

南开大学

恶意代码分析与防治技术课程实验报告

Lab12



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. 了解恶意代码常见的攻击技术，例如 DLL 注入、进程替换、挂钩注入等，并学习应对这些技术的方法。
2. 了解恶意代码的隐蔽启动。

二、实验原理

恶意代码分析的核心是通过动态和静态分析相结合的方法，探究其隐藏的行为和功能。静态分析主要通过工具（如 IDA Pro）逆向还原二进制代码，了解其执行逻辑及数据流；动态分析则通过运行恶意代码并监控其行为（如文件操作、进程注入、网络通信）来观察其真实作用。恶意代码常利用 DLL 注入、进程替换和挂钩等技术执行恶意功能，并通过编码、加密等手段保护自身。通过构建检测规则（如 Yara 规则）和自动化分析脚本，可以快速识别特定恶意代码样本的特征，从而实现高效检测与防护。

三、实验过程

• Lab12

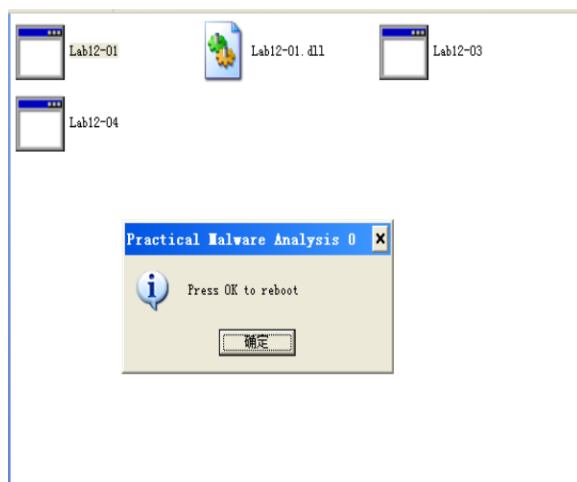
Lab 12-1

分析在 Lab12-01.exe 和 Lab12-01.dll 文件中找到的恶意代码，并确保在分析时这些文件在同一目录中。

问题

1. 在你运行恶意代码可执行文件时，会发生什么？

在进行虚拟机环境中的实验测试时，观察到的现象如下，实验结果显示，测试程序在特定的时间间隔内执行弹窗操作。



2. 哪个进程会被注入？

根据代码推断出目标注入的进程为” explorer.exe”。

```

.text:00401095
.text:00401095 loc_401095:                                     ; CODE XREF: sub_401000+58↑j
.text:00401095                                             ; sub_401000+76↑j
.text:00401095                                             ; size_t
.text:00401095 push    0Ch
.text:00401097 push    offset aExplorer_exe ; "explorer.exe"
.text:0040109C lea     ecx, [ebp+var_108]
.text:004010A2 push    ecx                ; char *
.text:004010A3 call    __strnicmp
.text:004010A8 add     esp, 0Ch
.text:004010AB test    eax, eax
.text:004010AD jnz     short loc_4010B6
.text:004010AF mov     eax, 1
.text:004010B4 jmp     short loc_4010C2

```

3. 你如何能够让恶意代码停止弹出窗口？

在 Process Explorer 工具中，用户可以观察到特定的进程。这些进程包括调用了 Lab12-01.dll 的实例。我推测通过重启 explorer.exe 进程，可以终止恶意代码触发的弹窗行为。

idaq.exe		86,340 K
explorer.exe	1.54	14,628 K
ProcessExplorer_v16.2...		11,840 K

4. 这个恶意代码样本是如何工作的？

通过分析我发现这个 exe 程序的主要作用是实现注入，而其真正的功能实际上包含在一个 dll 文件中。在对这个 dll 文件进行详细分析时，我们注意到其 dllMain 函数一开始就创建了一个线程，该线程运行的函数是 sub_10001030。

```

.text:100010A0      push     ebp
.text:100010A1      mov      ebp, esp
.text:100010A3      sub      esp, 8
.text:100010A6      cmp      [ebp+fdwReason], 1
.text:100010AA      jnz      short loc_100010C6
.text:100010AC      lea      eax, [ebp+ThreadId]
.text:100010AF      push     eax                ; lpThreadId
.text:100010B0      push     0                  ; dwCreationFlags
.text:100010B2      push     0                  ; lpParameter
.text:100010B4      push     offset sub_10001030 ; lpStartAddress
.text:100010B9      push     0                  ; dwStackSize
.text:100010BB      push     0                  ; lpThreadAttributes
.text:100010BD      call     ds:CreateThread
.text:100010C3      mov      [ebp+var_8], eax

