浏览器解析过程

浏览器在解析HTML文档时，无论按照什么顺序，主要有三个过程：HTML解析、JS解析 以及 URL解析。每个解析器负责HTML文档中各自对应部分的解析工作。

首先，浏览器接收到一个HTML文档时，会触发HTML解析器对HTML文档进行词法解析，这一过程完成HTML解码并创建DOM树。接下来JavaScript解析器会介入对内联脚本进行解析，这一过程完成JS的解码工作。如果浏览器遇到需要URL的上下文环境，这时URL解析器也会介入完成URL的解码工作，URL解析器的解码顺序会根据URL所在位置不同，可能在JavaScript解析器之前或之后解析。

基本概念：

1.)URL编码

URL编码是为了允许URL中存在汉字这样的非标准字符，本质是把一个字符转换为"%"+"UTF-8编码对应的两位16进制数字"，如："/"对应的URL编码为%2f。所以又称之为Percent-encoding

再例如：

j ====ASCII码对应的十进制====> 106

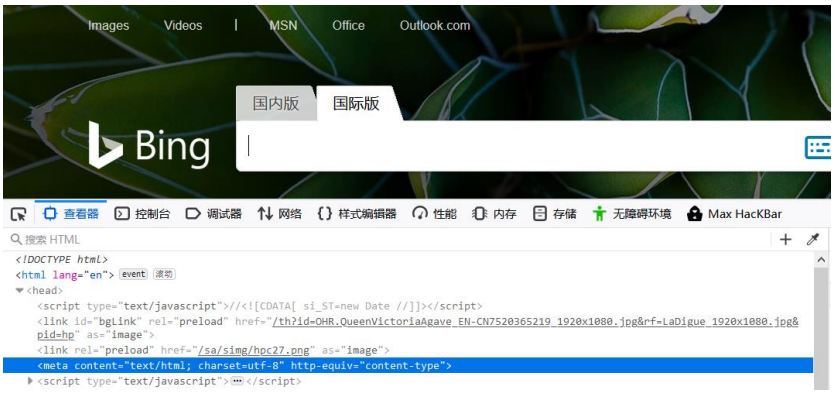
106 ====十六进制====> 6A

j ====URL编码====> %6A

在服务器端接收到请求时，会自动对请求进行一次URL解码。

2.)HTML编码/解码

当浏览器接收到服务器端发送来的二进制数据后，首先会对其进行HTML解码，呈现出来的就是我们看到的源代码。具体的解码方式依具体的情况而定。所以我们需要在页面中指定编码，以防止浏览器按照错误的方式解码，造成乱码，比如必应搜索首页就指定了解码方式为UTF-8



3.）HTML字符实体

在呈现HTML页面时，针对某些特殊字符，如"<"或">"，倘若直接使用，浏览器会误以为它们是标签的开始和结束，若想正确的在HTML页面呈现特殊字符，就需要用到其对应的字符实体。

字符实体是一个预先定义好的转义序列，它定义了一些无法在文本内容中输入的字符或符号。字符实体以"&"开头+预先定义的实体名称，以"；"结束。如："<"的实体名称为< 或者以"&"开头+"#"+字符的十进制数字+";"(亦即：Ascii码)。 或者"&"+"#"+'x'+字符的16进制数字+";"

如"<"的实体编号为<或者&#x3C

字符都是有实体编号的，但是有些字符没有实体名称。

显示结果	描述	实体名称	实体编号
<	小于号	<	<
>	大于号	>	>

浏览器对HTML解码之后就开始解析HTML，将标签转化为内容树中的DOM节点，此时识别标签的时候，HTML解析器是无法识别那些被实体编码的内容的，只有建立起DOM树，才能对每个节点的内容进行识别，如果出现实体编码，则会进行实体解码，只要是DOM节点里属性的值，都可以被HTML编码和解析。

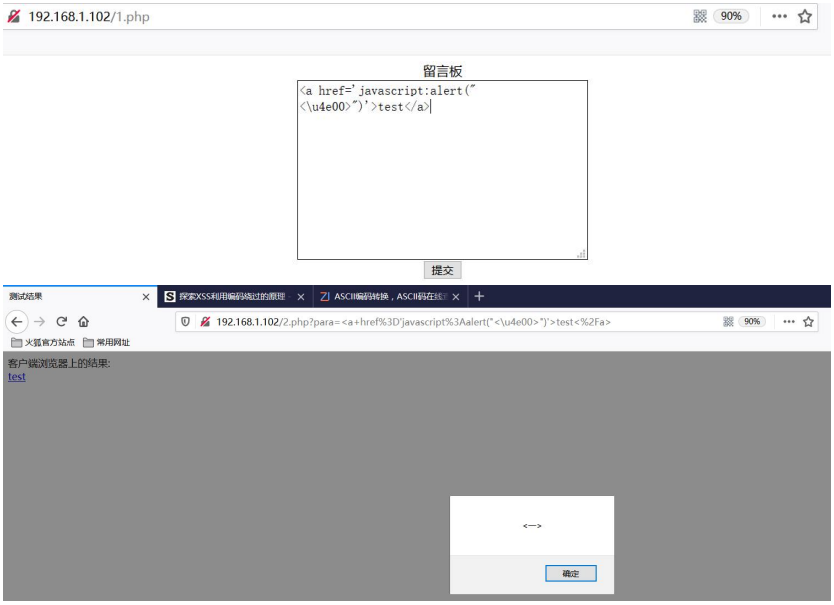
所以在PHP中，使用htmlspecialchars()函数把预定义的字符转换为HTML实体，只有等到DOM树建立起来后，才会解析HTML实体，起到了XSS防护作用。

