



南開大學  
Nankai University

网络空间安全学院  
恶意代码分析与防治技术课程实验报告

实验十：恶意代码行为

姓名：周钰宸

学号：2111408

专业：信息安全

2024 年 1 月 14 日

# 1 实验目的

1. 复习教材和课件内第 11 章的内容。
2. 完成 Lab11 的实验内容，对 Lab11 的三个样本进行依次分析，编写 Yara 规则，并尝试使用 IDAPython 的自动化分析辅助完成。

# 2 实验原理

## 2.1 IDAPro

IDA 是一款广泛使用的交互式反汇编工具，它支持多种平台和文件格式。IDA 的主要功能是将机器代码转换为人类可读的汇编代码，从而帮助研究人员理解和分析程序的功能和行为。

## 2.2 IDAPython

IDAPython 是一个 IDA 插件，允许用户使用 Python 语言来脚本化 IDA 的功能。这为自动化任务和复杂的分析提供了巨大的灵活性。可以调用 IDA 的所有 API 函数，使用 Python 的功能模块编写强大的自动分析脚本。

## 2.3 OllyDBG

OllyDbg 是一个 32 位的汇编级别的调试器，主要用于 Microsoft Windows。它是反向工程和软件分析中的一个流行工具。OllyDbg 的特点是其用户友好的界面、多窗口模式、直接修改代码、以及强大的插件支持。

## 2.4 Windows 恶意代码行为

本次实验旨在对 Lab11 中的 Windows 恶意代码进行全面分析。具体而谈，Windows 下的操作系统一般具有如下的一些常用的恶意软件技术：

1. **Windows 注册表修改**：恶意软件可以修改 Windows 注册表中的关键信息，以便在系统启动时自动运行或隐藏自己的存在。
2. **远程 Shell 后门**：恶意软件可以通过远程控制技术，获取用户的敏感信息或控制系统进行攻击。
3. **钓鱼攻击**：恶意软件可以伪装成合法的应用程序或文件，诱使用户点击并下载，从而感染系统。
4. **Windows 服务**：利用 Windows 服务框架的漏洞或弱点，来植入恶意代码或程序，从而在系统启动时自动运行并隐藏自己的存在。
5. **Rootkit**：这是一种隐藏在系统内核或驱动程序中的恶意软件，可以隐藏自己的存在并避免被常规的安全软件检测和清除。
6. **WindowsDLL 拦截**：利用 Windows 系统中的动态链接库加载顺序的漏洞来进行攻击的技术。恶意软件可以替换系统或应用程序所依赖的 DLL 文件，使其在程序启动时加载恶意的 DLL，从而执行恶意代码或进行其他攻击行为。

### 3 实验过程

#### 3.1 实验环境及工具

虚拟机软件	VMware Workstation 17 Pro
宿主机	Windows 11 家庭中文版
虚拟机操作系统	Windows XP Professional
实验工具 1	主机 WinDBG(X86).lnk
实验工具 2	OllyDBG 2.01
实验工具 3	IDAPro 6.6.14.1224
实验工具 5	虚拟机 Windbg_x86_6.12.2.633
配套工具	Python 2.7.2

表 1: 本次实验环境及工具

#### 3.2 Lab11-1

分析恶意代码 Lab11-1.exe 的恶意行为。

##### 3.2.1 静态分析

首先进行一些基础的静态分析，先使用 PEiD 查看其加壳情况和导入表：

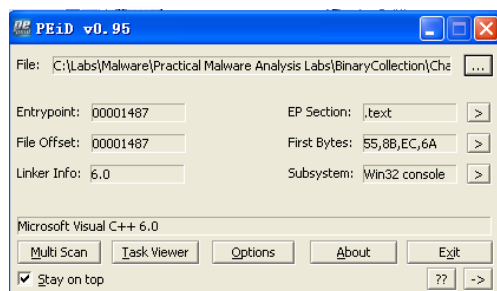


图 3.1: 加壳情况

图3.1看到病毒没有进行加壳，使用 VC6 编写，然后查看其细节，发现它额外具有一个资源节，先查看其导入表：

DllName	OriginalFirstThunk	TimeDateStamp	ForwarderChain	Name	FirstThunk
KERNEL32.dll	00007534	00000000	00000000	0000769E	0000700C
ADVAPI32.dll	00007528	00000000	00000000	000076D0	00007000

Thunk RVA	Thunk Offset	Thunk Value	Hint/Ordinal	API Name
0000700C	0000700C	00007632	0295	SizeOfResource
00007010	00007010	00007644	0105	LoadResource
00007014	00007014	00007654	01C7	LoadResource
00007018	00007018	00007622	02B8	VirtualAlloc
0000701C	0000701C	00007674	0124	GetModuleFileNameA
00007020	00007020	0000768A	0126	GetModuleHandleA
00007024	00007024	00007612	0086	FreeResource
00007028	00007028	00007664	00A3	FindResourceA

导入表 1

导入表 2

图 3.2: 查看导入表

图3.2能看到一些有用的导入表：

- 资源节处理: FindResourceA, SizeofResource 还有 LoadResource 都代表了恶意代码可能具有操作资源的功能, 结合之前发现的资源部分, 有必要深入分析。
- APVAPI32.dll: 其中包含了 RegSet 和 CreateValue 函数, 说明病毒具有对注册表键值的操作功能。

接下来使用 PEView 查看字符串:

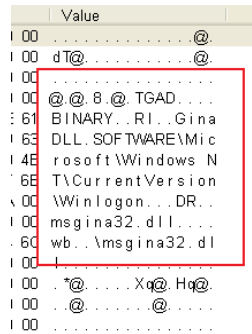


图 3.3: 字符串

图3.3可以看到一些有趣的字符串:

- **TGAD BINARY:** 这可能是某个可执行文件的名字, 先不管。
- **GinaDLL:** GinaDLL 是 Windows 操作系统中动态链接库的缩写。它是用户登录过程的一部分, 负责处理用户身份验证。通常攻击者可能会替换原始的 GinaDLL 以实现对用户凭据的窃取或植入恶意功能。
- **SOFTWARE\\Microsoft\\windowsNT\\CurrentVersion\\Winlogon:** 这是 Windows 注册表中的一个关键路径, 负责管理用户登录和注销过程的配置。推测攻击者可能会修改这个路径下的值, 以实现自动启动恶意代码或绕过系统安全措施。
- **msginan32.dll:** 这可能是与 GinaDLL 相关的文件, 通常位于系统目录下。攻击者可能会替换或修改这个文件, 以实现对用户身份验证的篡改或记录用户凭据。

结合这些信息, 我们推测可能病毒会尝试对用户信息进行恶意的篡改。然后我们在旁边的资源节展开后发现了:

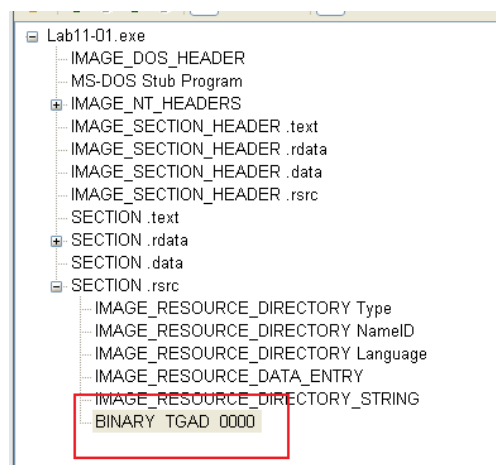
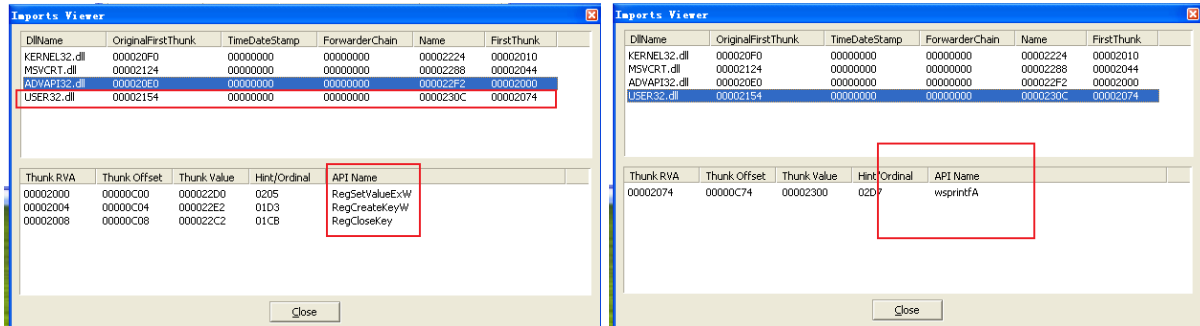


图 3.4: TGAD 资源节

图3.4看到一个命名为 **TGAD** 的资源节，结合字符串发现的这个名字，推测其是一个可执行文件，再结合导入表对其的 **Find** 和 **Load**，病毒可能会尝试加载这个恶意的 **exe**。

因此接下来我们去到 Resource Hacker 中，将这个 **TGAD** 直接保存下来保存为 **bin** 可执行文件，然后命名为 **Lab11-01-TGAD.src**。然后使用 **PEiD** 查看其导入表和导出表：



导入表 1

导入表 2

图 3.5: 查看导入表

图3.5看到一些有趣的导入表：

- **RegSet** 和 **CreateKey**：又可以这些注册表键值操作的函数，推测 **exe** 加载 **TGAD** 执行注册表修改。
- **User32.dll** 下的 **wsprintfA**：用于格式化字符串并将其输出到指定的缓冲区中。结合前面修改用户登录信息，可能病毒会盗号？然后假装用户打印一些东西。

发现他还有导出表：

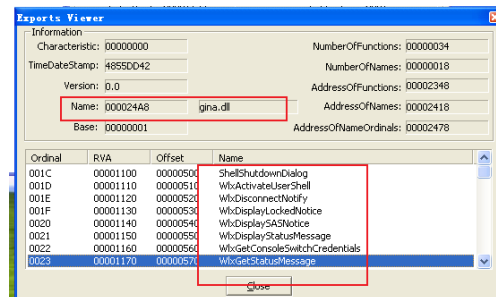


图 3.6: 导出表

图3.6 可以看到很多常见的 **Wlx** 即 **Windows** 操作系统中动态链接库（**DLL**）的命名前缀的函数，这些都是 **GINA** 拦截必须使用的：

- **WlxInitialize (GinaDLL)**: 初始化 **Gina DLL**。
- **WlxDisplaySASNotice (GinaDLL)**: 显示登录请求的安全注意事项。
- **WlxLoggedOutSAS (GinaDLL)** 处理用户注销的 **SAS** 操作。

不敢相信这个病毒会干什么了。。注销当前用户，替代我？细思极恐。再看看它的字符串：

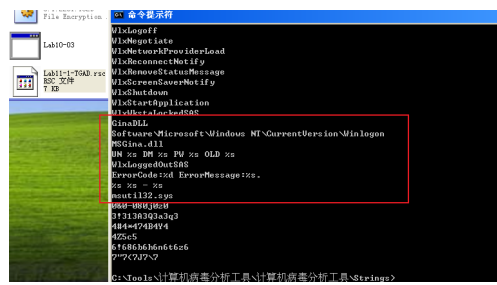


图 3.7: TGDA 字符串

图3.7使用 strings 看到一些有意思的字符串：

- Gina 与 Winlogon 注册表路径：这个和之前对 exe 的静态分析结果相同，会对 Gina 操作。
- UN %s DM %s PW %s OLD %s：结合其拦截 GINA 的行为，推测这个用来记录用户登录凭借的 log 日志。
- msutil32.sys：一个系统文件，暂时还不懂。

接下来就开始综合分析啦。

### 3.2.2 综合分析

当然先拍好快照。然后后面需要同时借用 IDA 静态分析加上一些 OllyDBG 等进行动态分析。

#### 1. Procmon 监视：

首先使用 Procmon 监视其行为，使用名为 Lab11-01 的过滤器：

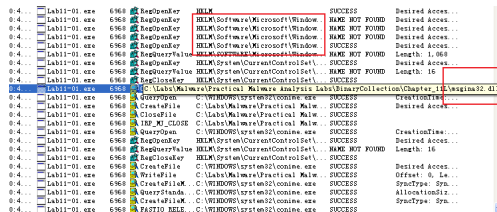


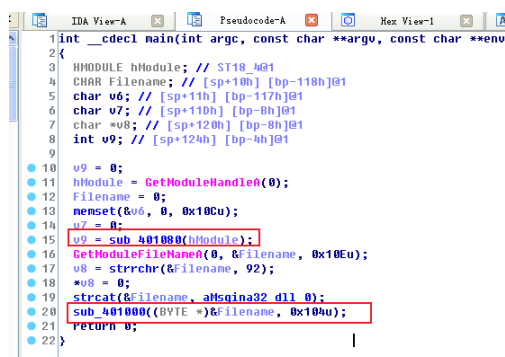
图 3.8: procomon 分析结果

图3.8看到病毒具有调用 CreateFile 创建文件 msgina32.dll，在启动目录位置。然后通过 Reg 相关操作将这个恶意 dll 插入到注册表项 SOFTWARE\Microsoft\windowsNT\CurrentVersion\Winlogon 中。这和我们静态分析结果相同，所以一旦系统重启，WinLogon 会重启该 dll。

并且经过刚才我们提取的 Lab11-01-TGAD 资源节和 msgina.dll 相比较 md5 码，发现两个东西其实同一个。于是我们将其重新命名为 msgina32.dll。

#### 2. IDA 静态分析：

首先分析 Lab11-01.exe，直接看 main 的反编译：

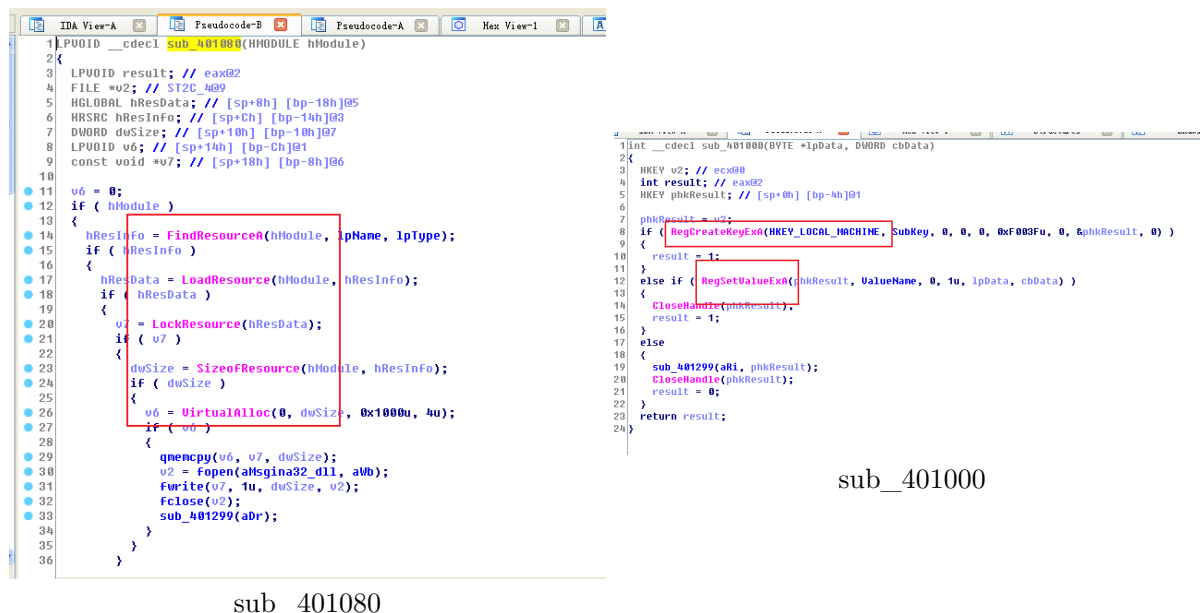


```

1 int __cdecl main(int argc, const char **argv, const char **envp)
2 {
3     HMODULE hModule; // ST18_401
4     CHAR filename; // [sp+10h] [bp-118h]@1
5     char u6; // [sp+11h] [bp-117h]@1
6     char u7; // [sp+110h] [bp-8h]@1
7     char u8; // [sp+120h] [bp-8h]@1
8     int u9; // [sp+124h] [bp-4h]@1
9
10    u9 = 0;
11    hModule = GetModuleHandle(0);
12    filename = 0;
13    memset(&u6, 0, 0x10Cu);
14    u7 = 0;
15    u9 = sub_401080(hModule);
16    GetModuleFileName(0, &filename, 0x10Eu);
17    u8 = strrchr(&filename, 92);
18    *u8 = 0;
19    strcat(&filename, "msgina32.dll");
20    sub_401000((BYTE *)&filename, 0x104u);
21    return 0;
22 }
    
```

