

南開大學

恶意代码分析与防治技术课程实验报告

Lab11



学 院 网络空间安全学院
专 业 信息安全
学 号 2213041
姓 名 李雅帆
班 级 信安班

一、实验目的

1. 了解恶意代码的行为；
2. 进一步熟悉静态分析与动态分析的过程。

二、实验原理

通过静态分析和动态分析相结合的方式，研究恶意软件的行为特征和技术机制，深入了解其隐蔽性与危害性。实验中使用现代逆向工程工具（如 IDA Pro、Dependency Walker、Process Monitor 等），解析恶意代码的逻辑结构、运行机制和攻击方法，揭示其对系统和用户的潜在威胁。

1. 静态分析

静态分析是在不运行恶意代码的前提下，通过反汇编和反编译技术，解析其代码结构、函数调用及资源特性，以推测其行为和功能。

方法：使用工具（如 IDA Pro）分析恶意代码的导入表、导出表 and 关键字符串，发现潜在的恶意行为，如注册表操作、DLL 劫持或代码注入。

结合函数调用图，定位与持久化机制、数据窃取或恶意通信相关的功能模块。

通过检查可疑资源（如嵌入的配置文件或密钥），揭示恶意软件的内置配置信息。

2. 动态分析

动态分析是在受控环境中运行恶意代码，实时监控其行为，以验证静态分析的推测并发现运行时特征。

方法：利用 Process Monitor 捕捉恶意代码对系统文件和注册表的修改行为，分析其持久化机制；使用 Wireshark 抓取恶意代码的网络通信数据，确认其是否与远程服务器通信以及是否存在数据泄露；借助内存调试工具（如 OllyDbg）跟踪恶意代码的运行流程，深入分析其注入、劫持和窃密过程。

3. 恶意行为特性

（1）下载器与启动器：恶意代码通过网络接口（如 HTTP、FTP）下载其他恶意组件或使用注入技术秘密加载恶意代码。

（2）后门与远程控制：通过注册表操作、DLL 劫持或代码注入，实现反向 Shell 或 RAT 控制，提供攻击者对系统的长期控制。

（3）登录凭证窃取：使用动态链接库注入或 API 钩取技术，窃取用户登录凭据并存储或上传到远程服务器。

(4) 按键记录：利用 SetWindowsHookEx、GetAsyncKeyState 等 API，记录用户键盘输入，用于窃取敏感数据。

(5) 存活机制：恶意代码通过修改注册表（如 Run、AppInit_DLLs、Winlogon）或劫持系统二进制文件，实现自启动和驻留。

(6) 用户态 Rootkit：采用 IAT Hook、Inline Hook 等技术，隐藏自身行为或修改系统函数，实现攻击功能的隐匿性。

三、实验过程

• Lab11 部分

Lab 11-1

分析恶意代码 Lab11-01.exe。

问题

1. 这个恶意代码向磁盘释放了什么？

在 IDA Python 中打开文件，查看其字符串。可以看到其中有许多 Wlx 开头的字符串，同时也看到 gina.dll，结合注册表的路径及 sys 驱动文件，推测其通过修改注册表拦截 GINA，借助驱动完成对应功能。

Address	Length	Type	String
.rdata:0040000000000013	C		GetLastActivePopup
.data:0040000000000008	C		GinaDLL
.rdata:004000000000000D	C		KERNEL32.dll
.rdata:004000000000000C	C		MessageBoxA
.rdata:0040000000000025	C		Microsoft Visual C++ Runtime Library
.rdata:0040000000000025	C		R6002\r\n- floating point not loaded\r\n
.rdata:004000000000002A	C		R6008\r\n- not enough space for arguments\r\n
.rdata:004000000000002C	C		R6009\r\n- not enough space for environment\r\n
.rdata:004000000000002C	C		R6016\r\n- not enough space for thread data\r\n
.rdata:004000000000002D	C		R6017\r\n- unexpected multithread lock error\r\n
.rdata:0040000000000021	C		R6018\r\n- unexpected heap error\r\n
.rdata:0040000000000029	C		R6019\r\n- unable to open console device\r\n
.rdata:0040000000000035	C		R6024\r\n- not enough space for _onexit/_atexit table\r\n
.rdata:0040000000000026	C		R6025\r\n- pure virtual function call\r\n
.rdata:0040000000000035	C		R6026\r\n- not enough space for stdio initialization\r\n
.rdata:0040000000000035	C		R6027\r\n- not enough space for lowio initialization\r\n
.rdata:0040000000000025	C		R6028\r\n- unable to initialize heap\r\n
.rdata:004000000000001A	C		Runtime Error!\n\nProgram:
.rdata:004000000000000D	C		SING error\r\n
.data:0040000000000036	C		SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon
.data:0040000000000005	C		TGAD
.rdata:004000000000000E	C		TLOSS error\r\n
.data:004000000000000E	C		\\msgina32.dll
.rdata:0040000000000008	C		\b h
.rdata:0040000000000021	C		\r\nabnormal program termination\r\n
.rdata:0040000000000017	C		__GLOBAL_HEAP_SELECTED
.rdata:0040000000000015	C		__MSVCRT_HEAP_SELECT
.data:0040000000000006	C		y !
.data:004000000000000D	C		msgina32.dll
.rdata:004000000000000A	C		ppxxxx\b\b
.rdata:004000000000000F	C		runtime error
.rdata:0040000000000008	C		near32.dll