由此可见：想要HTML标签被HTML解析器识别，转化为内容树中的DOM节点，就不能对HTML标签做编码操作

4.）JavaScript编码/解码

当HTML解析器产生DOM节点后，会根据DOM节点来做接下来的解析工作，比如在处理诸如<script>、<style>这样的标签时，解析器会自动切换到JS解析模式，而src、href后边加入的javascript为伪URL，也会进入JS的解析模式。

比如 <a href='javascript:alert("<\u4e00>")'>test ，javascript发出了JS解析器，JS解析器对alert("<\u4e00>")进行解析，里边有一个转义字符\u4e00，前导的u表示它是一个Unicode字符，根据后边的数字，解析为"--"，于是在完成JS的解析之后变成了：alert("<—>")'



最常用的如"\uXXXX" 这种写法为Unicode转义序列，表示一个字符，其中XXXX表示一个16进制数字，如："<"的Unicode编码为"\u003c"。

结合具体示例来讨论下浏览器的解析原理过程和XSS复合编码的一些内容：

示例：test

针对上述a标签，我们分析一下该环境中浏览器的解析顺序：

首先HTML解析器开始工作，对整个HTML文档进行HTML解码。接下来URL解析器会对href中的值进行URL解码，正常情况下URL值为一个正常的URL链接，如："https://www.xxx.com"。那么URL解析器工作完成后是不需要其他解码工作的，但是该环境中URL资源类型为JavaScript，因此该环境中最后一步JavaScript解析器会进行解码操作。

该环境中，整个解析顺序为3个环节：HTML解码 ==> URL解码 ==> JS解码

注意：

URL解析的一个细节：

不能对协议类型，例如"javascript:"(javascript: 为URL伪协议.注意href只是一个属性,它不是URL协议)进行任何编码操作(可以进行HTML实体编码，因为HTML解析器始终工作在URL解析器之前)，否则URL解析器会认为它无类型。就导致DOM节点中被编码的"javascript"没有解码，不会被URL解析器识别。就比如说http://www.baidu.com可以被编码为：http://%77%77%77%2e%62%61%69%64%75%2e%63%6f%6d，但是不能把协议也编码了：

%68%74%74%70%3a%2f%2f%77%77%77%2e%62%61%69%64%75%2e%63%6f%6d

JavaScript解析的一个细节：

假设JavaScript解析器工作时将\u0061\u006c\u0065\u0072\u0074进行JS解码后为"alert",而"alert"是一个有效的标识符名称，它是可以被正常解析的。但是像"(" 或者 ")"、“ ” 、“ ’ ”等等这些控制字符，在进行JS解析的时候仅会被解码为字符串文本或者标识符名称，例如：<script>alert('YISRC\u0027)</script>对控制字符"'"进行了Unicode编码，解析时\u0027会被解析成文本单引号，也就无法闭合前面的单引号，所以不能成功执行代码。

测试：

Test1:
HTML编码 test
目的：测试被HTML实体编码后的标签能否被正常执行

Test2:
HTML编码 href='javascript:alert(/xss/)'
目的：1.让HTML解析器能够识别<a>标签，从而测试js代码在HTML解码后能否被执行
2.测试对href进行编码操作后，"Test"能否被点击

Test3：
HTML编码 javascript:alert(/xss/)
目的：测试href未进行任何编码时，"Test"能否被点击

Test4：
HTML编码 alert(/xss/)

Test5:
URL编码 javascript:alert(/xss/)
目的：测试URL编码后，js代码在进行URL解码后能否被执行

Test6：
URL编码 alert(/xss/)

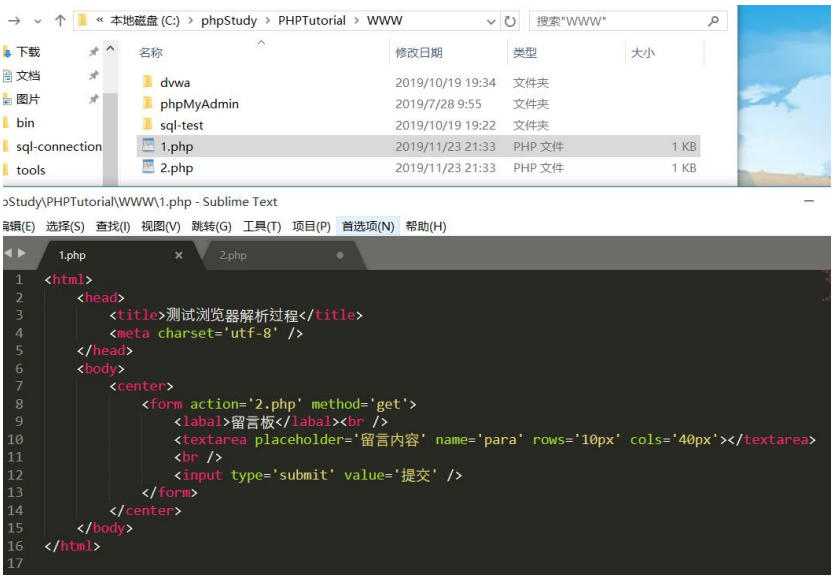
Test7:
URL编码 alert(/xss/)
HTML实体编码 javascript: (href='javascript: 在Test3测试过)
测试对javascript进行HTML编码操作后，【URL编码的alert(/xss/)】是否能够被URL解码

