



网 络 空 间 安 全 学 院

# 恶意代码分析与防治技术 实验报告

Lab 11：恶意代码行为



姓名：辛浩然

学号：2112514

年级：2021 级

专业：信息安全、法学

班级：信息安全、法学

实验目的

实验原理

实验环境

#### Lab 11-01

基本静态分析

导入表

字符串

资源节

基础动态分析

问题一：释放dll文件

问题二：恶意代码的驻留行为

分析 msgina32.dll

基本静态分析

IDA Pro 静态分析

WlxLoggedOnSAS

WlxLoggedOutSAS

问题三：窃取用户凭证

问题四：保存窃取记录

问题五：窃取登录凭证测试

#### Lab 11-02

基本静态分析

问题一：导出函数

问题二：动态运行恶意代码

问题三：Lab11-02.ini 的位置

问题四：恶意代码驻留

问题八：INI 文件的意义

问题七：恶意代码的攻击目标

问题五：HOOK 分析

分析 sub\_100013BD

安装inline挂钩，为inline挂钩创建trampoline

问题六：挂钩函数

问题九：抓取行为

#### Lab 11-03

问题一：基础静态分析

问题二：动态运行恶意代码

问题三：安装 Lab11-03.dll 并长期驻留

问题四：感染文件

问题五：Lab11-03.dll的行为

问题六：收集的数据

#### Yara 规则编写

#### IDA Python 自动化分析

编写脚本查看导入的动态链接库

编写脚本查找特定函数，分析其控制流，并识别其中的基本块和交叉引用

实验结论及实验心得

## 实验目的

---

- 熟悉恶意代码的一些常见功能，理解恶意代码存活、隐藏、实现功能的基本方法；
- 提高综合分析恶意代码的能力。

## 实验原理

---

恶意代码的常见功能有：

1. 下载器（Downloader）： 恶意代码专注于下载并安装其他恶意软件或组件，扩大攻击面。
2. 启动器（Dropper）： 恶意代码负责释放、部署和启动其他恶意软件，通常用于隐藏攻击的真实性质。
3. 后门（Backdoor）： 创建系统访问后门，使攻击者能够远程控制受感染的系统，执行恶意操作而不被察觉。
4. 远程控制工具（Remote Access Trojan, RAT）： 允许攻击者在远程地点控制受感染系统，执行指定的操作，通常用于窃取敏感信息或实施其他攻击。
5. 僵尸网络（Botnet）： 将大量受感染的计算机连接在一起，形成网络，攻击者通过远程控制这个网络进行协同攻击，如分布式拒绝服务攻击（DDoS）。
6. GINA拦截（GINA Hooking）： 修改Windows图形身份验证接口，以窃取用户凭据或绕过身份验证。
7. 口令哈希转储（Password Hash Dumping）： 通过攻击获取系统中存储的用户口令哈希，用于后续的破解和未授权访问。
8. 击键记录（Keylogging）： 记录和监视用户的击键活动，以获取敏感信息，如用户名、密码等。

恶意代码在系统中有多种方法存活并隐藏自身：

1. 修改系统注册表（Registry Modification）： 恶意代码通过更改系统注册表中的关键设置来维持其存在，使其在系统重启后仍然运行。
2. 提权（Privilege Escalation）： 利用漏洞或技术手段提升自身权限，以获取对系统更广泛的控制权。
3. 隐藏踪迹（Anti-Forensic Techniques）： 用户态的Rootkit等技术被用于在系统中隐藏恶意代码的存在，防止被安全工具或分析人员发现。

## 实验环境

---

虚拟机：关闭病毒防护的Windows XP SP3；每次病毒分析前拍摄快照，并在分析后恢复快照。

宿主机：Windows 11。

分析工具：

- 静态分析工具：IDA Pro、Resource Hacker、procmon、Process Explorer、RegShot等；
- 动态分析工具：OllyDbg等。

## Lab 11-01





### 基本静态分析

#### ▲ 导入表

首先查看导入表：






导入 `RegCreateKeyExA`、`RegSetValueExA` 函数，表明会创建注册表键、设置注册表键值。

导入 `FreeResource`、`FindResourceA`、`LockResource`、`LoadResource` 等函数，表明加载资源、查找资源等行为。

Address	Ordinal	Name	Library
 000000000004...		RegSetValueExA	ADVAPI32
 000000000004...		RegCreateKeyExA	ADVAPI32
 000000000004...		SizeofResource	KERNEL32
 000000000004...		LockResource	KERNEL32
 000000000004...		LoadResource	KERNEL32
 000000000004...		VirtualAlloc	KERNEL32
 000000000004...		GetModuleFileNameA	KERNEL32
 000000000004...		GetModuleHandleA	KERNEL32
 000000000004...		FreeResource	KERNEL32
 000000000004...		FindResourceA	KERNEL32

#### ▲ 字符串

接下来查看字符串信息。

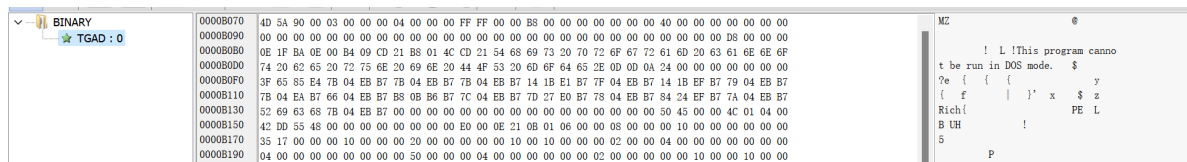
 .data:00...	00000007	C	BINARY
 .data:00...	00000008	C	GinaDLL
 .data:00...	00000036	C	SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion\\Winlogon
 .data:00...	0000000D	C	msgina32.dll
 .data:00...	0000000E	C	\\msgina32.dll

1. `SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion\\Winlogon`：这是一个注册表路径，指向Windows NT操作系统中Winlogon子系统的配置信息。Winlogon是负责处理用户登录、注销以及安全性验证的组件。

2. `msgina32.dll`：这是Microsoft Windows中的一个重要的DLL文件，通常用于实现GINA模块。GINA模块负责处理用户身份验证，以使用户能够登录到系统。

猜测它可能是一个拦截GINA的恶意代码。通过替换正常的GINA模块，恶意软件可以截获用户的登录凭据、绕过身份验证、或者在用户登录时执行恶意操作。

## ▲ 资源节

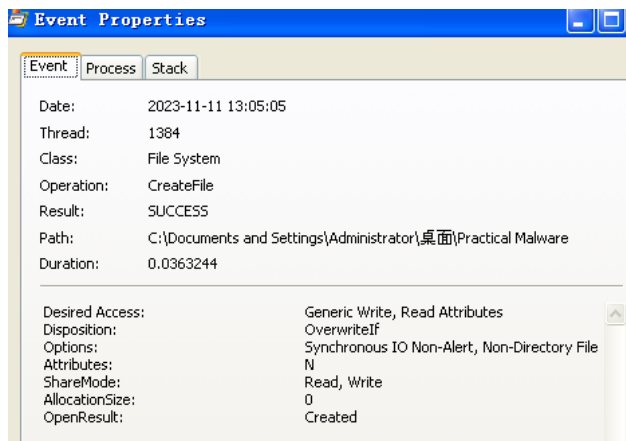


在Resource Hacker中打开，可以发现一个 `TGAD` 的资源节，包含一个内嵌的PE文件。将该PE文件导出。

## 基础动态分析

动态运行恶意代码，在procmon中设置进程名称过滤器。

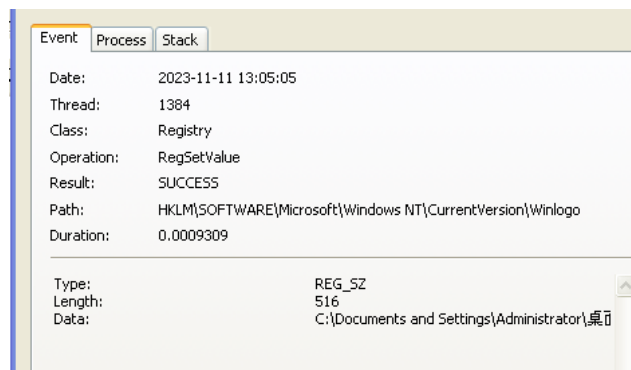
发现恶意代码在进程所在目录创建了一个名为 `msgina32.dll` 的文件。



将 `msgina32.dll` 文件与前面在资源节中导出的PE文件进行比较，发现是相同的。

之后，恶意代码将 `msgina32.dll` 的路径写入注册表 `HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\GinaDLL` 中。WindowsXP通过注册表 `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows`

`NT\CurrentVersion\Winlogon\GinaDLL` 来设置需要WinLogon加载的第三方DLL。因此，当系统重启时，WinLogon会加载这个DLL。



## 问题一：释放dll文件

- 这个恶意代码向磁盘释放了什么？

根据前面procmon过滤的行为分析，恶意代码将从名为TGAD资源节中**提取出的PE文件释放**到所在目录创建 `msgina32.dll` 中。

在IDA Pro中分析：

可以看到，首先打开、加载资源节；然后调用 `_fopen` 和 `_fwrite` 函数，将TGAD资源节中提取出的PE文件写入 `msgina32.dll` 中。



```

loc_40114A:
mov     ecx, [ebp+dwSize]
mov     esi, [ebp+Buffer]
mov     edi, [ebp+var_C]
mov     eax, ecx
shr     ecx, 2
rep movsd
mov     ecx, eax
and     ecx, 3
rep movsb
push    offset Mode           ; "wh"
push    offset FileName       ; "msgina32.dll"
call    fopen
add     esp, 8
mov     [ebp+Stream], eax
mov     ecx, [ebp+Stream]
push    ecx                  ; Stream
mov     edx, [ebp+dwSize]
push    edx                  ; ElementCount
push    1                    ; ElementSize
mov     eax, [ebp+Buffer]
push    eax                  ; Buffer
call    fwrite
add     esp, 10h
mov     ecx, [ebp+Stream]
push    ecx                  ; Stream
call    fclose

```

## 问题二：恶意代码的驻留行为

- 这个恶意代码如何进行驻留？

恶意代码将 `msgina32.dll` 的路径写入注册表 `HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\GinaDLL` 中。WindowsXP通过注册表 `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\GinaDLL` 来设置需要WinLogon加载的第三方DLL。因此，当系统重启时，WinLogon会加载这个DLL。因此，实现了在系统中的驻留。

```

push    0                    ; Reserved
push    offset SubKey        ; "SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\GinaDLL"
push    80000000h            ; hkey
call    ds:RegCreateKeyExA
test    eax, eax
jz      short loc_401032

loc_401032:
mov     ecx, [ebp+cbData]
push    ecx                  ; cbData
mov     edx, [ebp+lpData]
push    edx                  ; lpData
push    1                    ; dwType
push    0                    ; Reserved
push    offset ValueName     ; "GinaDLL"
mov     eax, [ebp+phkResult]
push    eax                  ; hKey
call    ds:RegSetValueExA
test    eax, eax
jz      short loc_401062

```

## 分析 msgina32.dll

exe文件仅仅是进行了资源释放和注册表设置，其他功能应该是在资源文件里实现的，因此，分析 `msgina32.dll` 文件。

### 基本静态分析

查看字符串信息。

.data:10...	00000010	C (16 bits) - UTF-16LE	GinaDLL
.data:10...	0000006C	C (16 bits) - UTF-16LE	Software\\Microsoft\\Windows NT\\CurrentVersion\\Winlogon
.data:10...	00000016	C (16 bits) - UTF-16LE	MSGina.dll
.data:10...	00000032	C (16 bits) - UTF-16LE	UN %s DM %s PW %s OLD %s
.data:10...	00000010	C	WlxLoggedOutSAS
.data:10...	00000040	C (16 bits) - UTF-16LE	ErrorCode:%d ErrorMessage:%s. \n
.data:10...	00000018	C (16 bits) - UTF-16LE	%s %s - %s
.data:10...	0000001A	C (16 bits) - UTF-16LE	msutil32.sys

值得关注的是 `UN %s DM %s PW %s OLD %s`，这是一个包含用户名、域名、密码和旧密码的字符串。可能涉及用户凭据的操作，例如获取、修改或传递给其他恶意功能。

而查看恶意代码的导入表，发现很多 `Wlx` 前缀的函数：

WlxActivateUserShell	00000000100012B0	30
WlxDisconnectNotify	00000000100012C0	31
WlxDisplayLockedNotice	00000000100012D0	32
WlxDisplaySASNotice	00000000100012E0	33
WlxDisplayStatusMessage	00000000100012F0	34
WlxGetConsoleSwitchCredentials	0000000010001300	35
WlxGetStatusMessage	0000000010001310	36
WlxInitialize	0000000010001320	37
WlxIsLockOk	0000000010001330	38
WlxIsLogoffOk	0000000010001340	39
WlxLoggedOnSAS	0000000010001350	40
WlxLoggedOutSAS	00000000100014A0	41
WlxLogoff	0000000010001360	42
WlxNegotiate	0000000010001370	43
WlxNetworkProviderLoad	0000000010001380	44
WlxReconnectNotify	0000000010001390	45
WlxRemoveStatusMessage	00000000100013A0	46
WlxScreenSaverNotify	00000000100013B0	47
WlxShutdown	00000000100013C0	48
WlxStartApplication	00000000100013D0	49
WlxWkstaLockedSAS	00000000100013E0	50
DllRegister	0000000010001440	51
DllUnregister	0000000010001490	52
DllEntryPoint	0000000010001735	[main en...