使用 Dependency Walker 查看其导入导出函数。可以看到与注册表操作相关的函数以及提取资源节的函数。

Dependency Walker - [Lab11-01]

File Edit View Options Profile Window Help

LAB11-01.EXE

KERNEL32.DLL

ADVAPI32.DLL

PI	Ordinal	Hint	Function	Entry Point
6	N/A	27 (0x001B)	CloseHandle	Not Bound
6	N/A	52 (0x0034)	CreateFileA	Not Bound
6	N/A	125 (0x007D)	ExitProcess	Not Bound
6	N/A	163 (0x00A3)	FindResourceA	Not Bound
6	N/A	170 (0x00AA)	FlushFileBuffers	Not Bound
6	N/A	178 (0x00B2)	FreeEnvironmentStringsA	Not Bound
6	N/A	179 (0x00B3)	FreeEnvironmentStringsW	Not Bound
6	N/A	182 (0x00B6)	FreeResource	Not Bound
6	N/A	185 (0x00B9)	GetACP	Not Bound
6	N/A	191 (0x00BF)	GetCPInfo	Not Bound
6	N/A	202 (0x00CA)	GetCommandLineA	Not Bound
6	N/A	247 (0x00F7)	GetCurrentProcess	Not Bound
6	N/A	262 (0x0106)	GetEnvironmentStrings	Not Bound
6	N/A	264 (0x0108)	GetEnvironmentStringsW	Not Bound
6	N/A	265 (0x0109)	GetEnvironmentVariableA	Not Bound
6	N/A	277 (0x0115)	GetFileType	Not Bound
6	N/A	282 (0x011A)	GetLastError	Not Bound
6	N/A	292 (0x0124)	GetModuleFileNameA	Not Bound

Dependency Walker - [Lab11-01]

File Edit View Options Profile Window Help

LAB11-01.EXE

KERNEL32.DLL

ADVAPI32.DLL

PI	Ordinal	Hint	Function	Entry Point
6	N/A	351 (0x015F)	RegCreateKeyExA	Not Bound
6	N/A	390 (0x0186)	RegSetValueExA	Not Bound

进一步的分析集中在名为“msgina1.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+dwSize]
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
push    ecx ; FILE *
mov     edx, [ebp+dwSize]
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_4@9
    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,
        0xF003Fu,
        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。

2. 这个恶意代码如何进行驻留？

```
.data:00408040 aBinary          db 'BINARY',0           ; DATA XREF: .data:lpTypeInfo
.data:00408047          align 4
.data:00408048 aRi              db 'RI',0Ah,0          ; DATA XREF: sub_401000:loc_401062fo
.data:0040804C ; CHAR ValueName[]
.data:0040804C ValueName      db 'GinaDLL',0           ; DATA XREF: sub_401000+3Efo
.data:00408054 ; CHAR SubKey[]
.data:00408054 SubKey          db 'SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon',0
.data:00408054          ; DATA XREF: sub_401000+17fo
.data:0040808A          align 4
.data:0040808C aDr              db 'DR',0Ah,0          ; DATA XREF: sub_401000+118fo
.data:00408090 ; char aMsgina32_dll[]
.data:00408090 aMsgina32_dll db 'msgina32.dll',0       ; DATA XREF: sub_401000+E6fo
.data:0040809D          align 10h
```

通过第一小题的分析我们可知，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\WindowsNT\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。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？

通过第一小题的分析我们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。

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 dll main 中实现。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]
1000100D      push     esi          ; lpProcName
1000100E      push     eax          ; hModule
1000100F      call     ds:GetProcAddress
10001015      test     eax, eax
10001017      jnz      short loc_1000103C
10001019      mov      ecx, esi
1000101B      shr      ecx, 10h
1000101E      jnz      short loc_10001034
10001020      push     esi
10001021      lea      edx, [esp+18h+var_10]
10001025      push     offset aD      ; "%d"
1000102A      push     edx            ; LPSTR
1000102B      call     ds:wprintfA
10001031      add      esp, 0Ch
10001034      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？