Test8:
JS编码 alert(/xss/)
URL编码 【JS编码后的alert(/xss/)】
HTML编码 “ javascript:【JS编码再URL编码后的alert(/xss/)】”

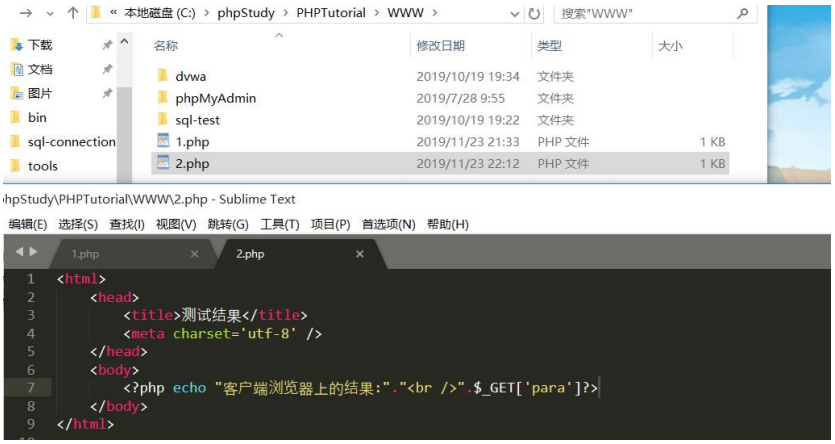
Test9：
JS编码 alert (这里要考虑JS解码的一个细节)
URL编码 ":"+【JS编码后的alert】+"(/xss/)"
HTML编码 【URL编码后的 “: +【JS编码后的alert】+ (/xss/)”】

Test10：
JS编码 alert (这里要考虑JS解码的一个细节)
URL编码 【JS编码后的alert】+"(/xss/)"
HTML编码 "javascript:" +【URL编码后的【JS编码后的alert】+ (/xss/)”】

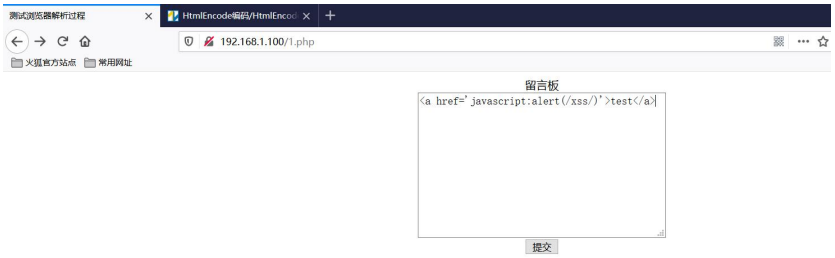
服务器端存在XSS漏洞的代码如下：



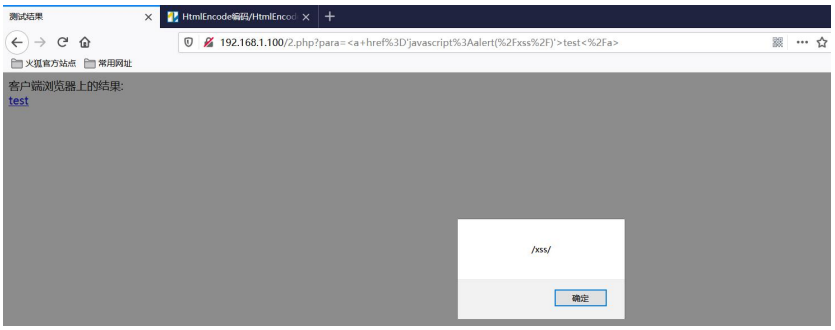
服务器端2.php文件未对用户输入的内容做任何校验就输出结果：



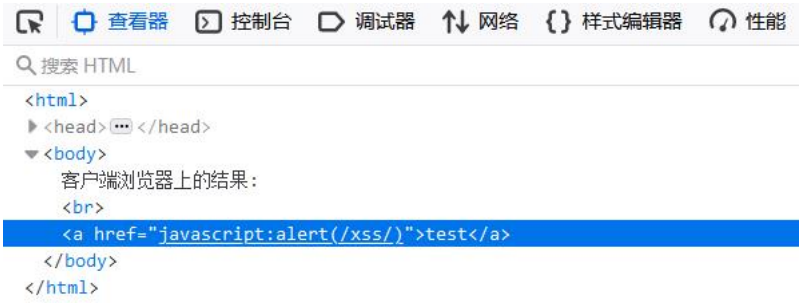
在开始Test1之前，先来看看未做任何编码的：



点击"test",弹框：



审查元素：



Test1:

HTML编码 test

目的：测试被HTML实体编码后的标签能否被正常执行

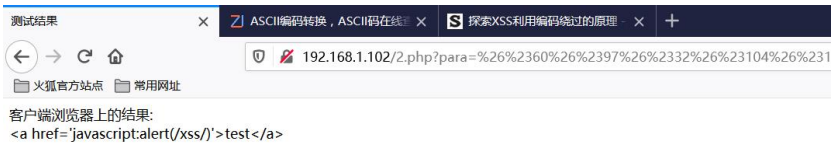
test

====HTML实体编码后====>

(ASCII在线编码转换：https://www.qqxiuzi.cn/bianma/ascii.htm)

输入：

&#60&#97&#32&#104&#114&#101&#102&#61&#39&#106&#97&#118&#97&#115&#99&#114&#105&#112&#116&#58&#97&#108&#101&#114&#116&#40&#47&#120&#115&#11