图 3.9: exe 的 main

图3.9看到最主要的两个函数 sub\_401080 和 sub\_401000，前者在使用句柄，后者传入了文件名作为参数。查看这两个：



sub\_401080

sub\_401000

图 3.10: 两个函数

可以看到：

- sub\_401080：通过使用文件句柄，不断地通过 Find 和 Load 等查找和加载资源节 TGAD 到 msgina32.dll。
- sub\_401000：根据修改 Reg 注册表的行为，推测和 GINA 有关，然后就是利用文件名写入键值。最后使用 sub\_401299 来打印信息。

因此该 exe 文件就是 msgina32.dll 的小弟，一个加载器。接下来继续静态分析验证前面结果，将 msgina32.dll 载入：



```

1 BOOL __stdcall DLLMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved)
2 {
3     BOOL result; // eax@2
4     WCHAR Buffer; // [esp+0h] [bp-200h]@2
5
6     if ( fdwReason == 1 )
7     {
8         DisableThreadLibraryCalls(hinstDLL);
9         hModule = hinstDLL;
10        GetSystemDirectoryW(Buffer, 0x100u);
11        lstrcatW(Buffer, L"\\MSGina");
12        hLibModule = LoadLibraryW(Buffer);
13        result = hLibModule != 0;
14    }
15    else
16    {
17        if ( !fdwReason )
18        {
19            if ( hLibModule )
20            {
21                FreeLibrary(hLibModule);
22            }
23            result = 1;
24        }
25        return result;
26    }
27 }

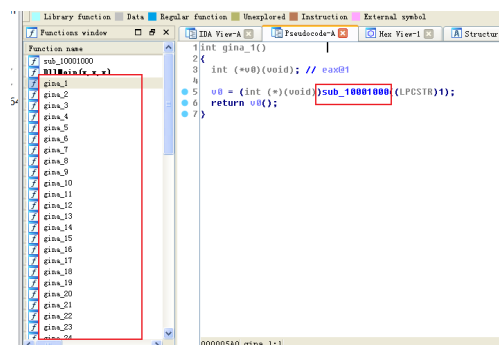
```

图 3.11: DLLMain

图3.11看到一些重要的代码段：

- 参数：hinstDLL 表示动态链接库的句柄，fdwReason 表示调用原因，lpvReserved 保留参数。
- 判断 fdwReason 是否为 1：1 代表了 DLL\_PROCESS\_ATTACH，即动态链接库被加载时的调用。
  - DisableThreadLibraryCalls 函数来禁用对线程的调用。然后将 hinstDLL 赋值给全局变量 hModule
  - 使用 GetSystemDirectoryW 函数获取系统目录，并将其存储在 Buffer 变量中。
  - 通过 lstrcatW 在 Buffer 后面追加”\_MSGina”。
  - 调用 LoadLibraryW 函数加载对应的库文件即”MSGina”，将返回的句柄存储在 hLibModule 变量中。这样是为了模拟系统正常调用 msgina.dll 库的过程。并将调用结果返回 result，即 True 或 False。
- 判断 fdwReason 是否为 0：0 代表了 DLL\_PROCESS\_DETACH。表示动态链接库被卸载时的调用。在这种情况下，代码会检查 hLibModule 是否为空，如果不为空则调用 FreeLibrary 函数释放对应的库文件。返回 True。

然后我们发现此时除了 DLLMain 之后，还有一些名为 gina\_1，gina\_2 等等多个同名的类似函数：



```

1 int gina_1(LPCSTR lpProcName)
2 {
3     int (*v0)(void); // eax@1
4
5     v0 = (int (*)(void))sub_10001000(LPCSTR)1;
6     return v0();
7 }

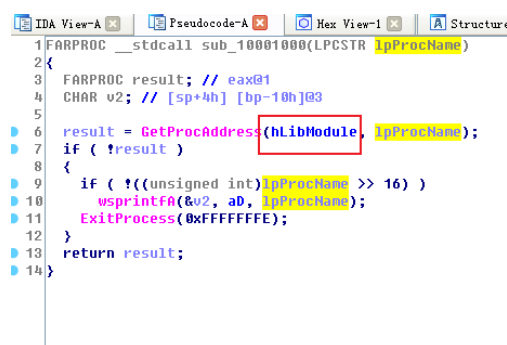
```

图 3.12: gina\_1

图3.12经过查看，这些函数都具有几乎相同的代码，都是去调用 sub\_10001000 位置的函数，目的是获取一个函数指针。它们唯一的区别就是传入的参数 LPCSTR lpProcName 不同，这应该是代表着不同的字符串。



因此我们进入到 sub\_10001000 位置：



```
1 FARPROC __stdcall sub_10001000(LPCSTR lpProcName)
2 {
3     FARPROC result; // eax@1
4     CHAR v2; // [sp+4h] [bp-10h]@3
5
6     result = GetProcAddress(hLibModule, lpProcName);
7     if ( !result )
8     {
9         if ( !((unsigned int)lpProcName >> 16) )
10            wsprintfA(&v2, aD, lpProcName);
11            ExitProcess(0xFFFFFFFF);
12        }
13    return result;
14 }
```

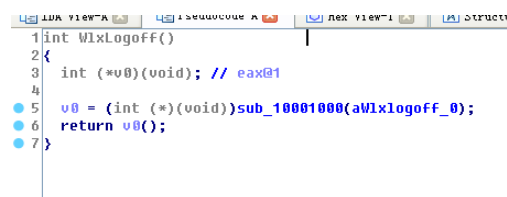
图 3.13: sub\_10001000

图3.13看到这个代码的一些地方值得注意：

- 它接受一个名为 lpProcName 的字符串参数，并返回一个 FARPROC 类型的值。前者应该是不同的函数名字符串。
- GetProcAddress 函数，该函数用于检索动态链接库中导出函数的地址。它的第一个参数是 hLibModule，这是一个全局变量，已经指向了被加载的 msgina.dll 的句柄。第二个参数是 lpProcName，表示要检索的函数名。
- 如果 GetProcAddress 返回的结果是空（即 0），则代码会进行进一步的检查。它首先使用位运算检查 lpProcName 的高 16 位是否为 0，如果是，则调用 wsprintfA 函数格式化字符串，将错误信息存储在 v2 变量中。然后调用 ExitProcess 函数终止当前进程。解析失败。

因此该函数通过传入的参数 LPCSTR 的不同中的值来作为函数地址偏移，来选取 msgina.dll 中的不同函数。GetProcAddress 获取到了函数地址。

如果进一步思考和查看：



```
1 int WlxLogoff()
2 {
3     int (*v0)(void); // eax@1
4
5     v0 = (int (*)(void))sub_10001000(aWlxLogoff_0);
6     return v0();
7 }
```

图 3.14: 大部分跳转

图3.14可以看到，很明显通过上述地址偏移找到的位置，直接跳转到的应该就是 GINA 需要的那些导出函数，包括 Wlx 前缀的那些导出函数。

但是有一个特例即 WlxLoggedOutSAS 函数，它本来的作用是处理用户注销时的安全身份验证的动态链接库文件。

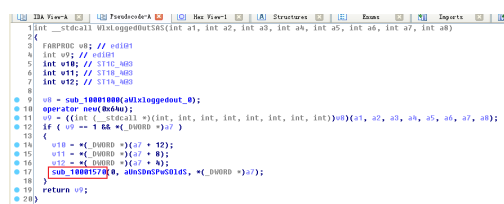


图 3.15: 特例: WlxLoggedOutSAS

图3.15看到这个 WlxLoggedOutSAS 并不是跳到了直接的对应 GINA 函数，而是转而去看了 sub\_10001570，看看这个：

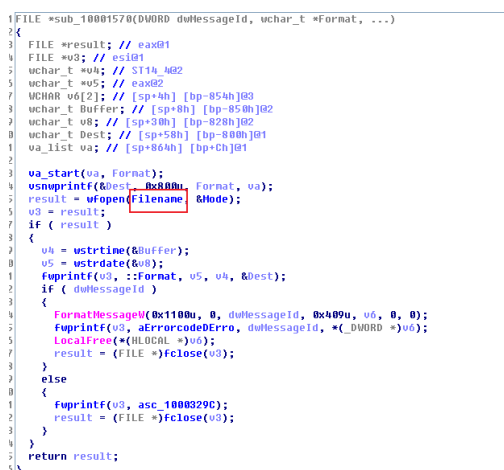


图 3.16: sub\_10001570

图3.16看到一些值得注意的地方：

- vsnwprintf(&Dest, 0x800u, Format, va): 这个调用使用可变参数列表 va 和格式化字符串 Format，将格式化后的字符串写入 Dest 中。这个函数通常用于将格式化后的字符串写入一个缓冲区中。
- wlopen(Filename, Mode): 这个调用创建并打开一个文件，‘Filename’是文件名，‘Mode’是文件打开模式。详细查看后,这里的 Filename 对应了之前静态分析发现的字符串 msutil32.sys，推测是向其中写入一些文件内容。
- fwprintf(v3, ::Format, v5, v4, &Dest): 这个调用使用 v3 指向的文件流，按照::Format‘指定的格式，将 v5、v4 和 Dest 的内容写入文件。其中 v5 还有 v4 对应的都是日期和时间的信息，而 Dest 储存的是格式化的消息字符串。
- FormatMessageW: 用来格式化调用获取错误信息的文本描述，对应的描述为“ErrorCode:%d ErrorMessage:%s”

综上所述,整个函数看起来是一个用来记录日志的函数。并把格式化的日志信息记录在 msutil32.sys 文件中。其中由于 Winlogon 位于 System32 目录，因此这个文件也是创建在这里，并且运行在 Winlogon 进程中。如果没有提供消息 ID，它可能会写入一个默认的日志消息。

最初我们推测字符串时候，想的是或许 msutil32.sys 根据其后缀表达，有可能是一个系统文件。但是由于它是在 WlxLoggedOutSAS 这个函数调用时候使用，并用来记录日志，或许还有别的可能。

因为 WlxLoggedOutSAS 的调用会默认遍历所有存在的用户名，这样才能实际地进行销毁和注销。因此这个过程会暴露所有的用户信息及登陆凭证。故 sub\_10005070 可能会在这个过程中窃取登陆的用户名和密码，然后再把这些信息写在 System32 下的 msutil32.sys 文件中，让我们误以为他只是是一个简单的系统文件，借此隐藏自己。

### 3.2.3 实验问题

在差不多分析完成后，我们可以来得出结论：Lab11-01.exe 其中包含的一个资源节，而可执行文件本身不过是一个对其资源节的 DLL 文件进行加载的安装器。而它实际上是是为了拦截 GINA，即用户登录认证时候的关键动态链接库。通过修改注册表，实现系统重启后在用户注销系统过程中盗取信息的恶意的目的。盗号专家！

现在我们可以来回答问题了。

1. Q1: 这个恶意代码向磁盘释放了什么？

回答：恶意代码通过将自己自带的名为 TGAD 的资源节取出一个名为 msgina32.dll 的文件，然后存储在磁盘中。

2. Q2: 这个恶意代码如何进行驻留？

回答：为了实现驻留或者说是自启动，exe 通过创建新的在 HKLM\SOFTWARE\Microsoft\WindowsNT\CurrentVersion\Winlogon\GINADLL 的注册表项，实现每次启动登陆时候，让 msgina32.dll 被当作 GINA DLL 安装。

3. Q3: 这个恶意代码如何窃取用户登录凭证？

回答：它会把自己装成一个 GINA 下的正常动态链接库文件 msgina32.dll，然后拦截所有提交到系统认证的用户登录凭证。

4. Q4: 这个恶意代码对窃取的证书做了什么处理？

回答：msgina32.dll 会将盗窃的证书实际上就是用户名和密码保存在 System32 下的 msutil32.sys 文件中。其中用户名、域名称、密码、时间戳都将被记录到该文件。并加入后缀为 sys 是为了伪装自己，不引人注目。

5. Q5: 如何在你的测试环境让这个恶意代码获得用户登录凭证？

回答：最后在这里补充对其动态执行一下。注意一定要用管理员权限运行这个病毒！



图 3.17: 注销再登陆

首先拍好快照，双击运行病毒。然后如3.17注销重启后再登陆：

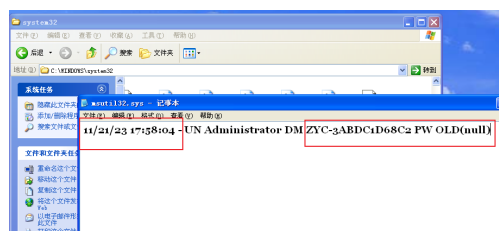


图 3.18: 结果

图3.18结果和我们预想的一致，带有详细的时间信息以及登录的详细凭证。

因此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，所以注销后再登录系统，就能看到记录到日志文件的登录凭证。如上面我们的结果一样呢。

### 3.3 Lab11-2

分析恶意代码 Lab11-02.dll。假设一个名为 Lab11-02.ini 的可疑文件与这个恶意代码一同被发现。

#### 3.3.1 静态分析

首先进行一些简单的静态分析，先使用 PEiD 查看其加壳情况，导入表和导出表等：

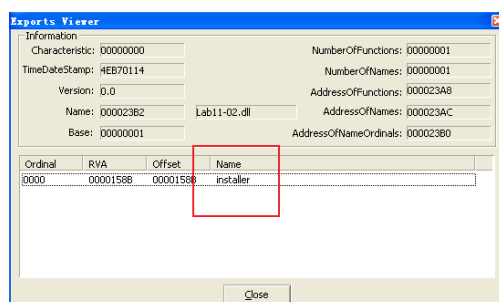
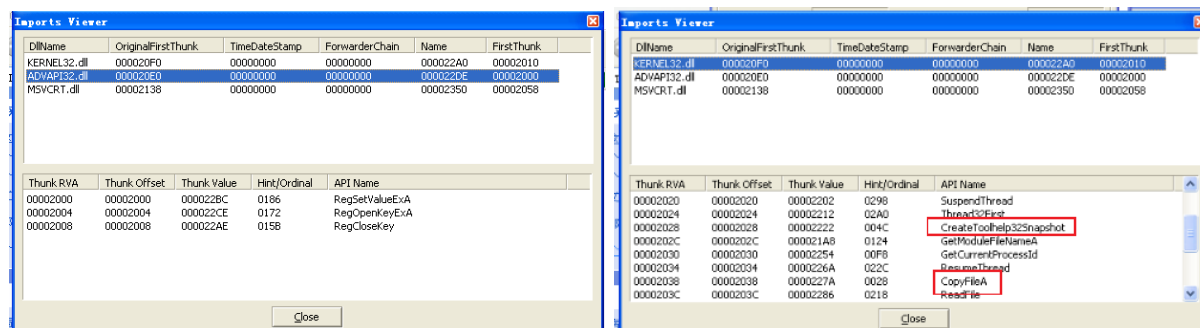