该恶意代码将窃取到的证书写入到创建在 C:\Windows\System32 目录下的 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？

注销或重启系统，再次登录，恶意代码即可获取用户的登录凭证。

Lab 11-2

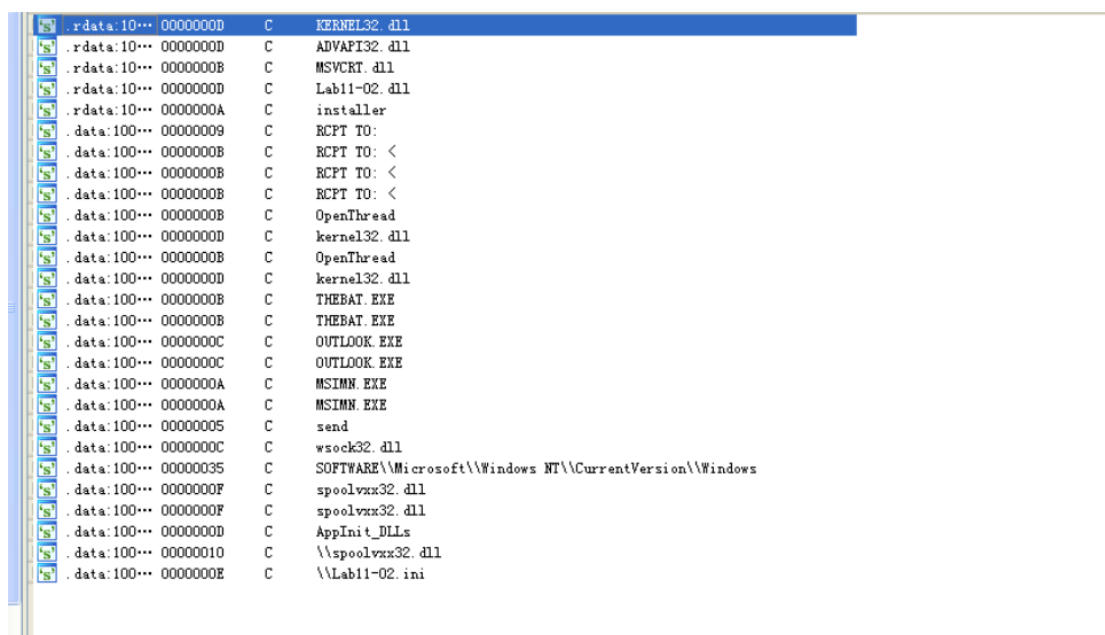
分析恶意代码 Lab11-02.dll。假设一个名为 Lab11-02.ini 的可疑文件与这个恶意代码一同被发现。

问题

1. 这个恶意 DLL 导出了什么？

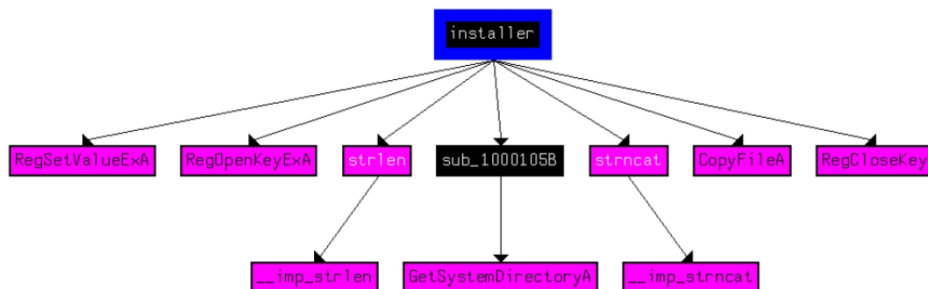
使用 IDA Pro 查看其字符串。可以看到 APPInit_DLLs 和一个注册表路径 SOFTWARE\Microsoft\WindowsNT\CurrentVersion\Windows，表明恶意代码使用 AppInit_DLLs 进行半永久安装。Lab11-02.ini 说明其使用了 ini 文件，而 RCPT 是一个命令，用来创建一个电子邮件的接收人。

然后查看 ini 文件，发现一个奇怪的字符串，推测已经加密。



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，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。

Name	Address	Ordinal
installer	000000001000158B	1
DllEntryPoint	00000000100017E9	



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。

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 rundll32 .exe Lab11-02.dll.installer 来启动程序。