以 `Wlx` 开头的函数通常是与 Windows 登录扩展相关的函数。这些函数的命名表明它们与用户登录、注销、屏幕锁定等过程相关。这样的函数通常在自定义登录过程中使用，以提供额外的功能或修改标准登录过程的行为。



## ▲ IDA Pro 静态分析

在IDA Pro中载入dll文件。

首先查看 `DllMain` 函数。

```
.text:10001050      mov     eax, [esp+fdwReason]
.text:10001054      sub     esp, 208h
.text:1000105A      cmp     eax, 1
.text:1000105D      jnz     short loc_100010B7
.text:1000105F      push    esi
.text:10001060      mov     esi, [esp+20Ch+hinstDLL]
.text:10001067      push    esi                ; hLibModule
.text:10001068      call    ds:DisableThreadLibraryCalls
.text:1000106E      lea     eax, [esp+20Ch+Buffer]
.text:10001072      push    104h                ; uSize
.text:10001077      push    eax                ; lpBuffer
.text:10001078      mov     hModule, esi
.text:1000107E      call    ds:GetSystemDirectoryW
.text:10001084      lea     ecx, [esp+20Ch+Buffer]
.text:10001088      push    offset String2      ; "\\MSGina"
.text:1000108D      push    ecx                ; lpString1
.text:1000108E      call    ds:lstrcatW
.text:10001094      lea     edx, [esp+20Ch+Buffer]
.text:10001098      push    edx                ; lpLibFileName
.text:10001099      call    ds:LoadLibraryW
.text:1000109F      xor     ecx, ecx
.text:100010A1      mov     hLibModule, eax
.text:100010A6      test    eax, eax
.text:100010A8      setnz   cl
.text:100010AB      mov     eax, ecx
.text:100010AD      pop     esi
.text:100010AE      add     esp, 208h
.text:100010B4      retn    0Ch
```

`DllMain` 首先检查参数 `fdwReason0`，这个传入的参数表示着DLL入口函数被调用的原因。恶意代码检查传入参数是否为 `DLL_PROCESS_ATTACH`，`DLL_PROCESS_ATTACH` 表示DLL被加载到进程的地址空间中。

如果参数符合条件，接下来会调用 `LoadLibraryW` 函数，获取Windows系统目录中 `msgina.dll` 的句柄，并将这个句柄保存到全局变量中。使用这个变量可以让这个DLL的导入函数恰当地调用Windows DLL `msgina.dll` 中的函数。既然 `msgina32.dll` 拦截Winlogon与 `msgina.dll` 之间的通信，则它必须恰当地调用 `msgina.dll` 中的函数，从而使系统操作继续正常运行。

前面分析提到，恶意dll文件包含很多 `wlx` 前缀的导出函数。接下来就进行详细分析。

首先可以发现这些导出函数几乎都调用了 `sub_10001000` 函数。

```

text:100012D0 WlxDisplayLockedNotice proc near          ; DATA XREF: .rdata:off_10002348↓o
text:100012D0
text:100012D0 pWlxContext      = dword ptr 4
text:100012D0
text:100012D0          push     offset aWlxdisplaylock_0 ; "WlxDisplayLockedNotice"
text:100012D5          call     sub_10001000
text:100012DA          jmp      eax
text:100012DA WlxDisplayLockedNotice endp
text:100012DA
text:100012DA ; -----
text:100012DC          align 10h
text:100012E0 ; Exported entry 33. WlxDisplaySASNotice
text:100012E0
text:100012E0 ; ===== S U B R O U T I N E =====
text:100012E0
text:100012E0 ; void __stdcall WlxDisplaySASNotice(PVOID pWlxContext)
text:100012E0          public WlxDisplaySASNotice
text:100012E0 WlxDisplaySASNotice proc near          ; DATA XREF: .rdata:off_10002348↓o
text:100012E0
text:100012E0 pWlxContext      = dword ptr 4
text:100012E0
text:100012E0          push     offset aWlxdisplaysasn_0 ; "WlxDisplaySASNotice"
text:100012E5          call     sub_10001000
text:100012EA          jmp      eax
text:100012EA WlxDisplaySASNotice endp
text:100012EA

```

查看该函数。

```

.text:10001000 ; int __stdcall sub_10001000(LPCSTR lpProcName)
.text:10001000 sub_10001000 proc near          ; CODE XREF: gina_1+2↓p
.text:10001000          ; gina_2+2↓p ...
.text:10001000
.text:10001000 var_10      = byte ptr -10h
.text:10001000 lpProcName    = dword ptr 4
.text:10001000
.text:10001000          mov     eax, hLibModule
.text:10001005          sub     esp, 10h
.text:10001008          push    esi
.text:10001009          mov     esi, [esp+14h+lpProcName]
.text:1000100D          push    esi          ; lpProcName
.text:1000100E          push    eax          ; hModule
.text:1000100F          call    ds:GetProcAddress
.text:10001015          test    eax, eax
.text:10001017          jnz     short loc_1000103C
.text:10001019          mov     ecx, esi
.text:1000101B          shr     ecx, 10h
.text:1000101E          jnz     short loc_10001034
.text:10001020          push    esi
.text:10001021          lea     edx, [esp+18h+var_10]
.text:10001025          push    offset aD          ; "%d"
.text:1000102A          push    edx          ; LPSTR
.text:1000102B          call    ds:wsprintfA
.text:10001031          add     esp, 0Ch
.text:10001034
.text:10001034 loc_10001034:          ; CODE XREF: sub_10001000+1E↑j
.text:10001034          push    0FFFFFFEh          ; uExitCode
.text:10001036          call    ds:ExitProcess

```

这里是从原本的 `dll` 中获取函数地址，然后返回函数地址。得到返回地址后，直接 `jmp` 跳转，这是一个函数转发。

## WlxLoggedOnSAS

分析函数 `WlxLoggedOnSAS`，调用 `sub_10001000`，只是简单地传递了真正的 `WlxLoggedOnSAS` 函数。

```
.text:10001350 ; int __stdcall WlxLoggedOnSAS(PVOID pWlxContext, DWORD dwSasType, PVOID pReserved)
.text:10001350 public WlxLoggedOnSAS
.text:10001350 WlxLoggedOnSAS proc near ; DATA XREF: .rdata:off_10002348lo
.text:10001350
.text:10001350 pWlxContext = dword ptr 4
.text:10001350 dwSasType = dword ptr 8
.text:10001350 pReserved = dword ptr 0Ch
.text:10001350
.text:10001350 push offset aWlxloggedonsas_0 ; "WlxLoggedOnSAS"
.text:10001355 call sub_10001000
.text:1000135A jmp eax
.text:1000135A WlxLoggedOnSAS endp
```

而得到真正函数地址后，它直接 `jmp` 跳转至该函数。这段代码将不会创建一个栈帧，或者将返回地址压入栈。当 `msgina.dll` 中 `WlxLoggedonSAS` 被调用时，它将直接返回到 Winlogon 上运行。

## WlxLoggedOutSAS

当系统注销时，会调用 `WlxLoggedOutSAS` 函数。`WlxLoggedOutSAS` 函数还是先调用 `sub_10001000` 获得到真正函数的地址并跳转至地址处执行。

```
.text:100014A0 push esi
.text:100014A1 push edi
.text:100014A2 push offset aWlxloggedoutsas_0 ; "WlxLoggedOutSAS"
.text:100014A7 call sub_10001000
```

接下来调用 `sub_10001570` 函数。在调用之前，压入了格式化字符串 `UN %s DM %s PW %s OLD %s`。

```
.text:100014FB push eax ; Args
.text:100014FC push offset aUnSDmSPwSOldS ; "UN %s DM %s PW %s OLD %s"
.text:10001501 push 0 ; dwMessageId
.text:10001503 call sub_10001570
```

`sub_10001570` 位置的代码看起来像是一个记录窃取登录凭证的函数，首先传入被当做参数传入的格式化字符串，然后打开 `msutil32.sys` 文件，因为 Winlogon 在 `C:\Windows\System32` 的目录下，所以这个文件也被创建在该目录下，接下来记录日期时间最后记录登录的凭证，所以说，这个 sys 文件并不是一个驱动，只是记录记录凭证信息的文件。

```

.text:1000158E      call     _vsnwprintf
.text:10001593      push    offset Mode      ; Mode
.text:10001598      push    offset FileName ; "msutil32.sys"
.text:1000159D      call    _wfopen
.text:100015A2      mov     esi, eax
.text:100015A4      add     esp, 18h
.text:100015A7      test    esi, esi
.text:100015A9      jz      loc_1000164F
.text:100015AF      lea     eax, [esp+858h+Buffer]
.text:100015B3      push    edi
.text:100015B4      lea     ecx, [esp+85Ch+var_850]
.text:100015B8      push    eax
.text:100015B9      push    ecx                ; Buffer
.text:100015BA      call    _wstrtime
.text:100015BF      add     esp, 4
.text:100015C2      lea     edx, [esp+860h+var_828]
.text:100015C6      push    eax
.text:100015C7      push    edx                ; Buffer
.text:100015C8      call    _wstrdate
.text:100015CD      add     esp, 4
.text:100015D0      push    eax
.text:100015D1      push    offset Format      ; "%s %s - %s "
.text:100015D6      push    esi                ; Stream
.text:100015D7      call    fprintf
.text:100015DC      mov     edi, [esp+870h+dwMessageId]
.text:100015E3      add     esp, 14h
.text:100015E6      test    edi, edi

```

### 问题三：窃取用户凭证

- 这个恶意代码如何窃取用户登录凭证？

恶意代码用GINA拦截窃取用户登录凭证。 `msgina32.dll` 作为GINA拦截器。恶意文件 `msgina32.dll` 在 Winlogon 和 `msgina.dll` 之间工作（类似于中间人攻击），此时恶意代码能够拦截提交给系统的所有登陆凭证。

所以说， `msgina32.dll` 拦截了两者之间的通信，为了能够让系统正常运行，它必须将登陆凭证传递给 `msgina.dll`。从而拦截器 `msgina32.dll` 必须包括 `msgina.dll` 的所有导出函数，也就是前缀是 `Wlx` 的那些函数。

### 问题四：保存窃取记录

- 这个恶意代码对窃取的证书做了什么处理？

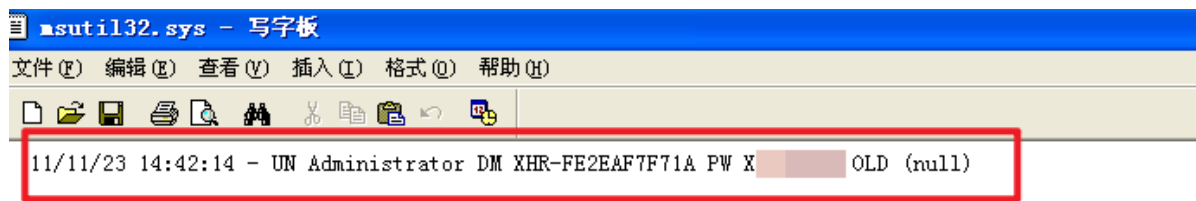
通过前面对 `WlxLoggedOutSAS` 函数的分析得知，恶意代码将被盗窃的登录凭证保存到 `C:\Windows\System32` 的目录下的 `msutil32.sys` 文件中。用户名、域名称、密码、时间戳都将被记录到该文件。