图 3.19: 导出表

图3.19知道这个病毒作为 DLL 文件，唯一的导出函数是 installer，提示他可能进行一些自安装的行为。另外他也没有加壳，继续查看导入表：



导入表 1

导入表 2

图 3.20: 查看导入表

图3.20看到了几个有趣的导入函数：

- ADVAPI32.dll: 其中包含了再次出现的 **RegOpen** 和 **SetValue** 等实现注册表操作的导入函数, 结合之前的 **installer**, 继续推测其具有驻留的行为。
- Kernel32.dll: 其中包含了非常可疑的 **CreateToolhelp32Snapshot**, 用于创建一个进程和模块的快照。它可以用来获取系统中当前运行的进程和模块的信息, 比如进程 ID、父进程 ID、模块路径等。这个函数通常与其他函数一起使用, 来遍历和获取进程和模块的详细信息。另外还有试图对文件系统进行改变的 **CopyFileA**。

然后使用 Strings 查看其字符串:

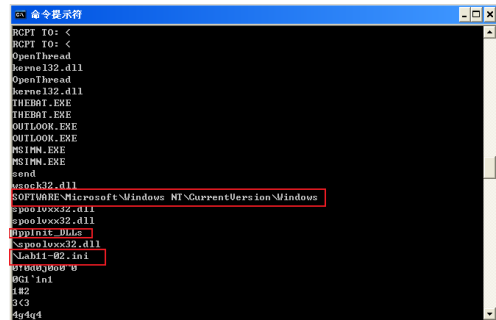


图 3.21: Strings

图3.21 图3.21可以看到一些有趣并且很有指向性的符号串:

- SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows: 注册表路径, 该路径包含了与 Windows 版本和配置相关的信息, 比如安装路径、产品密钥、安全设置等。结合之前发现的注册表项操作的函数, 推测其对这个路径下的注册表有操作。
- AppInit\_DLLs: 这是 Windows 操作系统中的一个注册表键, 用于指定在用户登录时自动加载的动态链接库文件。结合上面的注册表路径, 推断恶意代码可能利用这个注册表键来安装自己。如果真在这里成功了, 那么它会被加载到所有装载了 User32.dll 的进程中。
- Lab11-02.ini: 这是配套的配置文件, 意味着病毒使用了它。
- OUTLOOK.EXE, MSIMN.EXE 以及 THEBAT.EXE: 这个非常有意思, 这三个分别是 Microsoft Outlook, Microsoft Outlook Express 以及 The Bat! 的客户端可执行文件, 因此推测病毒可能会有邮件操作等行为。
- send 和 wsck32.dll: 结合上面的邮件信息, 推测可能是有网络行为, 并进行邮件的发送。同样的还有 “RCPT TO:” 作为 SMTP 命令来创建电子邮件接收人。

最后看一下配置文件 Lab11-2.ini:

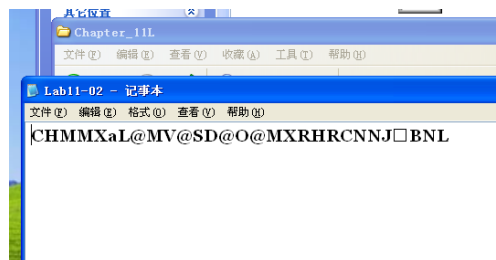


图 3.22: 配置文件

图3.22看到文件似乎经历了混淆处理，暂时无法解密。

### 3.3.2 综合分析

为了进一步探究病毒行为并解密配置文件，接下来深度静态和动态分析：

### 1. Procomon 动态执行:

接下来结合之前 Lab3 的经验，为了运行这些 DLL，我们必须使用 rundll32.exe，因此设置一个 rundll32.exe 的过滤器：

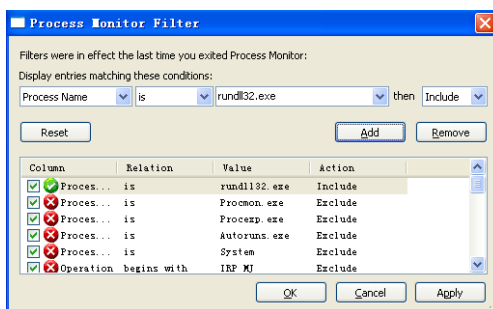


图 3.23: 设置 Procmon 过滤器

然后通过命令 `rundll32.exe Lab11-02.dll,instaler` 执行安装，观测到：

[illegible]

图 3.24: Procmon 观测结果

图3.24看到了:

- CreateFileA: 创建了一个名为 spoolvxx32.dll 的文件在 System32 下! 推测可能是恶意文件。
- Reg: 后面通过注册表操作, 将 spoolvxx32.dll 添加到了 AppInit\_DLLs 列表中。
- 在 System32 下访问 Lab11-02.ini: 之前没注意到, 因此我们得把在这个配置挪到 System32 下。



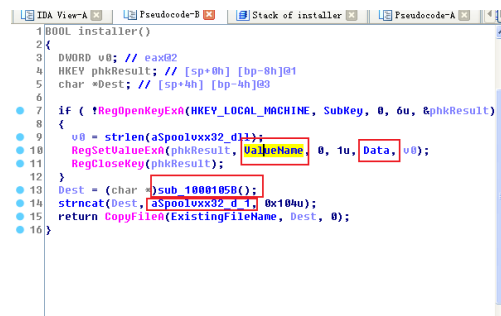


图 3.25: installer

这些都和我们静态分析的结果类似，其中 spoolvxx32.dll 我之前在使用 strings 分析字符时候也发现了，不过没想过它的含义。这些想要解密 ini 文件还远远不够，接下来使用 IDA 进行深度静态分析。

## 2. IDA 深度分析：

首先进来查看 installer 函数：图3.25可以看到以下值得注意的地方：

- RegOpenKeyExA: 打开注册表键，其中 HKEY 位置加上 SubKey 拼接起来是 SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows。和我们静态分析结果相同，如果成功打开了这个键，接下来的操作将会在这个键下进行。
- RegSetValueExA: 这里面的 ValueName 经过查看就是 AppInit\_DLLs，而 Data 就是 spoolvxx32.dll。其下设置了这个 DLL 文件的路径。然后关闭了注册表键。由此实现了病毒驻留的持久化，每次启动都会被加载。
- sub\_1000105B 的函数: 并将其返回的路径与”\spoolvxx32.dll” 拼接起来。
- CopyFileA 函数: 将文件从 ExistingFileName 复制到 Destination，并返回结果。

然后去看眼 sub\_1000105B:



图 3.26: sub\_1000105B

图3.26看到他只是 GetSystemDirectoryA 函数获取系统目录，并返回路径地址。现在使用交叉引用图查看 installer 具体调用的情况：

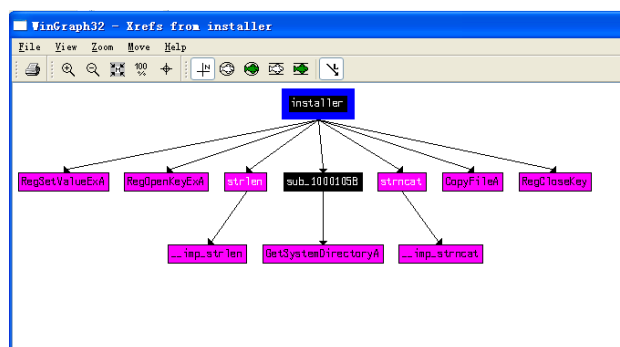


图 3.27: installer 交叉引用

图3.27显示的与我们刚才分析的也一致。

接下来分析 DLLMAIN:

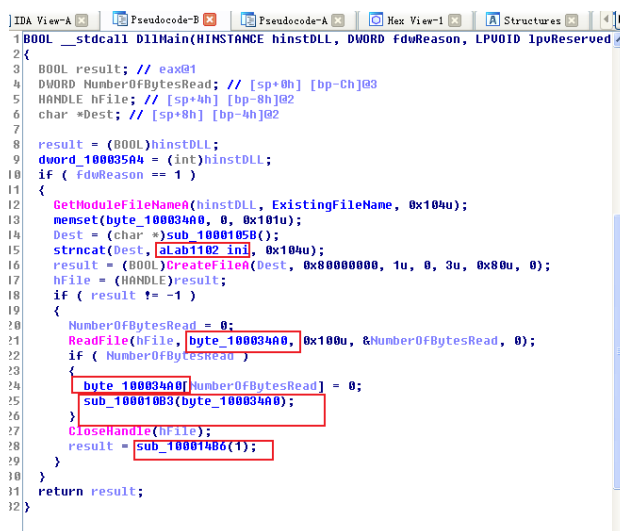


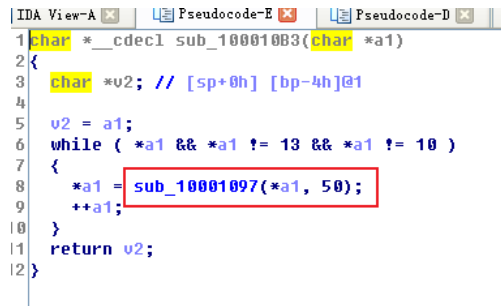
图 3.28: DLLMAIN

图??看到一些有意思的地方:

- fdwReason 参数表示 DLL 加载或卸载原因，这一点和之前的 Lab11-01 一样。具体而言，如果检查病毒已经在 DLL\_PROCESS\_ATTACH 即 fdwReason 为 1 的状态运行，如果不在，函数直接返回。
- sub\_1000105B: 之前我们分析过了，它会调用 GetSystemDirectoryA 函数获取系统目录，这里也一样，会返回配置文件的存放路径。
- 然后使用 strncat 在这个路径上拼接“\\Lab11-02.ini” 即 aLab1102\_ini 的值，构造出配置文件的完整路径。
- 调用 CreateFileA 函数打开配置文件，如果成功打开文件，则继续读取文件内容。如果文件读取成功，将文件内容传递给 sub\_100010B3 函数进行处理。
- 最后调用了一个名为 sub\_100014B6 的函数，传入参数 1，可能是用于通知其他部分 DLL 加载已经完成。这里先不深究。



由此可知，关键部分函数就是 sub\_100010B3，它可能隐藏着反混淆对 ini 进行解密的关键：



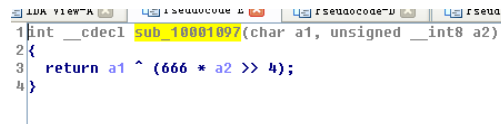
```

1 char * _cdecl sub_100010B3(char *a1)
2 {
3     char *u2; // [sp+0h] [bp-4h]@1
4
5     u2 = a1;
6     while ( *a1 && *a1 != 13 && *a1 != 10 )
7     {
8         *a1 = sub_10001097(*a1, 50);
9         ++a1;
10    }
11    return u2;
12 }

```

图 3.29: sub\_100010B3

图3.29不出我们所料，此时正在利用一些字符串和 13 以及 10，还有 50 等对应的 ASCII 码字符进行类似的解密操作，同样他还调用了 sub\_10001097：



```

1 int _cdecl sub_10001097(char a1, unsigned __int8 a2)
2 {
3     return a1 ^ (666 * a2 >> 4);
4 }

```

图 3.30: sub\_10001097

图3.29看到也在解密复杂的东西。到此我们就有目标了，由于我们之前 Procmon 观察到它会试图打开 system32 下的 ini，我们也把配置文件挪进去了。而且我们还知道关键的解密代码位置了，因此直接可以进行动态分析，来到此处，一步一步看到解密的结果啦！

### 3. OllyDBG 动态分析：

由于 IDA 和 OllyDBG 存在加载位置的不一致性，通过换算我们得出，我们需要在函数 sub\_100010B3 对应的 0x003D10B3，找到 0x003D16CA 处对 0x003D10B3 调用结束后打下断点。直接运行到此处：

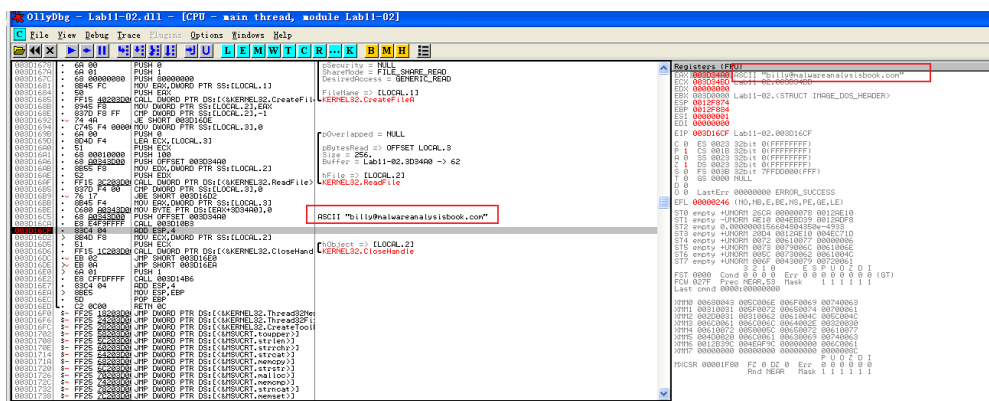


图 3.31: 解密结果

图3.31可以看到在最右侧的寄存器值中显示了解密的结果：billy@malwareanalysisbook.com。这里的 billy 明显是个名字，不过鉴于本书的作者 Michael Sikorski 和 Andrew Honig 不是 Billy，有点奇怪 hhh。

但可以看出这是一个邮箱地址, 结合之前的 send 函数推测其可能要向这个邮箱发送一些东西, 接下来回到 IDA 中进一步分析:

#### 4. IDA 深度分析 2

之前我们提到 sub\_100014B6 的函数在 DLLMAIN 的结尾, 根据解密的结果 result 进行进一步的行为, 因此我们接下来需要深入分析 sub\_100014B6 以找到最终的目的。



```
int cdecl sub_100014B6(int a1)
{
    u1 = 0; // eax
    size_t u2; // eax
    size_t u3; // eax
    size_t u4; // eax
    u1 = 0; // [sp+0h] [bp-4h]
    Buf1 = u1;
    if ( a1 )
    {
        sub_10001075(a1, (int)&Buf1);
        Buf1 = (u1 + 1) * sub_10001104((char *)Buf1);
        if ( Buf1 )
        {
            sub_1000102D(Buf1);
            u2 = strlen(aThebat_exe);
            if ( !memcmp(Buf1, aThebat_exe, u2) )
            {
                if ( u3 = strlen(aOutlook_exe), !memcmp(Buf1, aOutlook_exe, u3) )
                {
                    if ( u4 = strlen(aSim_exe), !memcmp(Buf1, aSim_exe, u4) )
                    {
                        sub_100013BD();
                        sub_100012A3(ModuleName, aSend, (int)sub_1000113D, (int)&word_10003404);
                        sub_10001499();
                    }
                }
            }
        }
    }
}
```

图 3.32: sub\_100014B6

图3.32可以看到一些地方:

- sub\_100014B6 调用了诸多函数, 具体而言:
  - sub\_10001075
  - sub\_10001104
  - sub\_1000102D
  - sub\_100013BD
  - sub\_100012A3
  - sub\_1000113D
  - sub\_10001499
- 除此之外, 出现了一些字符串。比如 aThebat\_exe, aOutlook\_exe 等, 这些通过验证都一次代表了静态分析字符串时候发现的 OUTLOOK.EXE 等客户端邮件的可执行文件。然后 aSend 对应 send。

因此我们想要得出更多结论, 必须得依次分析上述函数。而其中还存在着复杂的调用关系链, 总的来说, 最后发现一共调用了 11 个函数, 其中有两个作用完全相同:

```

1 int sub_10001075(HMODULE hModule, int a2)
2 {
3     DWORD result; // eax@1
4
5     result = GetModuleFileNameA(hModule, Filename, 0x100u);
6     *(DWORD *)a2 = Filename;
7     return result;
8 }
    
```

((a)) sub\_10001075

```

char * __cdecl sub_10001104(char *Str)
{
    char *result; // eax@2
    char *u2; // [sp+0h] [bp-4h]@1

    u2 = strrchr(Str, 92) + 1;
    if ( strlen(u2) )
        result = u2;
    else
        result = 0;
    return result;
}
    
```