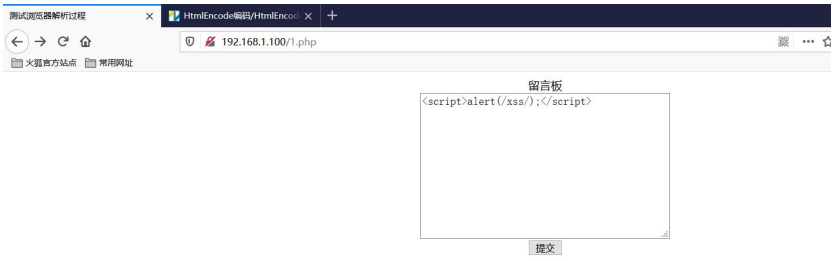


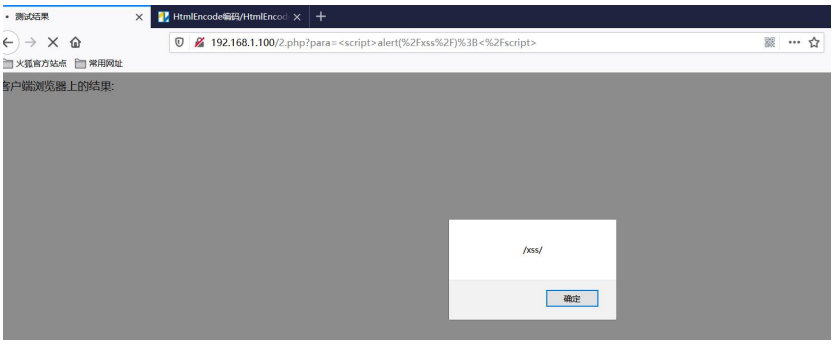
==> 可以看到，HTML标签被解析成了字符串，能够输出到浏览器页面上，XSS绕过失败。

因为服务器端传过来的HTML文档第一次被客户端浏览器解析HTML时，由DOM方法将HTML文档解析成树形节点，然后才对DOM节点里面的HTML实体进行解析。所以HTML编码后的 test 没有被HTML解析器识别为DOM节点。

不妨再试试：<script>alert(/xss/);</script>

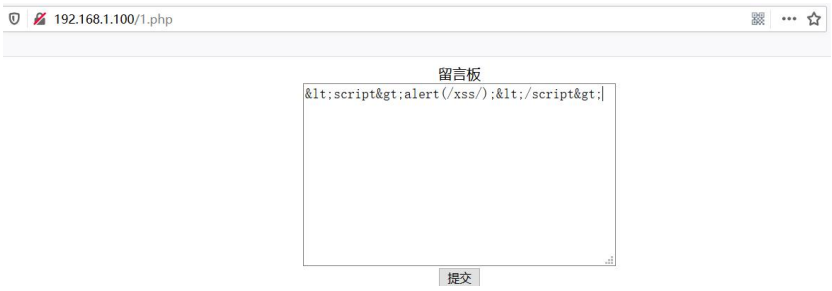
不做任何编码：





这次仅对"<"、">"进行HTML编码：(后面会介绍解析器)

<script>alert(/xss/);</script>



==> 同样被当成了字符串输出到页面上

Test2:

HTML编码 href='javascript:alert(/xss/)'

目的：1.让HTML解析器能够识别<a>标签，从而测试js代码在HTML解码后能否被执行

2.测试对href进行编码操作后，"Test"能否被点击

href='javascript:alert(/xss/)'

====HTML编码====>

&#104&#114&#101&#102&#61&#39&#106&#97&#118&#97&#115&#99&#114&#105&#112&#116&#58&#97&#108&#101&#114&#116&#40&#47&#120&#115&#115&#47&#41&#3

输入：<a

&#104&#114&#101&#102&#61&#39&#106&#97&#118&#97&#115&#99&#114&#105&#112&#116&#58&#97&#108&#101&#114&#116&#40&#47&#120&#115&#115&#47&#41&#3

==> HTML解析器识别了<a>标签，将<a>标签放入DOM树形节点数中，但是HTML解析器并没有对DOM节点里面的HTML实体进行解析

但是我们将 href='javascript:alert(/xss/)' 进行HTML实体编码后的内容带入，浏览器却会对其进行解码：

==> HTML解析器确实能够识别<a>标签，将其带入DOM节点树，但是<a>标签中的属性及属性值没有被HTML解析器解码

Test3：

HTML编码 javascript:alert(/xss/)

