恶意代码分析与防治技术实验报告

Lab11

学号：2011937　　姓名：姜志凯　　专业： 信息安全

一、 实验环境

- Windows10
- Windows xp

二、 实验工具

- IDA Pro
- ProcessMonitor
- Resource Hacker

三、 实验内容

Lab 11-1

Analyze the malware found in *Lab11-01.exe*.

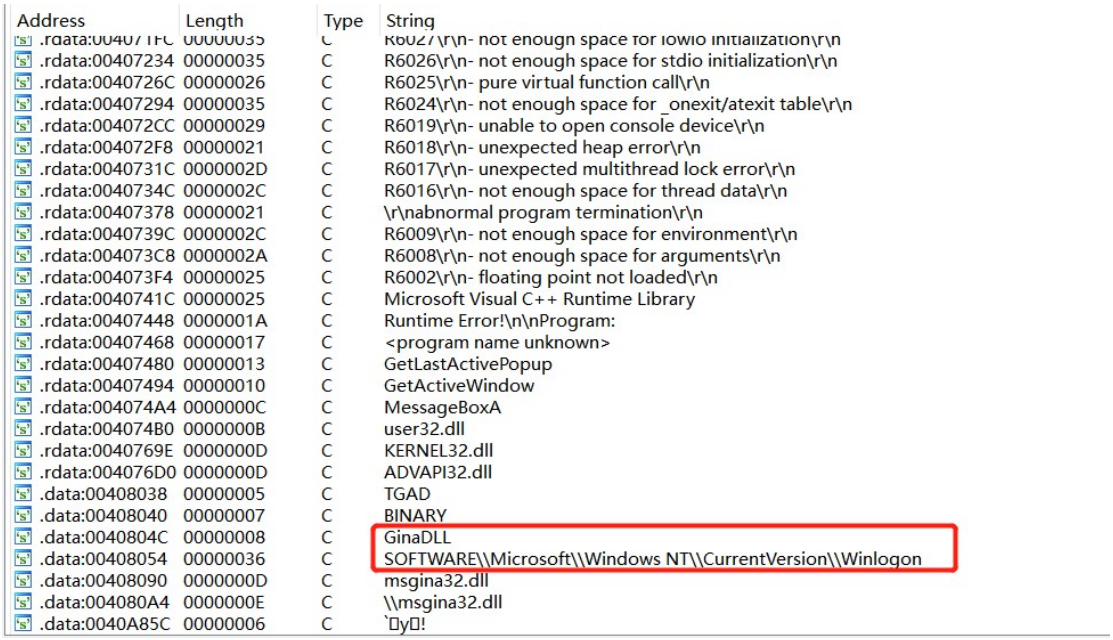
Questions

1. What does the malware drop to disk?
2. How does the malware achieve persistence?
3. How does the malware steal user credentials?
4. What does the malware do with stolen credentials?
5. How can you use this malware to get user credentials from your test environment?

1、这个恶意代码向磁盘中释放了什么？

用 IDA Pro 打开：

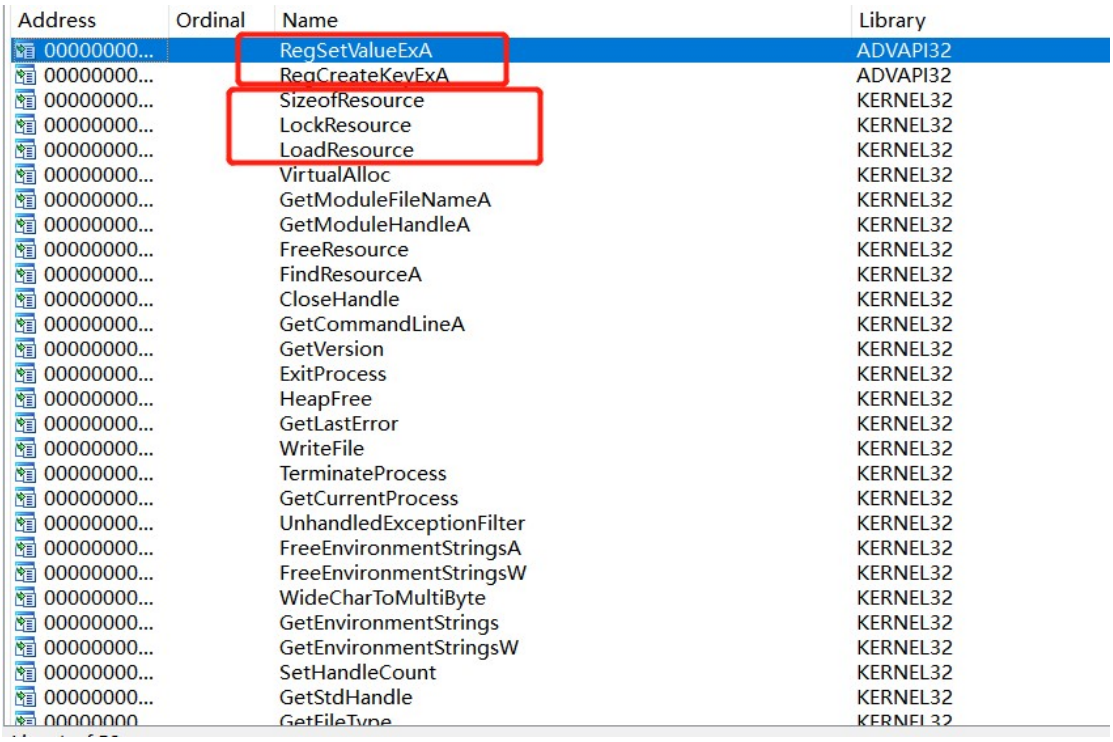
先来查看字符串



Address	Length	Type	String
.rdata:00407234	00000035	C	R6026\r\n- not enough space for stdio initialization\r\n
.rdata:0040726C	00000026	C	R6025\r\n- pure virtual function call\r\n
.rdata:00407294	00000035	C	R6024\r\n- not enough space for _onexit/atexit table\r\n
.rdata:004072CC	00000029	C	R6019\r\n- unable to open console device\r\n
.rdata:004072F8	00000021	C	R6018\r\n- unexpected heap error\r\n
.rdata:0040731C	0000002D	C	R6017\r\n- unexpected multithread lock error\r\n
.rdata:0040734C	0000002C	C	R6016\r\n- not enough space for thread data\r\n
.rdata:00407378	00000021	C	\r\nabnormal program termination\r\n
.rdata:0040739C	0000002C	C	R6009\r\n- not enough space for environment\r\n
.rdata:004073C8	0000002A	C	R6008\r\n- not enough space for arguments\r\n
.rdata:004073F4	00000025	C	R6002\r\n- floating point not loaded\r\n
.rdata:0040741C	00000025	C	Microsoft Visual C++ Runtime Library
.rdata:00407448	0000001A	C	Runtime Error!\n\nProgram:
.rdata:00407468	00000017	C	<program name unknown>
.rdata:00407480	00000013	C	GetLastActivePopup
.rdata:00407494	00000010	C	GetActiveWindow
.rdata:004074A4	0000000C	C	MessageBoxA
.rdata:004074B0	0000000B	C	user32.dll
.rdata:0040769E	0000000D	C	KERNEL32.dll
.rdata:004076D0	0000000D	C	ADVAPI32.dll
.data:00408038	00000005	C	TGAD
.data:00408040	00000007	C	BINARY
.data:0040804C	00000008	C	GinaDLL
.data:00408054	00000036	C	SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion\\Winlogon
.data:00408090	0000000D	C	msgina32.dll
.data:004080A4	0000000E	C	\\msgina32.dll
.data:0040A85C	00000006	C	`□y□!

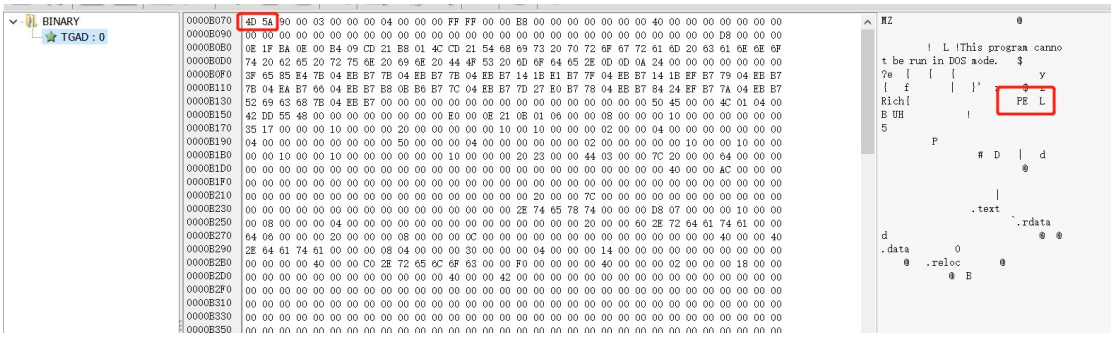
看到了 GinaDLL 和 Winlogon，猜测可能是一个拦截 GINA 的恶意代码，即恶意程序可以使用微软图形识别和验证界面（GINA）拦截技术来窃取用户的登录凭证。

再看一下导入表：

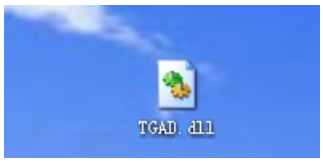


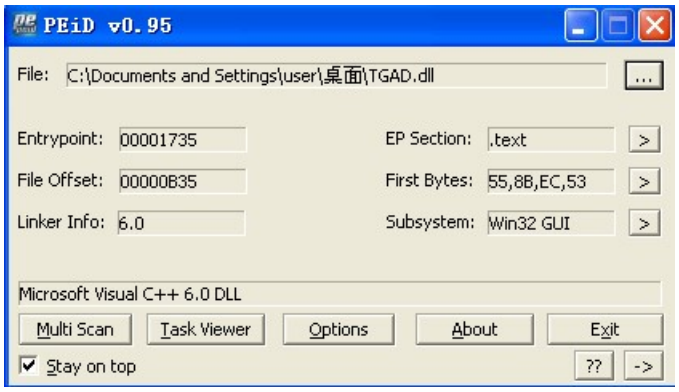
Address	Ordinal	Name	Library
00000000...		RegSetValueExA	ADVAPI32
00000000...		RegCreateKeyExA	ADVAPI32
00000000...		SizeofResource	KERNEL32
00000000...		LockResource	KERNEL32
00000000...		LoadResource	KERNEL32
00000000...		VirtualAlloc	KERNEL32
00000000...		GetModuleFileNameA	KERNEL32
00000000...		GetModuleHandleA	KERNEL32
00000000...		FreeResource	KERNEL32
00000000...		FindResourceA	KERNEL32
00000000...		CloseHandle	KERNEL32
00000000...		GetCommandLineA	KERNEL32
00000000...		GetVersion	KERNEL32
00000000...		ExitProcess	KERNEL32
00000000...		HeapFree	KERNEL32
00000000...		GetLastError	KERNEL32
00000000...		WriteFile	KERNEL32
00000000...		TerminateProcess	KERNEL32
00000000...		GetCurrentProcess	KERNEL32
00000000...		UnhandledExceptionFilter	KERNEL32
00000000...		FreeEnvironmentStringsA	KERNEL32
00000000...		FreeEnvironmentStringsW	KERNEL32
00000000...		WideCharToMultiByte	KERNEL32
00000000...		GetEnvironmentStrings	KERNEL32
00000000...		GetEnvironmentStringsW	KERNEL32
00000000...		SetHandleCount	KERNEL32
00000000...		GetStdHandle	KERNEL32
00000000		GetFileType	KERNEL32