((b)) sub\_10001104

```

1 int __cdecl sub_1000102D(int *a1)
2 {
3     int *result; // eax@1
4
5     while ( 1 )
6     {
7         result = a1;
8         if ( !*( _BYTE *)a1 )
9             break;
10        *(_BYTE *)a1 = toupper(*(_BYTE *)a1);
11        a1 = (int *)((char *)a1 + 1);
12    }
13    return result;
14}
    
```

((c)) sub\_1000102D

```

1 int sub_100013BD()
2 {
3     DWORD u0; // ST04_4@1
4
5     u0 = GetCurrentProcessId();
6     return sub_100012FE(u0);
7 }
    
```

((d)) sub\_100013BD

```

1 int __cdecl sub_100012FE(int a1)
2 {
3     HANDLE hThread; // [sp+0h] [bp-0h]@1
4     DWORD u2; // [sp+4h] [bp-4h]@2
5     HANDLE hSnapshot; // [sp+8h] [bp-8h]@3
6     int u3; // [sp+Ch] [bp-4h]@4
7
8     u3 = sub_10001104(Str, 92);
9     if ( u3 )
10        u2 = CreateToolhelp32Snapshot(TH32CS_SNAPALL, 0);
11    else
12        u2 = 0;
13    if ( !u2 )
14        return 0;
15    if ( !Thread32First(u2, &hThread) )
16        return 0;
17    do
18    {
19        if ( !Thread32Next(u2, &hThread, 0, &u3) )
20            break;
21        if ( hThread == GetCurrentThread() )
22            continue;
23        while ( !Thread32Next(u2, &hThread, 0, &u3) )
24            continue;
25    } while ( 1 );
26    return u3;
27}
    
```

((e)) sub\_100012FE

```

1 int __cdecl sub_10001000(LPCSTR lpFileName, LPCSTR lpProcName)
2 {
3     FARPROC result; // eax@2
4     MODULE hModule; // [sp+0h] [bp-0h]@1
5
6     hModule = LoadLibraryA(lpFileName);
7     if ( hModule )
8         result = GetProcAddress(hModule, lpProcName);
9     else
10        result = 0;
11    return result;
12}
    
```

((f)) sub\_10001000

```

1 int __cdecl sub_100012A3(LPCSTR lpModulePath, LPCSTR lpProcName, int a3, int a4)
2 {
3     MODULE hModule; // eax@1
4     MODULE hModule; // [sp+0h] [bp-0h]@2
5
6     result = GetModuleHandleA(lpModulePath);
7     hModule = result;
8     if ( result )
9     {
10        result = LoadLibraryA(lpModulePath);
11        hModule = result;
12    }
13    if ( hModule )
14    {
15        result = GetProcAddress(hModule, lpProcName);
16        if ( result )
17            result = (DWORD)sub_100012FE(result, a3, a4);
18    }
19    return result;
20}
    
```

((g)) sub\_100012A3

```

1 int __cdecl sub_10001203(LPCSTR lpModulePath, int a2, int a3)
2 {
3     void *u2; // eax@1
4     int result; // eax@2
5     DWORD u3; // [sp+0h] [bp-0h]@3
6     void *u4; // [sp+4h] [bp-4h]@4
7     int u5; // [sp+8h] [bp-8h]@5
8
9     u5 = a2 - (DWORD)lpModulePath - 5;
10    u4 = VirtualProtect(lpModulePath, u5, 0x40, 0x40);
11    if ( u4 )
12        u3 = *u4;
13    u2 = u3;
14    u4 = u3;
15    u5 = u3;
16    u4 = u3;
17    u5 = u3;
18    u4 = u3;
19    u5 = u3;
20    u4 = u3;
21    u5 = u3;
22    u4 = u3;
23    u5 = u3;
24    u4 = u3;
25    u5 = u3;
26    u4 = u3;
27    u5 = u3;
28    u4 = u3;
29    u5 = u3;
30    u4 = u3;
31    u5 = u3;
32    u4 = u3;
33    u5 = u3;
34    u4 = u3;
35    u5 = u3;
36    u4 = u3;
37    u5 = u3;
38    u4 = u3;
39    u5 = u3;
40    u4 = u3;
41    u5 = u3;
42    u4 = u3;
43    u5 = u3;
44    u4 = u3;
45    u5 = u3;
46    u4 = u3;
47    u5 = u3;
48    u4 = u3;
49    u5 = u3;
50    u4 = u3;
51    u5 = u3;
52    u4 = u3;
53    u5 = u3;
54    u4 = u3;
55    u5 = u3;
56    u4 = u3;
57    u5 = u3;
58    u4 = u3;
59    u5 = u3;
60    u4 = u3;
61    u5 = u3;
62    u4 = u3;
63    u5 = u3;
64    u4 = u3;
65    u5 = u3;
66    u4 = u3;
67    u5 = u3;
68    u4 = u3;
69    u5 = u3;
70    u4 = u3;
71    u5 = u3;
72    u4 = u3;
73    u5 = u3;
74    u4 = u3;
75    u5 = u3;
76    u4 = u3;
77    u5 = u3;
78    u4 = u3;
79    u5 = u3;
80    u4 = u3;
81    u5 = u3;
82    u4 = u3;
83    u5 = u3;
84    u4 = u3;
85    u5 = u3;
86    u4 = u3;
87    u5 = u3;
88    u4 = u3;
89    u5 = u3;
90    u4 = u3;
91    u5 = u3;
92    u4 = u3;
93    u5 = u3;
94    u4 = u3;
95    u5 = u3;
96    u4 = u3;
97    u5 = u3;
98    u4 = u3;
99    u5 = u3;
100   u4 = u3;
101   u5 = u3;
102   u4 = u3;
103   u5 = u3;
104   u4 = u3;
105   u5 = u3;
106   u4 = u3;
107   u5 = u3;
108   u4 = u3;
109   u5 = u3;
110   u4 = u3;
111   u5 = u3;
112   u4 = u3;
113   u5 = u3;
114   u4 = u3;
115   u5 = u3;
116   u4 = u3;
117   u5 = u3;
118   u4 = u3;
119   u5 = u3;
120   u4 = u3;
121   u5 = u3;
122   u4 = u3;
123   u5 = u3;
124   u4 = u3;
125   u5 = u3;
126   u4 = u3;
127   u5 = u3;
128   u4 = u3;
129   u5 = u3;
130   u4 = u3;
131   u5 = u3;
132   u4 = u3;
133   u5 = u3;
134   u4 = u3;
135   u5 = u3;
136   u4 = u3;
137   u5 = u3;
138   u4 = u3;
139   u5 = u3;
140   u4 = u3;
141   u5 = u3;
142   u4 = u3;
143   u5 = u3;
144   u4 = u3;
145   u5 = u3;
146   u4 = u3;
147   u5 = u3;
148   u4 = u3;
149   u5 = u3;
150   u4 = u3;
151   u5 = u3;
152   u4 = u3;
153   u5 = u3;
154   u4 = u3;
155   u5 = u3;
156   u4 = u3;
157   u5 = u3;
158   u4 = u3;
159   u5 = u3;
160   u4 = u3;
161   u5 = u3;
162   u4 = u3;
163   u5 = u3;
164   u4 = u3;
165   u5 = u3;
166   u4 = u3;
167   u5 = u3;
168   u4 = u3;
169   u5 = u3;
170   u4 = u3;
171   u5 = u3;
172   u4 = u3;
173   u5 = u3;
174   u4 = u3;
175   u5 = u3;
176   u4 = u3;
177   u5 = u3;
178   u4 = u3;
179   u5 = u3;
180   u4 = u3;
181   u5 = u3;
182   u4 = u3;
183   u5 = u3;
184   u4 = u3;
185   u5 = u3;
186   u4 = u3;
187   u5 = u3;
188   u4 = u3;
189   u5 = u3;
190   u4 = u3;
191   u5 = u3;
192   u4 = u3;
193   u5 = u3;
194   u4 = u3;
195   u5 = u3;
196   u4 = u3;
197   u5 = u3;
198   u4 = u3;
199   u5 = u3;
200   u4 = u3;
201   u5 = u3;
202   u4 = u3;
203   u5 = u3;
204   u4 = u3;
205   u5 = u3;
206   u4 = u3;
207   u5 = u3;
208   u4 = u3;
209   u5 = u3;
210   u4 = u3;
211   u5 = u3;
212   u4 = u3;
213   u5 = u3;
214   u4 = u3;
215   u5 = u3;
216   u4 = u3;
217   u5 = u3;
218   u4 = u3;
219   u5 = u3;
220   u4 = u3;
221   u5 = u3;
222   u4 = u3;
223   u5 = u3;
224   u4 = u3;
225   u5 = u3;
226   u4 = u3;
227   u5 = u3;
228   u4 = u3;
229   u5 = u3;
230   u4 = u3;
231   u5 = u3;
232   u4 = u3;
233   u5 = u3;
234   u4 = u3;
235   u5 = u3;
236   u4 = u3;
237   u5 = u3;
238   u4 = u3;
239   u5 = u3;
240   u4 = u3;
241   u5 = u3;
242   u4 = u3;
243   u5 = u3;
244   u4 = u3;
245   u5 = u3;
246   u4 = u3;
247   u5 = u3;
248   u4 = u3;
249   u5 = u3;
250   u4 = u3;
251   u5 = u3;
252   u4 = u3;
253   u5 = u3;
254   u4 = u3;
255   u5 = u3;
256   u4 = u3;
257   u5 = u3;
258   u4 = u3;
259   u5 = u3;
260   u4 = u3;
261   u5 = u3;
262   u4 = u3;
263   u5 = u3;
264   u4 = u3;
265   u5 = u3;
266   u4 = u3;
267   u5 = u3;
268   u4 = u3;
269   u5 = u3;
270   u4 = u3;
271   u5 = u3;
272   u4 = u3;
273   u5 = u3;
274   u4 = u3;
275   u5 = u3;
276   u4 = u3;
277   u5 = u3;
278   u4 = u3;
279   u5 = u3;
280   u4 = u3;
281   u5 = u3;
282   u4 = u3;
283   u5 = u3;
284   u4 = u3;
285   u5 = u3;
286   u4 = u3;
287   u5 = u3;
288   u4 = u3;
289   u5 = u3;
290   u4 = u3;
291   u5 = u3;
292   u4 = u3;
293   u5 = u3;
294   u4 = u3;
295   u5 = u3;
296   u4 = u3;
297   u5 = u3;
298   u4 = u3;
299   u5 = u3;
300   u4 = u3;
301   u5 = u3;
302   u4 = u3;
303   u5 = u3;
304   u4 = u3;
305   u5 = u3;
306   u4 = u3;
307   u5 = u3;
308   u4 = u3;
309   u5 = u3;
310   u4 = u3;
311   u5 = u3;
312   u4 = u3;
313   u5 = u3;
314   u4 = u3;
315   u5 = u3;
316   u4 = u3;
317   u5 = u3;
318   u4 = u3;
319   u5 = u3;
320   u4 = u3;
321   u5 = u3;
322   u4 = u3;
323   u5 = u3;
324   u4 = u3;
325   u5 = u3;
326   u4 = u3;
327   u5 = u3;
328   u4 = u3;
329   u5 = u3;
330   u4 = u3;
331   u5 = u3;
332   u4 = u3;
333   u5 = u3;
334   u4 = u3;
335   u5 = u3;
336   u4 = u3;
337   u5 = u3;
338   u4 = u3;
339   u5 = u3;
340   u4 = u3;
341   u5 = u3;
342   u4 = u3;
343   u5 = u3;
344   u4 = u3;
345   u5 = u3;
346   u4 = u3;
347   u5 = u3;
348   u4 = u3;
349   u5 = u3;
350   u4 = u3;
351   u5 = u3;
352   u4 = u3;
353   u5 = u3;
354   u4 = u3;
355   u5 = u3;
356   u4 = u3;
357   u5 = u3;
358   u4 = u3;
359   u5 = u3;
360   u4 = u3;
361   u5 = u3;
362   u4 = u3;
363   u5 = u3;
364   u4 = u3;
365   u5 = u3;
366   u4 = u3;
367   u5 = u3;
368   u4 = u3;
369   u5 = u3;
370   u4 = u3;
371   u5 = u3;
372   u4 = u3;
373   u5 = u3;
374   u4 = u3;
375   u5 = u3;
376   u4 = u3;
377   u5 = u3;
378   u4 = u3;
379   u5 = u3;
380   u4 = u3;
381   u5 = u3;
382   u4 = u3;
383   u5 = u3;
384   u4 = u3;
385   u5 = u3;
386   u4 = u3;
387   u5 = u3;
388   u4 = u3;
389   u5 = u3;
390   u4 = u3;
391   u5 = u3;
392   u4 = u3;
393   u5 = u3;
394   u4 = u3;
395   u5 = u3;
396   u4 = u3;
397   u5 = u3;
398   u4 = u3;
399   u5 = u3;
400   u4 = u3;
401   u5 = u3;
402   u4 = u3;
403   u5 = u3;
404   u4 = u3;
405   u5 = u3;
406   u4 = u3;
407   u5 = u3;
408   u4 = u3;
409   u5 = u3;
410   u4 = u3;
411   u5 = u3;
412   u4 = u3;
413   u5 = u3;
414   u4 = u3;
415   u5 = u3;
416   u4 = u3;
417   u5 = u3;
418   u4 = u3;
419   u5 = u3;
420   u4 = u3;
421   u5 = u3;
422   u4 = u3;
423   u5 = u3;
424   u4 = u3;
425   u5 = u3;
426   u4 = u3;
427   u5 = u3;
428   u4 = u3;
429   u5 = u3;
430   u4 = u3;
431   u5 = u3;
432   u4 = u3;
433   u5 = u3;
434   u4 = u3;
435   u5 = u3;
436   u4 = u3;
437   u5 = u3;
438   u4 = u3;
439   u5 = u3;
440   u4 = u3;
441   u5 = u3;
442   u4 = u3;
443   u5 = u3;
444   u4 = u3;
445   u5 = u3;
446   u4 = u3;
447   u5 = u3;
448   u4 = u3;
449   u5 = u3;
450   u4 = u3;
451   u5 = u3;
452   u4 = u3;
453   u5 = u3;
454   u4 = u3;
455   u5 = u3;
456   u4 = u3;
457   u5 = u3;
458   u4 = u3;
459   u5 = u3;
460   u4 = u3;
461   u5 = u3;
462   u4 = u3;
463   u5 = u3;
464   u4 = u3;
465   u5 = u3;
466   u4 = u3;
467   u5 = u3;
468   u4 = u3;
469   u5 = u3;
470   u4 = u3;
471   u5 = u3;
472   u4 = u3;
473   u5 = u3;
474   u4 = u3;
475   u5 = u3;
476   u4 = u3;
477   u5 = u3;
478   u4 = u3;
479   u5 = u3;
480   u4 = u3;
481   u5 = u3;
482   u4 = u3;
483   u5 = u3;
484   u4 = u3;
485   u5 = u3;
486   u4 = u3;
487   u5 = u3;
488   u4 = u3;
489   u5 = u3;
490   u4 = u3;
491   u5 = u3;
492   u4 = u3;
493   u5 = u3;
494   u4 = u3;
495   u5 = u3;
496   u4 = u3;
497   u5 = u3;
498   u4 = u3;
499   u5 = u3;
500   u4 = u3;
501   u5 = u3;
502   u4 = u3;
503   u5 = u3;
504   u4 = u3;
505   u5 = u3;
506   u4 = u3;
507   u5 = u3;
508   u4 = u3;
509   u5 = u3;
510   u4 = u3;
511   u5 = u3;
512   u4 = u3;
513   u5 = u3;
514   u4 = u3;
515   u5 = u3;
516   u4 = u3;
517   u5 = u3;
518   u4 = u3;
519   u5 = u3;
520   u4 = u3;
521   u5 = u3;
522   u4 = u3;
523   u5 = u3;
524   u4 = u3;
525   u5 = u3;
526   u4 = u3;
527   u5 = u3;
528   u4 = u3;
529   u5 = u3;
530   u4 = u3;
531   u5 = u3;
532   u4 = u3;
533   u5 = u3;
534   u4 = u3;
535   u5 = u3;
536   u4 = u3;
537   u5 = u3;
538   u4 = u3;
539   u5 = u3;
540   u4 = u3;
541   u5 = u3;
542   u4 = u3;
543   u5 = u3;
544   u4 = u3;
545   u5 = u3;
546   u4 = u3;
547   u5 = u3;
548   u4 = u3;
549   u5 = u3;
550   u4 = u3;
551   u5 = u3;
552   u4 = u3;
553   u5 = u3;
554   u4 = u3;
555   u5 = u3;
556   u4 = u3;
557   u5 = u3;
558   u4 = u3;
559   u5 = u3;
560   u4 = u3;
561   u5 = u3;
562   u4 = u3;
563   u5 = u3;
564   u4 = u3;
565   u5 = u3;
566   u4 = u3;
567   u5 = u3;
568   u4 = u3;
569   u5 = u3;
570   u4 = u3;
571   u5 = u3;
572   u4 = u3;
573   u5 = u3;
574   u4 = u3;
575   u5 = u3;
576   u4 = u3;
577   u5 = u3;
578   u4 = u3;
579   u5 = u3;
580   u4 = u3;
581   u5 = u3;
582   u4 = u3;
583   u5 = u3;
584   u4 = u3;
585   u5 = u3;
586   u4 = u3;
587   u5 = u3;
588   u4 = u3;
589   u5 = u3;
590   u4 = u3;
591   u5 = u3;
592   u4 = u3;
593   u5 = u3;
594   u4 = u3;
595   u5 = u3;
596   u4 = u3;
597   u5 = u3;
598   u4 = u3;
599   u5 = u3;
600   u4 = u3;
601   u5 = u3;
602   u4 = u3;
603   u5 = u3;
604   u4 = u3;
605   u5 = u3;
606   u4 = u3;
607   u5 = u3;
608   u4 = u3;
609   u5 = u3;
610   u4 = u3;
611   u5 = u3;
612   u4 = u3;
613   u5 = u3;
614   u4 = u3;
615   u5 = u3;
616   u4 = u3;
617   u5 = u3;
618   u4 = u3;
619   u5 = u3;
620   u4 = u3;
621   u5 = u3;
622   u4 = u3;
623   u5 = u3;
624   u4 = u3;
625   u5 = u3;
626   u4 = u3;
627   u5 = u3;
628   u4 = u3;
629   u5 = u3;
630   u4 = u3;
631   u5 = u3;
632   u4 = u3;
633   u5 = u3;
634   u4 = u3;
635   u5 = u3;
636   u4 = u3;
637   u5 = u3;
638   u4 = u3;
639   u5 = u3;
640   u4 = u3;
641   u5 = u3;
642   u4 = u3;
643   u5 = u3;
644   u4 = u3;
645   u5 = u3;
646   u4 = u3;
647   u5 = u3;
648   u4 = u3;
649   u5 = u3;
650   u4 = u3;
651   u5 = u3;
652   u4 = u3;
653   u5 = u3;
654   u4 = u3;
655   u5 = u3;
656   u4 = u3;
657   u5 = u3;
658   u4 = u3;
659   u5 = u3;
660   u4 = u3;
661   u5 = u3;
662   u4 = u3;
663   u5 = u3;
664   u4 = u3;
665   u5 = u3;
666   u4 = u3;
667   u5 = u3;
668   u4 = u3;
669   u5 = u3;
670   u4 = u3;
671   u5 = u3;
672   u4 = u3;
673   u5 = u3;
674   u4 = u3;
675   u5 = u3;
676   u4 = u3;
677   u5 = u3;
678   u4 = u3;
679   u5 = u3;
680   u4 = u3;
681   u5 = u3;
682   u4 = u3;
683   u5 = u3;
684   u4 = u3;
685   u5 = u3;
686   u4 = u3;
687   u5 = u3;
688   u4 = u3;
689   u5 = u3;
690   u4 = u3;
691   u5 = u3;
692   u4 = u3;
693   u5 = u3;
694   u4 = u3;
695   u5 = u3;
696   u4 = u3;
697   u5 = u3;
698   u4 = u3;
699   u5 = u3;
700   u4 = u3;
701   u5 = u3;
702   u4 = u3;
703   u5 = u3;
704   u4 = u3;
705   u5 = u3;
706   u4 = u3;
707   u5 = u3;
708   u4 = u3;
709   u5 = u3;
710   u4 = u3;
711   u5 = u3;
712   u4 = u3;
713   u5 = u3;
714   u4 = u3;
715   u5 = u3;
716   u4 = u3;
717   u5 = u3;
718   u4 = u3;
719   u5 = u3;
720   u4 = u3;
721   u5 = u3;
722   u4 = u3;
723   u5 = u3;
724   u4 = u3;
725   u5 = u3;
726   u4 = u3;
727   u5 = u3;
728   u4 = u3;
729   u5 = u3;
730   u4 = u3;
731   u5 = u3;
732   u4 = u3;
733   u5 = u3;
734   u4 = u3;
735   u5 = u3;
736   u4 = u3;
737   u5 = u3;
738   u4 = u3;
739   u5 = u3;
740   u4 = u3;
741   u5 = u3;
742   u4 = u3;
743   u5 = u3;
744   u4 = u3;
745   u5 = u3;
746   u4 = u3;
747   u5 = u3;
748   u4 = u3;
749   u5 = u3;
750   u4 = u3;
751   u5 = u3;
752   u4 = u3;
753   u5 = u3;
754   u4 = u3;
755   u5 = u3;
756   u4 = u3;
757   u5 = u3;
758   u4 = u3;
759   u5 = u3;
760   u4 = u3;
761   u5 = u3;
762   u4 = u3;
763   u5 = u3;
764   u4 = u3;
765   u5 = u3;
766   u4 = u3;
767   u5 = u3;
768   u4 = u3;
769   u5 = u3;
770   u4 = u3;
771   u5 = u3;
772   u4 = u3;
773   u5 = u3;
774   u4 = u3;
775   u5 = u3;
776   u4 = u3;
777   u5 = u3;
778   u4 = u3;
779   u5 = u3;
780   u4 = u3;
781   u5 = u3;
782   u4 = u3;
783   u5 = u3;
784   u4 = u3;
785   u5 = u3;
786   u4 = u3;
787   u5 = u3;
788   u4 = u3;
789   u5 = u3;
790   u4 = u3;
791   u5 = u3;
792   u4 = u3;
793   u5 = u3;
794   u4 = u3;
795   u5 = u3;
796   u4 = u3;
797   u5 = u3;
798   u4 = u3;
799   u5 = u3;
800   u4 = u3;
801   u5 = u3;
802   u4 = u3;
803   u5 = u3;
804   u4 = u3;
805   u5 = u3;
806   u4 = u3;
807   u5 = u3;
808   u4 = u3;
809   u5 = u3;
810   u4 = u3;
811   u5 = u3;
812   u4 = u3;
813   u5 = u3;
814   u4 = u3;
815   u5 = u3;
816   u4 = u3;
817   u5 = u3;
818   u4 = u3;
819   u5 = u3;
820   u4 = u3;
821   u5 = u3;
822   u4 = u3;
823   u5 = u3;
824   u4 = u3;
825   u5 = u3;
826   u4 = u3;
827   u5 = u3;
828   u4 = u3;
829   u5 = u3;
830   u4 = u3;
831   u5 = u3;
832   u4 = u3;
833   u5 = u3;
834   u4 = u3;
835   u5 = u3;
836   u4 = u3;
837   u5 = u3;
838   u4 = u3;
839   u5 = u3;
840   u4 = u3;
841   u5 = u3;
842   u4 = u3;
843   u5 = u3;
844   u4 = u3;
845   u5 = u3;
846   u4 = u3;
847   u5 = u3;
848   u4 = u3;
849   u5 = u3;
850   u4 = u3;
851   u5 = u3;
852   u4 = u3;
853   u5 = u3;
854   u4 = u3;
855   u5 = u3;
856   u4 = u3;
857   u5 = u3;
858   u4 = u3;
859   u5 = u3;
860   u4 = u3;
861   u5 = u3;
862   u4 = u3;
863   u5 = u3;
864   u4 = u3;
865   u5 = u3;
866   u4 = u3;
867   u5 = u3;
868   u4 = u3;
869   u5 = u3;
870   u4 = u3;
871   u5 = u3;
872   u4 = u3;
873   u5 = u3;
874   u4 = u3;
875   u5 = u3;
876   u4 = u3;
877   u5 = u3;
878   u4 = u3;
879   u5 = u3;
880   u4 = u3;
881   u5 = u3;
882   u4 = u3;
883   u5 = u3;
884   u4 = u3;
885   u5 = u3;
886   u4 = u3;
887   u5 = u3;
888   u4 = u3;
889   u5 = u3;
890   u4 = u3;
891   u5 = u3;
892   u4 = u3;
893   u5 = u3;
894   u4 = u3;
895   u5 = u3;
896   u4 = u3;
897   u5 = u3;
898   u4 = u3;
899   u5 = u3;
900   u4 = u3;
901   u5 = u3;
902   u4 = u3;
903   u5 = u3;
904   u4 = u3;
905   u5 = u3;
906   u4 = u3;
907   u5 = u3;
908   u4 = u3;
909   u5 = u3;
910   u4 = u3;
911   u5 = u3;
912   u4 = u3;
913   u5 = u3;
914   u4 = u3;
915   u5 = u3;
916   u4 = u3;
917   u5 = u3;
918   u4 = u3;
919   u5 = u3;
920   u4 = u3;
921   u5 = u3;
922   u4 = u3;
923   u5 = u3;
924   u4 = u3;
925   u5 = u3;
926   u4 = u3;
927   u5 = u3;
928   u4 = u3;
929   u5 = u3;
930   u4 = u3;
931   u5 = u3;
932   u4 = u3;
933   u5 = u3;
934   u4 = u3;
935   u5 = u3;
936   u4 = u3;
937   u5 = u3;
938   u4 = u3;
939   u5 = u3;
940   u4 = u3;
941   u5 = u3;
942   u4 = u3;
943   u
```