执行该命令后，恶意程序便成功安装。观察到监控工具捕获的活动，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。

```

REG_setval          HKEY_LOCAL_MACHINE
BA register autorun  spoolvxx32.dll

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。

4. 这个安装的恶意代码如何驻留?

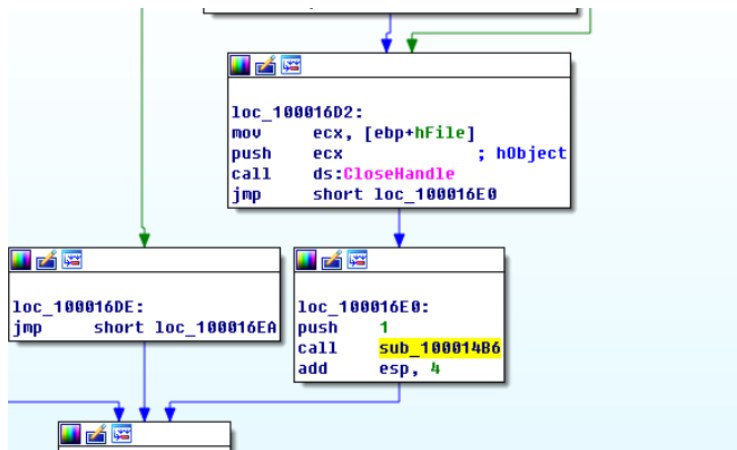
在进行恶意代码分析时, 特别是针对于其驻留机制, 我们观察到了一种特定的行为模式。该恶意代码通过修改 Windows 注册表来实现其持久性。具体而言, 它利用了 AppInit_DLLs 这一注册表键值。该机制的工作原理是, 恶意代码的安装程序 (installer) 将其代码复制到 Windows 的 system32 目录下。此外, 它还会在 AppInit_DLLs 的注册表项中添加一个新条目。

```
.text:10001591      lea     eax, [ebp+phkResult]
.text:10001594      push   eax                ; phkResult
.text:10001595      push   6                 ; sanDesired
.text:10001597      push   0                 ; ulOptions
.text:10001599      push   offset SubKey      ; "SOFTWARE\\Microsoft\\Windows NT\\Curren"...
.text:1000159E      push   80000002h         ; hKey
.text:100015A3      call   ds:RegOpenKeyExA
.text:100015A9      test    eax, eax
.text:100015AB      jnz     short loc_100015DD
.text:100015AD      push   offset aSpoolvxx32_dll ; "spoolvxx32.dll"
.text:100015B2      call   strlen
.text:100015B7      add     esp, 4
.text:100015BA      push   eax                ; cbData
.text:100015BB      push   offset Data        ; "spoolvxx32.dll"
.text:100015BD      push   1                 ; dwType
.text:100015C0      push   0                 ; Reserved
.text:100015C2      push   offset ValueName   ; "AppInit_DLLs"
.text:100015C9      mov     ecx, [ebp+phkResult]
.text:100015CC      push   ecx                ; hKey
.text:100015CD      call   ds:RegSetValueExA
.text:100015D3      mov     edx, [ebp+phkResult]
.text:100015D6      push   edx                ; hKey
```

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。

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。



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 callsub_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 ; ===== S U B R O U T I N E =====
.text:100014B6
.text:100014B6 ; Attributes: bp-based frame
.text:100014B6
.text:100014B6 sub_100014B6 proc near ; CODE XREF: DllMain(x,x,x)+024p
.text:100014B6
.text:100014B6 Buf1 = dword ptr -4
.text:100014B6 arg_0 = dword ptr 8
.text:100014B6
.text:100014B6 push ebp
.text:100014B6 mov ebp, esp
.text:100014B7 push ecx
.text:100014B8 cmp [ebp+arg_0], 0
.text:100014B9 jz loc_10001587
.text:100014C4 lea eax, [ebp+Buf1]
.text:100014C7 push eax ; int
.text:100014C8 push 0 ; hModule
.text:100014CA call sub_10001075
.text:100014CF add esp, 8
.text:100014D2 mov ecx, [ebp+Buf1]
.text:100014D5 push ecx ; Str
.text:100014D6 call sub_10001104
.text:100014D8 add esp, 4
.text:100014DE mov [ebp+Buf1], eax
.text:100014E1 cmp [ebp+Buf1], 0
.text:100014E5 jnz short loc_100014EC
.text:100014E7 jmp loc_10001587
.text:100014EC ;

