标签构造脚本

我们构造的脚本如下：

```
<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。

我们在原来的代码中做一些修改：

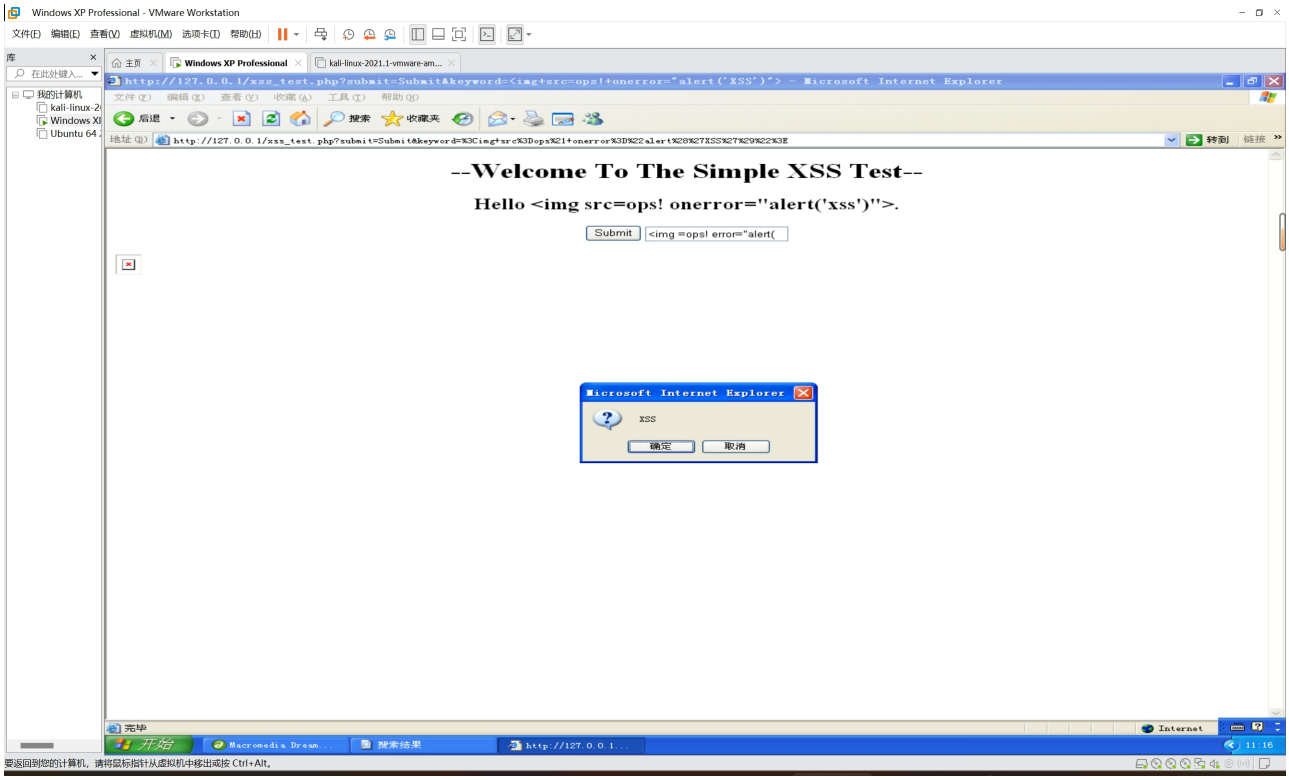
```
<!DOCTYPE html>
<head>
 <meta http-equiv="content-type" content="text/html;charset=utf-8">
 <script>
 window.alert = function()
 {
      confirm("Congratulations~");
 }
 </script>
</head>
<body>
  <h1 align=center>--Welcome To The Simple XSS Test--</h1>
  <?php
  ini_set( "display_errors", 0);
  $str=strtolower($_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='".htmlspecialchars($str4)."'>
  </form>
  </center>";
 ?>
 <img src="ops!" onerror="var
payload=String.fromCharCode(99,111,110,102,105,114,109,40,39,88,83,
83,39,41);eval(payload)">
</body>
</html>
```

我们来分析一下这段代码。

我在此处添加了一个`img`标签，并设置其`src`属性为一个无效的`URL ops!`。然后，我使用`onerror`事件来执行`JavaScript`代码。为了绕过过滤和替换操作，我将`JavaScript`代码以字符编码的方式嵌入到`onerror`属性中，并通过`eval`函数执行解码后的代码。在修改的源码中，`String.fromCharCode()` 被用于构建一个字符串变量 `payload`，其中的参数是一系列的 `Unicode` 值。这些 `Unicode` 值代表的是解码后的 `JavaScript` 代码，用于实现特定的功能，例如执行弹窗操作。在 `eval(payload)` 中，`eval()` 函数用于解析并执行该字符串中的 `JavaScript` 代码。这段代码通过一系列的 `Unicode` 值构建了一个字符串 '`confirm('XSS')`'，它表示调用 `confirm` 函数来显示一个确认对话框，内容为 '`XSS`'。通过这句核心代码，我们就能实现以`img`的方式完成一次跨站脚本攻击。

运行结果




可以看到，我们成功的输出了一个弹窗“`XSS`”，说明我们的攻击成功！

## 心得体会：

通过本次实验，我对XSS攻击与防御有了更深刻的认识：

**攻击手段多样：**XSS可通过`script`标签闭合、双写绕过过滤（如`scrscriptipt`）或`<img>`标签的`onerror`事件触发，利用编码（如`String.fromCharCode`）可进一步隐藏恶意代码，体现攻击灵活性。

**防御需综合措施：**单纯依赖关键字替换（如过滤`script`）易被绕过，需结合输入验证（白名单）、输出编码（`htmlspecialchars`）及上下文敏感处理，避免依赖黑名单。

**配置影响安全性：**服务器配置（如`magic_quotes_gpc`）直接影响漏洞利用，开发中应通过代码层防护（强制转义）而非环境假设。

**测试与安全意识：**黑盒测试定位漏洞，白盒分析优化逻辑；开发者需遵循“不信任用户输入”原则，优先使用安全框架（如React自动转义）减少人为疏漏。