它还调用 `sub_10001000`。因此它暂定了全部进程。目的可能是为了改变进程运行状态，比如改变内核或者安装内嵌的挂钩。

- `sub_10001000`: 调用 `LoadLibraryA` 和 `GetProcAddress` 获取当前进程 IP。
- `sub_100012A3`: 调用 `GetModuleHandleA` 和 `GetProcAddress`，以及 `sub_10001203`。同样也是在获取现在的进程号，这是为了后面的对进程处理埋下伏笔。
- `sub_10001203`: 通过 `VirtualProtect` 修改了内存的运行以及读写的权限。目的就是这样的可以将 `send` 函数的目标改为恶意代码指定的地址。由此实现重定向。这样实施挂钩，可以拦截和篡改邮件。最后 `VirtualProtect` 则恢复原有的保护。
- `sub_100011BD`:
- `sub_10001499`: 调用 `sub_100013DA`。
- `sub_100013DA`: 内容和 `sub_100012FE` 几乎一样。但这次 `sub_100013BD` 暂停了其他的所有线程。因此这里的 `sub_10001499` 调用将之前所有挂起的线程全部都恢复了。

现在我们回到 `sub_100014B6`，一切就很明朗了。

- **检查进程**: 这个函数通过获取当前运行的可执行文件的名称，然后转换为大写，最后再和三个预设的邮件可执行客户端进行比较来实现的。能够判断自己现在是否正在目标的三个邮件客户端进程中运行。
- **send 重定向与挂钩**: 为了能够发送指定的数据到指定的目标，需要对 `send` 函数进行修改。是新方法是在 `send` 函数中修改最开始的目标，然后实现重新定向，达到实施挂钩的目的。
- **数据劫持与篡改**: 最后一步，在 `send` 函数处理后的 `buffer` 中，内联函数寻找“REPT TO”的垃圾。而 `RCPT` 是 `SMTP` 命令，由此实现对指定电子邮件收件人的修改。在这之后，将一个新的 `REPT TO` 写进，不过这次发送的目标是 `Lab11-02.ini` 解密得来的 `billy@malwareanalysisbook.com`。

由此总结道。病毒目的是在特定的邮件客户端进程中拦截发送的邮件消息，并篡改其中的收件人信息，以便将邮件副本发送到恶意软件作者指定的地址。

### 3.3.3 实验问题

1. Q1: 这个恶意 DLL 导出了什么？

回答：唯一的导出函数是 `installer`。

2. Q2: 使用 `rundll32.exe` 安装这个恶意代码后，发生了什么？

回答：通过 `rundll32.exe Lab11-02.dll, installer` 命令，可以正常运行病毒。根据在 `Procmon` 中发现的结果，我们注意到：病毒创建了新的文件 `spoolvxx32.dll`，其实就是他自己的六耳猕猴。然后在 `AppInit_DLLs` 键值下实现配置以驻留。不过我们第一次打开时候没有把 `ini` 文件放在正确的位置。

3. Q3: 为了使这个恶意代码正确安装，`Lab11-02.ini` 必须放置在何处？

回答：根据 `Procmon` 观察，`Lab11-02ini` 必须位于 `%SystemRoot%\System32\` 目录下。

#### 4. Q4: 这个安装的恶意代码如何驻留?

回答:将自身安装到 AppInit\_DLLs 的注册表键值中,这可以使恶意代码加载到所有装载 User32.dll 的进程中。以此实现驻留。

#### 5. Q5: 这个恶意代码采用的用户态 Rootkit 技术是什么?

回答:通过对 send 函数的代码进行修改的重定向,实现了 send 函数安装了一个 inline 挂钩。

#### 6. Q6: 挂钩代码做了什么?

回答:它会检查所有已经发出去的缓冲区,看其中是否包含 RCPT TO: 的电子邮件信息,如果发现了这个字符串,则它会添加一个额外的 RCPT TO 行,来增加一个恶意的电子邮件账户。就是 billy@malwareanalysisbook.com。

#### 7. Q7: 哪个或者哪些进程执行这个恶意攻击,为什么?

回答:针对 MSIMN.exe、THEBAT.exe 和 OUTLOOK.exe 即这三个邮件客户端。因为病毒会优先检查自己所正在运行的客户端名字,由此判断是否是需要的空间,不是的话也不会安装 hook。

#### 8. Q8:.ini 文件的意义是什么?

回答:包含一个加密的邮件地址。使用 OllyDBG 解密 Lab11-02.ini 之后我们看到它包含 billy@malwareanalysisbook.com。

#### 9. Q9: 你怎样用 Wireshark 动态捕获这个恶意代码的行为?

回答:这里补做一下可以发现,Wireshark 抓取的包会包含一个虚假的邮件服务器也就是 billy@malwareanalysisbook.com 还有 Outlook Express 客户端。

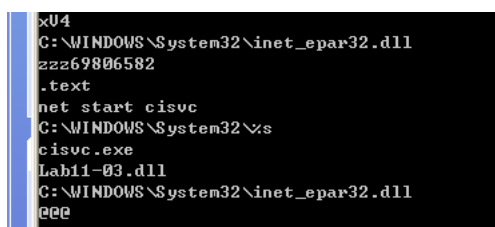
### 3.4 Lab11-3

分析恶意代码 Lab11-03.exe 和。确保这两个文件在分析时位于同一个目录中。

#### 3.4.1 静态分析

首先对两个文件进行静态分析:

##### 1. Lab11-03.exe: 首先使用 Strings 查看其字符串:



```
xU4
C:\WINDOWS\System32\inet_epar32.dll
zzz69806582
.text
net start cisvc
C:\WINDOWS\System32\cs
cisvc.exe
Lab11-03.dll
C:\WINDOWS\System32\inet_epar32.dll
qqq
```

图 3.34: exe 的字符串 1

图3.34看到了一些值得注意的地方:

- C:\Windows\System32\inet\_epar32.dll: inet\_epar32.dll 文件可能一些与网络通信或者互联网协议相关的功能,由此推测网络行为。

- zzz69806582: 这里最开始没有引我瞩目，不过后面会多次出现。
- net start cisvc 和 cisvc.exe: 在 Windows 操作系统中，使用”net start” 命令可以启动指定的服务，我们知道他有服务行为。但具体什么服务？
- Lab11-03.dll: 明显是另一个 dll，代表他可能对其进行操作。

再进一步看，又看到：

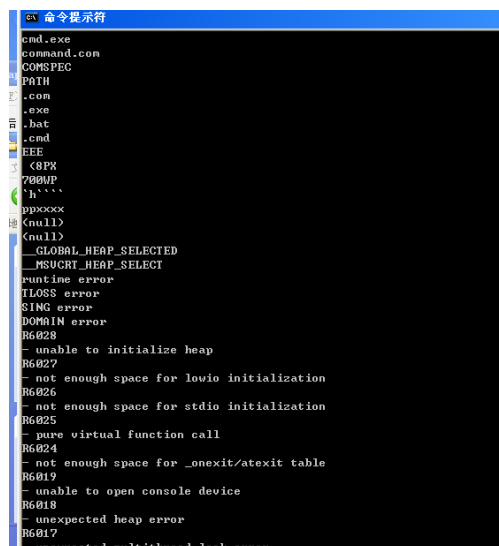


图 3.35: exe 的字符串 2

图3.35又看到了：

- cmd.exe 与 command.com: 推测可能又有远程 Shell 后门操纵？
- runtime/SING/TLOSS/DOMAIN error: 中间两个未知。第一个就是超市里，最后是域名错误。
- R60XX: 暂时未知。
- unable to XXX: 这是一个通用的错误消息格式，通常出现在命令行等地方，进一步验证了可能与 Shell 操作有关

接下来看看导入函数：

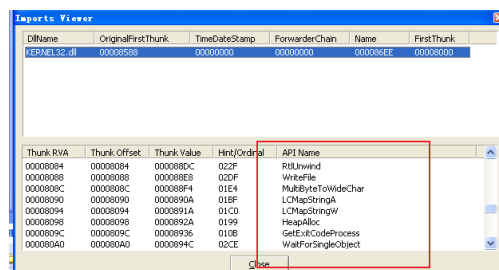


图 3.36: exe 导入函数

图3.36只有一些很普通的函数，其中唯一值得注意的只有 WriteFile。

2. **Lab11-03.dll**: 接着来查看 dll，依然是先使用 strings 查看其字符串；

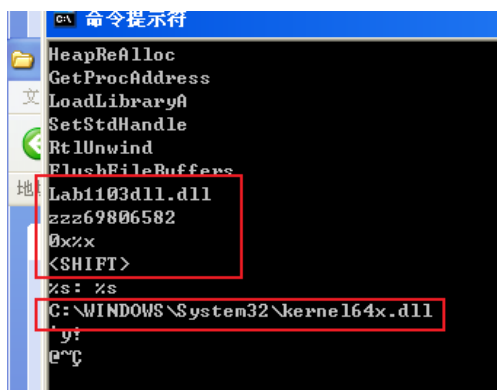


图 3.37: dll 的 strings

图3.37看到：

- C:\WINDOWS\System32\kernel64x.dll：推测这是 System32 下某个伪装起来的恶意目录。
- Lab1103dll.dll：非常可疑，疑似在混淆处理？
- zzz69806582：出现在 exe 中的再出现，也是要重点关注。
- <SHIFT>：明显是在表示电脑的 Shift 键，推测可能和键盘记录有关系。

接下来查看一下这个 dll 的导出函数：

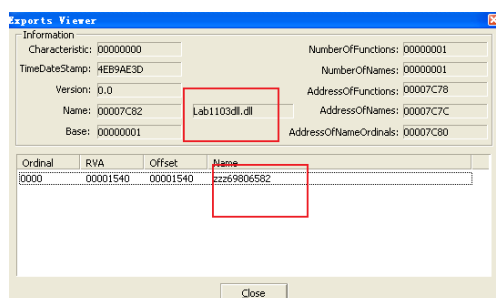


图 3.38: dll 的导出函数

图3.38的位置可以看到：**Lab1103dll.dll**，这个也挺奇怪的。**zzz69806582**，这里面也有奇怪的导出函数。

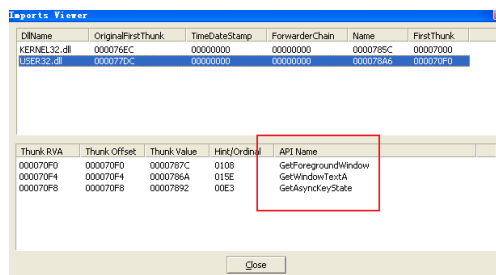


图 3.39: dll 的导入函数

图3.39展示了一些导出函数：

- GetAsyncKeyState: 用于检查指定虚拟键的状态。它可以检查按键是否当前被按下，或者在调用该函数之前是否被按下过。
- GetForegroundWindow: 用来确定用户当前正在与哪个窗口交互。
- GetWindowTextA: 用来获取窗口的标题，以便进行相应的处理。

综合上面这些，我们推测这可能是一个击键记录器，并把击键记录在 kernel64x.dll。

### 3.4.2 综合分析

接下来会进一步使用综合的深度静态以及动态分析来试图找到其中的恶意目的。

#### 1. Procmon 动态分析

首先我们推测主要运行的应该是可执行文件，因此设置对应名字为 Lab11-03.exe 的过滤器：

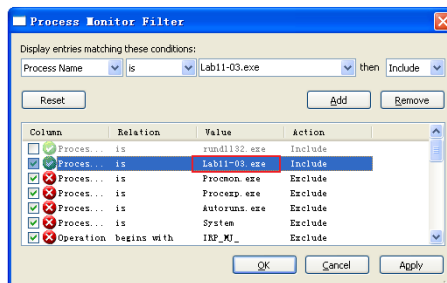


图 3.40: 过滤器

然后双击运行可执行文件，结果如下：

10:2...	Lab11-03.exe	6880	QueryDefin...	C:\Labs\Malware\Practical Malw...	SUCCESS	Size: 0
10:2...	Lab11-03.exe	6880	CreateFile	C:\WINDOWS\system32\inet_epar3...	SUCCESS	Desired Acces...
10:2...	Lab11-03.exe	6880	CreateFile	C:\WINDOWS\system32\...	SUCCESS	Desired Acces...
10:2...	Lab11-03.exe	6880	CreateFile	C:\WINDOWS\system32\...	SUCCESS	Desired Acces...
10:2...	Lab11-03.exe	6880	QueryAttrib...	C:\WINDOWS\system32\inet_epar3...	SUCCESS	FileSystemAtt...
10:2...	Lab11-03.exe	6880	QueryBasicI...	C:\WINDOWS\system32\inet_epar3...	SUCCESS	CreationTime...
10:2...	Lab11-03.exe	6880	QueryAttrib...	C:\Labs\Malware\Practical Malw...	SUCCESS	FileSystemAtt...
10:2...	Lab11-03.exe	6880	SetEndOfFil...	C:\WINDOWS\system32\inet_epar3...	SUCCESS	EndOfFile: 49...
10:2...	Lab11-03.exe	6880	CreateFileK...	C:\Labs\Malware\Practical Malw...	SUCCESS	SyncType: Syn...
10:2...	Lab11-03.exe	6880	QueryStanda...	C:\Labs\Malware\Practical Malw...	SUCCESS	AllocationSiz...
10:2...	Lab11-03.exe	6880	CreateFileK...	C:\Labs\Malware\Practical Malw...	SUCCESS	SyncType: Syn...
10:2...	Lab11-03.exe	6880	WriteFile	C:\WINDOWS\system32\inet_epar3...	SUCCESS	Offset: 0, Le...
10:2...	Lab11-03.exe	6880	SetBasicInf...	C:\WINDOWS\system32\inet_epar3...	SUCCESS	CreationTime...
10:26:56.819409	Lab11-03.exe	6880	CreateFile	C:\Labs\Malware\Practical Malw...	SUCCESS	
10:2...	Lab11-03.exe	6880	CreateFile	C:\WINDOWS\system32\inet_epar3...	SUCCESS	
10:2...	Lab11-03.exe	6880	CreateFile	C:\WINDOWS\system32\cisvc.exe	SUCCESS	Desired Acces...
10:2...	Lab11-03.exe	6880	QueryStanda...	C:\WINDOWS\system32\cisvc.exe	SUCCESS	AllocationSiz...
10:2...	Lab11-03.exe	6880	CreateFileK...	C:\WINDOWS\system32\cisvc.exe	SUCCESS	SyncType: Syn...
10:2...	Lab11-03.exe	6880	QueryStanda...	C:\WINDOWS\system32\cisvc.exe	SUCCESS	AllocationSiz...
10:2...	Lab11-03.exe	6880	CreateFile	C:\WINDOWS\system32\cisvc.exe	SUCCESS	
10:2...	Lab11-03.exe	6880	QueryOpen	C:\WINDOWS\system32\cmd.exe	SUCCESS	CreationTime...
10:2...	Lab11-03.exe	6880	CreateFile	C:\WINDOWS\system32\cmd.exe	SUCCESS	Desired Acces...
10:2...	Lab11-03.exe	6880	CreateFileK...	C:\WINDOWS\system32\cmd.exe	SUCCESS	SyncType: Syn...
10:2...	Lab11-03.exe	6880	CreateFileK...	C:\WINDOWS\system32\cmd.exe	SUCCESS	SyncType: Syn...
10:2...	Lab11-03.exe	6880	RegOpenKey	HKEY\System\CurrentControlSet\...	NAME NOT FOUND	Desired Acces...

图 3.41: Lab11-03.exe 行为

图3.41能够看到如下值得注意的地方：

- CreateFile of inet\_epar32.dll: 可以看到它在 System32 的目录下创建了这个 inet\_epar32.dll。
- WirteFile of inet\_epar32.dll: 可以看到对 inet\_epar32.dll 进行写内容操作。
- CreateFile cisvc.exe: 可以看到病毒会对 cisvc.exe 文件进行创建，不过没有发现写入的操作，不过有创建文件映射的地方。



经过分许，之所以上面不会出现 WriteFile 的原因：在于我们在最开始不知道它的行为时候没有采用管理员权限运行，导致出现了和 Lab11-01 一样无法向 System32 下文件进行写入的问题。恢复快照后，重新管理员运行：

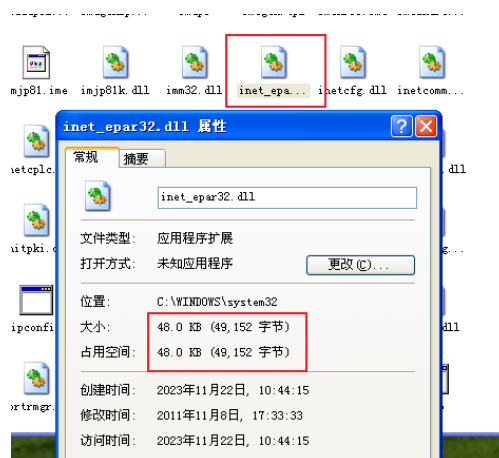


图 3.42: inet\_epar32.dll

图3.42看到此时在 System32 目录下出现了文件 inet\_epar32.dll，其大小为 48KB

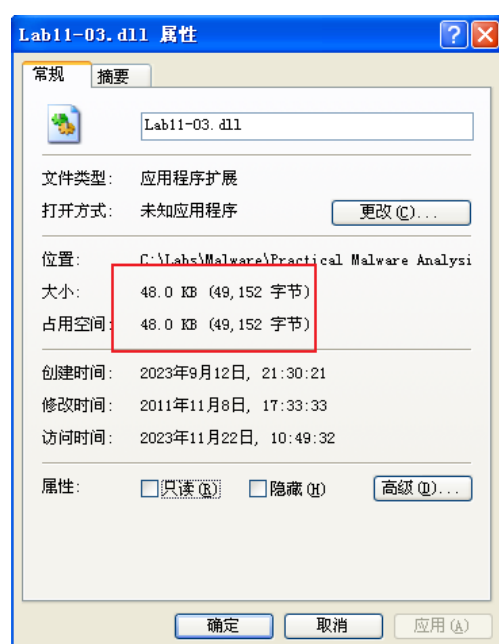


图 3.43: Lab11-03.dll

图3.43看到 Lab11-03.dll 大小也是 48KB，因此得出推测 inet\_epar32.dll 不过是 exe 将 dll 文件直接复制到 System32 目录下的新名字。

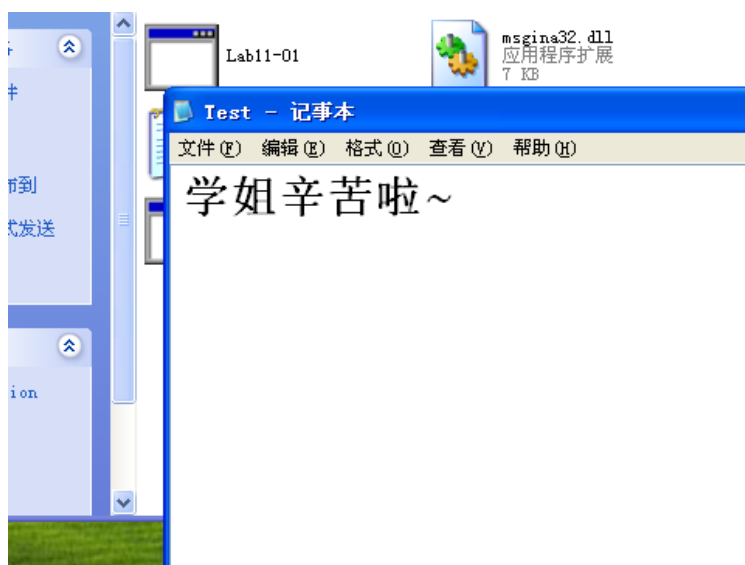


图 3.44: 小彩蛋 学姐辛苦啦

然后既然我们推测了他是一个击键记录器，我们就试图打开一个 txt 文件写入一点内容验证，如图3.44所示，学姐辛苦啦！

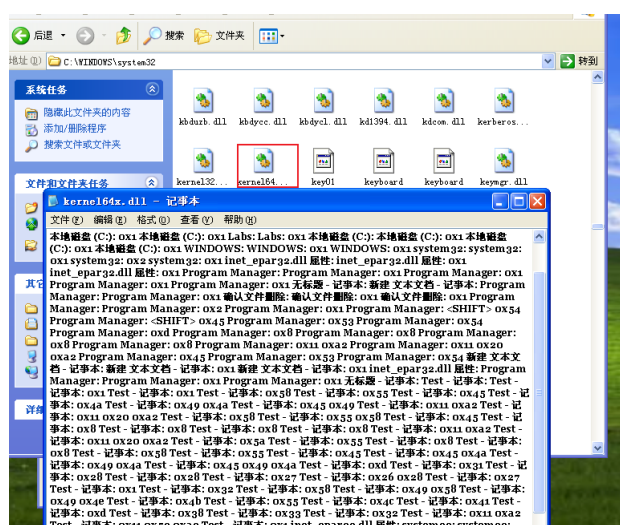


图 3.45: cisvc.exe 内容

图3.45看到使用记事本打开这个文件后的结果，能清楚地看到许多击键记录，这证明了我们的猜想。

## 2. IDA 分析 exe

在动态分析，证明了我们的猜想后，我们接下来使用 IDA 进一步分析其行为，首先查看 exe 的 main 函数：

```

.text:00401200 ; int __cdecl main(int argc, const char **argv, const char **envp)
.text:00401200 _main proc near
; CDE XREF: start+0fip
; byte ptr -100h
; dword ptr 8
; dword ptr 0Ch
; dword ptr 10h
push ebp
mov ebp, esp
sub esp, 100h
push 0 ; Dfa11f1Exists
push offset NewFileName ; "C:\\WINDOWS\\System32\\inet_epar32.dll"
push offset ExistingFileName ; "Lab11-03.dll"
call @_imp__File
push offset aCisvc_exe ; "cisvc.exe"
push offset aWindowsSyst_0 ; "C:\\WINDOWS\\System32\\%s"
lea eax, [ebp+FileName]
push eax ; char *
call sprintf
add esp, 8Ch
lea ecx, [ebp+FileName]
push ecx ; lpFileName
call sub_401070
add esp, 4
push offset aNetStartCisvc ; "net start cisvc"
call system
add esp, 4
xor eax, eax
mov esp, ebp
pop ebp
ret
.text:00401225 _main endp