发现有许多关于资源的 API 函数，还有修改注册表的相关函数，猜测可能暗藏资源，用 Resource Hacker 打开：



发现 PE 和 MZ，说明隐藏一个可执行文件，提取资源，用 PEiD 打开得知，这是一个从 TGAD 资源节提取出来的 dll 文件。

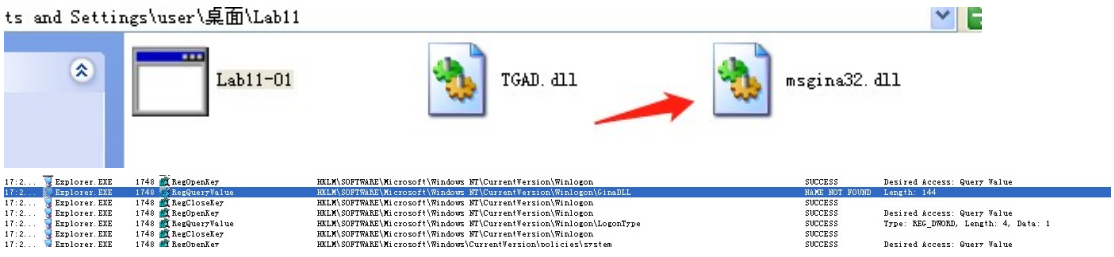




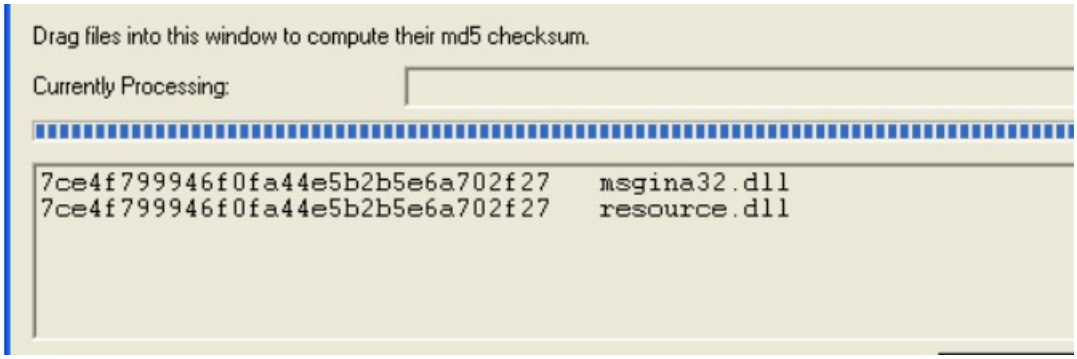
猜测这个可能就是程序向磁盘中释放的资源，下面运行一下看看：

用 processmontior 监控：

双击程序运行，发现目录下出现了一个 msgina32.dll 文件，查看注册表变化，发现调用了一个 RegCreateKey 然后创建了一个 HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon，然后通过函数 RegSetValue 来设置了一个值为 HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\GinaDLL



将这个 msgina32.dll 文件与之前提取出来的进行对比，计算 MD5 值，发现二者完全一样



所以，该恶意程序从名为 TGAD 的资源节提取出一个名为 msgina32.dll 的文件，将其释放到磁盘中该程序所在的目录。

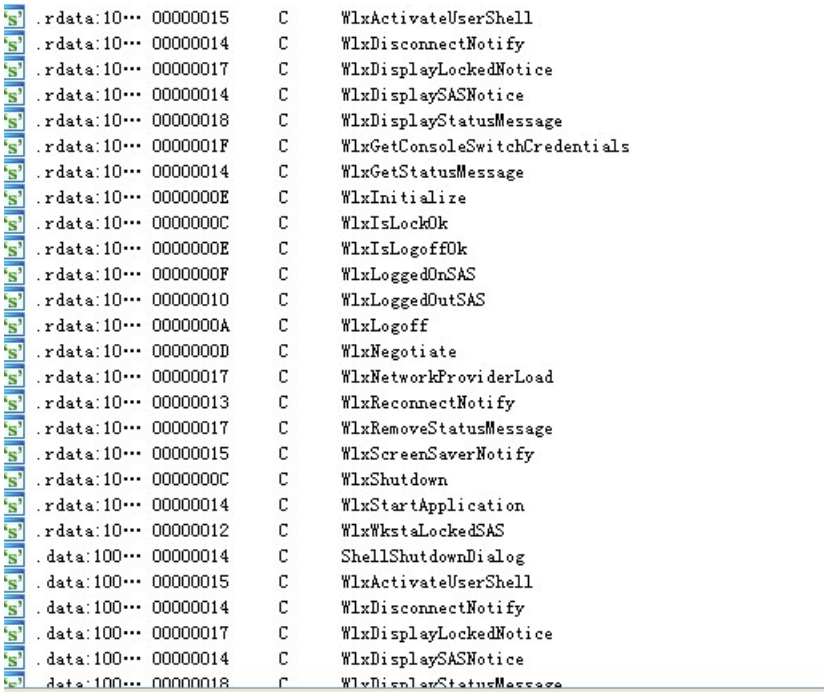
2、该恶意代码如何进行驻留？

由上一问的分析，该恶意代码运行后会修改注册表，添加键值 GINADLL，每次，msgina32.dll 作为 GINADLL 安装，系统重启后就会加载 msgina32.dll，达到驻留的目的。

3、这个恶意代码如何窃取用户登录凭证？

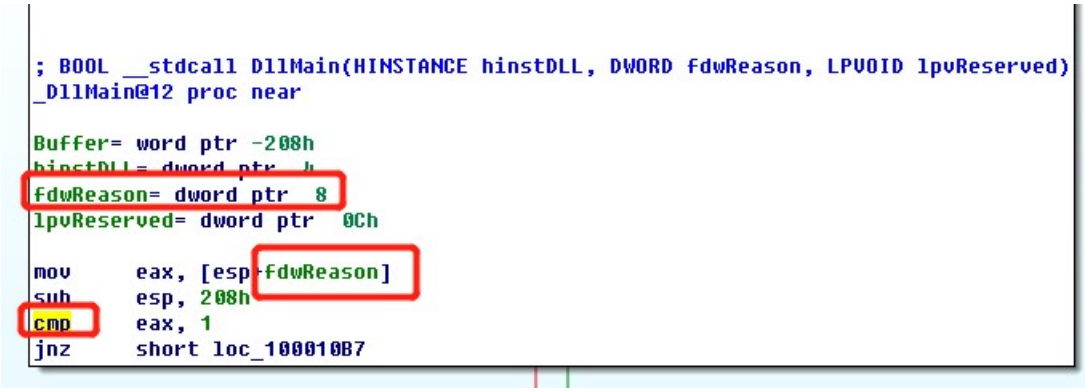
这个 exe 程序就是为了加载 msgina32.dll 文件到主机中，所以真正实现功能的还是 msgina32.dll 文件，接下来分析 msgina32.dll：

查看 Strings：



有好多 Wlx 开头的函数，而 GINA 的导出函数大部分都是这种形式的，这些函数是进行 Winlogon 和 magina.dll 之间通信的函数，所以 msgina32.dll 这个恶意程序可能截获了 Winlogon 与 msgina.dll 之间的通信，然后作为中间人，使系统正常运行，然后在这个过程中获取凭证。还发现了一个 msuti123.sys 文件，可疑。

进入 IDA 分析：

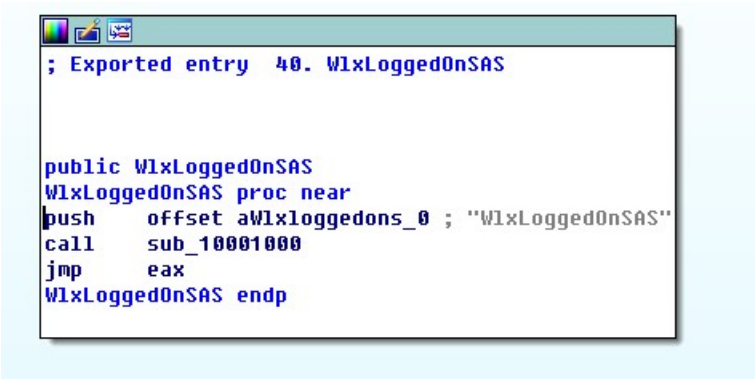


看到 fdwreason 参数，这个参数表示 DLL 函数被调用的原因，然后这个参数会与 1 进行比较，也就是在判断当前这个 DLL 文件是不是在加载时被调用的，如果是，则执行以下程序：

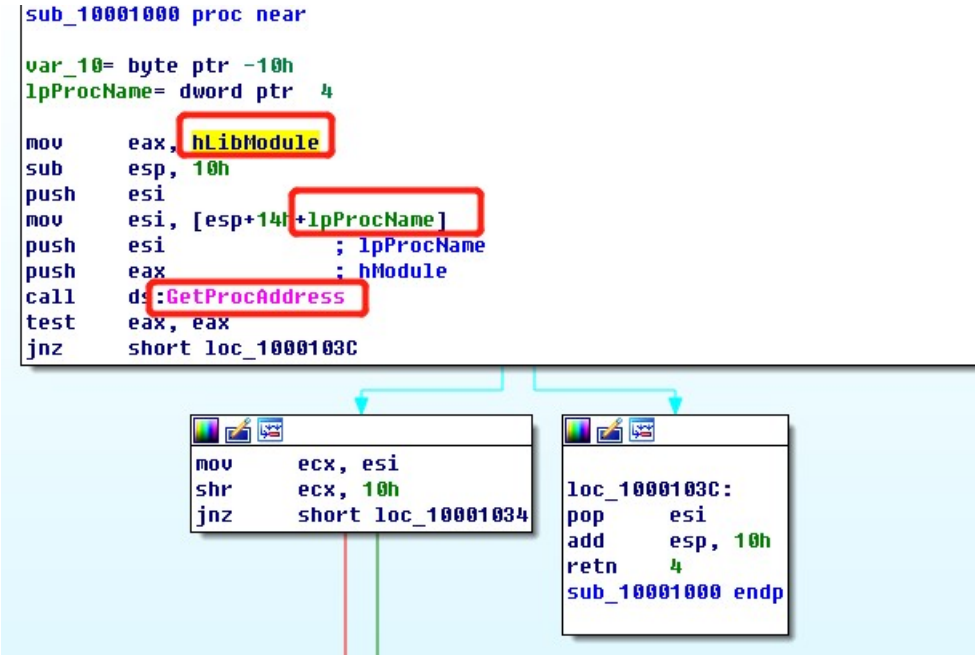
程序调用 GetSystemDirectoryW 函数，获取系统目录，然后调用 lstrcatW 把上面获取的目录与"\\MSGina"组合在一起，得到 msgina.all 的目录，再使用 LoadLibraryW 函数获取 msgina.dll 的句柄，返回值存在 eax 中，然后赋值给 hLibModule。此番操作就是向这个恶意程序中导入 msgina.dll 中的所有函数，使得恶意程序截获通信后，系统仍可以正常运行。