### 问题五：窃取登录凭证测试

- 如何在你的测试环境让这个恶意代码获得用户登录凭证？

释放并且安装恶意代码后，必须重启系统才能启动GINA拦截。仅当用户注销时，恶意代码才记录登录凭证，所以注销然后再登录系统，就能看到记录到日志文件的登录凭证。

关机重启后，可以在 `msutil32.sys` 文件中发现登陆记录，记录了账号和密码：



## Lab 11-02

### 基本静态分析

首先分析Lab11-02.dll的导入表。可以发现导入的函数涉及到对Windows系统的注册表操作、线程操作、文件操作和内存操作等。

- 注册表操作函数：`RegSetValueExA`、`RegOpenKeyExA` 和 `RegCloseKey`；
- 内存操作函数：
  - `VirtualProtect` 用于更改指定内存区域的保护属性，可能用于修改代码段或数据段的权限。
  - `malloc`，`free`，`memset`，`memcpy` 用于动态内存分配、释放和操作。
- 线程操作函数：
  - `Thread32Next`，`Thread32First`：用于遍历系统中的线程信息。
  - `GetCurrentThreadId` 和 `GetCurrentProcessId`：获取当前线程和进程的标识符。
    - `SuspendThread` 和 `ResumeThread`：用于暂停和恢复线程的执行。
- 文件操作函数：
  - `CopyFileA`，`ReadFile`，`CreateFileA`：用于复制文件、读取文件和创建文件。
- 模块操作函数：
  - `GetModuleHandleA`：获取指定模块的句柄。

Address	Ordinal	Name	Library
00000000100...		RegSetValueExA	ADVAPI32
00000000100...		RegOpenKeyExA	ADVAPI32
00000000100...		RegCloseKey	ADVAPI32
00000000100...		VirtualProtect	KERNEL32
00000000100...		GetModuleHandleA	KERNEL32
00000000100...		Thread32Next	KERNEL32
00000000100...		CloseHandle	KERNEL32
00000000100...		SuspendThread	KERNEL32
00000000100...		Thread32First	KERNEL32
00000000100...		CreateToolhelp32Snapshot	KERNEL32
00000000100...		GetModuleFileNameA	KERNEL32
00000000100...		GetCurrentProcessId	KERNEL32
00000000100...		ResumeThread	KERNEL32
00000000100...		CopyFileA	KERNEL32
00000000100...		ReadFile	KERNEL32
00000000100...		CreateFileA	KERNEL32
00000000100...		GetSystemDirectoryA	KERNEL32
00000000100...		GetProcAddress	KERNEL32
00000000100...		LoadLibraryA	KERNEL32
00000000100...		GetCurrentThreadId	KERNEL32
00000000100...		toupper	MSVCRT
00000000100...		strlen	MSVCRT
00000000100...		strchr	MSVCRT
00000000100...		strcat	MSVCRT
00000000100...		memcpy	MSVCRT
00000000100...		strstr	MSVCRT
00000000100...		malloc	MSVCRT
00000000100...		memcmp	MSVCRT
00000000100...		strncat	MSVCRT
00000000100...		memset	MSVCRT
00000000100...		free	MSVCRT
00000000100...		_initterm	MSVCRT
00000000100...		_adjust_fdiv	MSVCRT

分析字符串信息，可以发现几个有趣的字符串：

- **RCPT TO**：这是与电子邮件传输协议 (SMTP) 相关的命令，用于指定邮件的接收者。
- **AppInit\_DLLs** 和 **SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows**：是 **AppInit\_DLLs** 键值存储了在系统启动时自动加载的动态链接库 (DLL) 文件的路径。这说明恶意代码可能使用 **AppInit\_DLLs** 来永久安装自身。
- 这个键值的特殊之处在于任何使用到 User32.dll 的 EXE、DLL、OCX 等类型的 PE 文件都会读取这个地方 并且根据约定的规范将这个键值下指向的 DLL 文件进行加载，加载的方式是调用 **LoadLibrary**
- **\Lab11-02.ini**：INI 文件通常用于存储配置信息。
- **Process Names (THEBAT.EXE, OUTLOOK.EXE, MSIMN.EXE)**：这些字符串包含一些进程的名称，可能是针对特定应用程序的操作。

## 问题一：导出函数

- 这个恶意 DLL 导出了什么？

导出了 **installer** 函数。

Name	Address	Ordinal
installer	000000001000158B	1
DllEntryPoint	00000000100017E9	[main entry]

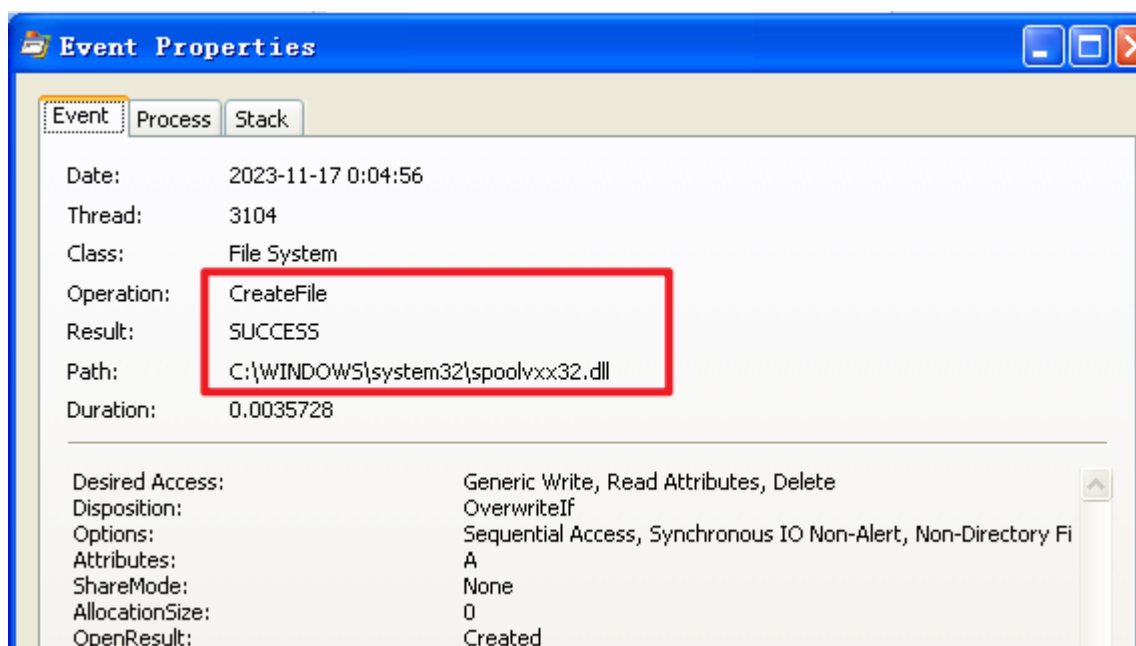
## 问题二：动态运行恶意代码

- 使用rundll32.exe安装这个恶意代码后，发生了什么？

拍摄快照，打开procmon。输入命令 `rundll32.exe Lab11-02.dll,installer`，使用 `installer` 安装恶意代码。

```
C:\Documents and Settings\Administrator\桌面\Practical Malware Analysis Labs\BinaryCollection\Chapter_11L>rundll32.exe Lab11-02.dll,installer
```

在procmon中过滤 `rundll32.exe` 的行为，过滤到一条 `CreateFile` 的记录。可以看到，它在系统目录下创建文件 `spoolvxx32.dll`。将该文件与 `Lab11-02.dll` 比较，发现二者是一样的。



在安装后再次拍摄快照并进行比较。可以发现，恶意代码将 `spoolvcx32.dll` 添加到注册表键 `AppInit_DLLs` 中，这将导致恶意代码被加载到所有装载 `User32.dll` 的进程中。

```
Values modified: 7
HKLM\SOFTWARE\Microsoft\Cryptography\ RNG\Seed: 7F FE B7 3D 6B 4B E2 EB 80 F2 11 9F 1B AB 22 55
HKLM\SOFTWARE\Microsoft\Cryptography\ RNG\Seed: 9B D2 66 D3 ED 79 1B 03 1C 09 12 55 05 EF 50 5F
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows\AppInit_DLLs: ""
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows\AppInit_DLLs: "spoolvxx32.dll"
HKU\S-1-5-21-1993962763-2025429265-1606980848-500\Software\Microsoft\Windows\CurrentVersion\E
```



### 问题三：Lab11-02.ini 的位置

- 为了使这个恶意代码正确安装，Lab11-02.ini必须放置在何处？

Lab11-02.ini必须放置在系统目录 `C:\WINDOWS\system32\Lab11-02.ini` 下。

在IDA Pro中分析dll，查看 `DLLMain` 函数。

首先保存当前路径到一个全局变量。然后调用 `sub_1000105B` 函数。

```
.text:10001629 loc_10001629, CODE XREF: DllMain(x,x,x)+141J
.text:10001629      push     104h                ; nSize
.text:1000162E      push     offset ExistingFileName ; lpFileName
.text:10001633      mov     ecx, [ebp+hinstDLL]
.text:10001636      push     ecx                ; hModule
.text:10001637      call    ds:GetModuleFileNameA
.text:1000163D      push     101h                ; Size
.text:10001642      push     0                  ; Val
.text:10001644      push     offset byte_100034A0 ; void *
.text:10001649      call    memset
.text:1000164E      add     esp, 0Ch
.text:10001651      call    sub_1000105B
.text:10001656      mov     [ebp+Destination], eax
.text:10001659      push     104h                ; Count
.text:1000165E      push     offset aLab1102Ini ; "\\Lab11-02.ini"
.text:10001663      mov     edx, [ebp+Destination]
.text:10001666      push     edx                ; Destination
.text:10001667      call    strncat
```

`sub_1000105B` 获取系统目录，返回系统目录的路径。

```
.text:1000105B sub_1000105B proc near ; CODE XREF: installer:loc_100015DD↓p
.text:1000105B      ; DllMain(x,x,x)+41↓p
.text:1000105B      push    ebp
.text:1000105C      mov     ebp, esp
.text:1000105E      push    104h                ; uSize
.text:10001063      push    offset Buffer        ; lpBuffer
.text:10001068      call    ds:GetSystemDirectoryA
.text:1000106E      mov     eax, offset Buffer
.text:10001073      pop     ebp
.text:10001074      retn
.text:10001074 sub_1000105B endp
.text:10001074
```

之后调用了 `strncat` 拼接字符串，拼接后的字符串为 `C:\WINDOWS\system32\Lab11-02.ini`。

之后调用 `CreateFileA` 试图打开ini文件，如果不能打开则返回。

```
.text:10001667      call    strncat
.text:1000166C      add     esp, 0Ch
.text:1000166F      push    0                  ; hTemplateFile
.text:10001671      push    80h ; 'E'          ; dwFlagsAndAttributes
.text:10001676      push    3                  ; dwCreationDisposition
.text:10001678      push    0                  ; lpSecurityAttributes
.text:1000167A      push    1                  ; dwShareMode
.text:1000167C      push    80000000h          ; dwDesiredAccess
.text:10001681      mov     eax, [ebp+Destination]
.text:10001684      push    eax                ; lpFileName
.text:10001685      call    ds>CreateFileA
.text:1000168B      mov     [ebp+hFile], eax
.text:1000168E      cmp     [ebp+hFile], 0FFFFFFFh
.text:10001692      iz      short loc_100016DE
```