```

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 M SIMN.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:100014F8      push     offset aThebat_exe ; "THEBAT.EXE"
.text:100014FD      call     strlen
.text:10001502      add     esp, 4
.text:10001505      push     eax                ; Size
.text:10001506      push     offset aThebat_exe_0 ; "THEBAT.EXE"
.text:1000150B      mov     eax, [ebp+Buf1]
.text:1000150E      push     eax                ; Buf1
.text:1000150F      call     memcmp
.text:10001514      add     esp, 0Ch
.text:10001517      test    eax, eax
.text:10001519      jz      short loc_10001561
.text:1000151B      push     offset aOutlook_exe ; "OUTLOOK.EXE"
.text:10001520      call     strlen
.text:10001525      add     esp, 4
.text:10001528      push     eax                ; Size
.text:10001529      push     offset aOutlook_exe_0 ; "OUTLOOK.EXE"
.text:1000152E      mov     ecx, [ebp+Buf1]
.text:10001531      push     ecx                ; Buf1
.text:10001532      call     memcmp
.text:10001537      add     esp, 0Ch
.text:1000153A      test    eax, eax
.text:1000153C      jz      short loc_10001561
.text:1000153E      push     offset aMsimn_exe ; "MSIMN.EXE"
.text:10001543      call     strlen
.text:10001548      add     esp, 4
.text:1000154B      push     eax                ; Size
.text:1000154C      push     offset aMsimn_exe_0 ; "MSIMN.EXE"

.text:10001561 loc_10001561:                ; CODE XREF: sub_100014B6+63↑j
                                ; sub_100014B6+86↑j
.text:10001561      call     sub_1000138D
.text:10001566      push     offset dword_10003484 ; int
.text:1000156B      push     offset sub_1000113D ; int
.text:10001570      push     offset aSend        ; "send"
.text:10001575      push     offset ModuleName ; "wssock32.dll"
.text:1000157A      call     sub_100012A3
.text:1000157F      add     esp, 10h
.text:10001582      call     sub_10001499
.text:10001587

```

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“RCPTTO:”。如果包含，它会在字符串中额外添加以下内容：“RCPTTO: <billy@malwareanalysisbook.com>\r\n”。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      push     ebp
.text:1000113D      mov      ebp, esp
.text:1000113E      sub      esp, 204h
.text:10001140      push     offset SubStr ; "RCPT TO:"
.text:10001140      mov      eax, [ebp+Str]
.text:10001141      push     eax ; Str
.text:10001142      call     strstr
.text:10001143      add      esp, 8
.text:10001144      test     eax, eax
.text:10001145      jz        loc_100011E4
.text:10001146      push     offset Str ; "RCPT TO: <"
.text:10001147      call     strlen
.text:10001148      add      esp, 4
.text:10001149      push     eax ; Size
.text:1000114A      push     offset aRcptTo_1 ; "RCPT TO: <"
.text:1000114B      lea      ecx, [ebp+Dst]
.text:1000114C      push     ecx ; Dst
.text:1000114D      call     memcpy
.text:1000114E      add      esp, 0Ch
.text:1000114F      push     101h ; Size
.text:10001150      push     offset byte_100034A0 ; Src
.text:10001151      push     offset aRcptTo_2 ; "RCPT TO: <"
.text:10001152      call     strlen
.text:10001153      add      esp, 4
.text:10001154      lea      edx, [ebp+eax+Dst]
.text:10001155      push     edx ; Dst
.text:10001156      call     memcpy
.text:10001157      add      esp, 0Ch
.text:10001158      push     offset Source ; ">\r\n"
.text:10001159      lea      eax, [ebp+Dst]
.text:1000115A      push     eax ; Dest
.text:1000115B      call     strcat

```

7. 哪个或者哪些进程执行这个恶意攻击, 为什么?

该恶意攻击仅针对 MSIML.EXE、THEBAT.EXE 和 OUTLOOK.EXE, 经比较, 以上三者均为电子邮件客户端。

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, 之后在使用 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。

```

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

```

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?