```

进一步分析 sub_10001030 函数，我们发现它实际上是一个无限循环，循环的主要操作是创建线程并随后休眠一分钟。结合先前观察到的字符串和运行时发现的恶意样本行为，我们可以推断出，这个循环的行为是周期性地创建线程以显示相关内容。

```

.text:10001030      mov      eax, 1
.text:10001042      test     eax, eax
.text:10001044      jz       short loc_10001088
.text:10001046      mov      ecx, [ebp+var_18]
.text:10001049      push     ecx
.text:1000104A      push     offset aPracticalMalwa ; "Practical Malware Analysis %d"
.text:1000104F      lea      edx, [ebp+Parameter]
.text:10001052      push     edx                ; char *
.text:10001053      call     _sprintf
.text:10001058      add      esp, 0Ch
.text:1000105B      push     0                  ; lpThreadId
.text:1000105D      push     0                  ; dwCreationFlags
.text:1000105F      lea      eax, [ebp+Parameter]
.text:10001062      push     eax                ; lpParameter
.text:10001063      push     offset StartAddress ; lpStartAddress
.text:10001068      push     0                  ; dwStackSize
.text:1000106A      push     0                  ; lpThreadAttributes
.text:1000106C      call     ds:CreateThread
.text:10001072      push     0EA60h             ; dwMilliseconds
.text:10001077      call     ds:Sleep
.text:1000107D      mov      ecx, [ebp+var_18]
.text:10001080      add      ecx, 1
.text:10001083      mov      [ebp+var_18], ecx
.text:10001086      jmp      short loc_1000103D

```

该恶意代码执行 DLL 注入攻击，将 Lab12-01.dll 注入到 explorer.exe 中，使得每分钟弹出一个消息提示框并予以计数。

Lab 12-2

分析在 Lab12-02.exe 文件中找到的恶意代码。

观察 Lab12-02.exe 导入表包含创建进程的函数。

CreateProcessA、GetThreadContext 以及 SetThreadContext 暗示着这个程序创建新的进程，并修改进程中线程的上下文。

导入函数 ReadProcessMemory 和 WriteProcessMemory 告诉我们这个程序对进程内存空间进行了直接的读写。

导入函数 LockResource 和 SizeOfResource 告诉我们这个进程比较重要的数据可能保存在哪里。

00404028	GetProcAddress	KERNEL32
0040402C	GetModuleHandleA	KERNEL32
00404030	ReadProcessMemory	KERNEL32
00404034	GetThreadContext	KERNEL32
00404038	CreateProcessA	KERNEL32
0040403C	FreeResource	KERNEL32
00404040	SizeOfResource	KERNEL32
00404044	LockResource	KERNEL32
00404048	LoadResource	KERNEL32
0040404C	FindResourceA	KERNEL32
00404050	GetSystemDirectoryA	KERNEL32
00404054	Sleep	KERNEL32
00404058	GetCommandLineA	KERNEL32
0040405C	GetVersion	KERNEL32
00404060	ExitProcess	KERNEL32
00404064	TerminateProcess	KERNEL32
00404068	GetCurrentProcess	KERNEL32
.....