因此，Lab11-02.ini必须放置在系统目录 `C:\WINDOWS\system32\Lab11-02.ini` 下。



## 问题四：恶意代码驻留

这个安装的恶意代码如何驻留？

通过动态分析和静态分析恶意代码的导出函数，恶意代码的 `installer` 函数会把自己复制到 `C:\WINDOWS\system32\spoolvxx32.dll`，以及给注册表 `AppInit_DLLs` 添加项，使恶意代码被加载到所有装载 `User32.dll` 的进程中。

```
.text:1000158B      public installer
.text:1000158B      installer      proc near                                ; DATA XREF: .rdata:off_100023A8↓o
.text:1000158B
.text:1000158B      phkResult      = dword ptr -8
.text:1000158B      Destination    = dword ptr -4
.text:1000158B
.text:1000158B      push         ebp
.text:1000158C      mov          ebp, esp
.text:1000158E      sub          esp, 8
.text:10001591      lea          eax, [ebp+phkResult]
.text:10001594      push         eax                                ; phkResult
.text:10001595      push         6                                ; samDesired
.text:10001597      push         0                                ; ulOptions
.text:10001599      push         offset SubKey                    ; "SOFTWARE\Microsoft\Windows NT\Curren...
.text:1000159E      push         80000003h                        ; hKey
.text:100015A3      call         ds:RegOpenKeyExA
.text:100015A9      test         eax, eax
.text:100015AB      jnz          short loc_100015DD
.text:100015AD      push         offset aSpoolvxx32Dll1 ; "spoolvxx32.dll"
.text:100015B2      call         strlen
.text:100015B7      add          esp, 4
.text:100015BA      push         eax                                ; cbData
.text:100015BB      push         offset Data                    ; "spoolvxx32.dll"
.text:100015C0      push         1                                ; dwType
.text:100015C2      push         0                                ; Reserved
.text:100015C4      push         offset ValueName                ; "AppInit_DLLs"
.text:100015C9      mov          ecx, [ebp+phkResult]
.text:100015CC      push         ecx                                ; hKey
.text:100015CD      call         ds:RegSetValueExA
.text:100015D3      mov          edx, [ebp+phkResult]
.text:100015D6      push         edx                                ; hKey
.text:100015D7      call         ds:RegCloseKey

-----
.text:100015DD      loc_100015DD:                                ; CODE XREF: installer+20↑j
.text:100015DD      call         sub_1000105B
.text:100015E2      mov          [ebp+Destination], eax
.text:100015E5      push         104h                                ; Count
.text:100015EA      push         offset aSpoolvxx32Dll1_1 ; "\\spoolvxx32.dll"
.text:100015EF      mov          eax, [ebp+Destination]
.text:100015F2      push         eax                                ; Destination
.text:100015F3      call         strncat
.text:100015F8      add          esp, 0Ch
.text:100015FB      push         0                                ; bFailIfExists
.text:100015FD      mov          ecx, [ebp+Destination]
.text:10001600      push         ecx                                ; lpNewFileName
.text:10001601      push         offset ExistingFileName ; lpExistingFileName
.text:10001606      call         ds:CopyFileA
.text:1000160C      mov          esp, ebp
.text:1000160E      pop          ebp
.text:1000160F      retn
```

## 问题八：INI 文件的意义

INI 文件的意义是什么？

INI 文件中包含一个加密的邮件地址。

恶意代码成功打开INI文件后，会将文件读到全局变量缓冲区中。接下来调用 `sub_100010B3` 函数。

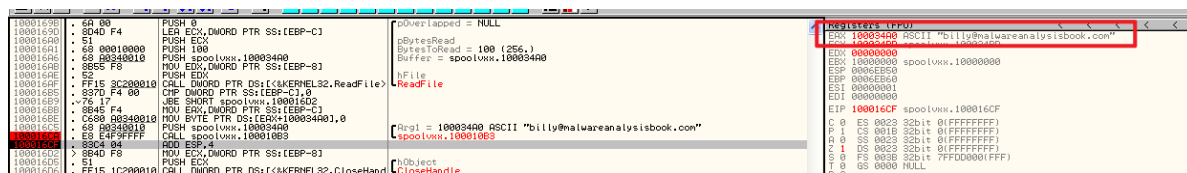
```

.text:1000169D      push     ecx, [ebp+NumberOfBytesRead]
.text:100016A0      push     ecx ; lpNumberOfBytesRead
.text:100016A1      push     100h ; nNumberOfBytesToRead
.text:100016A6      push     offset byte_100034A0 ; lpBuffer
.text:100016AB      mov      edx, [ebp+hFile]
.text:100016AE      push     edx ; hFile
.text:100016AF      call     ds:ReadFile
.text:100016B5      cmp      [ebp+NumberOfBytesRead], 0
.text:100016B9      jbe      short loc_100016D2
.text:100016BB      mov      eax, [ebp+NumberOfBytesRead]
.text:100016BE      mov      byte_100034A0[eax], 0
.text:100016C5      push     offset byte_100034A0
.text:100016CA      call     sub_100010B3
.text:100016CF      add      esp, 4

```

在OllyDbg中在调用后的地址处打上一个断点。然后执行至断点处。

发现 `eax` 的值为 `100034A0`，该地址存放一个字符串 `billy@malwareanalysisbook.com`。这是一个邮箱地址。因此，这个函数应该是一个解密函数，将INI的文件内容解密，得到邮箱地址。



## 问题七：恶意代码的攻击目标

- 哪个或者哪些进程执行这个恶意攻击，为什么？

恶意代码劫持邮件，针对邮件程序进行攻击，攻击目标仅针对 `MSIMN.exe`、`THEBAT.exe` 和 `OUTLOOK.exe`。除非恶意代码运行在这些进程空间中，否则它不会安装挂钩。下面是分析过程：

前面分析到，dll文件读取ini信息，然后进行解密操作，解密完成之后保存在全局变量里。继续分析后面的行为，调用了 `sub_100014B6` 函数。

该函数内部第一个调用是 `sub_10001075` 函数。再查看后者，发现调用 `GetModuleFileNameA`，且 `hModule` 参数的值被设置为了 `0`，因此这个函数会返回加载了这个 `DLL` 的进程的绝对路径。

```

.text:100014B6      push    ebp
.text:100014B7      mov     ebp, esp
.text:100014B9      push    ecx
.text:100014BA      cmp     [ebp+arg_0], 0
.text:100014BE      jz      loc_10001587
.text:100014C4      lea     eax, [ebp+Buf1]
.text:100014C7      push    eax                    ; int
.text:100014C8      push    0                    ; hModule
.text:100014CA      call    sub_10001075
.text:100014CF      add     esp, 8
.text:100014D2      mov     ecx, [ebp+Buf1]
.text:100014D5      push    ecx                    ; Str
.text:100014D6      call    sub_10001104
.text:100014DB      add     esp, 4
.text:100014DE      mov     [ebp+Buf1], eax
.text:100014E1      cmp     [ebp+Buf1], 0
.text:100014E5      jnz     short loc_100014EC
.text:100014E7      jmp     loc_10001587

.text:10001075      push    ebp
.text:10001076      mov     ebp, esp
.text:10001078      push    104h                 ; nSize
.text:1000107D      push    offset Filename ; lpFilename
.text:10001082      mov     eax, [ebp+hModule]
.text:10001085      push    eax                    ; hModule
.text:10001086      call    ds:GetModuleFileNameA
.text:1000108C      mov     ecx, [ebp+arg_4]
.text:1000108F      mov     dword ptr [ecx], offset Filename
.text:10001095      pop     ebp
.text:10001096      retn
.text:10001096      sub_10001075      endp

```

得到这个返回值之后，之后会剔除多余的路径名，并且进行大写变换，得到当前的进程名。之后将进程名与 `THEBAT.EXE`、`OUTLOOK.EXE`、`MSIMN.EXE` 比较，如果不是三者之一，则退出。

```

push    eax                ; Size
push    offset aThebatExe_0 ; "THEBAT.EXE"
mov     eax, [ebp+Buf1]
push    eax                ; Buf1
call    memcmp
add     esp, 0Ch
test    eax, eax
jz      short loc_10001561

```

```

push    offset aOutlookExe ; "OUTLOOK.EXE"
call    strlen
add     esp, 4
push    eax                ; Size
push    offset aOutlookExe_0 ; "OUTLOOK.EXE"
mov     ecx, [ebp+Buf1]
push    ecx                ; Buf1
call    memcmp
add     esp, 0Ch
test    eax, eax
jz      short loc_10001561

```

```

push    offset aMsimnExe ; "MSIMN.EXE"
call    strlen
add     esp, 4
push    eax                ; Size
push    offset aMsimnExe_0 ; "MSIMN.EXE"
mov     edx, [ebp+Buf1]
push    edx                ; Buf1

```

如果进程名称是三者之一，那么继续执行，设置hook。

```

.text:10001561 loc_10001561:                                ; CODE XREF: sub_100014B6+63↑j
.text:10001561                                           ; sub_100014B6+86↑j
.text:10001561      call    sub_100013BD
.text:10001566      push    offset dword_10003484 ; int
.text:10001568      push    offset sub_1000113D ; int
.text:10001570      push    offset aSend      ; "send"
.text:10001575      push    offset ModuleName ; "wsock32.dll"
.text:1000157A      call    sub_100012A3
.text:1000157F      add     esp, 10h
.text:10001582      call    sub_10001499

```

## 问题五：HOOK 分析

- 这个恶意代码采用的用户态Rootkit技术是什么？

这个恶意代码针对 `send` 函数安装了一个 `inline` 挂钩，使其跳转到挂钩函数。在程序调用 `send` 函数时，其第一个指令会使得转换到 `sub1000113D` 处运行。而运行结束后，会调用trampoline的代码，运行 `send` 函数的前三条原始指令，并跳回 `send` 函数的第五字节，使 `send` 函数可以正常运行。

下面是分析过程。

继续向后分析，分析进程符合条件后的代码。

```

.text:10001561 loc_10001561:                                ; CODE XREF: sub_100014B6+63↑j
.text:10001561                                           ; sub_100014B6+86↑j
.text:10001561      call     sub_100013BD
.text:10001566      push     offset dword_10003484 ; int
.text:1000156B      push     offset sub_1000113D ; int
.text:10001570      push     offset aSend ; "send"
.text:10001575      push     offset ModuleName ; "wssock32.dll"
.text:1000157A      call     sub_100012A3
.text:1000157F      add      esp, 10h
.text:10001582      call     sub_10001499

```

## ▲ 分析 sub\_100013BD

分析它的第一个函数调用 `sub_100013BD` 。

```

.text:100013BD      push     ebp
.text:100013BE      mov      ebp, esp
.text:100013C0      push     ecx
.text:100013C1      call     ds:GetCurrentProcessId
.text:100013C7      mov      [ebp+var_4], eax
.text:100013CA      mov      eax, [ebp+var_4]
.text:100013CD      push     eax
.text:100013CE      call     sub_100012FE
.text:100013D3      add      esp, 4
.text:100013D6      mov      esp, ebp
.text:100013D8      pop      ebp
.text:100013D9      retn
.text:100013D9      sub_100013BD      endp

```

首先调用了 `GetCurrentProcessId` 函数，然后是 `sub_100012FE` 函数。这个函数会返回当前运行进程内的所有线程标识符，`sub_100012FE` 就会调用 `CreateToolHelp32Snapshot` 来遍历所有的线程。如果这个线程不是当前的线程，就会挂起这个线程，所以这个函数会挂起所有不是当前线程的所有线程。

```

.text:100012FE      push     ebp
.text:100012FF      mov      ebp, esp
.text:10001301      sub      esp, 2Ch
.text:10001304      push     offset ProcName ; "OpenThread"
.text:10001309      push     offset LibFileName ; "kernel32.dll"
.text:1000130E      call     sub_10001000
.text:10001313      add      esp, 8
.text:10001316      mov      [ebp+var_4], eax
.text:10001319      cmp      [ebp+var_4], 0
.text:1000131D      jnz      short loc_10001326
.text:1000131F      xor      eax, eax
.text:10001321      jmp      loc_100013B9
; -----
.text:10001326      loc_10001326:
.text:10001326      call     ds:GetCurrentThreadId ; CODE XREF: sub_100012FE+1F↑j
.text:1000132B      mov      [ebp+var_28], eax
.text:1000132C      push     0 ; th32ProcessID
.text:1000132F      push     4 ; dwFlags
.text:10001333      call     CreateToolhelp32Snapshot
.text:10001338      mov      [ebp+hSnapshot], eax
.text:1000133B      cmp      [ebp+hSnapshot], 0FFFFFFFFh
.text:1000133F      jnz      short loc_10001345
.text:10001341      xor      eax, eax
.text:10001343      jmp      short loc_100013B9
; -----