然后分析导出函数，看看程序到底干嘛：

找到 WlxLoggedOnSAS 进入查看：

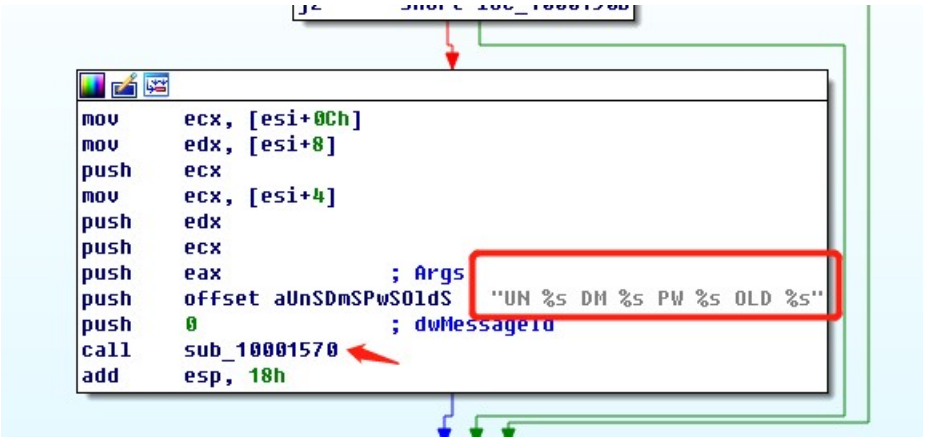


可以看到，这里现将字符串“WlxLoggedOnSAS”作为参数压栈，然后调用了 sub_10001000 这个函数，进入这个函数：



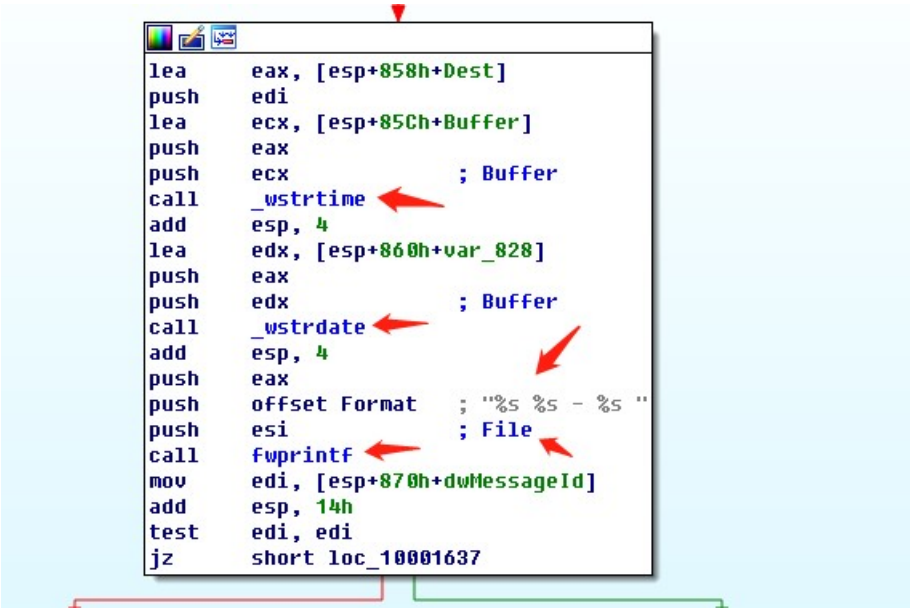
可以看到，这里使用到了我们刚刚所获取的 msgina.dll 的句柄 hLibModule 以及 WlxLoggedOnSAS（lpProcName）这个字符串，然后调用 GetProcAddress，获取 WlxLoggedOnSAS 的地址，其他的导出函数功能类似，接下来分析系统退出函数，因为这里可能有对凭证的存储操作：

查看 WlxLoggedOutSAS：



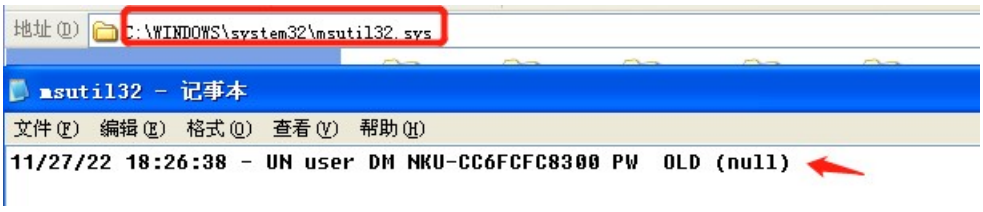
关键，有一个格式化字符串，可能是用于输出信息的，之前一堆 push 也都是函数 sub_10001570 的参数，进入这个函数：

发现这个文件名，创建这个文件然后打开；



这里将时间、日期、登录凭证写入 msutil32.sys 中。目录为 C:\WINDOWS\system32\

如果我们重启主机就会在这个文件里看到用户名和密码：



恶意程序使用 GINA 拦截并窃取用户登陆凭证。msgina32.dll 能够拦截所有提交到系统的用户登陆凭证。然后将凭证存到 msutil32.sys 文件中，放在系统目录 C:\WINDOWS\system32\中。

4、这个恶意代码对窃取的证书做了什么处理？

这个恶意代码会将凭证保存在 C:\WINDOWS\system32\msutil32.dll 中

5、如何在测试环境中让这个恶意代码获取用户登录凭证？

重启系统，会弹出一个登录界面，然后输入密码登录，这个过程会被恶意程序拦截并做如上处理。

Lab 11-2

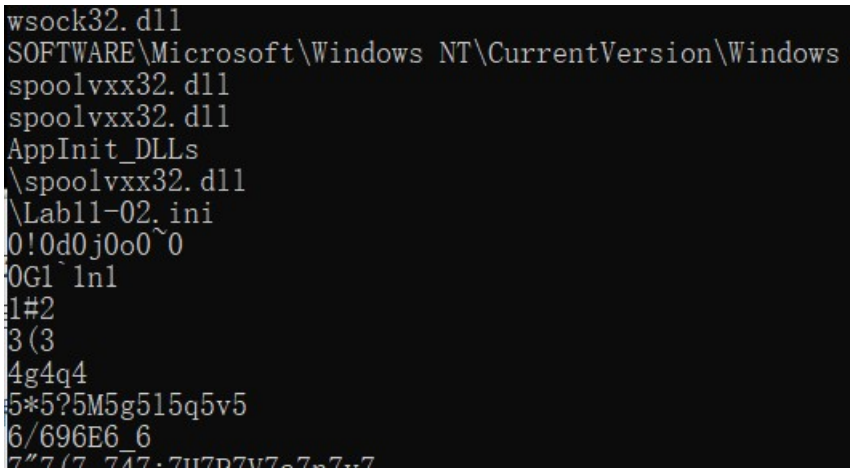
Analyze the malware found in *Lab11-02.dll*. Assume that a suspicious file named *Lab11-02.ini* was also found with this malware.

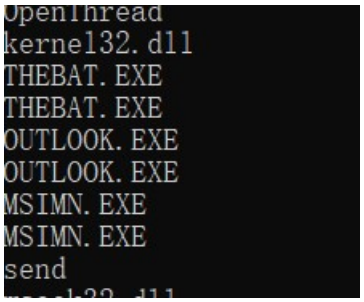
Questions

1. What are the exports for this DLL malware?
2. What happens after you attempt to install this malware using *rundll32.exe*?
3. Where must *Lab11-02.ini* reside in order for the malware to install properly?
4. How is this malware installed for persistence?
5. What user-space rootkit technique does this malware employ?
6. What does the hooking code do?
7. Which process(es) does this malware attack and why?
8. What is the significance of the *.ini* file?
9. How can you dynamically capture this malware's activity with Wireshark?

1、这个恶意 DLL 导出了什么？

查看 Strings：





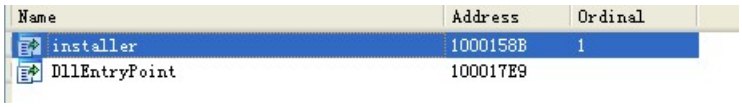
这里看到了 AppInit_DLLs 以及 SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows 这两个字符串，说明恶意程序使用 AppInit_DLLs 来实现永久的自我安装。还看到了 \Lab11-02.ini 说明有可能会调用这个文件，进程名 THEBAT.EXE、OUTLOOK.EXE 以及 MSIMN.EXE 是邮件客户端，RCPT TO:是用来创建一个邮件的收件人，send 和 wsock32.dll

表明恶意程序有会进行一些网络通信，所以这个恶意程序可能会对邮件发送进行一些不为人知的操作。

看 ini 文件：

无有效信息，可能是被加密了。

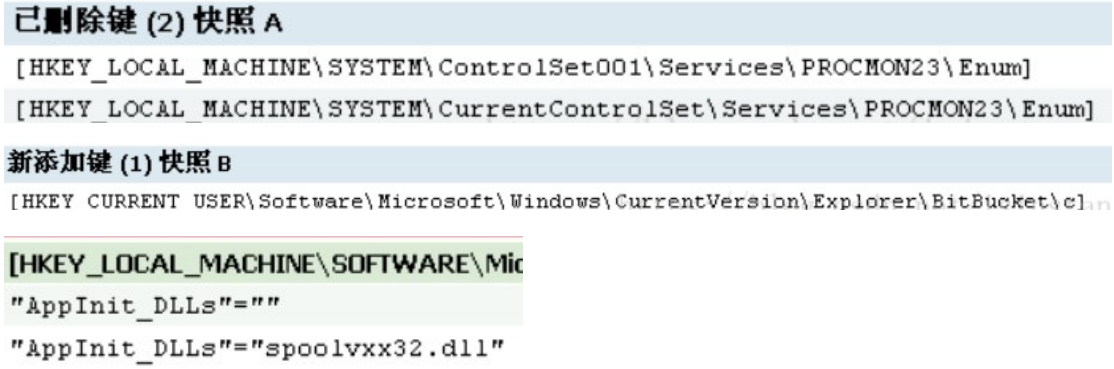
用 IDA 打开，查看导出表：



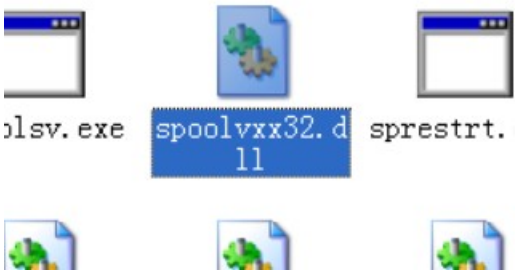
有导出函数 installer

2、使用 rundll32.exe 安装这个恶意代码后，发生了什么？

用 processmontior 监控，命令行输入：rundll32.exe Lab11-02.dll, installer



Procmon 中发现，在 C:\WINDOWS\system32\目录下，创建了一个 spoolvxx32.dll 文件



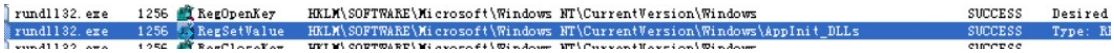
经过比较，这个文件和 Lab11-02.dll 文件完全一样。

所以，安装这个恶意代码后吗，会在系统目录下拷贝一个文件，名为 spoolvxx32.dll，与原文件一样。

3、为了使这个恶意代码正确安装，Lab11-02.ini 必须放在何处？

恶意代码会在拷贝的目录下，也就是 C:\WINDOWS\system32\下，打开 ini 文件，所以将 Lab11-02.ini 放到系统目录下就好了。

4、这个恶意代码如何驻留？




```
[HKEY_LOCAL_MACHINE\SOFTWARE\Mi
"AppInit_DLLs"=""
"AppInit_DLLs"="spoolvxx32.dll"
```