发现了对内存读写操作的 API 函数，以及对资源的操作。通过深入分析 createProcess 函数，发现其第四个参数指示创建的进程处于挂起状态，直到主进程调用此函数后才启动。进一步的分析揭示了 GetThreadContext 函数的调用，表明程序访问了进程上下文。

.text:0040113D E8 4E 04 00 00	call	_memset
.text:00401142 83 C4 0C	add	esp, 0Ch
.text:00401145 8D 55 E8	lea	edx, [ebp+ProcessInformation]
.text:00401148 52	push	edx ; lpProcessInformation
.text:00401149 8D 45 A4	lea	eax, [ebp+StartupInfo]
.text:0040114C 50	push	eax ; lpStartupInfo
.text:0040114D 6A 00	push	0 ; lpCurrentDirectory
.text:0040114F 6A 00	push	0 ; lpEnvironment
.text:00401151 6A 04	push	4 ; dwCreationFlags
.text:00401153 6A 00	push	0 ; bInheritHandles
.text:00401155 6A 00	push	0 ; lpThreadAttributes
.text:00401157 6A 00	push	0 ; lpProcessAttributes
.text:00401159 6A 00	push	0 ; lpCommandLine
.text:0040115B 8B 40 08	mov	ecx, [ebp+lpApplicationName]
.text:0040115E 51	push	ecx ; lpApplicationName
.text:0040115F FF 15 38 40 40 00	call	ds:CreateProcessA
.text:00401165 85 C0	test	eax, eax
.text:00401167 0F 84 A6 01 00 00	jz	loc_401313

为了更精确地分析，程序添加了一个结构体到 IDA Pro 的结构窗口。

```

push      4 ; flProtect
push      1000h ; flAllocationType
push      2CCh ; dwSize
push      0 ; lpAddress
call      ds:VirtualAlloc
mov       [ebp+lpContext], eax
mov       edx, [ebp+lpContext]
mov       dword ptr [edx], 10007h
mov       eax, [ebp+lpContext]
push      eax ; lpContext
mov       ecx, [ebp+ProcessInformation.hThread]
push      ecx ; hThread
call      ds:GetThreadContext

```

通过这种方法，识别出 0A4h 是上下文结构体中的一个参数，它通过引用这个参数来获取 ebx 寄存器的值，该寄存器包含指向进程的 PEB 块的指针。

CONSOLE_SELECTION_INFO	struct _CONSOLE_SELECTION_INFO
CONTENTRESTRICTION	struct tagCONTENTRESTRICTION
CONTEXT	struct _CONTEXT
CONTEXTMENUITEM	struct _CONTEXTMENUITEM

接下来，程序中的操作包括将 ebx 的值放入 ecx，以及修改 ecx 以指向程序的入口地址。

```

004011D1 call    ds:ReadProcessMemory
004011D7 push    offset ProcName ; "NtUnmapViewOfSection"
004011DC push    offset ModuleName ; "ntdll.dll"
004011E1 call    ds:GetModuleHandleA
004011E7 push    eax                ; hModule
004011E8 call    ds:GetProcAddress
004011EE mov     [ebp+var_64], eax
004011F1 cmp     [ebp+var_64], 0
004011F5 jnz     short loc_4011FE

```

分析 GetProcAddress 函数，发现它用于获取 NtUnmapViewOfSection 函数的地址，以便释放新创建进程的内存空间并填充恶意代码。通过字符串比较分析，观察到程序检测 MZ 和 PE 字符串以判断 PE 文件。

```

mov     eax, [ebp+var_4]
mov     ecx, [ebp+lpBuffer]
add     ecx, [eax+3Ch]
mov     [ebp+var_8], ecx
mov     edx, [ebp+var_8]
cmp     dword ptr [edx], 'EP'
jnz     loc_401319

```

分析显示程序调整 ecx 以指向 PE 文件头，并计算映像基址，然后使用 VirtualAllocEx 函数分配内存。

```

loc_401269:
mov     ecx, [ebp+var_8]
xor     edx, edx
mov     dx, [ecx+6]
cmp     [ebp+var_70], edx
jge     short loc_4012B9