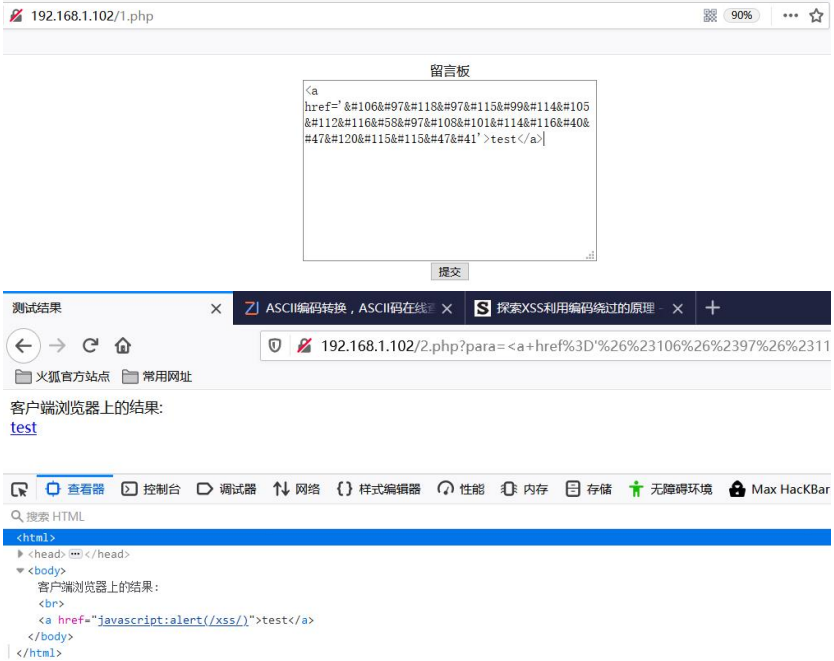
目的：测试href未进行任何编码时，"Test"能否被点击

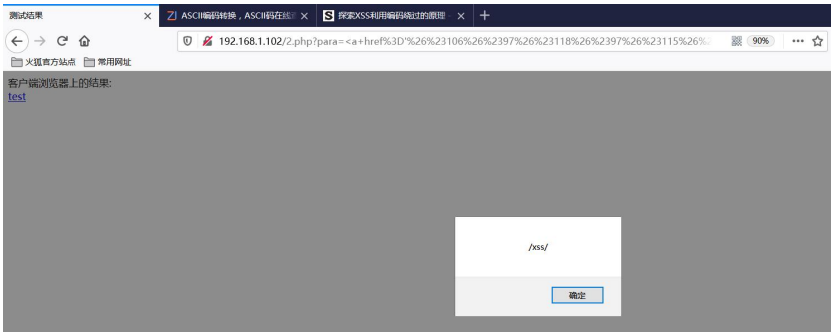
javascript:alert(/xss/)

====HTML编码====>

&#106&#97&#118&#97&#115&#99&#114&#105&#112&#116&#58&#97&#108&#101&#114&#116&#40&#47&#120&#115&#115&#47&#41

输入：test





==> 与Test2的区别在于，我们这次没有对<a>标签的属性"href"进行HTML编码，在形成DOM树后，HTML解析器对<a>标签中的内容进行了解析。

补充：

test

（1）对"href"中的某个字符进行HTML实体编码：

输入：

<a h&#114ef='&#106&#97&#118&#97&#115&#99&#114&#105&#112&#116&#58&#97&#108&#101&#114&#116&#40&#47&#120&#115&#115&#47&#41'>test

提交后：

==> HTML解析器未对<a>标签中的内容进行解码

（2）这次对"="进行HTML实体编码：

输入：

<a href&#61'&#106&#97&#118&#97&#115&#99&#114&#105&#112&#116&#58&#97&#108&#101&#114&#116&#40&#47&#120&#115&#115&#47&#41'>test

提交后：

==> HTML解析器未对<a>标签中的内容进行解码

（3）对两个单引号进行HTML实体编码：

输入：

test

提交后：

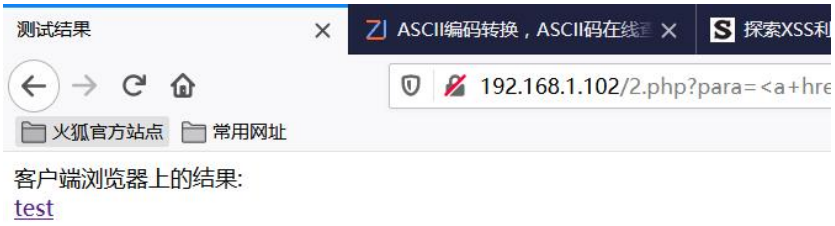
Forbidden

You don't have permission to access /'javascript:alert(/xss/)' on this server.

==> 对HTML实体编码后的单引号进行了HTML解码，但是由错误页面的报错信息可以看到，解码后的单引号变成了字符串文本。审查代码可以看到多添加了两个 ″

换成双引号：

test



Forbidden

You don't have permission to access /"javascript:alert(/xss/)" on this server.

==> 原因是：在<a>标签被识别为DOM节点数时，由于属性中的&#34还未进行解码，javascript:alert(/xss/)没有引号包裹，浏览器会自动给其添加引号来包裹javascript:alert(/xss/)。之后HTML解析器进行HTML解码，将&#34解码成 ″ ，但是外面已有引号包裹，所以&#34解码成的 ″ 变成了字符串文本。

Test4：

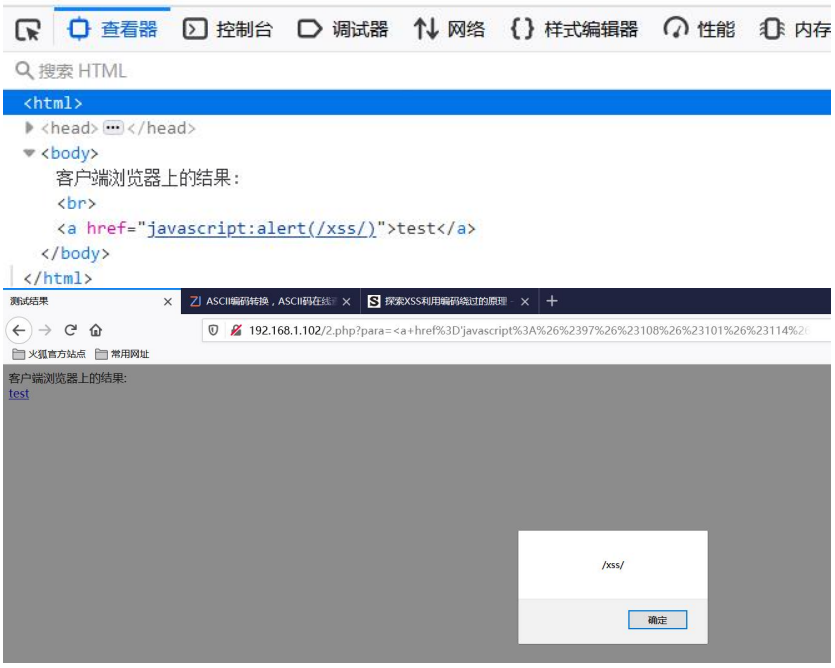
HTML编码 alert(/xss/)

alert(/xss/)