恶意程序将会添加到 AppInit_DLLs 中，即 user32.dll 中，所有装载 user32.dll 的进程，都将加载这个恶意程序，达到永久驻留的目的。

5、这个恶意代码采用的用户态 rootkit 是什么？

首先看看导出函数 installer

```
sub     esp, 8
lea     eax, [ebp+phkResult]
push    eax             ; phkResult
push    6               ; samDesired
push    0               ; ulOptions
push    offset SubKey   ; "SOFTWARE\\Microsoft\\Windows NT\\Curren"..
push    80000002h       ; hKey
call    ds:RegOpenKeyExA
test    eax, eax
jnz     short loc_100015DD
```

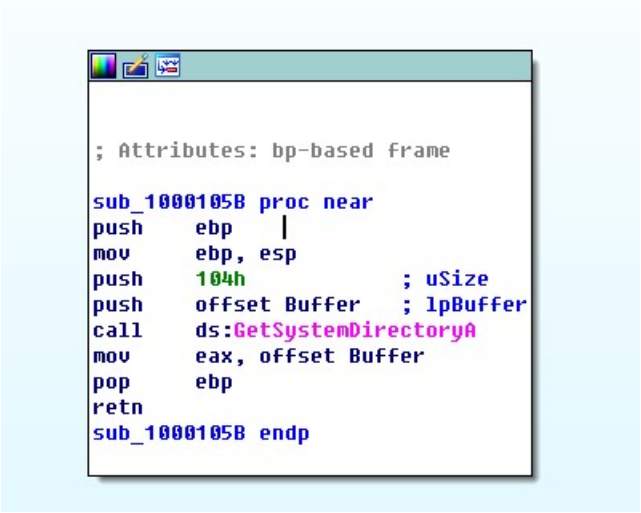
```
push    offset aSpoolvxx32_dll ; "spoolvxx32.dll"
call    strlen
add     esp, 4
push    eax             ; cbData
push    offset Data     ; "spoolvxx32.dll"
push    1               ; dwType
push    0               ; Reserved
push    offset ValueName ; "AppInit_DLLs"
mov     ecx, [ebp+phkResult]
push    ecx             ; hKey
call    ds:RegSetValueExA
```

将自己进行重命名操作，然后访问注册表 SOFTWARE\\Microsoft\\Windows NT\\Curren，找到 AppInit_DLLs 的位置，实现永久驻留。

然后看 DLLmain：

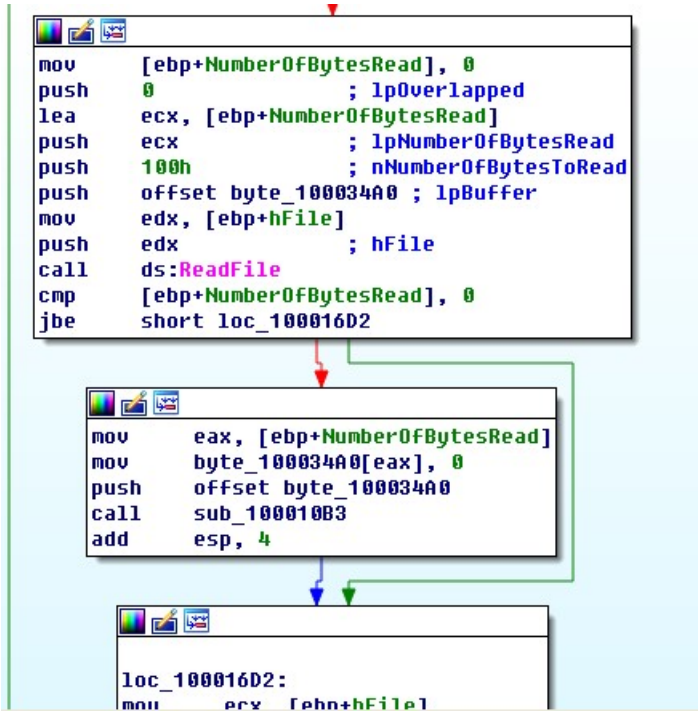
```
push    101h            ; Size
push    0               ; Val
push    offset byte_100034A0 ; Dst
call    memset
add     esp, 0Ch
call    sub_1000105B
mov     [ebp+Dest], eax
push    104h            ; Count
push    offset aLab1102_ini ; "\\Lab11-02.ini"
mov     edx, [ebp+Dest]
push    edx             ; Dest
call    strncat
add     esp, 0Ch
push    0               ; hTemplateFile
push    80h             ; dwFlagsAndAttributes
push    3               ; dwCreationDisposition
push    0               ; lpSecurityAttributes
push    1               ; dwShareMode
push    80000000h       ; dwDesiredAccess
mov     eax, [ebp+Dest]
push    eax             ; lpFileName
call    ds:CreateFileA
mov     [ebp+hFile], eax
cmp     [ebp+hFile], 0FFFFFFFFh
jz      short loc_100016DE
```

首先调用了 sub_1000105B 函数，进入



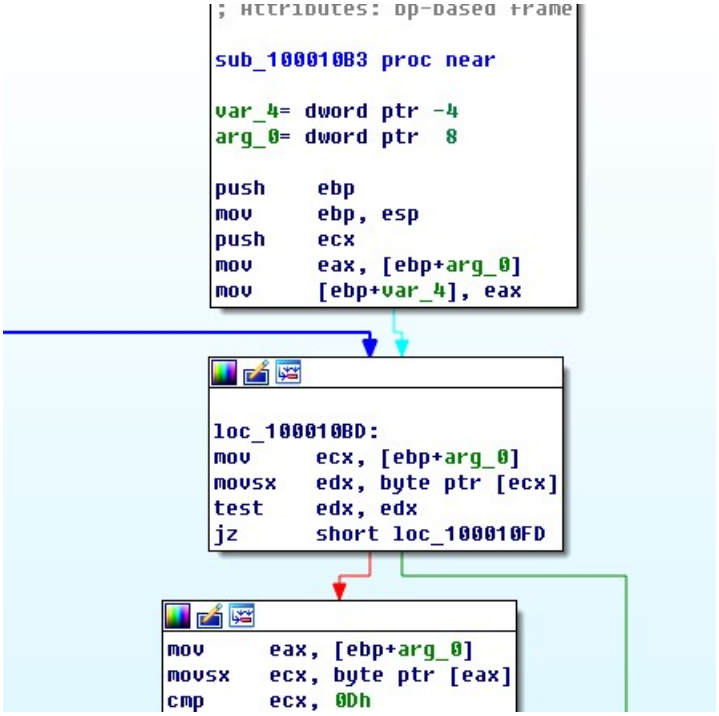
```
; Attributes: bp-based frame

sub_1000105B proc near
push    ebp
mov     ebp, esp
push    104h            ; uSize
push    offset Buffer   ; lpBuffer
call    ds:GetSystemDirectoryA
mov     eax, offset Buffer
pop     ebp
retn
sub_1000105B endp
```

获取系统目录，然后用 strncat 函数将上面获取到的目录与"\\Lab11-02.ini"组合到一起形成一个新的目录，然后用 CreateFile 函数打开上边的目录，如果打开成功（说明驻留成功），就会执行以下操作：

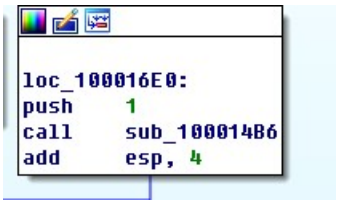


```
mov     [ebp+NumberOfBytesRead], 0
push    0               ; lpOverlapped
lea     ecx, [ebp+NumberOfBytesRead]
push    ecx             ; lpNumberOfBytesRead
push    100h            ; nNumberOfBytesToRead
push    offset byte_100034A0 ; lpBuffer
mov     edx, [ebp+hFile]
push    edx             ; hFile
call    ds:ReadFile
cmp     [ebp+NumberOfBytesRead], 0
jbe     short loc_100016D2

mov     eax, [ebp+NumberOfBytesRead]
mov     byte_100034A0[eax], 0
push    offset byte_100034A0
call    sub_100010B3
add     esp, 4

loc_100016D2:
```

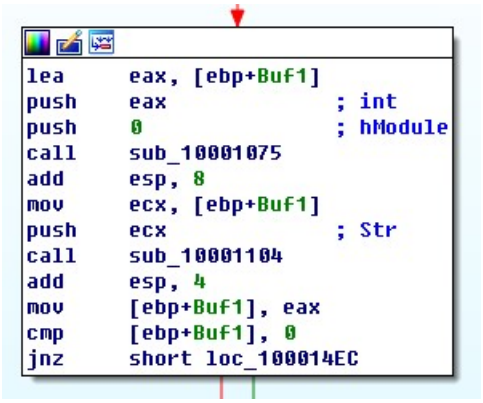
这里调用了 readfile 函数，读取文件放到缓冲区 lpBuffer 里面。后面有一个 cmp 语句来确保读取的文件大于 0。下面可以看到 push 语句会将 ini 文件压入栈然后作为参数传给 sub_100010B3 调用：



经过分析，这个函数为解密函数，解释了开头的疑问，继续：



调用了一个 sub_100014B6 函数然后程序结束，所以这个函数很关键，进入查看：

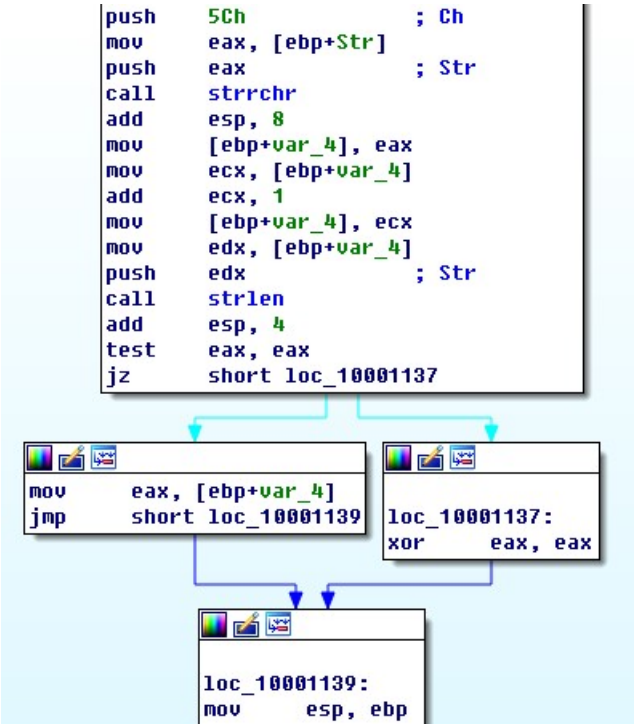


调用了 1075 函数，进入：

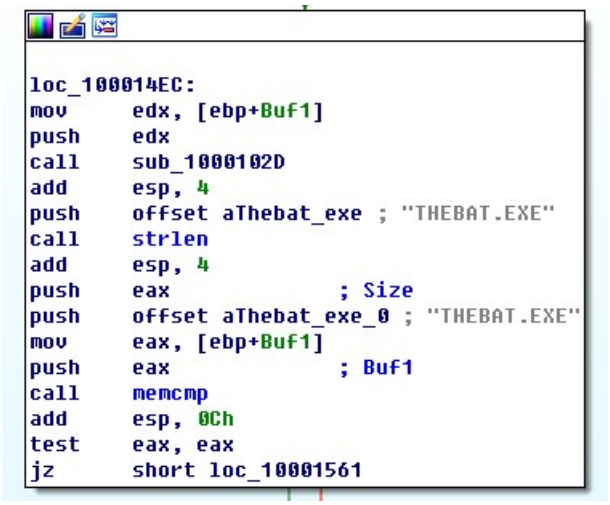
```
; Attributes: bp-based frame

; int __cdecl sub_10001075(HMODULE hModule, int)
sub_10001075 proc near

hModule= dword ptr  8
arg_4= dword ptr  0Ch

push    ebp
mov     ebp, esp
push    104h            ; nSize
push    offset Filename ; lpFilename
mov     eax, [ebp+hModule]
push    eax             ; hModule
call    ds:GetModuleFileNameA
mov     ecx, [ebp+arg_4]
mov     dword ptr [ecx], offset Filename
pop     ebp
retn
sub_10001075 endp
```