抓取系统的网络数据包即可看到一个假冒的服务器以及 Outlook Press 客户端。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。

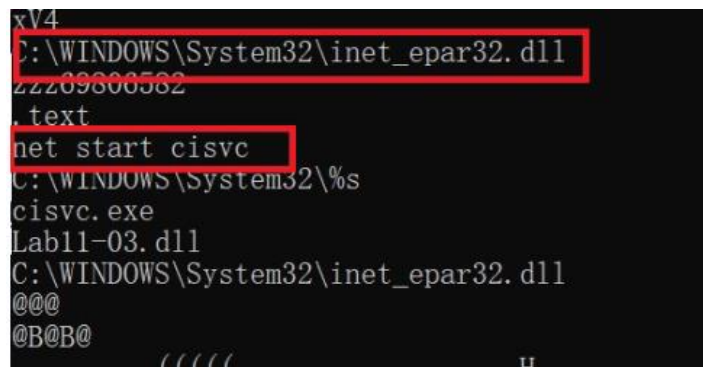
Lab 11-3

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

问题

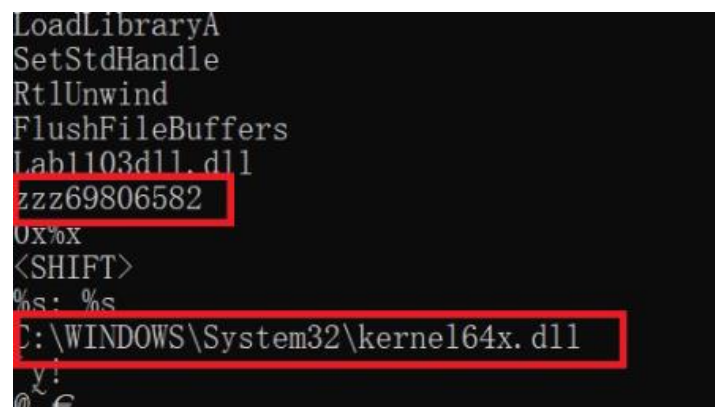
1. 使用基础的静态分析过程, 你可以发现什么有趣的线索?

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文件, 这暗示恶意代码可能在该位置创建一个 DLL 文件。



```
xV4
C:\WINDOWS\System32\inet_epar32.dll
zzz09806582
.net
net start cisvc
C:\WINDOWS\System32\%s
cisvc.exe
Lab11-03.dll
C:\WINDOWS\System32\inet_epar32.dll
@@@
@B@B@
((((H
```

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。



```
LoadLibraryA
SetStdHandle
RtlUnwind
FlushFileBuffers
Lab1103dll.dll
zzz69806582
0x%x
<SHIFT>
%s: %s
C:\WINDOWS\System32\kernel64x.dll
y!
@ f
```

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

2. 当运行这个恶意代码时, 发生了什么?

恶意代码在系统目录下创建了一个 inet_epar32.dll 的文件, 猜测这个 dll 文件和 lab11-03.dll 是同一个文件。之后, 恶意代码打开了 cisvc.exe, 但是并没有进行任何 写文件的操作。最后, 恶意代码通过命令 net start cisvc 来启动该索引服务。通过 Process Explorer 我们观察到 cisvc.exe 正在运行。

vmtoolsd.exe		12,004 K	3,752 K	1964 VMware Tools Core Ser...	VMware, Inc.
alg.exe		1,272 K	212 K	732 Application Layer Gat...	Microsoft Corporation
cisvc.exe	1.39	2,540 K	676 K	3512 Content Index service	Microsoft Corporation
cidaemon.exe		1,188 K	212 K	2608 Indexing Service filt...	Microsoft Corporation
lsass.exe	0.69	3,904 K	1,404 K	684 LSA Shell (Export Ver...	Microsoft Corporation
er.exe		27,784 K	13,752 K	1476 Windows Explorer	Microsoft Corporation

将 Lab11-03.dll 文件复制到系统目录下新创建的 inet_epar32.dll 文件中, 向 cisvc.exe 写入数据并启动服务, 记录键盘输入到 kernel64x.dll 文件中。

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

用 IDA Python 打开 Lab11-03.exe 文件, 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:004012D0 55
.text:004012D1 8B EC
.text:004012D3 81 EC 04 01 00 00
.text:004012D9 6A 00
.text:004012DB 68 08 91 40 00
.text:004012E0 68 08 91 40 00
.text:004012E5 FF 15 1C 80 40 00
.text:004012E8 68 9C 91 40 00
.text:004012F0 68 84 91 40 00
.text:004012F5 8D 85 FC FE FF FF
.text:004012FB 50
.text:004012FC E8 51 01 00 00
.text:00401301 83 C4 0C
.text:00401304 8D 8D FC FE FF FF
.text:0040130A 51
.text:0040130B E8 60 FD FF FF
.text:00401310 83 C4 04
.text:00401313 68 74 91 40 00