```

而在后面调用的 `sub_10001499` 函数中，它使用 `ResumeThread` 恢复所有的线程。

接下来，分析之间调用的 `sub_100012A3` 函数。在前面的调用可以看到，传入了参数 `send` 和 `wsock32.dll`。函数内部会使用 `GetModuleHandleA` 函数获得 `wsock32.dll` 的句柄，然后在 `wsock32.dll` 中查找 `send` 函数的地址，并且把这个地址赋值到 `lpAddress` 中保存。之后就会调用 `sub_10001203`，传递 `lpAddress` 和两个 `int` 型参数。

```
.text:100012A3 sub_100012A3 proc near ; CODE XREF: sub_100014B6+C4!p
.text:100012A3
.text:100012A3 lpAddress = dword ptr -8
.text:100012A3 hModule = dword ptr -4
.text:100012A3 lpModuleName = dword ptr 8
.text:100012A3 lpProcName = dword ptr 0Ch
.text:100012A3 arg_8 = dword ptr 10h
.text:100012A3 arg_C = dword ptr 14h
.text:100012A3
.text:100012A3 push ebp
.text:100012A4 mov ebp, esp
.text:100012A6 sub esp, 8
.text:100012A9 mov eax, [ebp+lpModuleName]
.text:100012AC push eax ; lpModuleName
.text:100012AD call ds:GetModuleHandleA
.text:100012B3 mov [ebp+hModule], eax
.text:100012B6 cmp [ebp+hModule], 0
.text:100012BA jnz short loc_100012C9
.text:100012BC mov ecx, [ebp+lpModuleName]
.text:100012BF push ecx ; lpLibFileName
.text:100012C0 call ds:LoadLibraryA
.text:100012C6 mov [ebp+hModule], eax
.text:100012C9 loc_100012C9: ; CODE XREF: sub_100012A3+17!j
.text:100012C9 loc_100012C9: ; CODE XREF: sub_100012A3+17!j
.text:100012C9 cmp [ebp+hModule], 0
.text:100012CD jz short loc_100012FA
.text:100012CF mov edx, [ebp+lpProcName]
.text:100012D2 push edx ; lpProcName
.text:100012D3 mov eax, [ebp+hModule]
.text:100012D6 push eax ; hModule
.text:100012D7 call ds:GetProcAddress
.text:100012DD mov [ebp+lpAddress], eax
.text:100012E0 cmp [ebp+lpAddress], 0
.text:100012E4 jz short loc_100012FA
.text:100012E6 mov ecx, [ebp+arg_C]
.text:100012E9 push ecx ; int
.text:100012EA mov edx, [ebp+arg_8]
.text:100012ED push edx ; int
.text:100012EE mov eax, [ebp+lpAddress]
.text:100012F1 push eax ; lpAddress
.text:100012F2 call sub_10001203
.text:100012F7 add esp, 0Ch
```

这两个参数其实就是调用 `sub_100013BD` 时传入的参数，`arg_C` 是 `sub_1000113D` 的地址，`arg_8` 是 `dword_10003484` 的地址。

```
.text:10001561 call sub_100013BD
.text:10001566 push offset dword_10003484 ; int
.text:1000156B push offset sub_1000113D ; int
```

## ▲ 安装inline挂钩，为inline挂钩创建trampoline

首先计算 `send` 函数地址和 `sub_1000113D` 的地址之间的地址差值，减去 5 之后，存储在变量 `var_4` 中。减去这额外的5字节是为了：随后的代码使用 `var_4` 变量时，在它的前面加上 `0xE9` (`jmp` 的操作码)，使这个5字节的指令能够跳转到 `sub_1000113D`。



```

• .text:10001203      push    ebp
• .text:10001204      mov     ebp, esp
• .text:10001206      sub     esp, 0Ch
• .text:10001209      mov     eax, [ebp+arg_4]
• .text:1000120C      sub     eax, [ebp+lpAddress]
• .text:1000120F      sub     eax, 5
• .text:10001212      mov     [ebp+var_4], eax

```

之后代码会调用函数 `VirtualProtect`，传入的 `lpAddress` 参数的值是 `send` 函数的地址，这个函数的作用是改变了这个地址的保护属性。这个调用修改了内存的运行、读以及写等保护权限，因此可以使恶意代码修改 `send` 函数的执行。

```

• .text:10001215      lea     ecx, [ebp+flOldProtect]
• .text:10001218      push    ecx                ; lpflOldProtect
• .text:10001219      push    40h                ; flNewProtect
• .text:1000121B      push    5                  ; dwSize
• .text:1000121D      mov     edx, [ebp+lpAddress]
• .text:10001220      push    edx                ; lpAddress
• .text:10001221      call    ds:VirtualProtect

```

之后，这个函数会调用 `malloc` 函数，分配了一个 `0FFh` 大小的空间，这个内存将作为 trampoline。

```

• .text:10001227      push    0FFh                ; Size
• .text:1000122C      call    malloc
• .text:10001231      add     esp, 4

```

接下来调用 `memcpy`，将上面分配好的空间复制了 `send` 函数的前五个字节，因为恶意代码覆盖了 `send` 函数的前五个字节，这为了方便跳转回来之后保证 `send` 函数代码的完整性。

```

• .text:1000124B      push    eax                ; Src
• .text:1000124C      mov     ecx, [ebp+var_8]
• .text:1000124F      add     ecx, 5
• .text:10001252      push    ecx                ; void *
• .text:10001253      call    memcpy

```

继续分析后面的行为：

恶意代码在 trampoline 代码中添加一个 `jmp` 指令。通过从 `send` 函数的地址中减去 trampoline 的地址，来计算跳转地址。这使得 trampoline 能跳转回 `send` 函数的第五个字节，使 `send` 函数能正常运行。

```

.text:10001246      push     5                ; Size
.text:10001248      mov     eax, [ebp+lpAddress]
.text:1000124B      push     eax              ; Src
.text:1000124C      mov     ecx, [ebp+var_8]
.text:1000124F      add     ecx, 5
.text:10001252      push     ecx              ; void *
.text:10001253      call    memcpy
.text:10001258      add     esp, 0Ch
.text:1000125B      mov     edx, [ebp+var_8]
.text:1000125E      mov     byte ptr [edx+0Ah], 0E9h
.text:10001262      mov     eax, [ebp+lpAddress]
.text:10001265      sub     eax, [ebp+var_8]
.text:10001268      sub     eax, 0Ah
.text:1000126B      mov     ecx, [ebp+var_8]
.text:1000126E      mov     [ecx+0Bh], eax

```

这段代码将机器码 `0xE9` 复制到 `send` 函数的开头，之后代码复制 `var_4` 到 `0xE9` 之后的内存。因此，这段代码，在 `send` 函数的开始部分放置了一个 `jmp` 指令，从而让代码跳转到DLL中 `sub_1000113D` 函数。

```

.text:10001271      mov     edx, [ebp+lpAddress]
.text:10001274      mov     byte ptr [edx], 0E9h
.text:10001277      mov     eax, [ebp+lpAddress]
.text:1000127A      mov     ecx, [ebp+var_4]
.text:1000127D      mov     [eax+1], ecx

```

最后，将全局变量  `dword_10003484`  变量设置为trampoline的地址。

## 问题六：挂钩函数

挂钩代码做了什么？

分析挂钩函数。可以看到，它首先查找字符串 `RCPT TO:`。

```

.text:1000113D      push     ebp
.text:1000113E      mov     ebp, esp
.text:10001140      sub     esp, 204h
.text:10001146      push     offset SubStr    ; "RCPT TO:"
.text:1000114B      mov     eax, [ebp+Str]
.text:1000114E      push     eax              ; Str
.text:1000114F      call    strstr
.text:10001154      add     esp, 8
.text:10001157      test    eax, eax
.text:10001159      jz      loc_100011E4

```

如果查找失败，恶意代码则只调用trampoline\_function，这不会发生任何恶意行为，`send` 函数操作与没有安装挂钩前一样。



```

.text:100011E4 loc_100011E4:                                ; CODE XREF: sub_1000113D+1C↑j
.text:100011E4      mov     edx, [ebp+arg_C]
.text:100011E7      push    edx
.text:100011E8      mov     eax, [ebp+arg_8]
.text:100011EB      push    eax
.text:100011EC      mov     ecx, [ebp+Str]
.text:100011EF      push    ecx
.text:100011F0      mov     edx, [ebp+arg_0]
.text:100011F3      push    edx
.text:100011F4      call    dword_10003484
.text:100011FA      add     esp, 10h
.text:100011FD      mov     esp, ebp
.text:100011FF      pop     ebp
.text:10001200      retn    10h
.text:10001200 sub_1000113D      endp

```

而如果查找成功，会创建一个添加到向外传输的缓冲区中的字符串。这个字符串以 `RCPT TO: <` 开头，随后是邮件地址，最后以 `>\r\n` 结束。这个邮件地址就是在ini文件中提取到的邮件地址，为 `billy@malwareanalysisbook.com`。这向所有的发出邮件中添加了一个收件人。

```

.text:1000116D      push    offset aRcptTo_1 ; "RCPT TO: <"
.text:10001172      lea     ecx, [ebp+Destination]
.text:10001178      push    ecx                ; void *
.text:10001179      call    memcpy
.text:1000117E      add     esp, 0Ch
.text:10001181      push    101h                ; Size
.text:10001186      push    offset eaddress ; Src
.text:1000118B      push    offset aRcptTo_2 ; "RCPT TO: <"
.text:10001190      call    strlen
.text:10001195      add     esp, 4
.text:10001198      lea     edx, [ebp+eax+Destination]
.text:1000119F      push    edx                ; void *
.text:100011A0      call    memcpy
.text:100011A5      add     esp, 0Ch
.text:100011A8      push    offset Source        ; ">\r\n"
.text:100011AD      lea     eax, [ebp+Destination]
.text:100011B3      push    eax                ; Destination
.text:100011B4      call    strcat

```

## 问题九：抓取行为

- 你怎样用Wireshark动态捕获这个恶意代码的行为？

可以通过抓取网络流量的方法，通过 Wireshark 抓取的网路数据，可以看到一个假冒的邮件服务器以及Outlook Express客户端。

## Lab 11-03

### 问题一：基础静态分析

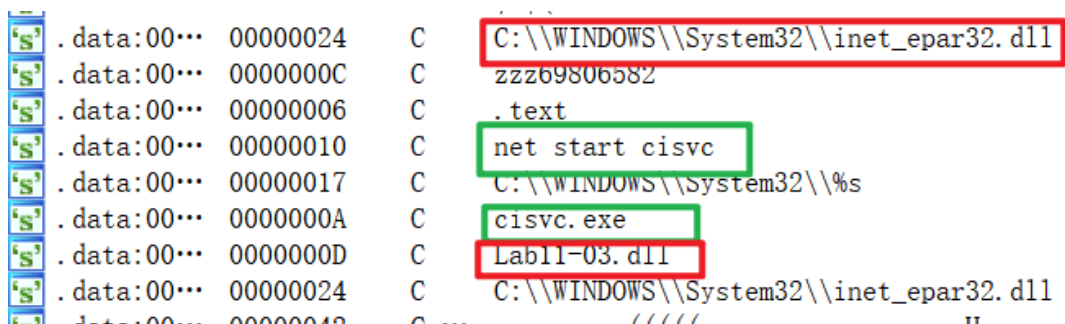
- 使用基础的静态分析过程，你可以发现什么有趣的线索？

首先查看Lab11-03.exe的字符串信息，注意到：

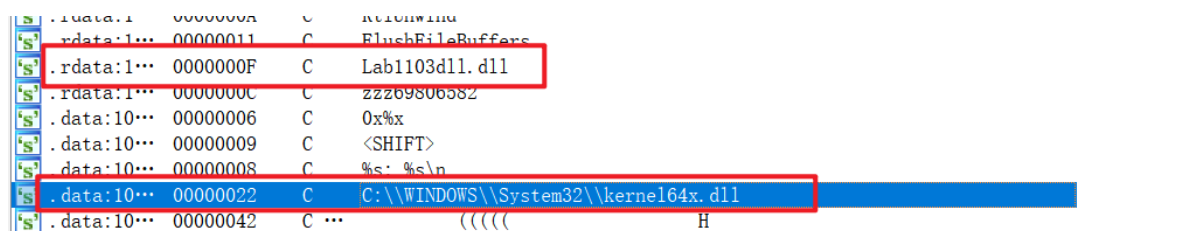
- `C:\WINDOWS\System32\inet_eap32.dll` 和 `Lab11-03.dll`

- exe程序: `cisvc.exe`
- 命令: `net start cisvc`

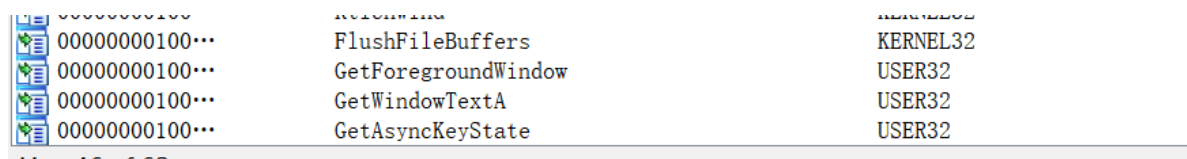
其中, `cisvc.exe` 是 Indexing Service 的主要执行文件。Indexing Service 会扫描计算机上的文件和文档, 并创建一个包含关键词和其他元数据的索引。`cisvc.exe` 负责运行和管理索引服务。在系统启动时, 它会自动启动, 并在后台运行, 处理文件索引的创建、更新和维护。这有助于确保文件搜索功能的高效性。而 `net start cisvc` 则是启动这个索引服务的命令。



而分析Lab11-03.dll的字符串信息, 同样发现两个dll: `Lab1103dll.dll` 和 `C:\\WINDOWS\\System32\\kernel64x.dll`。



查看它的导入表。导入了 `GetForegroundWindow`、`GetWindowTextA`、`GetAsyncKeyState` 三个导入函数。初步推断这是个击键记录器, 将击键记录保存到 `kernel64x.dll`。



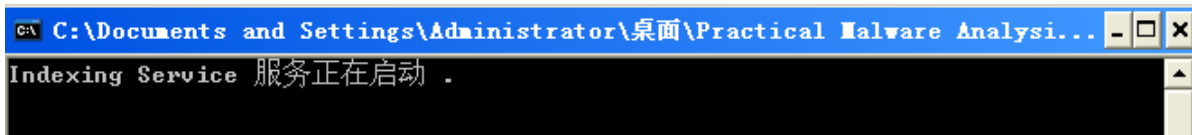
查看它的导出表。发现导出了一个奇怪的函数: `zzz69806582`。

Name	Address	Ordinal
zzz69806582	0000000010001540	1
DllEntryPoint	0000000010001968	[main en...