发现这里调用了函数 GetModuleFileNameA，这是获取进程名称的函数，由于它的 hModule 参数的值为 0，所以函数会返回加载这个 dll 进程的绝对路径；

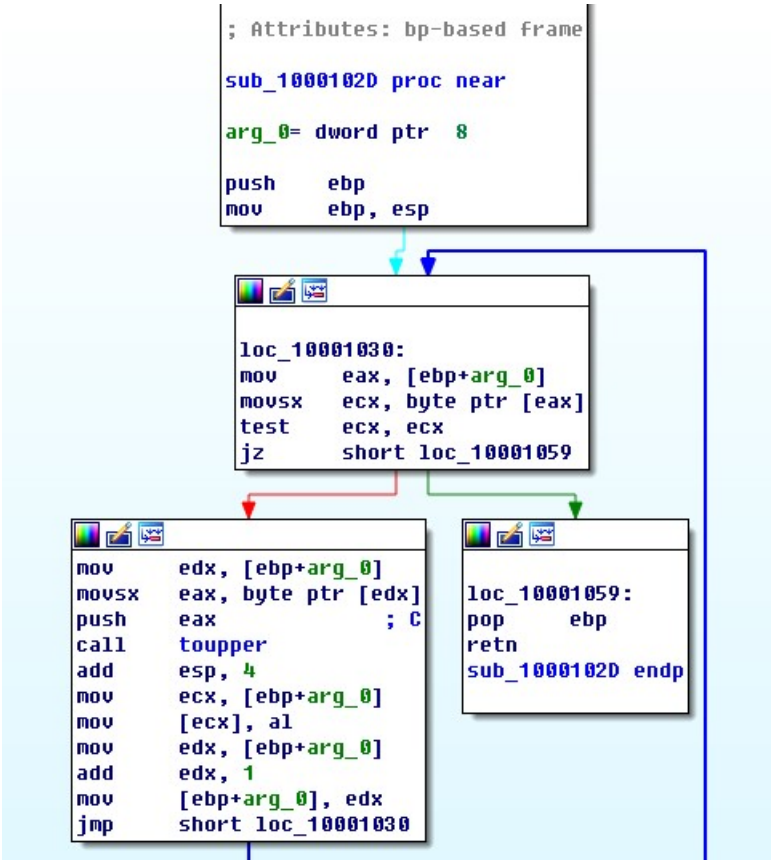
然后调用 1104 函数：



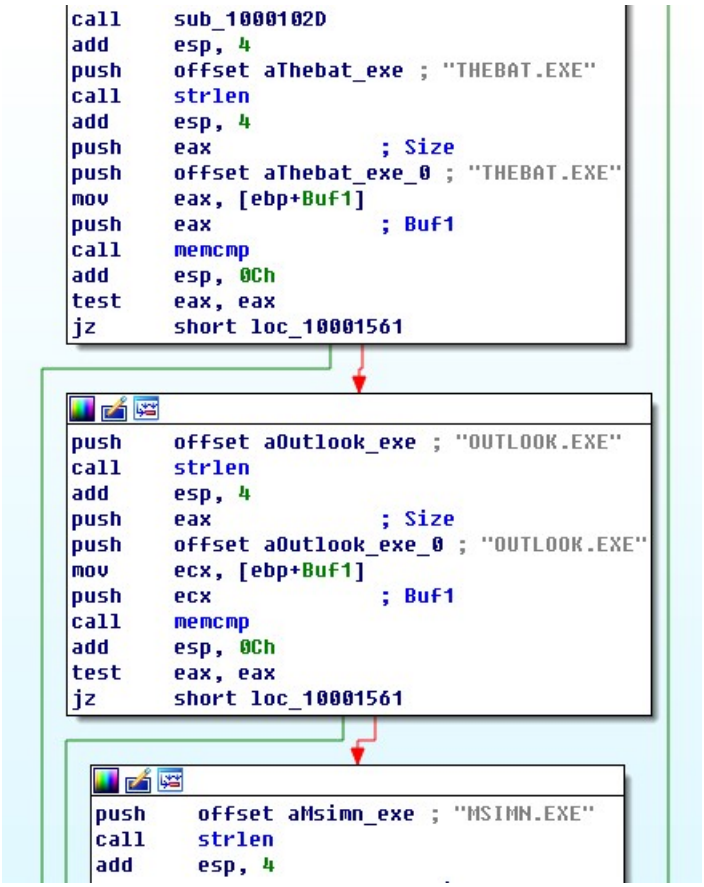
这个函数将调取的路径中的名称提取出来，继续；



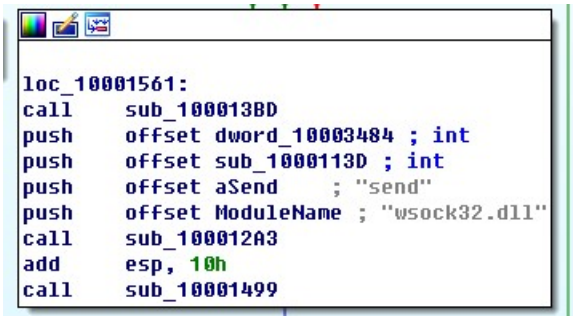
调用函数 102D，进入：



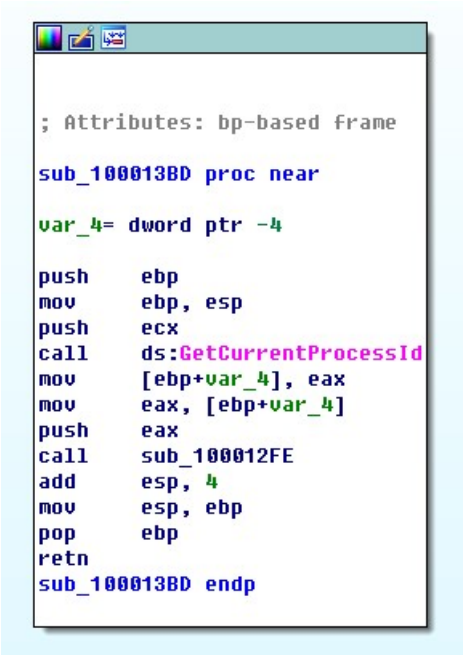
将进程名称转换为大写字符，继续；



将转换完的进程大写字符与"THEBAT.EXE"，"OUTLOOK.EXE"，"MSIMN.EXE"进行对比，任何一个不成功都会退出，都对了则继续：

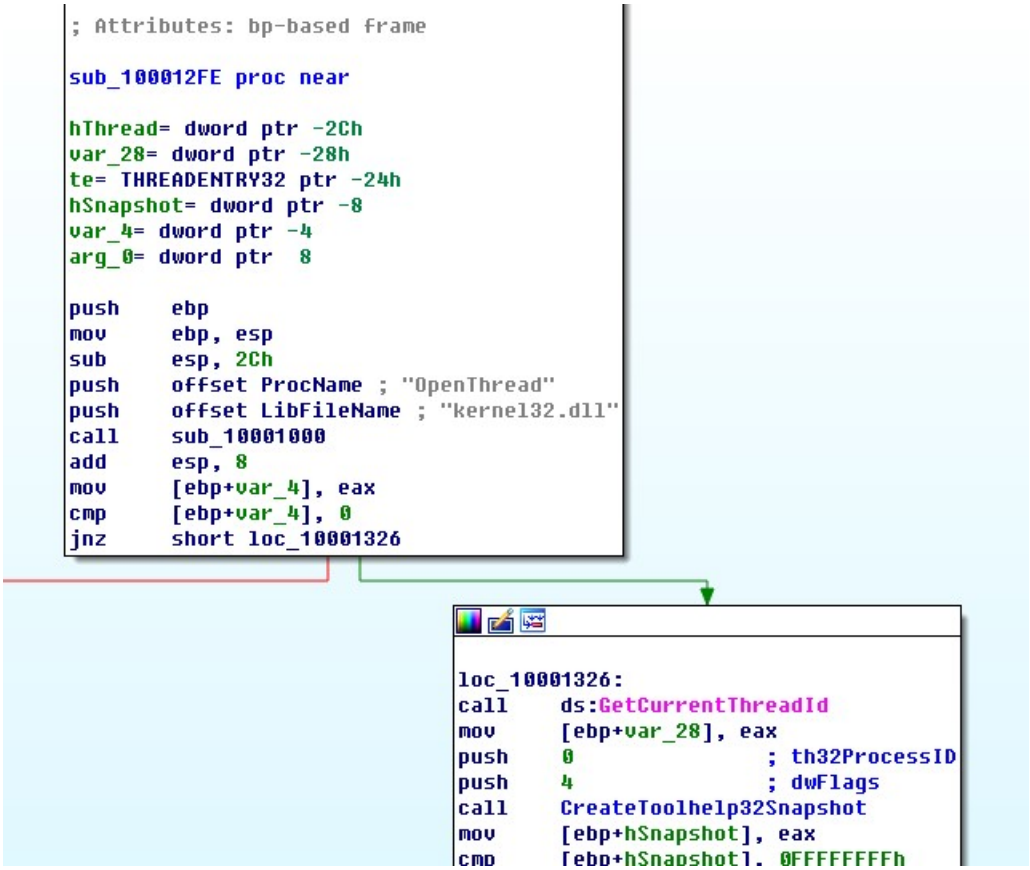


先调用了 13BD 函数，进入：

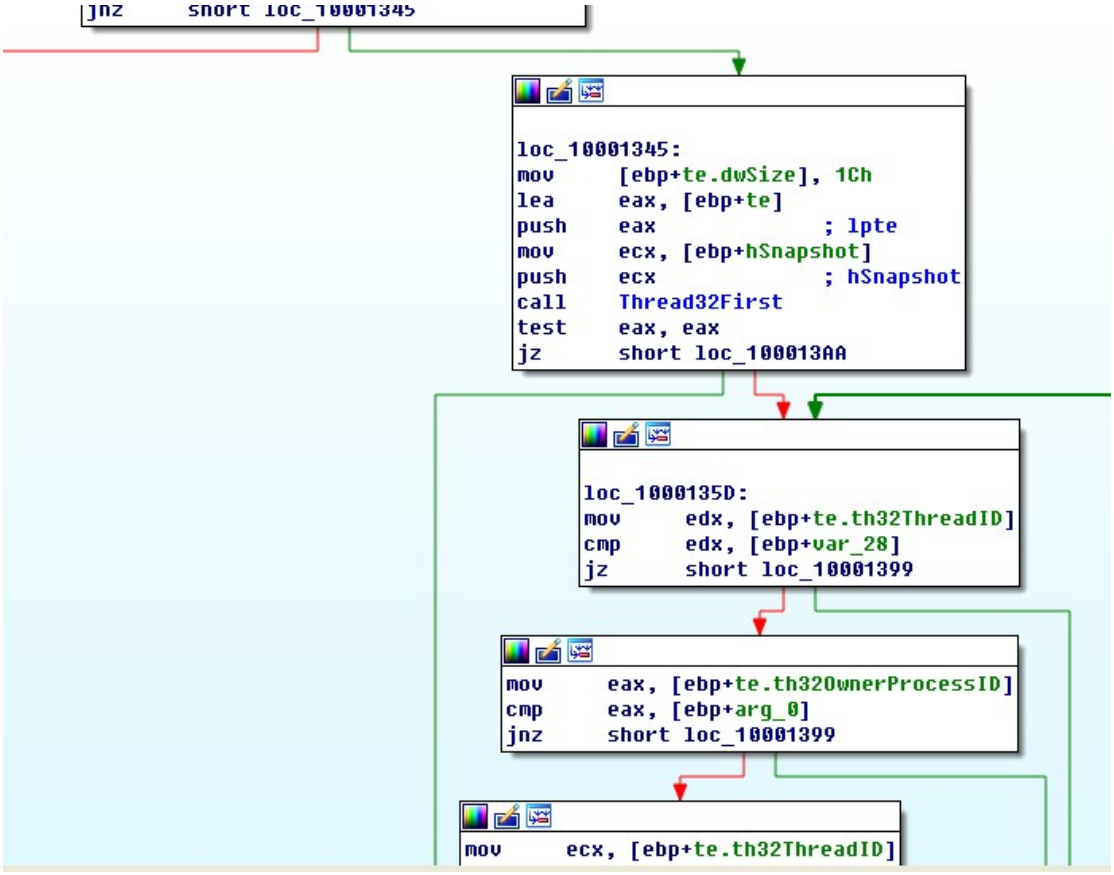


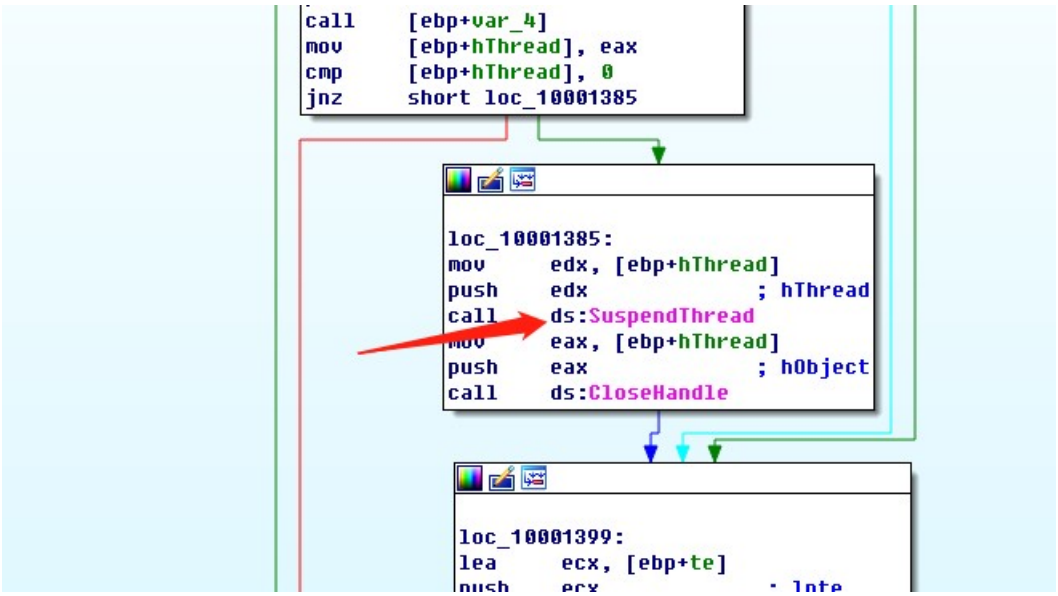
```
; Attributes: bp-based frame

sub_100013BD proc near

var_4= dword ptr -4

push    ebp
mov     ebp, esp
push    ecx
call    ds:GetCurrentProcessId
mov     [ebp+var_4], eax
mov     eax, [ebp+var_4]
push    eax
call    sub_100012FE
add     esp, 4
mov     esp, ebp
pop     ebp
retn
sub_100013BD endp
```

调用了 GetCurrentProcessId 函数，得到当前的进程，然后是 sub_100012FE 函数。查看函数 sub_100012F：



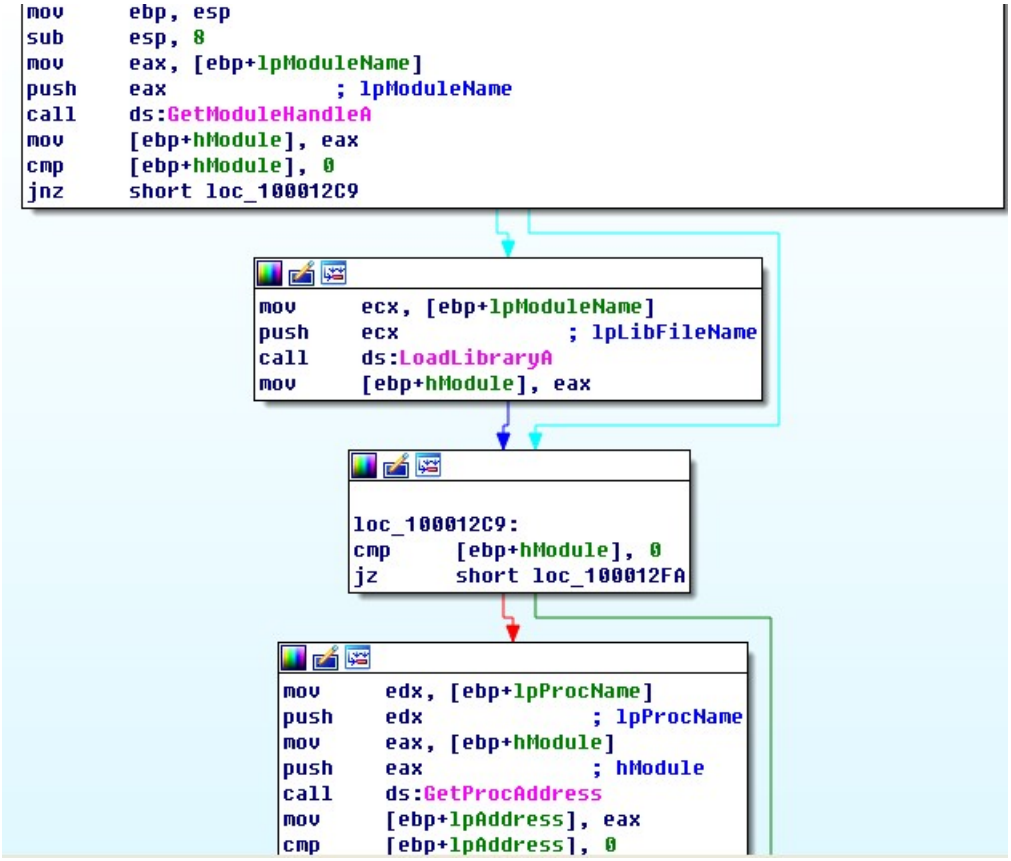
```
; Attributes: bp-based frame

sub_100012FE proc near

hThread= dword ptr -2Ch
var_28= dword ptr -28h
te= THREADENTRY32 ptr -24h
hSnapshot= dword ptr -8
var_4= dword ptr -4
arg_0= dword ptr  8

push    ebp
mov     ebp, esp
sub     esp, 2Ch
push    offset ProcName ; "OpenThread"
push    offset LibFileName ; "kernel32.dll"
call    sub_10001000
add     esp, 8
mov     [ebp+var_4], eax
cmp     [ebp+var_4], 0
jnz     short loc_10001326

loc_10001326:
call    ds:GetCurrentThreadId
mov     [ebp+var_28], eax
push    0               ; th32ProcessID
push    4               ; dwFlags
call    CreateToolhelp32Snapshot