====HTML编码后====>

&#97&#108&#101&#114&#116&#40&#47&#120&#115&#115&#47&#41

带入：test



Test5:

URL编码 javascript:alert(/xss/)

目的：测试URL编码后，js代码在进行URL解码后能否被执行

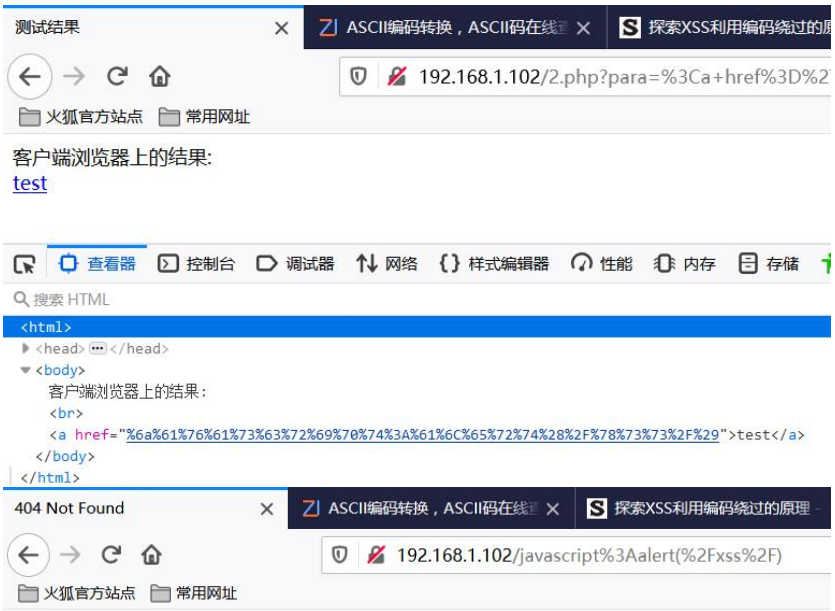
javascript:alert(/xss/)

====URL编码====>

%6a%61%76%61%73%63%72%69%70%74%3A%61%6C%65%72%74%28%2F%78%73%73%2F%29

输入：

test



Not Found

The requested URL /javascript:alert(/xss/) was not found on this server.

==> ULR解析器未进行解码操作

对javascript协议进行了编码操作，URL解析器认为它无类型，导致DOM节点中被编码的"javascript"没有解码，不会被URL解析器识别

Test6：

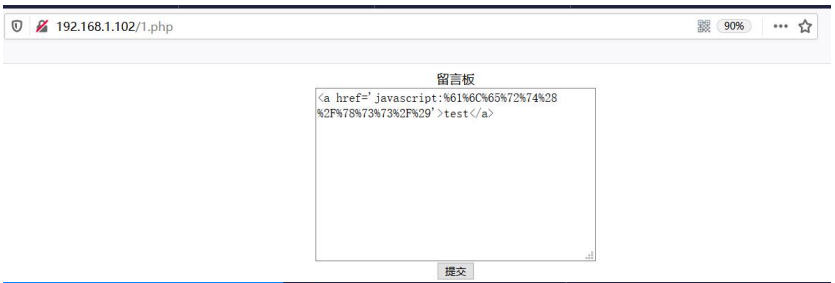
URL编码 alert(/xss/)

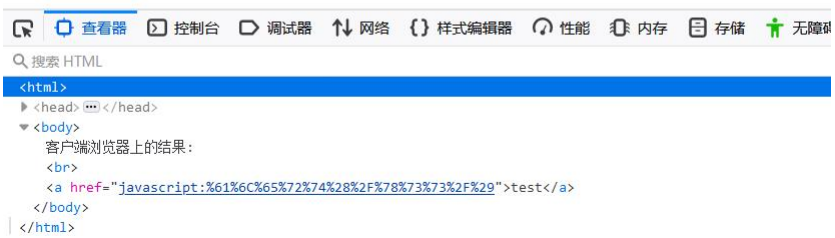
alert(/xss/)

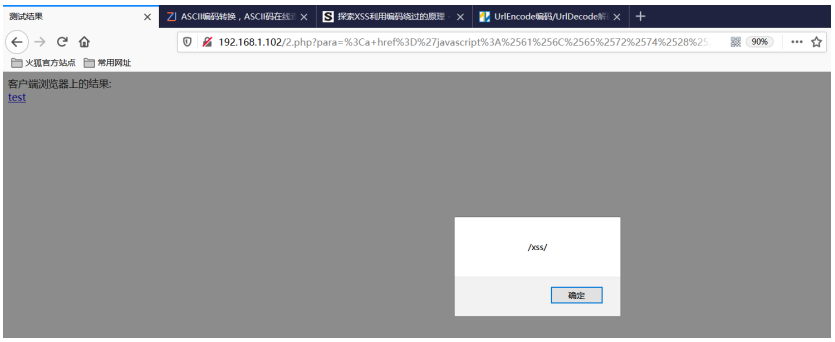
====URL编码====>

%61%6C%65%72%74%28%2F%78%73%73%2F%29

输入：test







==> 可以看到HTML源码中URL解析器未对 %61%6C%65%72%74%28%2F%78%73%73%2F%29 进行解码

当点击"test"时，URL解析器对 %61%6C%65%72%74%28%2F%78%73%73%2F%29 ==解码==> /xss/

Test7:

URL编码 alert(/xss/)

HTML实体编码 javascript: （href='javascript: 在Test3测试过）

测试对javascript进行HTML编码操作后，【URL编码的alert(/xss/)】是否能够被URL解码

alert(/xss/)

====URL编码===>

%61%6C%65%72%74%28%2F%78%73%73%2F%29

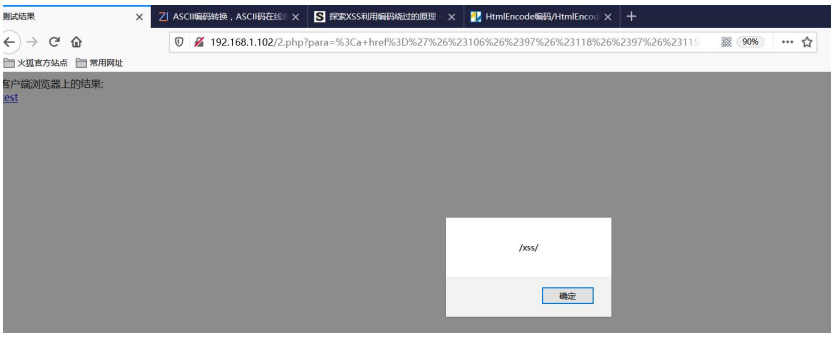
javascript:

====HTML实体编码===>

&#106&#97&#118&#97&#115&#99&#114&#105&#112&#116&#58

输入：

test



==> 点击"test"时，URL解析器对 %61%6C%65%72%74%28%2F%78%73%73%2F%29 ==解码==> /xss/

Test8:

JS编码 alert(/xss/)

URL编码 【JS编码后的alert(/xss/)】

HTML编码 “ javascript:【JS编码再URL编码后的alert(/xss/)】”

alert(/xss/)

====JS编码====>

\u0061\u006C\u0065\u0072\u0074\u0028\u002F\u0078\u0073\u0073\u002F\u0029

\u0061\u006C\u0065\u0072\u0074\u0028\u002F\u0078\u0073\u0073\u002F\u0029

====URL编码====> (asciizm.exe工具 + calc.exe)

%5c%75%30%30%36%31%5c%75%30%30%36%63%5c%75%30%30%36%35%5c%75%30%30%37%32%5c%75%30%30%37%34%5c%75%30%30%32%38%5c%75%30%30%32%46%5c%75%30%30%37%3

javascript:%5c%75%30%30%36%31%5c%75%30%30%36%63%5c%75%30%30%36%35%5c%75%30%30%37%32%5c%75%30%30%37%34%5c%75%30%30%32%38%5c%75%30%30%32%46%5c%7

====HTML编码====> (HTML实体编码在线网站：http://www.ofmonkey.com/encode/ascii)

&#106&#97&#118&#97&#115&#99&#114&#105&#112&#116&#58%5c%75%30%30%36%31

提交：

<a

href='&#106&#97&#118&#97&#115&#99&#114&#105&#112&#116&#58%5c%75%30%30%36%&#5

'>test

==> 点击"test"不跳转到任何页面

Test9：

JS编码 alert (这里要考虑JS解码的一个细节)

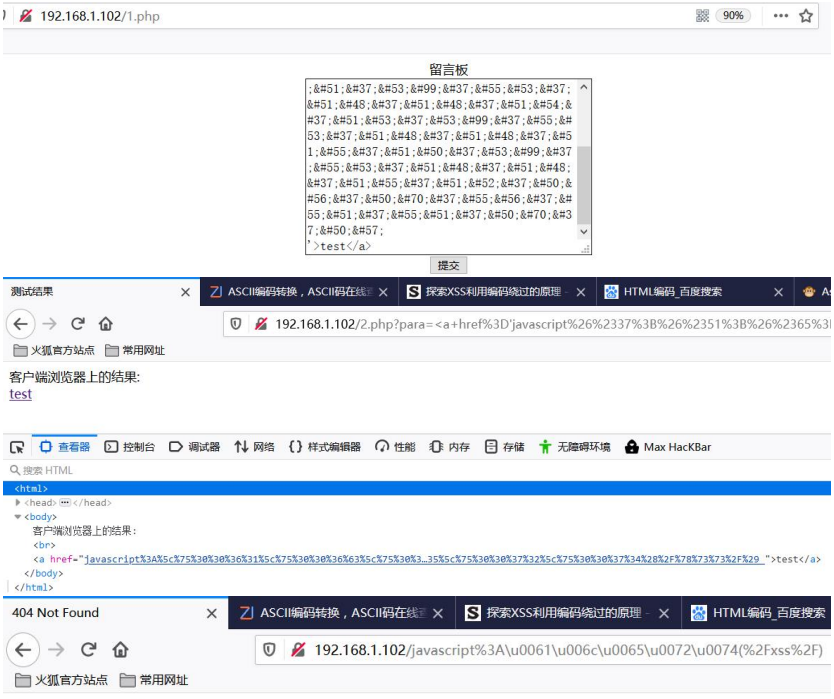
URL编码 ":"+【JS编码后的alert】+"(/xss/)"
HTML编码 【URL编码后的 ": +【JS编码后的alert】+ (/xss/)"】

alert
====JS编码====>
\u0061\u006C\u0065\u0072\u0074

:\u0061\u006C\u0065\u0072\u0074(/xss/)
====URL编码====>
%3A%5c%75%30%30%36%31%5c%75%30%30%36%63%5c%75%30%30%36%35%5c%75%30%30%37%32%5c%75%30%30%37%34%28%2F%78%73%73%2F%29