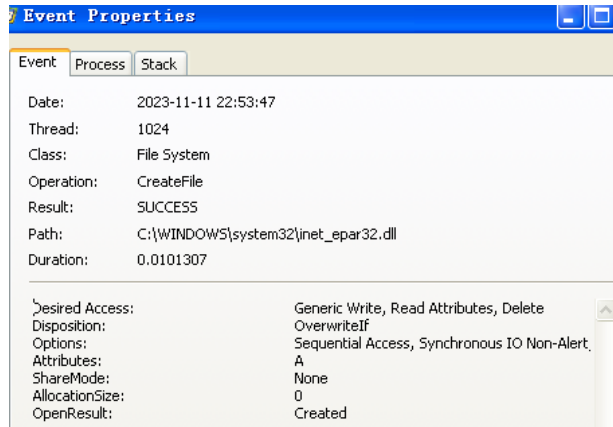
## 问题二: 动态运行恶意代码

启动Process Explorer、启动procmon并过滤 `Lab11-03.exe`, 动态运行恶意代码。

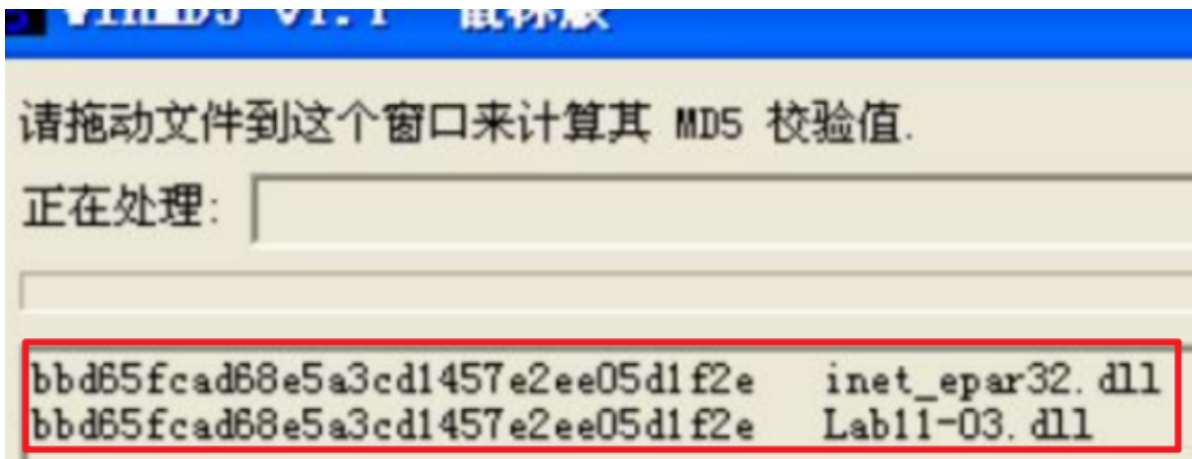
首先弹出窗口, 显示 `Indexing Service`服务正在启动, 这说明恶意代码启动了索引服务。



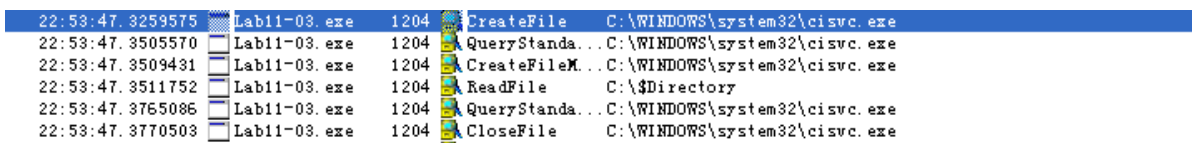
在procmon中分析，看到恶意代码创建了 `C:\Windows\System32\inet_epar32.dll` 。



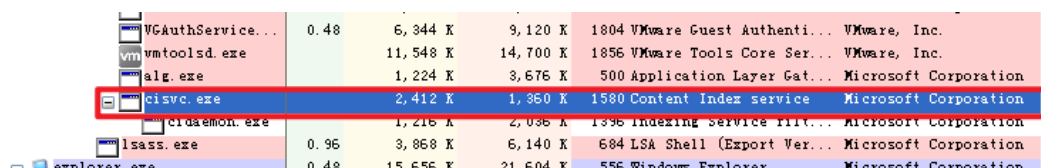
进行MD5比对能够发现， `inet_epar32.dll` 和 `Lab11-03.dll` 是相同文件，这儿说明恶意代码复制了Lab11-03.dll到windows系统目录。



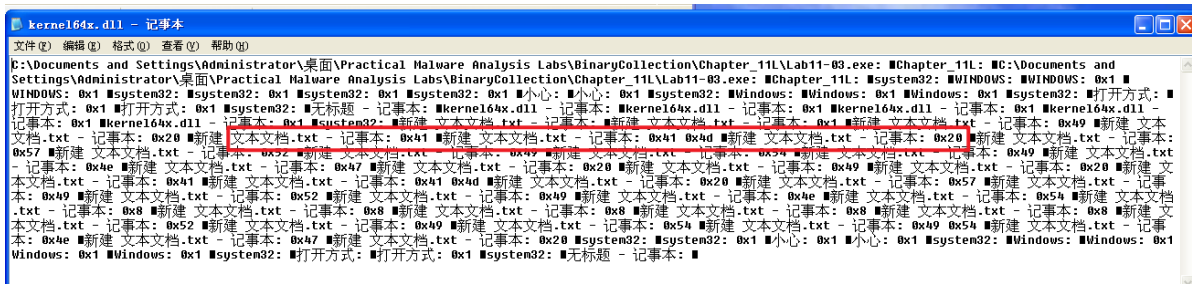
在procmon中还发现恶意代码创建打开了 `cisvc.exe` 的句柄，但是没有看到任何 `WriteFile` 操作。



在Process Explorer中，观察到 `cisvc.exe` 正在系统中运行。



打开记事本并且输入一串字符。发现创建了 `C:\WINDOWS\system32\kernel64x.dll` 文件。打开文件，发现记录了刚才的键盘记录和窗体记录。



### 问题三：安装 Lab11-03.dll 并长期驻留

Lab11-03.exe如何安装Lab11-03.dll使其长期驻留？

总的来说，恶意代码将 `shellcode` 写入索引服务的可执行文件 `cisvc.exe`，并通过 `cisvc.exe` 的入口重定向来允许 `shellcode`。`shellcode` 加载了Lab11-03.dll复制后的备份文件，并调用了它的导出函数。下面具体分析：

加载Lab11-03.exe到IDA Pro，检查 `main` 函数，

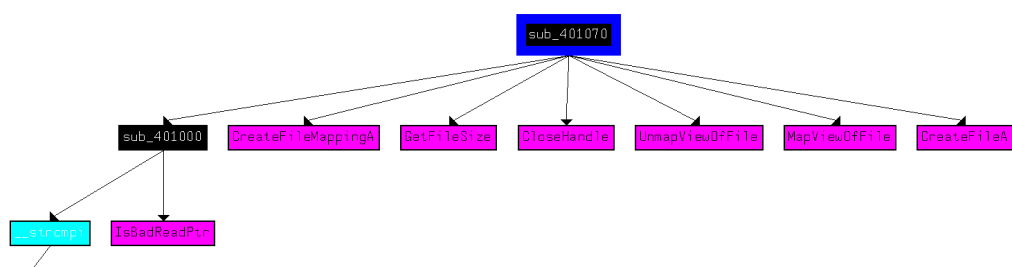
```
.text:004012D0 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:004012D0 _main      proc near      ; CODE XREF: start+AF↓p
.text:004012D0
.text:004012D0 Buffer      = byte ptr -104h
.text:004012D0 argc       = dword ptr  8
.text:004012D0 argv       = dword ptr  0Ch
.text:004012D0 envp       = dword ptr  10h
.text:004012D0
.text:004012D0 push      ebp
.text:004012D1 mov       ebp, esp
.text:004012D3 sub       esp, 104h
.text:004012D9 push      0 ; bFailIfExists
.text:004012DB push      offset NewFileName ; "C:\\WINDOWS\\System32\\inet_epar32.dll"
.text:004012E0 push      offset ExistingFileName ; "Lab11-03.dll"
.text:004012E5 call      ds:CopyFileA
.text:004012EB push      offset aCisvcExe ; "cisvc.exe"
.text:004012F0 push      offset Format ; "C:\\WINDOWS\\System32\\%s"
.text:004012F5 lea       eax, [ebp+Buffer]
.text:004012FB push      eax ; Buffer
.text:004012FC call      _sprintf
.text:00401301 add       esp, 0Ch
.text:00401304 lea       ecx, [ebp+Buffer]
.text:0040130A push      ecx ; lpFileName
.text:0040130B call      sub_401070
.text:00401310 add       esp, 4
.text:00401313 push      offset Command ; "net start cisvc"
.text:00401318 call      _system
.text:0040131D add       esp, 4
.text:00401320 xor       eax, eax
.text:00401322 mov       esp, ebp
.text:00401324 pop       ebp
.text:00401325 retn
.text:00401325 _main      endp
```

可以发现，首先调用 `CopyFileA` 复制 `Lab11-03.dll` 到 `C:\WINDOWS\system32\inet_epar32.dll`。

然后创建一个字符串 `C:\windows\system32\cisvc.exe`，将其作为参数传递给了 `sub_401070`；

之后是使用系统运行命令 `net start cisvc`，来启动索引服务。

重点分析一下 `sub_401070` 函数，看看它对 `cisvc.exe` 做了什么。首先查看其交叉引用图。



sub\_401070 调用的函数有：

- `createfile`：创建文件
- `createfilemappingA`：创建一个文件映射内核对象
- `MapViewOfFile`：将一个文件映射对象映射到当前应用程序的地址空间，将cisvc.exe 映射到内存中。
  - `MapViewOfFile` 返回的内存映射视图的基地址（`lpBaseAddress`）可以被读写。
- `UnmapViewOfFile`，调用该函数后，对这个文件做的任何修改都会被写入到硬盘，这

这些函数都有访问这个文件的权限。

继续分析，

```
.text:0040127C
.text:0040127C loc_40127C:
.text:0040127C mov     edi, [ebp+lpBaseAddress]
.text:0040127F add     edi, [ebp+var_28]
.text:00401282 mov     ecx, 4Eh ; 'N'
.text:00401287 mov     esi, offset dword_409030
.text:0040128C rep movsd
.text:0040128E movsw
.text:00401290 mov     eax, [ebp+var_14]
.text:00401293 mov     ecx, [ebp+var_28]
.text:00401296 sub     ecx, [eax+14h]
.text:00401299 mov     edx, [ebp+var_24]
.text:0040129C add     ecx, [edx+2Ch]
.text:0040129F mov     eax, [ebp+var_24]
.text:004012A2 mov     [eax+28h], ecx
.text:004012A5 mov     ecx, [ebp+hFile]
.text:004012A8 push    ecx ; hObject
.text:004012A9 call    ds:CloseHandle
.text:004012AF mov     edx, [ebp+hFileMappingObject]
.text:004012B2 push    edx ; hObject
.text:004012B3 call    ds:CloseHandle
.text:004012B9 mov     eax, [ebp+lpBaseAddress]
.text:004012BC push    eax ; lpBaseAddress
.text:004012BD call    ds:UnmapViewOfFile
.text:004012C3 xor     eax, eax
```