```

进一步的分析揭示了程序在复制 PE 文件的可执行段到挂起进程中。通过分析 image_dos_header、image_nt_headers 和 image_section_header 结构，能够更清楚地看到 PE 文件结构中的数据被复制到另一个内存空间

```

mov     eax, [ebp+var_4]
mov     ecx, [ebp+lpBuffer]
add     ecx, [eax+3Ch]
mov     edx, [ebp+var_70]
imul    edx, 28h
lea     eax, [ecx+edx+0F8h]
mov     [ebp+var_74], eax
push    0 ; lpNumberOfBytesWritten
mov     ecx, [ebp+var_74]
mov     edx, [ecx+10h]

```

SetThreadContext 函数用于修改 eax 寄存器的数据，将其设置为可执行文件的加载入口点。随后，ResumeThread 函数的调用表明成功地将 createThread 函数创建的进程替换为另一个进程 A。

```

add     ecx, 8
push    ecx ; lpBaseAddress
mov     edx, [ebp+ProcessInformation.hProcess]
push    edx ; hProcess
call    ds:WriteProcessMemory
mov     eax, [ebp+var_8]
mov     ecx, [ebp+lpBaseAddress]
add     ecx, [eax+28h]
mov     edx, [ebp+lpContext]
mov     [edx+000h], ecx
mov     eax, [ebp+lpContext]
push    eax ; lpContext
mov     ecx, [ebp+ProcessInformation.hThread]
push    ecx ; hThread
call    ds:SetThreadContext
mov     edx, [ebp+ProcessInformation.hThread]
push    edx ; hThread
call    ds:ResumeThread
jmp     short loc_40130B

```

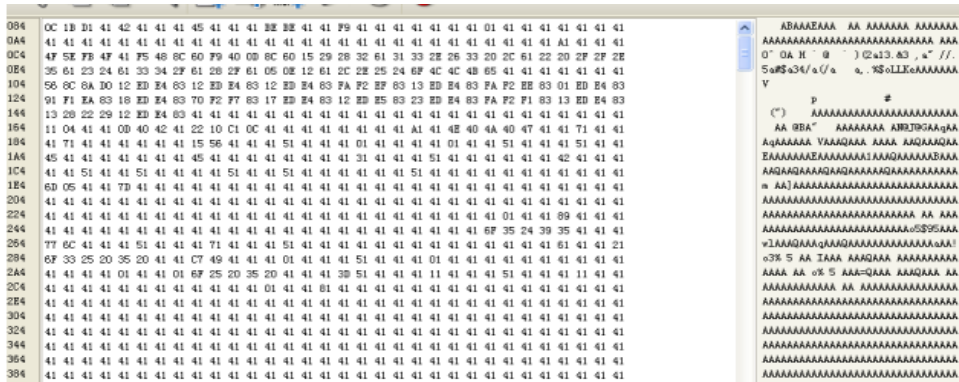
为了确定进程 A 的身份，分析发现进程名称是通过特定的内存偏移确定的。

```

push    0 ; lpCommandLine
mov     ecx, [ebp+lpApplicationName]
push    ecx ; lpApplicationName
call    ds:CreateProcess

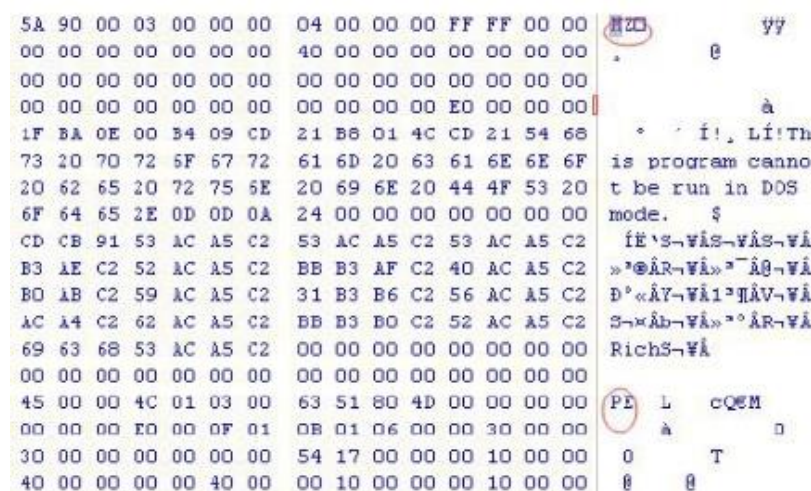
```