```

图 3.46: exe 的 main 函数

图3.46可以看到一些重要的地方，因为需要做展示字符串，因此这里没有用反汇编：

- CopyFileA("Lab11-03.dll", "C:\\\\WINDOWS\\System32\\inet\_epar32.dll", 0):  
根据调用 CopyFileA 前面 push 进入的参数，我们可以还原调用的情况，即实现了将 Lab11-03.dll 复制到 System32 目录下，并命名为 inet\_epar32.dll。
- sprintf(Buffer, "C:\\\\WINDOWS\\System32\\%s", "cisvc.exe")  
通过 sprintf 将完整路径拼接起来后放入 buffer 中。
- buffer 作为参数调用 sub\_401070。
- system 系统调用：传入参数 net start cisvc，即启动索引服务。

其中不明确的只有这个 sub\_401070，想必它应该保存了使用 cisvc.exe 做的事情，查看它发现其中内部调用了多个函数，因此首先看看它的导交叉引用图，看看他大概做了什么：

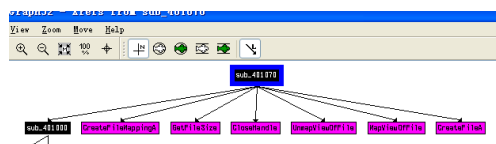


图 3.47: 交叉引用

图3.47看到调用关系非常复杂，其中 sub\_401000 又调用了很多函数，我们重点关注它在函数内部调用的：

- CreateFileA：创建了 cisvc.exe。
- CreateFileMappingA 和 MapViewOfFile：创建文件 cisvc 的文件映射，然后映射进入内存中。其中 MapViewOfFile 返回的内存映射视图基地址可以被读写。由此给了它们必要的权限。
- UnmapViewOfFile：再关闭映射，结果就是可以把所有在内存中对文件的修改也同样写入磁盘。这样就解释了为什么在 Procom 中没有观察到 WriteFile 函数，实际上是通过映射的方式在内存中操作的。

接下来去顺序查看整个代码，其中发现了一处值得注意到地方：

```

.text:0040127C ; CODE XREF: sub_401070+196fj
.text:0040127C loc_40127C: ; sub_401070+205fj
.text:0040127C mov edi, [ebp+lpBaseAddress]
.text:0040127F add edi, [ebp+var_28]
.text:00401282 mov ecx, 4Eh
.text:00401287 mov esi, offset byte_409030
.text:0040128C rep movsd
.text:0040128E movsui
.text:00401290 mov eax, [ebp+var_14]
.text:00401293 mov ecx, [ebp+var_28]
.text:00401296 sub ecx, [eax+1Ah]
.text:00401299 mov edx, [ebp+var_24]
.text:0040129C add ecx, [edx+2Ch]
.text:0040129F mov eax, [ebp+var_24]
.text:004012A2 mov [eax+20h], ecx
.text:004012A5 mov ecx, [ebp+hFile]
.text:004012A8 push ecx ; hObject
.text:004012AB call ds:CloseHandle

```

图 3.48: 使用字节 byte\_409030

图3.48可以看到此处是再通过 var\_28 等变量来调整一些偏移量，然后一直关键的地方在于它将 byte\_409030 放去了 esi 中，目的就是通过下面的 rep movsd 复制到映射文件中。因此我们推测这个可能是写入文件 cisvc.exe 的内容，过去查看：

```

.data:00409030 ; char byte_409030[]
.data:00409030 byte_409030 db 55h
.data:00409031 unk_409031 db 89h
.data:00409032 ; char byte_409032[]
.data:00409032 byte_409032 db 8E5h
.data:00409033 ; char byte_409033[]
.data:00409033 byte_409033 db 87h
.data:00409034 db 8ECh
.data:00409035 db 40h
.data:00409036 db 0

```

图 3.49: shellcode

图3.49看到一些原始字节，我们不知道是什么，但可以试着按下 C 著那位反汇编表示，发现了它确实是一段人工构造的汇编代码即 shellcode！

```

.data:00409030 ;
.data:00409030 loc_409030:
.data:00409030 push ebp
.data:00409031 loc_409031:
.data:00409031 mov ebp, esp
.data:00409032 loc_409032:
.data:00409033 loc_409033:
.data:00409033 sub esp, 40h
.data:00409034 jmp loc_409134

```

图 3.50: shellcode 解密结果

图3.50看到解密的结果，不过不是很好看懂，跟着内容来到 shellcode 的结尾 0x00409139 处：

```

.data:00409139 loc_409139: call sub_409000 ; CODE XREF: .data:00409039fj
.data:0040913A
.data:0040913B aC:\Windows\System32\inet_epar32.dll",0
.data:0040913C azzz69806582 db zzz69806582",0
.data:0040913D db 0
.data:0040913E db 0
.data:0040913F db 0
.data:00409140 ; char a_text[]

```

图 3.51: shellcode 末尾字符串

图3.51是原本有一些零散的串，这里将它们拼接起来，我们看到了：

- System32 下的 inet\_epar32.dll: 这是病毒加载的 dll，经过前面验证我们应该知道了就是复制过去的 Lab11-03.dll。
- zzz69806582: 它是 inet\_epar32.dll 的导出函数。

因此为了进一步分析恶意目的，我们需要看看这个 zzz69806582 究竟在干什么。

### 3. IDA 分析 dll

使用 IDA 加载 Lab11-03.dll 后，发现它确实有一个 zzz69806582 的函数。

```

.text:10001540 zzz69806582 proc near ; DATA XREF: .rdataoff_10007C78jo
.text:10001540
.text:10001540 var_4 = dword ptr -4
.text:10001540
.text:10001540 push ebp
.text:10001540 mov ebp, esp
.text:10001540 push ecx ; lpThreadId
.text:10001540 push 0 ; dwCreationFlags
.text:10001540 push 0 ; lpParameter
.text:10001540 push offset StartAddress ; lpStartAddress
.text:10001540 push 0 ; dwStackSize
.text:10001540 push 0 ; lpThreadAttributes
.text:10001551 call ds:CreateThread
.text:10001555 mov [ebp+var_4], eax
.text:1000155C cmp [ebp+var_4], 0
.text:10001560 jz short loc_10001566
.text:10001562 var eax, eax
.text:10001564 jmp short loc_10001568
.text:10001566

```

图 3.52: zzz69806582

图3.52看到了这个函数的特点：它唯一调用的函数就是 `CreateThread` 创建一个新线程，运行的函数就是 `StartAddress`，这个类似于计网多线程 `socket` 编程中使用的 `CreateThread`。

接下来去查看 `StartAddress` 函数：

```

1 HANDLE __stdcall StartAddress()
2 {
3     HANDLE result; // eax@3
4     HANDLE hObject; // [sp+8h] [bp-818h]@3
5     HANDLE hFile; // [sp+Ch] [bp-814h]@4
6     char v3; // [sp+10h] [bp-810h]@5
7     unsigned __int8 v4; // [sp+14h] [bp-80Ch]@1
8     int u5; // [sp+14h] [bp-80Ch]@1
9     unsigned __int8 v6; // [sp+18h] [bp-808h]@1
10    int v7; // [sp+18h] [bp-808h]@1
11    HANDLE v8; // [sp+1Ch] [bp-804h]@5
12
13    v7 = 102h;
14    u5 = 102h;
15    _nbsncpy(&v6, &byte_10000D28, 0x400u);
16    _nbsncpy(&v8, &byte_10000D28, 0x400u);
17    if (OpenMutexA(0x1F001u, 0, Name))
18        exit(0);
19    result = CreateMutexA(0, 1, Name);
20    hObject = result;
21    if (result)
22    {
23        result = CreateFileA(FileNames, 0xC0000000, 1u, 0, Au, 0x80u, 0);
24        hFile = result;
25        if (result)
26        {
27            SetFilePointer(result, 0, 0, 2u);
28            v8 = hFile;
29            sub_10001380(&v3);
30            CloseHandle(hFile);
31            result = (HANDLE)CloseHandle(hObject);
32        }
33    }
34    return result;
35}

```

图 3.53: StartAddress

图3.53其中有一些应该注意的地方：

- `OpenMutexA` 和 `CreateMutexA`：这招我们前面的 Lab 见过，就是通过创建一个互斥量（其中这里的 `Name` 是”MZ “），来实现保证任何时刻病毒只有一个实例在运行，避免引人注目。
- `CreateFileA`：创建并试图打开文件在 `System32` 下的 `kernel64x.dll`。如果打开成功，`result` 为真。
- `sub_10001380`：我们看到如果打开结果为真，在里面通过一个指针 `FilePointer` 依次试图在遍历文件，然后使用 `sub_10001380` 进行文件中的处理。

根据我们动态分析的结果，推荐是写入的日志，因此查看 `sub_10001380`：

```

1 signed int __cdecl sub_10001380(int a1)
2 {
3     DWORD NumberOfBytesWritten; // [sp+4h] [bp-404h]@3
4     char Buffer; // [sp+8h] [bp-400h]@3
5
6     while ( !sub_10001400(a1) )
7     {
8         if ( !_DWORD_>0 )
9         {
10            sprintf(Buffer, &v5, a1 + 4, a1 + 1022);
11            WriteFile((HANDLE_>)(a1 + 2060), &Buffer, strlen(Buffer), &NumberOfBytesWritten, 0);
12            Sleep(800u);
13        }
14        return 1;
15    }
16}

```

图 3.54: sub\_10001380

图3.54看到一些地方：

- 循环调用 `sub_10001380`。只要结果为真。
- `sprintf`: 对传入的 `Buffer` 进行格式化。变成格式化字符串。
- `WriteFile`: 将 `Buffer` 写入文件对应的句柄。实现记录!
- `Sleep(0xAu)` 代表睡眠时间为 10 (0xA 的十进制值) 个单位。即睡眠 10 毫秒。间隔 10 毫秒行动。

现在我们已经知道会写入内容了, 写了什么呢? 因此必须要去 `sub_10001380` 查看了:



```

27  if ( v1 )
28  {
29      _msbncpy((unsigned __int8 *)(&v1 * 4), (const unsigned __int8 *)&v1, 4);
30      *(_DWORD *)v1 = 1;
31  }
32  if ( GetAsyncKeyState(0x00000000) & 0x8000 )
33  {
34      GetAsyncKeyState(0x00000000) & 0x8000
35      GetAsyncKeyState(0x00000000) & 0x8000
36      GetAsyncKeyState(0x00000000) & 0x8000
37      sprintf((char *)Buffer, "%c", v1);
38      for ( v1 = 0; (unsigned __int8)v1 < 0xFF; ++v1 )
39      {
40          if ( GetAsyncKeyState(v1) & 0x8000 )
41          {
42              for ( i = 0; i < 0xFF; ++i )
43              {
44                  if ( GetAsyncKeyState(i) & 0x8000 )
45                  {
46                      ++i;
47                      sprintf((char *)Buffer, "%c", i);
48                      strcat((char *)Buffer, Buffer);
49                  }
50              }
51              break;
52          }
53      }
54      if ( v1 )
55      {
56          _msbncpy((unsigned __int8 *)(&v1 * 4), 0, 4);
57          *(_DWORD *)v1 = 1;
58      }
59      else
60      {
61          _msbncpy((unsigned __int8 *)(&v1 * 4), 0, 4);
62      }
63  }
64  }
65  if ( v1 )
66  {
67      _msbncpy((unsigned __int8 *)(&v1 * 4), 0, 4);
68      *(_DWORD *)v1 = 1;
69  }
70  else
71  {
72      _msbncpy((unsigned __int8 *)(&v1 * 4), 0, 4);
73  }

```

图 3.55: `sub_10001380`

图3.55展示了 `sub_10001380`, 最关键的是 `GetAsyncKeyState` 的调用, 实现了用户态的击键记录, 最后返回通过 `WriteFile` 写入日志! 到此分析结束。

### 3.4.3 实验问题

在分析结束后, 我们来回答问题:

#### 1. Q1: 使用基础的静态分析过程, 你可以发现什么有趣的线索?

回答: 在使用 PEiD 分析导入函数导出函数时候, 我们发现了:

- `Lab11-03.exe` 有 `WriteFile`, 字符串有 `Lab11-03.dll`, 意味着它可能加载 `dll`; `inet_epar32.dll` 和 `net start cisvc` 意味着它可能启动一些服务。
- `Lab11-03.dll` 导入了 API 函数 `GetAsyncKeyState` 和 `GetForegroundWindow`; 又有字符串 `C:\WINDOWS\System32\kernel64x.dll` 和一个 `Lab1103dll.dll` 和 `zzz69806582`。

这些都让我们推测它是一个记录到文件 `kernel64x.dll` 的击键记录器。不过 `Lab1103dll.dll` 的含义我们到最后也没有发现。

#### 2. Q2: 当运行这个恶意代码时, 发生了什么?

回答: 注意要以管理员权限运行。它会首先复制 `Lab11-03.dll` 到 `System32` 下的 `inet_epar32.dll`, 然后向 `cisvc.exe` 写入数据并且启动索引服务 (不过并不是通过 `WriteFile` 而是映射的方式)。同时它也会向 `C:\Windows\System32\kernel64x.dll` 写入击键记录。

#### 3. Q3: `Lab11-03.exe` 如何安装 `Lab11-03.d11` 使其长期驻留?

回答: 通过入口点重定向进行特洛伊木马化索引服务来实现永久安装和驻留。实际上是通过 `shellcode` 加载实现的。

#### 4. Q4: 这个恶意代码感染 Windows 系统的哪个文件?

回答: 为了创建 inet\_epar32.dll 实现驻留, 它感染了 cisvc.exe, 然后调用了 inet\_epar32.dll 的导出函数 zzz69806582。

#### 5. Q5: Lab11-03.dll 做了什么?

回答: 轮询的密记录器。主要功能在导出函数 zz6986582 中。

#### 6. Q6: 这个恶意代码将收集的数据存放在何处?

回答: 存储击键记录和窗体输入记录, 其中击键记录被存入到 C:\Windows\System32\kernel64x.dll

### 3.5 Yara 检测

#### 3.5.1 Sample 提取

利用课程中老师提供的 Scan.py 程序, 将电脑中所有的 PE 格式文件全部扫描, 提取后打开 sample 文件夹查看相关信息:

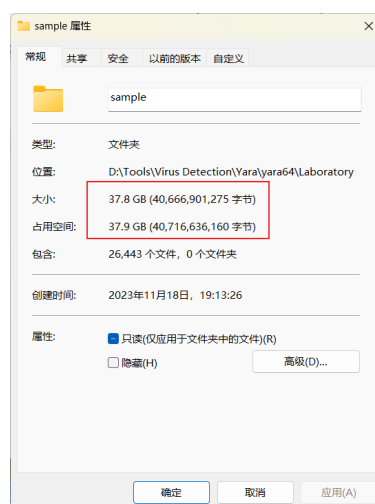


图 3.56: Sample 信息

图3.56可以看到从电脑中提取了所有 PE 格式文件后的文件及 sample 大小为 37.9GB。然后由于本次实验的中 msgina32.dll 是我在 xp 中提取的, 因此直接放在 sample 文件夹中。可以看到 sample 包含一共 26443 个文件。Yara 编写的规则将目标从 sample 中识别成功检测出本次 Lab11 的全部五个恶意代码包括 exe 和 dll。