mov     [ebp+hSnapshot], eax
cmp     [ebp+hSnapshot], 0FFFFFFFFh
```



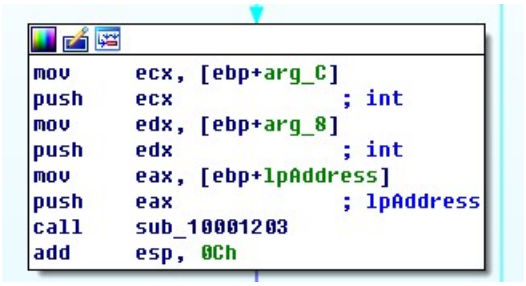


发现 12FE 这个函数，调用了 GetCurrentThreadId 函数，这个函数用于获取当前运行线程的线程标识符。使用 CreateToolhelp32Snapshot 来拍摄快照，并且不断循环当前的线程。如果当前线程不是我们之前获取的线程，那么就调用 SuspendThread 来挂起线程。所以，sub_100012FE 这个函数的目的就是为了挂起除当前线程以外的线程。

挂起之后，将 send 和 wsock32.dll 作为参数调用 12A3 函数，进入：



首先使用 GetModuleHandleA 函数来获取 wsock32.dll 的句柄。然后通过 LoadLibraryA 以及 GetProcAddress 这两个函数可以获取 wsock32.dll 中的 send 函数的地址，然后将这个地址，连同 arg_8 和 arg_C 进行了压栈的操作，也就是当作了 sub_10001203 函数的参数



进入 1203 函数：

将 arg_4 的起始地址与 send 函数的内存地址相减，获取二者地址的偏移量。然后从中减去 5 个字节，再将差值移入 var_4，继续：

```
push    eax             ; src
mov     ecx, [ebp+var_8]
add     ecx, 5
push    ecx             ; Dst
call    memcpy
add     esp, 0Ch
mov     edx, [ebp+var_8]
mov     byte ptr [edx+0Ah], 0E9h
mov     eax, [ebp+lpAddress]
sub     eax, [ebp+var_8]
sub     eax, 0Ah
mov     ecx, [ebp+var_8]
mov     [ecx+0Bh], eax
mov     edx, [ebp+lpAddress]
mov     byte ptr [edx], 0E9h
mov     eax, [ebp+lpAddress]
mov     ecx, [ebp+var_4]
mov     [eax+1], ecx
```

程序将机器码 0xE9 复制到 send 函数的开头，之后将 var_4，也就是四个字节偏移地址复制到 0xE9 的后面，继续：

```
mov     edx, [ebp+lpAddress]
push    edx             ; lpAddress
call    ds:VirtualProtect
push    0FFh            ; Size
call    malloc
add     esp, 4
mov     [ebp+var_8], eax
mov     eax, [ebp+var_8]
mov     ecx, [ebp+lpAddress]
mov     [eax], ecx
```

调用了 VirtualProtect 函数，它可以修改内存的运行、读和写等保护权限，因此可以使恶意程序修改 send 函数的执行。恶意程序使用 malloc 来分配 0xFF 字节的内存空间，并且将返回值也就是所分配空间的起始地址存入 var_8 里面，存储 send 函数原先的开头的 5 个字节，便于后续恢复然后正常执行，继续分析：