进一步分析表明，svchost.exe 文件被作为参数传递给了一个函数，该函数构造了 svchost.exe 的路径，表明有进程替换了 svchost.exe。



最后，通过 Resource Hacker 检查资源中的内容，发现资源数据可能经过加密处理。继续分析发现了异或操作，用于解密资源中的数据。通过这种方法能够解析出原始文本，完成了文件的分析。

继续分析，发现其调用 sub_401000 函数。步入分析，可以看到其中有异或操作，对象为 arg_8，向前查找确定为 41h。接着即为解密操作。使用 WinHex 软件将从资源节中提取出的 exe 文件进行解密分析，可以看到 MZ、PE 等文件标识。



问题

1. 这个程序的目的是什么？

通过上述分析可知这个程序的目的是隐蔽启动另一个程序。

2. 启动器恶意代码是如何隐蔽执行的？

使用进程替换，将 svchost.exe 替换为 Lab12-02.exe，来隐蔽执行本程序。即先将正常程序挂起，然后逐模块替换。

3. 恶意代码的负载存储在哪里？

保存在这个程序资源节中的恶意有效载荷是经过 XOR 编码过的。这个解码例程可以在 sub_40132C 处找到，而 XOR 字节在 0x0040141B 处可以找到。

4. 恶意负载是如何被保护的？

上述恶意代码的负载是经过异或编码的。用于解码的函数是 sub_40132C，它会调用 sub_401000 将加密的负载与 0x41 进行异或操作，达到解密的目的。

5. 字符串列表是如何被保护的？

在 sub_401000 函数中进行的异或操作被用于对字符串进行编码。这种编码方法是通过对字符串中的每个字符与 41h 进行异或运算，从而对原始数据进行编码。这是一种常见的简单加密方法，用于隐藏或保护数据的真实内容，以防止未经授权的访问或理解。

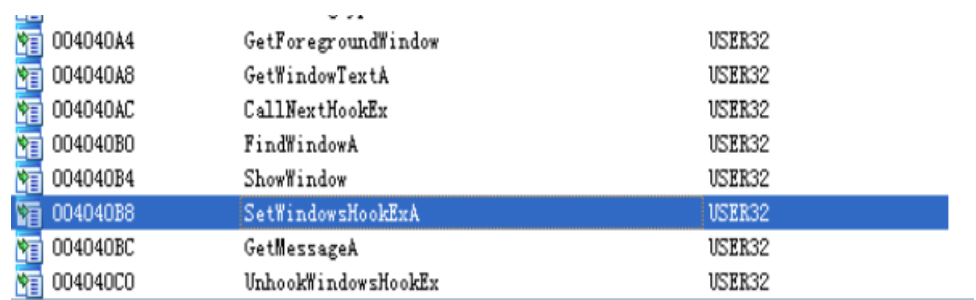
Lab 12-3

分析在 Lab 12-2 实验过程中抽取出的恶意代码样本，或者使用 Lab12-03.exe 文件

问题

1. 这个恶意负载的目的是什么？

查看文件的导入导出函数，可以看到 SetWindowsHookExA 等函数，用于应用程序挂钩或监控 Windows 内部事件。



004040A4	GetForegroundWindow	USER32
004040A8	GetWindowTextA	USER32
004040AC	CallNextHookEx	USER32
004040B0	FindWindowA	USER32
004040B4	ShowWindow	USER32
004040B8	SetWindowsHookExA	USER32
004040BC	GetMessageA	USER32
004040C0	UnhookWindowsHookEx	USER32

SetWindowsHookExA 函数的 idhook 参数指明了钩子的类型。通过查阅 MSDN 文档，可以了解到参数值 D 用于监控键盘消息。另一个重要参数 lpfn 指向了 hook 的地址，这表明 fn 函数启动了键盘监控。此函数的作用是接收击键记录，从而允许程序捕获用户的键盘输入。程序中还调用了 GetMessageA 函数。这个函数的调用是必需的，因为 Windows 系统不会主动发送消息给进程。因此，GetMessageA 函数的存在确保了程序能够接收到 Windows 消息队列中的消息，从而有效地执行其键盘监控功能。