push    ebp
mov     ebp, esp
sub     esp, 104h
push    0 ; bFailIfExists
push    offset NewFileName ; "C:\\WINDOWS\\System32\\inet_epar32.dll"
push    offset ExistingFileName ; "Lab11-03.dll"
call    ds:CopyFileA
push    offset aCisvc_exe ; "cisvc.exe"
push    offset aCWindowsSyst_0 ; "C:\\WINDOWS\\System32\\%s"
lea     eax, [ebp+FileName]
push    eax ; char *
call    _sprintf
add     esp, 0Ch
lea     ecx, [ebp+FileName]
push    ecx ; lpFileName
call    sub_401070
add     esp, 4
push    offset aNetStartCisvc ; "net start cisvc"

```

接下来分析函数 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      0
               db 55h ; DATA XREF: sub_401070+19D1r
               ; sub_401070+1FF1w ...
unk_409031     db 89h ; ; DATA XREF: sub_401070+1AD1r
byte_409032     db 0E5h ; ; DATA XREF: sub_401070+1BD1r
byte_409033     db 81h ; ; DATA XREF: sub_401070+1CD1r
               db 0ECh ;
               db 40h ; @

```

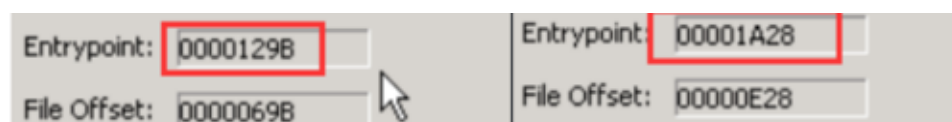
对 byte_409030 处的数据进行了查看，发现这里存储的是写入 cisvc.exe 的 shell-code。分析 shellcode 的结尾部分，我们发现了字符串 “C:\WINDOWS\System32\inet_eap32.dll” 和 “zzz69806582”，这表明 shellcode 加载了这个 DLL 文件，并调用了其中的导出函数。

```

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 ; N
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	. 8D83 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 F8	MOV DWORD PTR SS:[EBP-8],EAX	
01001821	. FF55 F8	CALL DWORD PTR SS:[EBP-8]	
01001824	. 89EC	MOV ESP,EBP	
01001826	. 5D	POP EBP	
01001827	. E9 6FF7FFFF	JMP cisvc.0100129B	

综上可知，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。**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`（即 `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`。

4. Lab11-03.dll 做了什么？

首先，该程序初始化了一个线程，在此线程中，它执行了对互斥量的判断。

```

.text:10001540 ; ===== SUBROUTINE =====
.text:10001540 ; Attributes: bp-based frame
.text:10001540
.text:10001540      public zzz69806582
.text:10001540 zzz69806582      proc near                ; DATA XREF: .rdata:off_10007C78↓o
.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
.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:1000146C      push     edx                ; unsigned __int8 *
.text:1000146D      call     _nbsnbcpy
.text:10001472      add      esp, 0Ch
.text:10001475      push     offset Name        ; "H2"
.text:1000147A      push     0                  ; bInheritHandle
.text:1000147C      push     1F0001h           ; dwDesiredAccess
.text:10001481      call     ds:OpenMutexA
.text:10001487      mov      [ebp+hObject], eax
.text:1000148D      cmp      [ebp+hObject], 0
.text:10001494      jz       short loc_1000149D
.text:10001496      push     0                  ; int
.text:10001498      call     _exit
.text:1000149D
.text:1000149D      loc_1000149D:                ; CODE XREF: StartAddress+84↑j
.text:1000149D      push     offset Name        ; "H2"
.text:100014A2      push     1                  ; bInitialOwner
.text:100014A4      push     0                  ; lpMutexAttributes
.text:100014A6      call     ds:CreateMutexA
.text:100014AC      mov      [ebp+hObject], eax
.text:100014B2      cmp      [ebp+hObject], 0
.text:100014B9      jnz      short loc_100014BD
.text:100014BB      jmp      short loc_10001530
.text:100014BD

```

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



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？

通过之前的分析可知，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。

• yara 规则

1. 编写依据

通过上述分析综合编写 Yara 检测规则。

2. yara 规则

```
rule lab1101exe{
strings:
    $string1 = "UN %s DM %s PW %s OLD %s" nocase
    $reg = "Software\\Microsoft\\Windows NT\\CurrentVersion\\Winlogon"
    $dll1 = "MSGina.dll"
    $dll2 = "GinaDLL"
condition:
    filesize < 100KB and uint16(0) == 0x5A4D and uint16(uint16(0x3C))==0x00004550 and 1
of them
}

rule lab1102dll{
strings:
    $reg1 = "SOFTWARE\\Microsoft\\Windows NT\\CurrentVersion\\Windows"
    $dll1 = "spoolvxx32.dll"
    $exe1 = "THEBAT.EXE"
    $exe2 = "OUTLOOK.EXE"
    $exe3 = "MSIMN.EXE"
condition:
    filesize < 100KB and uint16(0) == 0x5A4D and uint16(uint16(0x3C))==0x00004550 and 3
of them
}

rule lab1102ini{
strings:
    $string1 = "BNL"
condition:
    filesize < 100KB and 1 of them
}
```