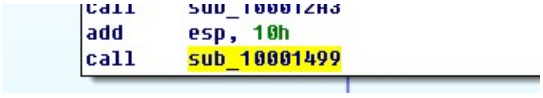
```
mov     ecx, [ebp+lpAddress]
mov     [eax], ecx
mov     edx, [ebp+var_8]
mov     byte ptr [edx+4], 5
push    5               ; Size
mov     eax, [ebp+lpAddress]
push    eax             ; Src
mov     ecx, [ebp+var_8]
add     ecx, 5
push    ecx             ; Dst
call    memcpy
add     esp, 0Ch
mov     edx, [ebp+var_8]
mov     byte ptr [edx+0Ah], 0E9h
mov     eax, [ebp+lpAddress]
sub     eax, [ebp+var_8]
sub     eax, 0Ah
mov     ecx, [ebp+var_8]
mov     [ecx+0Bh], eax
mov     edx, [ebp+lpAddress]
mov     byte ptr [edx], 0E9h
mov     eax, [ebp+lpAddress]
mov     ecx, [ebp+var_4]
mov     [eax+1], ecx
lea     edx, [ebp+flOldProtect]
push    edx             ; lpflOldProtect
mov     eax, [ebp+flOldProtect]
push    eax             ; flNewProtect
push    5               ; dwSize
```

调用 memcpy 函数，将 send 函数的前五个字节的内容保存到刚才分配的空间里。然后向分配的空间中添加了 0xE9 这个机器码，接着又添加了跳转地址 sub_1000113D，这样程序就可

以跳回 send 函数继续执行，所以这个程序使用了钩子技术，之后就是一些恢复现场的工作了；

最后程序又调用了 VirtualProtect 这个函数，目的就是恢复原始的内存保护设置；

12A3 函数完。



之后主程序调用的 1499 函数，作用于 13BD 相反，调用了 ResumeThread 用于线程的恢复，即挂起后的恢复线程。

所以这个程序利用了 HOOK 技术。

6、挂钩代码做了什么？

由之前的分析可知，挂钩代码的地址为 sub_1000113D，查看：

```
; Attributes: bp-based frame

; int __stdcall sub_1000113D(int, char *Str, int, int)
sub_1000113D proc near

Dst= byte ptr -204h
arg_0= dword ptr  8
Str= dword ptr  0Ch
arg_8= dword ptr  10h
arg_C= dword ptr  14h

push    ebp
mov     ebp, esp
sub     esp, 204h
push    offset SubStr   ; "RCPT TO:"
mov     eax, [ebp+Str]
push    eax             ; Str
call    strstr
add     esp, 8
test    eax, eax
jz      loc_100011E4
```

```
push    offset Str      ; "RCPT TO: <"
call    strlen
add     esp, 4
push    eax             ; Size
push    offset aRcptTo_1 ; "RCPT TO: <"
lea     ecx, [ebp+Dst]
push    ecx             ; Dst
call    memcpy
add     esp, 0Ch
push    101h            ; Size
push    offset byte_100034A0 ; Src
push    offset aRcptTo_2 ; "RCPT TO: <"
call    strlen
add     esp, 4
lea     edx, [ebp+eax+Dst]
push    edx             ; Dst
call    memcpy
add     esp, 0Ch
push    offset Source   ; ">\r\n"
lea     eax, [ebp+Dst]
push    eax             ; Dest
call    strcat
add     esp, 8
mov     ecx, [ebp+arg_C]
push    ecx
lea     edx, [ebp+Dst]
push    edx             ; Str
call    strlen
add     esp, 4
push    eax
lea     eax, [ebp+Dst]
push    eax
mov     ecx, [ebp+arg_0]
```

```
loc_100011E4:
mov     edx, [ebp+arg_C]
push    edx
mov     eax, [ebp+arg_8]
push    eax
mov     ecx, [ebp+Str]
push    ecx
mov     edx, [ebp+arg_0]
push    edx
call    dword_10003484
add     esp, 10h
mov     esp, ebp
pop     ebp
retn    10h
sub_1000113D endp
```

这个函数首先在传入的字符串中查找“RCPT TO:”。如果存在，那么恶意程序就会创建一个字符串，用于向外进行传输。这个字符串以“RCPT TO: <”作为开头，以“>\r\n”结束。

所以，这段代码的作用就是检查向外发出的数据包，查看这些数据包里面是否含有“RCPT TO:”的电子邮件信息，如果发现了这个字符串，那么就会添加一个额外的“RCPT TO”，来增加一个恶意的电子邮件账户。

7、哪个或者哪些进程执行这个恶意攻击，为什么？

由之前的分析，会进行进程比对，比对的三个进程为，MSIMN.exe、THEBAT.exe 和 OUTLOOK.exe，这三个都是电子邮件的客户端程序，因此这个恶意代码就是对发出的电子邮件进行检查，对某些邮件，额外发给另一个恶意邮件账户。

8、.ini 文件的意义是什么？

这个.ini 文件里面包含一个经过加密的邮件地址，解密之后发现就是一个邮箱 billy@malwareanalysisbook.com

9、你怎样用 wireshark 动态抓获这个恶意代码的行为？

运行程序后，随便发送一个邮件，在 wireshark 中会发现邮件发送给了两个人，一个是你自己发送的那个目的账户，同时还发送给了恶意代码的作者。

Lab 11-3

Analyze the malware found in *Lab11-03.exe* and *Lab11-03.dll*. Make sure that both files are in the same directory during analysis.

Questions

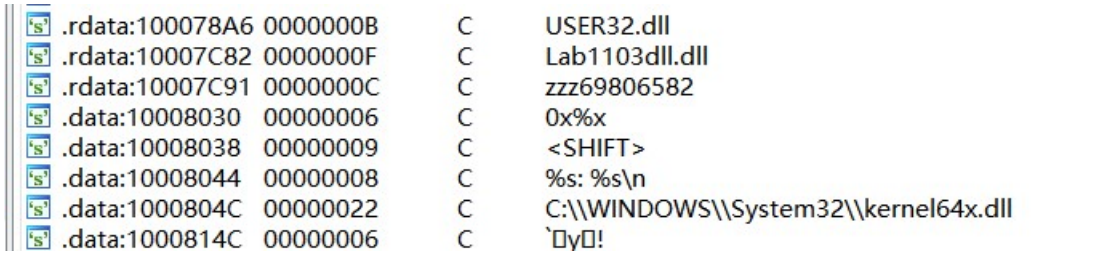
1. What interesting analysis leads can you discover using basic static analysis?
2. What happens when you run this malware?
3. How does *Lab11-03.exe* persistently install *Lab11-03.dll*?
4. Which Windows system file does the malware infect?
5. What does *Lab11-03.dll* do?
6. Where does the malware store the data it collects?

1、使用基础的静态分析过程，你可以发现什么有趣的线索?

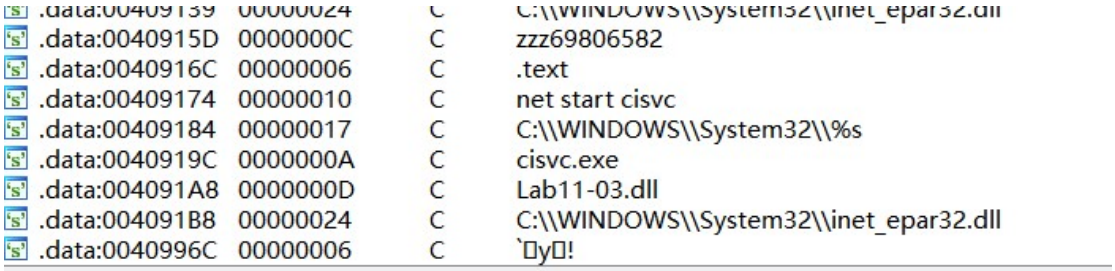
Dll：导入表有关于按键的函数

导出一个函数：

Strings 很奇怪：



Exe：Strings 很奇怪



2、当运行这个恶意代码时，发生了什么?

把自己的 dll 复制到了 system32 目录下，改名为 inet_epar32.dll，修改了 cisvc 程序，然后通过 cmd 执行了命令：cmd.exe /c net start cisvc 启动了一个服务

动作	路径	参数
EXEC_create	C:\Documents and Settings\Administrator\桌面\Lab11-03.exe	parent_pid:1588 cmdline:'"C:\Documents and Settings\Administrator\桌面\Lab
FILE_open	C:\Documents and Settings\Administrator\桌面\Lab11-03.dll	access:0x00120089 alloc_size:0 attrib:0x00000000 share_access:0x00000001 d
FILE_touch	C:\WINDOWS\System32\inet_epar32.dll	access:0x00130196 alloc_size:0 attrib:0x00000020 share_access:0x00000000 d
FILE_open	C:\WINDOWS\System32\inet_epar32.dll	access:0x00130196 alloc_size:0 attrib:0x00000000 share_access:0x00000000 d
FILE_write	C:\WINDOWS\System32\inet_epar32.dll	offset:0x00000000 datalen:0x0000C000
FILE_modified	C:\WINDOWS\System32\inet_epar32.dll	
FILE_modified	C:\WINDOWS\System32\cisvc.exe	
FILE_open	C:\WINDOWS\system32\cmd.exe	access:0x001000A1 alloc_size:0 attrib:0x00000000 share_access:0x00000005 d
FILE_open	C:\WINDOWS\system32\apphelp.dll	access:0x00100020 alloc_size:0 attrib:0x00000000 share_access:0x00000005 d
FILE_open	C:\WINDOWS\system32\apphelp.dll	access:0x00100020 alloc_size:0 attrib:0x00000000 share_access:0x00000005 d
FILE_open	C:\WINDOWS\AppPatch\sysmain.sdb	access:0x00120089 alloc_size:0 attrib:0x00000000 share_access:0x00000001 d
FILE_readdir	C:\WINDOWS\system32	filter:'cmd.exe'
FILE_readdir	C:\	filter:'WINDOWS'
FILE_readdir	C:\WINDOWS	filter:'system32'
FILE_readdir	C:\WINDOWS\system32	filter:'cmd.exe'
FILE_open	C:\WINDOWS\system32\cmd.exe	access:0x00100020 alloc_size:0 attrib:0x00000000 share_access:0x00000005 d
FILE_open	C:\WINDOWS\system32\cmd.exe	access:0x00120089 alloc_size:0 attrib:0x00000000 share_access:0x00000005 d
FILE_open	C:\WINDOWS\system32\cmd.exe	access:0x00100020 alloc_size:0 attrib:0x00000000 share_access:0x00000005 d
FILE_open	C:\WINDOWS\system32\cmd.exe	access:0x00120089 alloc_size:0 attrib:0x00000000 share_access:0x00000005 d
FILE_readdir	C:\	filter:'WINDOWS'
FILE_readdir	C:\WINDOWS	filter:'system32'
FILE_readdir	C:\WINDOWS\system32	filter:'cmd.exe'
FILE_readdir	C:\	filter:'WINDOWS'
FILE_readdir	C:\WINDOWS	filter:'system32'
FILE_readdir	C:\WINDOWS\system32	filter:'cmd.exe'
FILE_read	C:\WINDOWS\system32\cmd.exe	offset:0x00000000 datalen:0x00073000
PROC_exec	C:\WINDOWS\system32\cmd.exe	target_pid:2736 cmdline:'C:\WINDOWS\system32\cmd.exe /c net start cisvc'
THRD_resume	C:\WINDOWS\system32\cmd.exe	target_pid:2736 target_tid:2120
EXEC_destroy	C:\Documents and Settings\Administrator\桌面\Lab11-03.exe	parent_pid:1588 cmdline:'"C:\Documents and Settings\Administrator\桌面\Lab

3、Lab11-03.exe 如何安装 Lab11-03.dll 使其长期驻留?