该程序的主要目的是利用 `setwindowhook` 函数进行键盘记录。这种行为通常与恶意软件相关，其目的是记录用户的击键，可能用于窃取敏感信息，如密码和其他个人资料。

2. 恶意负载是如何注入自身的？

该程序使用了挂钩技术进行注入，目的是偷取按键记录。挂钩注入是一种常见的恶意软件行为，它通过注入代码或修改系统进程的正常行为来监控或更改用户的输入输出。

3. 这个程序还创建了哪些其他文件？

```
.text:004010C7 ; ===== S U B R O U T I N E =====
.text:004010C7
.text:004010C7 ; Attributes: bp-based frame
.text:004010C7
.text:004010C7 ; int __cdecl sub_4010C7(int Buffer)
.text:004010C7 sub_4010C7 proc near ; CODE XREF: Fn+21↑p
.text:004010C7
.text:004010C7 var_C = dword ptr -0Ch
.text:004010C7 hFile = dword ptr -8
.text:004010C7 NumberOfBytesWritten= dword ptr -4
.text:004010C7 Buffer = dword ptr 8
.text:004010C7
.text:004010C7 push ebp
.text:004010C8 mov ebp, esp
.text:004010CA sub esp, 0Ch
.text:004010CD mov [ebp+NumberOfBytesWritten], 0
.text:004010D4 push 0 ; hTemplateFile
.text:004010D6 push 80h ; dwFlagsAndAttributes
.text:004010D8 push 4 ; dwCreationDisposition
.text:004010DA push 0 ; lpSecurityAttributes
.text:004010DC push 2 ; dwShareMode
.text:004010DE push 40000000h ; dwDesiredAccess
.text:004010E0 push offset FileName ; "practicalmalwareanalysis.log"
.text:004010E2 call ds:CreateFileA
.text:004010E4 mov [ebp+hFile], eax
.text:004010E6 cmp [ebp+hFile], 0FFFFFFFFh
.text:004010E8 jnz short loc_4010FF
.text:004010EA jmp loc_40143D
.text:004010FF : -----
```

该恶意软件在其执行过程中创建了一个名为“practicalmalwareanalysis.log”的日志文件。

Lab 12-4

分析在 Lab12-04.exe 文件中找到的恶意代码。

使用 `Strings` 查看文件字符串，可以看到一些与注册表相关的字符串，还可以看到 `winlogon.exe` 等字符串。

```

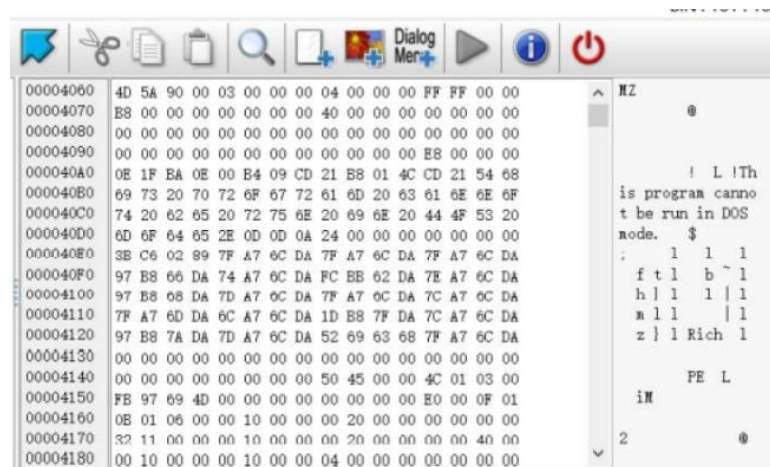
winlogon.exe
<not real>
SeDebugPrivilege
sfc_os.dll
\system32\wupdmgr.exe
%s%s
BIN
#101
EnumProcessModules
psapi.dll
GetModuleBaseNameA
psapi.dll
EnumProcesses
psapi.dll
\system32\wupdmgr.exe
%s%s
\winup.exe