%3A%5c%75%30%30%36%31%5c%75%30%30%36%63%5c%75%30%30%36%35%5c%75%30%30%37%32%5c%75%30%30%37%34%28%2F%78%73%73%2F%29
====HTML编码====>
%3A%5c%75%30%30%36%31%5c%75%&#5

输入：
<a
href='javascript%3A%5c%75%30%30%36%31%5c%&
'>test



==> 说明“javascript:”是一个完整的URL协议，不能对":"进行URL编码

Test10：
JS编码 alert （这里要考虑JS解码的一个细节）
URL编码 【JS编码后的alert】+"(/xss/)"
HTML编码 "javascript:" +【URL编码后的【JS编码后的alert】+ (/xss/)"】

alert
====JS编码====>
\u0061\u006C\u0065\u0072\u0074

\u0061\u006C\u0065\u0072\u0074(/xss/)
====URL编码====>
%5c%75%30%30%36%31%5c%75%30%30%36%63%5c%75%30%30%36%35%5c%75%30%30%37%32%5c%75%30%30%37%34%28%2F%78%73%73%2F%29

javascript:%5c%75%30%30%36%31%5c%75%30%30%36%63%5c%75%30%30%36%35%5c%75%30%30%37%32%5c%75%30%30%37%34%28%2F%78%73%73%2F%29

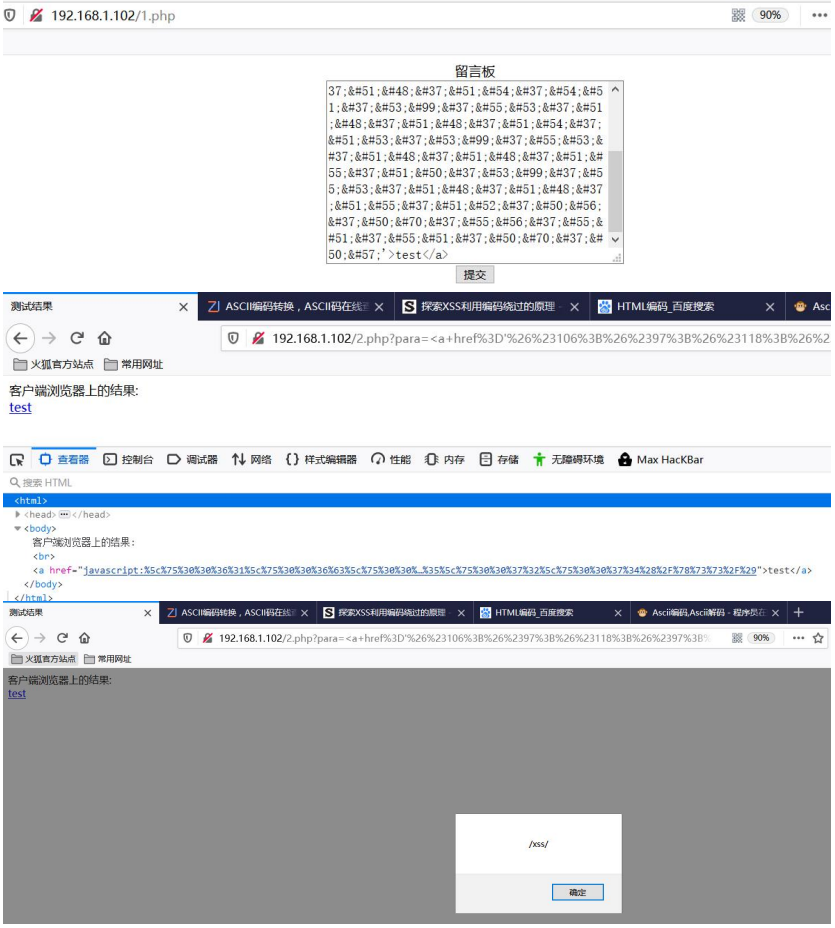
====HTML编码====>

javascript:%5c%75%30%30%36%&

输入：

<a

href='javascript:%5c%75%30%30%3&#54



==> 触发XSS

最后，结合上面的内容，分析一下value1和value2，处要防御XSS需要怎么做组合编码？

```
<a onclick="window.open('value1')" href="javascript:window.open('value2')">
```

(value值中带有协议类型，URL协议会触发URL解析器进行解码)

浏览器解码顺序：

value1：HTML解码 ==> JS解码 ==> URL解码

value2：HTML解码 ==> URL解码 ==> JS解码 ==> URL解码

对于Web应用中普遍存在的XSS问题，只有深入理解了浏览器解析过程原理及解析器的协同工作，并结合上下文环境深入分析编码原理，只有这样，才能在XSS漏洞挖掘过程中对于一些编码不合理的场景进行绕过（当然了，也有很多其他绕过方法，利用编码只是其中之一），才能在XSS防御中合理编码用户输入的内容，杜绝因编码不合理造成的XSS漏洞。