这段指令进行的工作是：首先将文件的映射位置移入到 `EDI`，用 `Var_28` 调整一些偏移量。`rep movsd` 循环，用 `ecx (4E)` 计数来写入该数量的 `DWORD`。因此总字节数为 `0x4E*4=312` 字节。之后 `byte_409030` 被移入 `ESI`，复制 `byte_409030` 的数据到映射文件中。

而查看 `byte_409030` 处的数据，看到以下内容：



```

.data:00409030 dword_409030 dd 81E58955h ; DATA XREF: sub_401070+19D↑r
.data:00409030 ; sub_401070+1FF↑w ...
.data:00409034 db 0ECh
.data:00409035 db 40h ; @
.data:00409036 db 0
.data:00409037 db 0
.data:00409038 db 0
.data:00409039 db 0E9h
.data:0040903A db 0F6h
.data:0040903B db 0
.data:0040903C db 0
.data:0040903D db 0
.data:0040903E db 56h ; V
.data:0040903F db 57h ; W
.data:00409040 db 8Bh
.data:00409041 db 74h ; t
.data:00409042 db 24h ; $
.data:00409043 db 0Ch
.data:00409044 db 31h ; 1
.data:00409045 db 0FFh
.data:00409046 db 0FCh
.data:00409047 db 31h ; 1
.data:00409048 db 0C0h
.data:00409049 db 0ACh
.data:0040904A db 38h ; 8
.data:0040904B db 0E0h
.data:0040904C db 74h ; t
.data:0040904D db 0Ah
.data:0040904E db 0C1h
.data:0040904F db 0CFh
.data:00409050 db 0Ah

```

在IDA Pro中按C键，得到反汇编显示。

```

.data:00409030 |
.data:00409030 loc_409030: ; DATA XREF: sub_401070+19D↑r
.data:00409030 ; sub_401070+1FF↑w ...
.data:00409030 push ebp
.data:00409031 mov ebp, esp ; DATA XREF: sub_401070+1AD↑r
.data:00409031 ; sub_401070+1BD↑r
.data:00409033 sub esp, 40h ; DATA XREF: sub_401070+1CD↑r
.data:00409039 jmp loc_409134
.data:0040903E

```

很明显，这是一段 `shellcode`。分析一下，这段 `shellcode` 做了什么。

跳转 `jmp loc_409134`：发现这里出现了 `inet_epar32.dll` 和 `cisvc.exe` 的导出函数 `zzz698068582`。所以可以猜测，向 `cisvc.exe` 写入了 `shellcode`，`shellcode` 加载了这个 `dll`，并调用了它的导出函数。

```

00409130 78 56 34 12 E8 A4 FF FF FF 43 3A 5C 57 49 4E 44 xv4.瑜...C:\WIND
00409140 4F 57 53 5C 53 79 73 74 65 6D 33 32 5C 69 6E 65 OWS\System32\ine
00409150 74 5F 65 70 61 72 33 32 2E 64 6C 6C 00 7A 7A 7A t_epar32.dll.zzz
00409160 36 39 38 30 36 35 38 32 00 00 00 00 2E 74 65 78 69806582....tex

```

接下来IDA分析一下 `cisvc.exe`，第一个调用是 `sub_1001AD5` 函数。

```
; Attributes: bp-based frame

public start
start proc near

; FUNCTION CHUNK AT .text:01001B2C SIZE 00000036 BYTES

push    ebp
mov     ebp, esp
sub     esp, 40h
jmp     loc_1001B2C
start endp ; sp-analysis failed
```

```
; START OF FUNCTION CHUNK FOR start

loc 1001B2C:
call    sub_1001AD5
inc     ebx
cmp     bl, [edi+edx*2+49h]
dec     esi
inc     esp
dec     edi
push    edi
push    ebx
pop     esp
push    ebx
jns     short near ptr word_1001BB2
```

查看该函数。

```
.text:01001AD5 ; __unwind { // __SEH_prolog
.text:01001AD5         pop     ebx
.text:01001AD6         call    sub_1001AB4
.text:01001ADB         mov     edx, eax
.text:01001ADD         push    753A4FCh
.text:01001AE2         push    edx
.text:01001AE3         call    sub_1001A57
.text:01001AE8         mov     [ebp-4], eax
.text:01001AEB         push    7C0DFCAAh
.text:01001AF0         push    edx
.text:01001AF1         call    sub_1001A57
.text:01001AF6         mov     [ebp-0Ch], eax
.text:01001AF9         lea     eax, [ebx+0]
.text:01001AFF         push    0
.text:01001B04         push    0
.text:01001B09         push    eax
.text:01001B0A         call    dword ptr [ebp-4]
.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]
.text:01001B1E         mov     [ebp-8], eax
.text:01001B21         call    dword ptr [ebp-8]
.text:01001B24         mov     esp, ebp
.text:01001B26         pop     ebp
.text:01001B27         jmp     loc_100129B
.text:01001B27 ; } // starts at 1001AD5
```



在OllyDbg中设置断点，分析内部进行的 `call dword ptr ss:[ebp-4]` 指令。

01001AFF	. 68 00000000	PUSH 0	
01001B04	. 68 00000000	PUSH 0	
01001B09	. 50	PUSH EAX	
01001B0A	. FF55 FC	CALL DWORD PTR SS:[EBP-4]	kernel32.LoadLibraryExA
01001B0D	. 8945 F0	MOV DWORD PTR SS:[EBP-10],EAX	
01001B10	. 8D83 24000000	LEA EAX,DWORD PTR DS:[EBX+24]	
01001B16	. 50	PUSH EAX	
01001B17	. 8B45 F0	MOV EAX,DWORD PTR SS:[EBP-10]	

提示出用的是 `LoadLibraryExA` 函数，而查看参数得知是将 `inet_epar32.dll` 载入内存。

0007FF70	01001B0D	CALL to LoadLibraryExA from cisvc.01001B0A
0007FF74	01001B31	FileName = "C:\WINDOWS\System32\inet_epar32.dll"
0007FF78	00000000	hFile = NULL
0007FF7C	00000000	Flags = 0
0007FF80	00001FE0	

同理接着分析下一个函数调用 `call dword ptr ss:[ebp-c]`。

01001B1A	. 50	PUSH EAX	
01001B1B	. FF55 F4	CALL DWORD PTR SS:[EBP-C]	kernel32.GetProcAddress
01001B1E	. 8945 F8	MOV DWORD PTR SS:[EBP-8],EAX	

调用 `GetProcAddress` 函数，作用是获取导出函数的地址。

0007FF74	01001B1E	CALL to GetProcAddress from cisvc.01001B1B
0007FF78	10000000	hModule = 10000000
0007FF7C	01001B55	ProcNameOrOrdinal = "zzz69806582"
0007FF80	00001FE0	

调用后，将返回值 `eax` 赋值给 `ebp-8`，之后是 `call dword ptr ss:[ebp-8]`，这说明接下来调用了这个导出函数。

01001B1B	. FF55 F4	CALL DWORD PTR SS:[EBP-C]	kernel32.GetProcAddress
01001B1E	. 8945 F8	MOV DWORD PTR SS:[EBP-8],EAX	
01001B21	. FF55 F8	CALL DWORD PTR SS:[EBP-8]	
01001B24	. 89EC	MOV ESP,EBP	

最后恶意代码跳转到原始的入口点，从而使服务正常执行。

## 问题四：感染文件

- 这个恶意代码感染Windows系统的哪个文件？

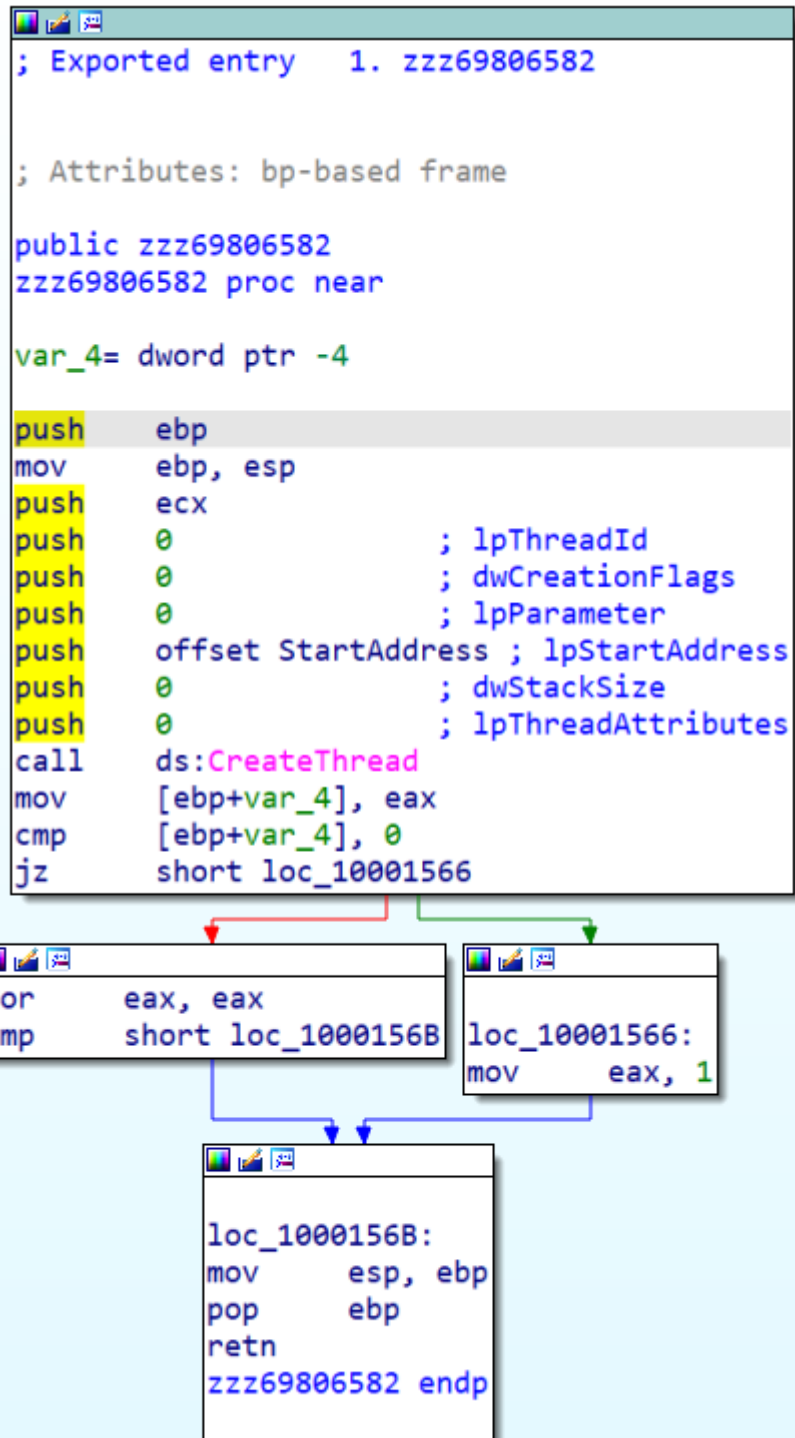
通过前面分析很容易知道，它感染了索引服务的可执行文件 `cisvc.exe`，向其写入了 `shellcode` 并使其入口重定向使其先执行 `shellcode`。

## 问题五：Lab11-03.dll的行为

- Lab11-03.dll做了什么？

Lab11-03.dll 是一个轮询的记录器，这在它的导出函数 `zzz69806582` 中得到实现。

具体Lab11-03.dll的导出函数 `zzz69806582`：



查看创建线程的 `lpStartAddress` 参数。可以发现，这里首先尝试打开MZ互斥量，如果不存在，则调用 `CreateMutexA` 创建一个。

```

text:1000146C push    edx                ; Dest
text:1000146D call     __mbstncpy
text:10001472 add     esp, 0Ch
text:10001475 push    offset Name        ; "MZ"
text:1000147A push    0                        ; bInheritHandle
text:1000147C push    1F0001h             ; dwDesiredAccess
text:10001481 call     ds:OpenMutexA
text:10001487 mov     [ebp+hObject], eax
text:1000148D cmp     [ebp+hObject], 0
text:10001494 jz      short loc_1000149D