#### 3.5.2 Yara 规则编写

本次 Yara 规则的编写基于上述的病毒分析和实验问题, 主要是基于静态分析的 Strings 字符串和 IDA 分析结果。为了能够更好地进行 Yara 规则的编写, 首先对之前分析内容进行回归。分别对 Lab11-01.exe, msgina32.dll, Lab11-02.dll, Lab11-01.exe 和 Lab11-03.dll 可以利用的病毒特征进行分条总结如下:

##### 1. Lab11-01.exe:

- **msgina32.dll**: 即它的资源节提取出来创建的恶意文件名字。被当作 GINA DLL, 用来拦截 GINA。

## 2. msgina32.dll:

- **msutil32.sys**: 即它会将自己伪装成的 System32 下的驱动文件名字, 但实际上是存取登陆的用户名和密码的日志文件。
- **UN %s DM %s PW %s OLD %s**: 即**拦截到的登陆凭证自带的日期时间等信息**。

## 3. Lab11-02.dll:

- **AppInit\_DLLs**: 即为了实现驻留和自启动, 病毒将自己加入到注册表的这个位置的一项中, 由此实现一旦重启会载入到任何用户进程中。
- **Lab11-02.ini**: 这个将会是绝杀, 代表着使用的另一个 ini 配置文件。
- **OUTLOOK.EXE**: 代表着病毒会使用发送的邮件客户端, 不过它会先检测自己是否正在运行在这个进程之中。

## 4. Lab11-03.exe:

- **zzz69806582**: Lab11-03.dll 加载的导出的函数名字。
- **net start cisvc**: 启动感染的索引服务实现的驻留。
- **Lab11-03.dll**: 这个将会是绝杀, 代表着它所载入的另一个 Lab11-03 的 dll 文件。

## 5. Lab11-03.dll:

- **Lab1103dll.dll**: 至今的为解谜题。不过也是一个绝杀。
- **zzz69806582**: 加载的导出函数, 主要来实现对于击键的记录。

因此加上必要的一些修饰符如 wide、ascii 以及 nocase 后, 编写如下 Yara 规则:

```
1 rule Lab11_01_exe
2 {
3   meta:
4     description = "Lab11_01_exe:Yara Rules"
5     date = "2023/11/22"
6     author = "ErwinZhou"
7   strings:
8     $clue1 = "msgina32.dll" wide ascii
9
10  condition:
11    all of them //Lab11-01.exe
12 }
13
14
15 rule msgina32_dll
16 {
```



```
17 meta:
18     description = "msgina32_dll:Yara Rules"
19     date = "2023/11/22"
20     author = "ErwinZhou"
21
22 strings:
23     $clue1 = "msutil32.sys" wide ascii
24     $clue2 = "UN %s DM %s PW %s OLD %s" wide ascii nocase
25
26 condition:
27     all of them //msgina32_dll
28 }
29
30
31 rule Lab11_02_dll
32 {
33 meta:
34     description = "Lab11_02:Yara Rules"
35     date = "2023/11/22"
36     author = "ErwinZhou"
37
38 strings:
39     $clue1 = "AppInit_DLLs" wide ascii
40     $clue2 = "Lab11-02.ini" wide ascii
41     $clue3 = "OUTLOOK.EXE" wide ascii
42
43 condition:
44     all of them //Lab11-02.exe
45 }
46
47 rule Lab11_03_exe
48 {
49 meta:
50     description = "Lab11_03_exe:Yara Rules"
51     date = "2023/11/22"
52     author = "ErwinZhou"
53
54 strings:
55     $clue1 = "zzz69806582" wide ascii
56     $clue2 = "net start cisvc" wide ascii nocase
57     $clue3 = "Lab11-03.dll" wide ascii
58
```

```
59 condition:
60     all of them //Lab11-03.exe
61 }
62
63 rule Lab11_03_dll
64 {
65 meta:
66     description = "Lab11_03_dll:Yara Rules"
67     date = "2023/11/22"
68     author = "ErwinZhou"
69
70 strings:
71     $clue1 = "Lab1103dll.dll" wide ascii
72     $clue2 = "zzz69806582" wide ascii
73
74 condition:
75     all of them //Lab11-03.dll
76 }
```

然后使用如下 Python 代码进行对 sample 的扫描：

```
1 import os
2 import yara
3 import time
4 # 加载YARA规则
5 rules = yara.compile('D:\Tools\Virus Detection\Yara\yara64\Lab11.yar')
6 # 初始化计数器
7 total_files_scanned = 0
8 total_files_matched = 0
9 def scan_folder(folder_path):
10     global total_files_scanned
11     global total_files_matched
12     # 检查文件夹是否存在
13     if os.path.exists(folder_path) and os.path.isdir(folder_path):
14         # 遍历文件夹内的文件和子文件夹
15         for root, dirs, files in os.walk(folder_path):
16             for filename in files:
17                 total_files_scanned += 1
18                 file_path = os.path.join(root, filename)
19                 with open(file_path, 'rb') as file:
20                     data = file.read()
21                     # 扫描数据
22                     matches = rules.match(data=data)
```

```
23         # 处理匹配结果
24         if matches:
25             total_files_matched += 1
26             print(f"File '{filename}' in path '{root}' matched YARA
27                 rule(s):")
28             for match in matches:
29                 print(f"Rule: {match.rule}")
30         else:
31             print(f'The folder at {folder_path} does not exist or is not a folder.')
32 # 文件夹路径
33 folder_path = 'D:\Tools\Virus Detection\Yara\yara64\Laboratory\sample'
34 # 记录开始时间
35 start_time = time.time()
36 # 递归地扫描文件夹
37 scan_folder(folder_path)
38 # 记录结束时间
39 end_time = time.time()
40 # 计算运行时间
41 runtime = end_time - start_time
42 print(f"Program runtime: {runtime} seconds.")
43 print(f"Total files scanned: {total_files_scanned}")
44 print(f"Total files matched: {total_files_matched}")
```

### 3.5.3 Yara 规则执行效率测试

扫描结果如下图所示：

```
File 'Lab11-01.exe' in path 'D:\Tools\Virus Detection\Yara\yara64\Laboratory\sample' m
atched YARA rule(s):
Rule: Lab11_01_exe
Rule: msgina32_dll
File 'Lab11-02.dll' in path 'D:\Tools\Virus Detection\Yara\yara64\Laboratory\sample' m
atched YARA rule(s):
Rule: Lab11_02_dll
File 'Lab11-03.dll' in path 'D:\Tools\Virus Detection\Yara\yara64\Laboratory\sample' m
atched YARA rule(s):
Rule: Lab11_03_dll
File 'Lab11-03.exe' in path 'D:\Tools\Virus Detection\Yara\yara64\Laboratory\sample' m
atched YARA rule(s):
Rule: Lab11_03_exe
File 'msgina32.dll' in path 'D:\Tools\Virus Detection\Yara\yara64\Laboratory\sample' m
atched YARA rule(s):
Rule: msgina32_dll
Program runtime: 125.77075290679932 seconds.
Total files scanned: 26443
Total files matched: 5
```

图 3.57: Yara 检测结果

图3.57可以看到能够成功地从 26443 个文件中唯一地识别检测到五个病毒文件包括 Lab11-01 的资源文件 msgina32.dll。

并且仅用时 125.77 秒，时间性能较快。总的来说，Yara 规则编写和检测较为成功。

### 3.6 IDAPython 辅助样本分析

根据要求，分别选择病毒分析过程中一些系统的过程编写如下的 Python 脚本：

#### 3.6.1 解密函数

这段代码中使用了异或操作，通过对字符串的每个字符与第一个字符进行异或，实现了简单的加密和解密过程。本次实验中由于 Lab11-03 出现需要解密 shellcode 的地方，因此可以用这个函数进行反混淆。

```
1 def decrypt(data):
2     # 获取输入字符串的长度
3     length = len(data)
4     # 初始化计数器
5     c = 1
6     # 初始化输出字符串
7     o = ""
8     # 使用异或解密
9     while c < length:
10        # 对输入字符串的第一个字符和后续字符进行异或操作，并将结果转换为字符
11        o += chr(ord(data[0]) ^ ord(data[c]))
12        # 更新计数器，移动到下一个字符
13        c += 1
14    # 返回解密后的字符串
15    return o
```

#### 3.6.2 查找并跟踪参数传递

这段代码的作用是在给定地址 addr 的上下文中，向前遍历指令，找到最近的一条 mov 指令，并检查是否将值赋给 esi 寄存器。如果是，则打印找到的参数值。本次实验中涉及的病毒具有复杂的调整行为比如 Lab11-02 的 11 个函数，这个函数可以查找参数并理解函数的参数传递方式。

```
1 def find_function_arg(addr):
2     # 进入一个无限循环
3     while True:
4         # 获取当前指令的前一个指令地址
5         addr = idc.PrevHead(addr)
6         # 判断当前指令是否是"mov"指令，并且目标操作数包含"esi"
7         if GetMnem(addr) == "mov" and "esi" in GetOpnd(addr, 0):
8             # 打印找到的参数值，并使用十六进制格式输出
9             print("We found it at 0x%x" % GetOperandValue(addr, 1))
10            # 跳出循环
11            break
```

### 3.6.3 综合分析

这段代码的主要目的是在给定地址的上下文中，查找特定函数的参数，然后获取该参数指向的字符串，并尝试通过异或解密算法对字符串进行解密。这对于本次实验中复杂的样本进行综合分析很有帮助。可以辅助获得对应参数的字符串并获得函数汇编代码。比如对 Lab11-02 复杂的分析检查交叉引用。

```
1 def find_function_arg(addr):
2     # 进入一个无限循环
3     while True:
4         # 获取当前指令的前一个指令地址
5         addr = idc.PrevHead(addr)
6         # 判断当前指令是否是 "mov" 指令，并且目标操作数包含 "esi"
7         if GetMnem(addr) == "mov" and "esi" in GetOpnd(addr, 0):
8             # 返回找到的参数值
9             return GetOperandValue(addr, 1)
10        # 如果未找到，返回空字符串
11        return ""
12 # 用于获取以给定地址为起始的字符串
13 def get_string(addr):
14     # 初始化一个空字符串
15     out = ""
16     # 进入一个无限循环
17     while True:
18         # 如果当前地址处的字节不为0
19         if Byte(addr) != 0:
20             # 将当前字节转换为字符并追加到输出字符串
21             out += chr(Byte(addr))
22         else:
23             # 如果当前字节为0，跳出循环
24             break
25         # 移动到下一个地址
26         addr += 1
27     # 返回获取的字符串
28     return out
29 # 定义一个函数，用于解密字符串
30 def decrypt(data):
31     # 获取输入字符串的长度
32     length = len(data)
33     # 初始化计数器
34     c = 1
35     # 初始化输出字符串
36     o = ""
```

```
37 # 使用异或解密
38 while c < length:
39     # 对输入字符串的第一个字符和后续字符进行异或操作，并将结果转换为字符
40     o += chr(ord(data[0]) ^ ord(data[c]))
41     # 更新计数器，移动到下一个字符
42     c += 1
43 # 返回解密后的字符串
44 return o
45 # 打印提示信息
46 print("[*] Attempting to decrypt strings in malware")
47 # 遍历对特定地址 (0x00405BF0) 进行引用的交叉引用
48 for x in XrefsTo(0x00405BF0, flags=0):
49     # 查找函数参数
50     ref = find_function_arg(x.frm)
51     # 获取字符串
52     string = get_string(ref)
53     # 解密字符串
54     dec = decrypt(string)
55     # 打印引用地址和解密后的字符串
56     print("Ref Addr: 0x%x | Decrypted: %s" % (x.frm, dec))
```

### 3.6.4 函数判断

这段代码用于检查当前地址是否在一个函数中，并根据结果打印相应的消息。用于在本次实验中一些过于复杂的样本代码中快速定位其所在的函数信息。同样也是针对 Lab11-02 那种。

```
1 def main():
2     # 打印横幅 (banner)
3     printBanner()
4     # 获取当前函数的函数对象
5     functionObject = idaapi.get_func(here())
6     # 尝试执行以下代码块
7     try:
8         # 如果当前地址不在一个函数中 (functionObject为None)
9         if (functionObject is None):
10             # 打印消息并退出
11             print("This is not a function! Exiting!")
12             return -1
13     except:
14         # 如果出现异常，说明当前地址在一个函数中，继续执行
15         print("Looks like this is a function. Continuing...")
16 # 调用主函数
```

```
17 main()
```

### 3.6.5 定位返回地址块的位置及汇编代码

这段代码的目的是找到包含 `ret` 或 `retn` 指令的基本块，并获取该基本块的起始地址。在流程图数组中查找包含 `ret/retn` 指令的基本块的偏移量。这对于理解本次实验中 Lab11-02 这种复杂的病毒样本的控制流变化行为有很大作用。

```
1 def findReturnBlockOffset(flowChart):
2     # 初始化基本块偏移量
3     returnBlockOffset = 0
4     # 遍历流程图数组
5     while (returnBlockOffset < flowChart.size):
6         # 检查当前基本块的类型是否为返回 (ret) 类型
7         if flowChart[returnBlockOffset].type == idaapi.fcb_ret:
8             # 如果是，退出循环
9             break
10        else:
11            # 否则，继续检查下一个基本块
12            returnBlockOffset += 1
13    # 检查是否找到返回基本块
14    if (returnBlockOffset >= flowChart.size):
15        # 如果没有找到，打印错误消息并返回-1
16        print("Something is seriously wrong! Couldn't find the basic block that
17              returns execution to the caller!")
18        return -1
19    else:
20        # 如果找到，打印消息并返回基本块的偏移量
21        print("Found return block at offset %d in the flowChart array!" %
22              returnBlockOffset)
23        return returnBlockOffset
24    # 调用findReturnBlockOffset函数，获取包含ret/retn指令的基本块的偏移量
25    returnBlockOffset = findReturnBlockOffset(flowChart)
26    # 检查是否成功找到基本块
27    if (returnBlockOffset == -1):
28        return -1
29    # 获取包含ret/retn指令的基本块的起始地址
30    returnBlockStartEA = flowChart[returnBlockOffset].startEA
31    # 其他代码...
```

使用上述 Python 脚本在 IDAPro 中便可以辅助进行分析。经过测试全部与使用 IDA 静态分析的结果相同，编写较为成功。

## 4 实验结论及心得体会

### 4.1 实验结论

本次实验，通过结合使用静态分析工具和动态分析方法，对 Lab11 的五个恶意代码进行了全面的分析，并依次回答了书中的问题。并在其中重点分析了多种多样的病毒恶意行为。然后结合之前的分析和 IDA 的 String 模块编写了 Yara 规则，对病毒样本进行了成功检测并且时间性能较强。

最后为本次实验中编写了对应可以辅助分析的 IDAPython 脚本，并且成功实现了辅助功能。

**总的来说，实验非常成功。**

### 4.2 心得体会

本次实验，我收获颇丰，这不仅是课堂中的知识，更是我解决许多问题的能力，具体来说：

1. 首先我进一步熟练掌握到了将课堂中学习的病毒分析工具 IDAPro，用于更全面的动态分析；
2. 其中我们是精进了 OllyDBG 动态分析的能力，实现了反混淆的解密；
3. 另外我还发现了多种多样的病毒恶意行为，**可谓是十分大开眼界，其中发邮件的那个真的很厉害哈哈，佩服写病毒的人。**
4. 除此之外我还精进了我编写更为高效的 Yara 规则的能力，更好地帮助我识别和检测病毒；
5. 我还精进了使用 IDAPython 脚本对病毒进行辅助分析。

总的来说，本次通过亲自实验让我感受到了很多包括 IDA 和 IDAPython，也加深了我对病毒分析综合使用静态和动态分析的能力，最重要的是学习到了对各种 Windows 下病毒恶意行为的识别能力。培养了我对病毒分析安全领域的兴趣。我会努力学习更多的知识，辅助我进行更好的病毒分析。

感谢助教学姐审阅:)