通过修改服务文件 cisvc.exe 实现驻留，除了去读反汇编分析改了哪些内容，还可以通过对比二进制文件来了解：

这里有两个字符串，分别是 dll 路径和它的导出函数，大概率这里是加载这个导出函数来运行，拖入 OD 便运行便配合 IDA 进行分析，功能和推断一致：

```
.text:01001AD5                pop     ebx
.text:01001AD6                call    sub_1001AB4     ; 获取kernel32的modulebase
.text:01001ADB                mov     edx, eax
.text:01001ADD                push    753A4FCh
.text:01001AE2                push    edx
.text:01001AE3                call    sub_1001A57     ; 获取LoadLibraryExA函数地址
.text:01001AE8                mov     [ebp-4], eax
.text:01001AEB                push    7C0DFCAAh
.text:01001AF0                push    edx
.text:01001AF1                call    sub_1001A57     ; 获取GetProcAddress函数地址
.text:01001AF6                mov     [ebp-0Ch], eax
.text:01001AF9                lea     eax, [ebx+0]    ; 模块地址
.text:01001AFF                push    0
.text:01001B04                push    0
.text:01001B09                push    eax
.text:01001B0A                call    dword ptr [ebp-4] ; LoadLibraryExA
.text:01001B0D                mov     [ebp-10h], eax
.text:01001B10                lea     eax, [ebx+24h]
.text:01001B16                push    eax
.text:01001B17                mov     eax, [ebp-10h]
.text:01001B1A                push    eax
.text:01001B1B                call    dword ptr [ebp-0Ch] ; GetProcAddress
.text:01001B1E                mov     [ebp-8], eax
.text:01001B21                call    dword ptr [ebp-8] ; 调用获取的函数
.text:01001B24                mov     esp, ebp
.text:01001B26                pop     ebp
.text:01001B27                jmp     loc_100129B     ; 回到真正的OEP
```

4、这个恶意代码感染 Windows 系统的哪个文件?

根据之前的分析，恶意代码感染的是 cisvc.exe 文件

5、Lab11-03.dll 做了什么?

创建了一个线程：

```
text:10001540                public zzz69806582
text:10001540 zzz69806582    proc near               ; DATA XREF: .rdata:off_10007C78↓o
text:10001540
text:10001540 var_4          = dword ptr -4
text:10001540
text:10001540                push    ebp
text:10001541                mov     ebp, esp
text:10001543                push    ecx
text:10001544                push    0               ; lpThreadId
text:10001546                push    0               ; dwCreationFlags
text:10001548                push    0               ; lpParameter
text:1000154A                push    offset StartAddress ; lpStartAddress
text:1000154F                push    0               ; dwStackSize
text:10001551                push    0               ; lpThreadAttributes
text:10001553                call    ds:CreateThread ; 创建了一个线程
text:10001559                mov     [ebp+var_4], eax
text:1000155C                cmp     [ebp+var_4], 0
text:10001560                jz      short loc_10001566
text:10001562                xor     eax, eax
text:10001564                jmp     short loc_1000156B
text:10001566 ; ---------------------------------------------------------------------------
text:10001566
text:10001566 loc_10001566:                          ; CODE XREF: zzz69806582+20↑j
text:10001566                mov     eax, 1
text:1000156B
text:1000156B loc_1000156B:                          ; CODE XREF: zzz69806582+24↑j
text:1000156B                mov     esp, ebp
text:1000156D                pop     ebp
text:1000156E                retn
text:1000156E zzz69806582    endp
text:1000156F
```

线程函数首先判断互斥量：

```
.text:1000147A                push    0               ; bInheritHandle
.text:1000147C                push    1F0001h         ; dwDesiredAccess
.text:10001481                call    ds:OpenMutexA   ; 打开互斥量
.text:10001487                mov     [ebp+hObject], eax
.text:1000148D                cmp     [ebp+hObject], 0
.text:10001494                jz      short loc_1000149D
.text:10001496                push    0               ; Code
.text:10001498                call    _exit
.text:1000149D ; ---------------------------------------------------------------------------
.text:1000149D
.text:1000149D loc_1000149D:                         ; CODE XREF: StartAddress+84↑j
.text:1000149D                push    offset Name     ; "MZ"
.text:100014A2                push    1               ; bInitialOwner
.text:100014A4                push    0               ; lpMutexAttributes
.text:100014A6                call    ds:CreateMutexA ; 创建互斥量
.text:100014AC                mov     [ebp+hObject], eax
.text:100014B2                cmp     [ebp+hObject], 0
.text:100014B9                jnz     short loc_100014BD
.text:100014BB                jmp     short loc_10001530
.text:100014BD ; ---------------------------------------------------------------------------
```

然后接下来创建了一个文件，进入了一个函数，这个函数的功能就是按键记录器

```
.text:100014BD loc_100014BD:                           ; CODE XREF: StartAddress+A9↑j
.text:100014BD                 push    0               ; hTemplateFile
.text:100014BF                 push    80h ; '€'       ; dwFlagsAndAttributes
.text:100014C4                 push    4               ; dwCreationDisposition
.text:100014C6                 push    0               ; lpSecurityAttributes
.text:100014C8                 push    1               ; dwShareMode
.text:100014CA                 push    0C0000000h      ; dwDesiredAccess
.text:100014CF                 push    offset FileName ; "C:\\WINDOWS\\System32\\kernel64x.dll"
.text:100014D4                 call    ds:CreateFileA
.text:100014DA                 mov     [ebp+hFile], eax
.text:100014E0                 cmp     [ebp+hFile], 0
.text:100014E7                 jnz     short loc_100014EB
.text:100014E9                 jmp     short loc_10001530
.text:100014EB ; ---------------------------------------------------------------------------
.text:100014EB
.text:100014EB loc_100014EB:                           ; CODE XREF: StartAddress+D7↑j
.text:100014EB                 push    2               ; dwMoveMethod
.text:100014ED                 push    0               ; lpDistanceToMoveHigh
.text:100014EF                 push    0               ; lDistanceToMove
.text:100014F1                 mov     eax, [ebp+hFile]
.text:100014F7                 push    eax             ; hFile
.text:100014F8                 call    ds:SetFilePointer
.text:100014FE                 mov     ecx, [ebp+hFile]
.text:10001504                 mov     [ebp+var_4], ecx
.text:10001507                 lea     edx, [ebp+var_810]
.text:1000150D                 push    edx
.text:1000150E                 call    sub_10001380
.text:10001513                 add     esp, 4
.text:10001516                 mov     eax, [ebp+hFile]
.text:1000151C                 push    eax             ; hObject
.text:1000151D                 call    ds:CloseHandle
.text:10001523                 mov     ecx, [ebp+hObject]
.text:10001529                 push    ecx             ; hObject
.text:1000152A                 call    ds:CloseHandle
```

6、这个恶意代码将收集到的信息存到了哪里？

都存到了 C:\Windows\System32\kernel64x.dll

Yara 规则

```
rule Lab11_1_feature{
    meta:
        description = "Lab11-01.exe's features"
    strings:
        $s1 = "GinaDLL" fullword ascii
        $s2 = "msgina32.dll" fullword ascii
        $s3 = "TGAD" fullword ascii
    condition:
        $s1 and $s2 and $s3
}
```

```
rule Lab11_2_feature{
    meta:
        description = "Lab11-02.exe’s features"
    strings:
        $s1 = "THEBAT.EXE" fullword ascii
        $s2 = "OUTLOOK.EXE" fullword ascii
        $s3 = "MSIMN.EXE" fullword ascii
    condition:
        $s1 and $s2 and $s3
}

rule Lab11_3_feature{
    meta:
        description = "Lab11-03.exe’s features"
    strings:
        $s1 = "Lab1103dll.dll" fullword ascii
        $s2 = "SunMonTueWedThuFriSat" fullword ascii
        $s3 = "JanFebMarAprMayJunJulAugSepOctNovDec" fullword ascii
    condition:
        $s1 and $s2 and $s3
}
```

结果：

```
D:\College Course\大三\恶意代码\Lab11>yara64 lab11.yara -r "D:\malware tools\Chapter_11L"
Lab11_2 D:\malware tools\Chapter_11L\Lab11-02.dll
Lab11_1 D:\malware tools\Chapter_11L\Lab11-01.exe
Lab11_3 D:\malware tools\Chapter_11L\Lab11-03.dll
```

IDA Python 脚本：

1、快速定位到 init_array 函数：

```
def goInitarray(self):
    # _get_modules 是 idc 提供的接口
```

```
for module in idc._get_modules():
    # 遍历所有 module，找到 linker
    module_name = module.name
    if 'linker' in module_name:
        print 'linker address is ' + str(hex(module.base + 0x2464))
        # 0x2464 是 Android 某个版本的 init_array 的偏移地址,
        # jumpto 可以直接跳转到目标地址
        idc.jumpto(module.base + 0x2464)
        # 在 init_array 上下个断点
        idc.add_bpt(module.base + 0x2464, 1)
        # makecode 更不用说了，相当于 C
        idaapi.auto_make_code(module.base + 0x2464)
```

2、保存日志、函数名字

即保存某些寄存器的值或者某个函数名之类的，方便快速回到调试之前

```
# 通过起始地址，终止地址，以及偏移地址去保存日志
def saveDebugMessage(self):
    # create file first
    # 用个轻量级的存储 shelve
    f = shelve.open(self.id)
    # 保存日志的起始地址
    addr_start = int(self.address_start, 16)
    # 保存日志的终止地址
    addr_end = int(self.address_end, 16)
    log_dict = {}
    log_dict_list = []
    for num in range(addr_start, addr_end):
        # 获取我们当前地址的日志
        com = idc.GetCommentEx(num, True)
        if com != None:
            #获取函数名
```

```
                fun_name = idc.GetFunctionName(num)
                print fun_name
                if fun_name != None and not 'sub' in fun_name:
                    log_dict = {'offset': str(num - addr_start), 'msg': str(com), 'funtion_name':
str(fun_name)}
                else:
                    log_dict = {'offset': str(num - addr_start), 'msg': str(com)}
                log_dict_list.append(log_dict)
                pass
        print(log_dict_list)
        # 保存日志
        f['info'] = log_dict_list
        f.close()
    # 通过起始地址即可，会自动判断长度，并且获取偏移地址去设置日志
def loadDebugMessage(self):
        f = shelve.open(self.id)
        data = f['info']
        addr_start = int(self.address_start, 16)
        for num in range(0, len(data)):
            offset = data[num]['offset']
            msg = data[num]['msg']
            fun_name = data[num]['funtion_name']
            idc.MakeRptCmt(addr_start + int(offset), msg)
            if fun_name is not None and fun_name != '':
                idc.SetFunctionCmt(addr_start + int(offset), fun_name, False)
```

四、 实验心得

通过本次实验，了解了恶意代码的行为，从静态、动态等多方面了解恶意代码，资

源文件、注册表、进程监控，对于各种工具的使用更娴熟，也复习了 yara 规则和 ida python 脚本编写，收获颇丰。