```

```

.text:1000149D
.text:1000149D loc_1000149D:
.text:1000149D push    offset Name        ; "MZ"
.text:100014A2 push    1                        ; bInitialOwner
.text:100014A4 push    0                        ; lpMutexAttributes
.text:100014A6 call     ds:CreateMutexA
.text:100014AC mov     [ebp+hObject], eax
.text:100014B2 cmp     [ebp+hObject], 0
.text:100014B9 jnz     short loc_100014BD

```

之后，创建了文件 `C:\WINDOWS\System32\kernel64x.dll`。这就是之前所发现的记录击键记录的文件

```

.text:100014BD
.text:100014BD loc_100014BD:                ; hTemplateFile
.text:100014BD push    0
.text:100014BF push    80h ; '€'            ; dwFlagsAndAttributes
.text:100014C4 push    4                        ; dwCreationDisposition
.text:100014C6 push    0                        ; lpSecurityAttributes
.text:100014C8 push    1                        ; dwShareMode
.text:100014CA push    0C000000h             ; 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

```

之后调用 `SetFilePointer`，作用是在一个文件中设置新的读取位置，`dwMoveMethod` 值为 2，开始点为文件的结尾位置。

```
.text:100014EB
.text:100014EB loc_100014EB:                ; dwMoveMethod
.text:100014EB push    2
.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
```

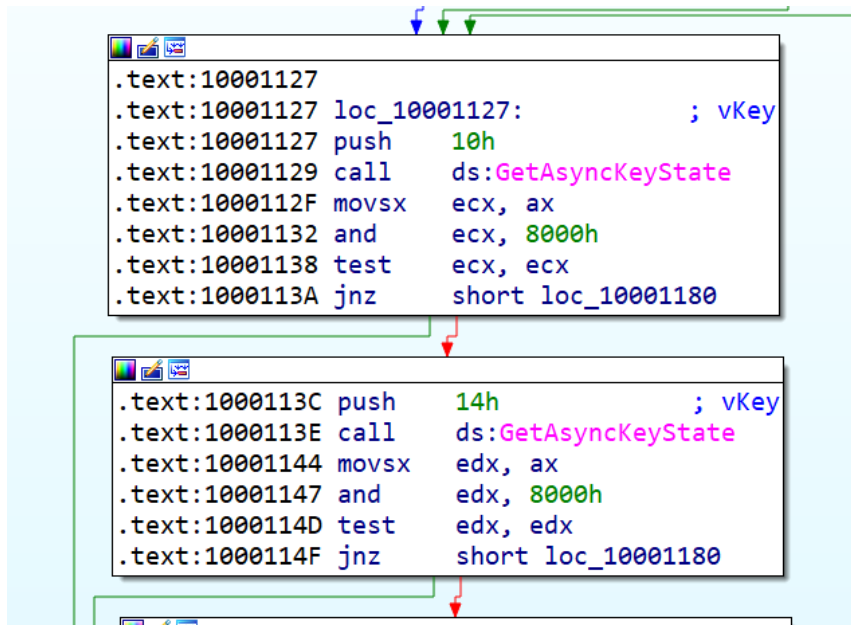
接下来调用关系为 `sub_10001380` → `sub_10001030` → `sub_10001000`

查看 `sub_10001000` :

```
.text:10001000 push    ebp
.text:10001001 mov     ebp, esp
.text:10001003 push    ecx
.text:10001004 call    ds:GetForegroundWindow
.text:1000100A mov     [ebp+hWnd], eax
.text:1000100D mov     eax, [ebp+nMaxCount]
.text:10001010 push    eax                ; nMaxCount
.text:10001011 mov     ecx, [ebp+lpString]
.text:10001014 push    ecx                ; lpString
.text:10001015 mov     edx, [ebp+hWnd]
.text:10001018 push    edx                ; hWnd
.text:10001019 call    ds:GetWindowTextA
.text:1000101F test     eax, eax
.text:10001021 jnz     short loc_1000102A
```

调用了 `GetForegroundWindow` 和 `GetWindowTextA` 函数，用来判断当前哪个程序正在输入和获取当前的标题。

返回 `sub_10001030` 函数后，调用 `GetAsyncKeyState` 判断一个按键是被按下还是弹起。



## 问题六：收集的数据

- 这个恶意代码将收集的数据存放在何处？

恶意代码存储击键记录和窗体输入记录，保存到创建的文件 `C:\WINDOWS\System32\kernel64x.dll` 中。

## Yara 规则编写

根据前面的分析，编写Yara规则如下：

```
rule Lab11_01exe {
    meta:
        description = "Lab11-01.exe"
    strings:
        $s1 = "gina.dll" fullword ascii
        $s2 = "msgina32.dll" fullword ascii
        $s3 = "DllRegister" fullword ascii
        $s4 = "DllUnregister" fullword ascii
    condition:
        uint16(0) == 0x5a4d and
        uint32(uint32(0x3c)) == 0x00004550 and filesize < 200KB and
        all of them
}

rule Lab11_02dll {
    meta:
        description = "Lab11-02.dll"
    strings:
```

```

    $s1 = "spoolvxx32.dll" fullword ascii
    $s2 = "THEBAT.EXE" fullword ascii
    $s3 = "MSIMN.EXE" fullword ascii
    $s4 = "Lab11-02.dll" fullword ascii
    $s5 = "\\Lab11-02.ini" fullword ascii
    $s6 = "AppInit_DLLs" fullword ascii
    $s7 = "RCPT TO: <" fullword ascii
    condition:
        uint16(0) == 0x5a4d and
        uint32(uint32(0x3c))==0x00004550 and filesize < 60KB and
        all of them
}

rule Lab11_03dll {
    meta:
        description = "Lab11-03.dll"
    strings:
        $s1 = "C:\\WINDOWS\\System32\\kernel64x.dll" fullword ascii
        $s2 = "Lab1103dll.dll" fullword ascii
    condition:
        uint16(0) == 0x5a4d and
        uint32(uint32(0x3c))==0x00004550 and filesize < 100KB and
        all of them
}

rule Lab11_03exe {
    meta:
        description = "Lab11-03.exe"
    strings:
        $s1 = "C:\\WINDOWS\\System32\\inet_epar32.dll" fullword ascii
        $s2 = "cisvc.exe" fullword ascii
        $s3 = "Lab11-03.dll" fullword ascii
        $s4 = "net start cisvc" fullword ascii
        $s5 = "command.com" fullword ascii
        $s6 = "COMSPEC" fullword ascii
        $s7 = "zzz69806582" fullword ascii
    condition:
        uint16(0) == 0x5a4d and
        uint32(uint32(0x3c))==0x00004550 and filesize < 100KB and
        all of them
}

```

能够扫描到相应的病毒样本，验证了Yara规则的正确性：

```

PS D:\NKU\23Fall\恶意代码分析与防治技术\yara-4.3.2-2150-win64> ./yara64 Lab11.yar Chapter_11L
Lab11_01exe Chapter_11L\Lab11-01.exe
Lab11_02dll Chapter_11L\Lab11-02.dll
Lab11_03dll Chapter_11L\Lab11-03.dll
Lab11_03exe Chapter_11L\Lab11-03.exe

```

能够扫描到相应的病毒样本，验证了Yara规则的正确性：

	sample
类型:	文件夹
位置:	D:\NKU\code\Python
大小:	17.1 GB (18,457,236,441 字节)
占用空间:	17.2 GB (18,484,576,256 字节)
包含:	14,589 个文件, 0 个文件夹
创建时间:	2023年10月4日, 18:39:02

```
Microsoft Visual Studio 调试 × + v
扫描到的文件:
Lab11_01.exe D:\NKU\code\Python\sample\Lab11-01.exe
Lab11_03.dll D:\NKU\code\Python\sample\Lab11-03.dll
Lab11_02.dll D:\NKU\code\Python\sample\Lab11-02.dll
Lab11_03.exe D:\NKU\code\Python\sample\Lab11-03.exe
运行时间为 12 s
```

## IDA Python 自动化分析

### 编写脚本查看导入的动态链接库

```
import idaapi

def get_imported_libraries():
    # 创建一个用于存储导入库的集合
    imported_libraries = set()

    # 遍历导入表中的所有模块
    for i in range(idaapi.get_import_module_qty()):
        modname = idaapi.get_import_module_name(i)
        if modname:
            imported_libraries.add(modname)

    return sorted(imported_libraries)
```

```
def main():
    # 获取导入库列表
    imported_libraries = get_imported_libraries()

    if imported_libraries:
        print("导入的动态链接库：")
        for lib in imported_libraries:
            print(lib)
    else:
        print("没有找到导入的动态链接库。")

if __name__ == "__main__":
    main()
```

分析 Lab11-02.dll 导入的动态链接库：

导入的动态链接库：  
 ADVAPI32  
 KERNEL32  
 MSVCRT

## 编写脚本查找特定函数，分析其控制流，并识别其中的基本块和交叉引用

```
import idaapi
import idutils
import idc

def analyze_function(function_name):
    # 查找二进制文件中的函数地址
    func_ea = idc.get_name_ea(idc.BADADDR, function_name)

    if func_ea == idc.BADADDR:
        print(f"函数 '{function_name}' 未找到")
        return

    # 分析函数的控制流
    f = idaapi.get_func(func_ea)
    if not f:
        print(f"无法获取函数 '{function_name}'")
        return

    print(f"分析函数 '{function_name}', 入口地址：0x{func_ea:08X}")

    for block in idaapi.FlowChart(f):
```



```

print(f"基本块: 0x{block.start_ea:08X}")

# 使用idautils.XrefsTo来获取交叉引用
for head in idautils.Heads(block.start_ea, block.end_ea):
    disasm = idc.GetDisasm(head)
    print(f"    0x{head:08X}: {disasm}")

# 识别交叉引用
for ref in idautils.XrefsTo(head):
    print(f"        引用自: 0x{ref.frm}")

if __name__ == '__main__':
    target_function = "sub_10606"

    analyze_function(target_function)

```

分析 `Lab11-02.dll` 的函数 `sub_10001203`，结果如下：

分析函数 'sub\_10001203', 入口地址: 0x10001203

基本块: 0x10001203

```
0x10001203: push    ebp
             引用自: 0x268440306
0x10001204: mov     ebp, esp
             引用自: 0x268440067
0x10001206: sub     esp, 0Ch
             引用自: 0x268440068
0x10001209: mov     eax, [ebp+arg_4]
             引用自: 0x268440070
0x1000120C: sub     eax, [ebp+lpAddress]
             引用自: 0x268440073
0x1000120F: sub     eax, 5
             引用自: 0x268440076
0x10001212: mov     [ebp+var_4], eax
             引用自: 0x268440079
0x10001215: lea     ecx, [ebp+flOldProtect]
             引用自: 0x268440082
0x10001218: push    ecx; lpflOldProtect
             引用自: 0x268440085
0x10001219: push    40h ; '@'; flNewProtect
             引用自: 0x268440088
0x1000121B: push    5; dwSize
             引用自: 0x268440089
0x1000121D: mov     edx, [ebp+lpAddress]
             引用自: 0x268440091
0x10001220: push    edx; lpAddress
             引用自: 0x268440093
0x10001221: call    ds:VirtualProtect
             引用自: 0x268440096
0x10001227: push    055h; Size
```

---

## 实验结论及实验心得

### 1. 深入理解恶意代码行为:

- 通过静态和动态分析, 我深入了解了不同类型的恶意代码如何操纵系统、窃取信息以及劫持特定应用程序。这对于理解威胁行为模式和构建有效的防御策略至关重要。

### 2. 动态与静态相结合的方法:

- 将动态和静态分析相结合的方法对于全面理解恶意代码至关重要。静态分析帮助我识别关键函数、字符串和导入的库, 而动态分析则让我验证实际行为并捕获运行时信息。

### 3. 工具的熟练应用:

- 实验过程中我熟练使用了安全工具，如IDA Pro、OllyDbg等，以及动态分析环境，这提高了我的工具使用能力，同时也让我更深入地理解了这些工具的原理和功能。

#### 4. 自动化分析和脚本编写：

- 编写Yara规则和IDA Python脚本的经验提高了我的自动化分析能力。这不仅有助于快速检测恶意代码，还提高了我的编程和脚本编写技能。