```

}

rule lab1103exe{
strings:
    $dll11 = "C:\\WINDOWS\\System32\\inet_epar32.dll"
    $dll12 = "Lab11-03.dll"
    $exe1 = "cisvc.exe"
    $string1 = "net start cisvc"
    $func1 = "zzz69806582"
condition:
    filesize < 100KB and uint16(0) == 0x5A4D and uint16(uint16(0x3C))==0x00004550 and 3
of them
}

rule lab1103dll{
strings:
    $dll11 = "C:\\WINDOWS\\System32\\kernel64x.dll"
    $func1 = "VirtualAlloc"
    $func2 = "RtlUnwind"
    $sys1 = "Lab10-03.sys"
    $string1 = "<SHIFT>"
condition:
    filesize < 100KB and uint16(0) == 0x5A4D and uint16(uint16(0x3C))==0x00004550 and 3
of them
}

```

3. 运行结果

```

PS C:\Users\lenovo> cd D:\fan\homework3\恶意代码分析与防治技术\计算机病毒分析工具\yara
PS D:\fan\homework3\恶意代码分析与防治技术\计算机病毒分析工具\yara> .\yara64.exe -r .\Lab11.y .\Chapter_11L\
lab1102dll .\Chapter_11L\Lab11-02.dll
lab1102ini .\Chapter_11L\Lab11-02.ini
lab1101exe .\Chapter_11L\Lab11-01.exe
lab1103dll .\Chapter_11L\Lab11-03.dll
PS D:\fan\homework3\恶意代码分析与防治技术\计算机病毒分析工具\yara>

```

• IDA Python 自动化分析

1. 功能

遍历所有函数，排除库函数或简单跳转函数，当反汇编的助记符为 call 或者 jmp 且操作数为寄存器类型时，输出该行反汇编指令

2. 代码

```

import idutils
ea=idc.ScreenEA()
funcName=idc.GetFunctionName(ea)
func=idaapi.get_func(ea)
print("FuncName:%s"%funcName) # 获取函数名

```

```

print "Start:0x%x,End:0x%x" % (func.startEA, func.endEA) # 获取函数开始地址和结束地址
# 分析函数属性
flags = idc.GetFunctionFlags(ea)
if flags&FUNC_NORET:
    print "FUNC_NORET"
if flags & FUNC_FAR:
    print "FUNC_FAR"
if flags & FUNC_STATIC:
    print "FUNC_STATIC"
if flags & FUNC_FRAME:
    print "FUNC_FRAME"
if flags & FUNC_USERFAR:
    print "FUNC_USERFAR"
if flags & FUNC_HIDDEN:
    print "FUNC_HIDDEN"
if flags & FUNC_THUNK:
    print "FUNC_THUNK"
if not(flags & FUNC_LIB or flags & FUNC_THUNK):# 获取当前函数中 call 或者 jmp 的指令
    dism_addr = list(idautils.FuncItems(ea))
    for line in dism_addr:
        m = idc.GetMnem(line)
        if m == "call" or m == "jmp":
            print "0x%x %s" % (line, idc.GetDisasm(line))

```

3. 运行结果

```

Types applied to 0 names.
Using FLIRT signature: Microsoft VisualC 2-11/net runtime
Propagating type information...
Function argument information has been propagated
The initial autoanalysis has been finished.
FuncName:_main
Start:0x4011d0,End:0x401299
FUNC_FRAME
0x4011e5 call    ds:GetModuleHandleA
0x40120f call    sub_401080
0x401228 call    ds:GetModuleFileNameA
0x401237 call    _strrchr
0x401288 call    sub_401000

```

四、实验结论及心得体会

通过实验,我学会了如何使用工具(如 IDA Pro)对恶意软件进行静态分析,分解其逻辑结构并理解其行为特征。同时,我通过动态分析观察了恶意软件的实际运行行为,深入了解其持久化机制和数据窃取方式。实验让我认识到恶意软件如何通过修改系统文件、注入代码和注册表操作来隐藏自身和执行恶意操作。总

体来说，这次实验不仅提升了我的技术能力，还增强了我对系统安全和恶意软件防御的理解，强化了对恶意代码分析的系统性认知。