```

查看导入导出函数，可以看到用于创建远程线程的 `CreateRemoteThread` 和用于操作资源的 `LoadResource` 函数。

00402000	OpenProcessToken	ADVAPI32
00402004	LookupPrivilegeValueA	ADVAPI32
00402008	AdjustTokenPrivileges	ADVAPI32
00402010	GetProcAddress	KERNEL32
00402014	LoadLibraryA	KERNEL32
00402018	WinExec	KERNEL32
0040201C	WriteFile	KERNEL32
00402020	CreateFileA	KERNEL32
00402024	SizeofResource	KERNEL32
00402028	CreateRemoteThread	KERNEL32
0040202C	FindResourceA	KERNEL32
00402030	GetModuleHandleA	KERNEL32

使用 Resource Hacker 查看资源，可以看到 MZ、PE 等信息，确定为一个 PE 文件。



运行文件，可以看到 Process Monitor 捕获到恶意代码行为。其对 Temp 文件夹进行一定操作，之后更新了 wupdmgr.exe 文件。经过对比分析，确认与前面资源节中文件完全相同。

双击运行该程序，发现它打开了一个网页，从 Procmon 中我们看到该恶意代码创建了 文件 winup.exe，并且覆盖了 wupdmgr.exe 的 Windows 更新二进制文件。比较恶意代码释放的 wupdmgr.exe 和 在上面资源节中提取的 BIN 文件，发现它们是相同的。且使用 netcat 监测 80 端口，我们可以发现恶意代码试图从 www.practicalmalwareanalysis.com 中获取 updater.exe。

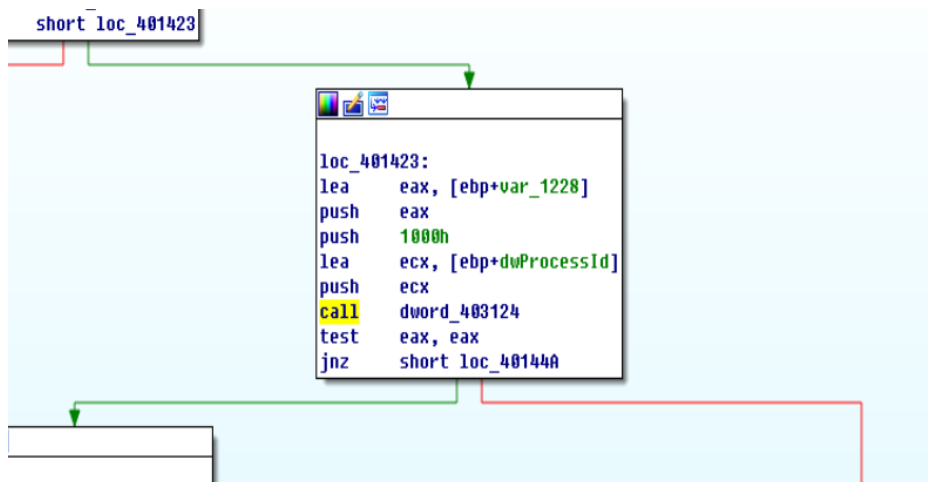


No.	Time	Source	Destination	Protocol	Length	Info
24	3.36222400	192.168.1.105	52.137.90.34	HTTP	269	GET / HTTP/1.1
27	3.42888100	192.168.1.105	192.0.78.25	HTTP	261	GET /updater.exe HTTP/1.1
29	3.61583300	192.0.78.25	192.168.1.105	HTTP	450	HTTP/1.1 301 Moved Permanently
30	3.61744500	52.137.90.34	192.168.1.105	HTTP	404	HTTP/1.1 302 Redirect (temporary)
39	3.69096300	192.168.1.105	192.0.78.25	HTTP	257	GET /updater.exe HTTP/1.1
43	3.80753100	192.0.78.25	192.168.1.105	HTTP	450	[TCP Retransmission] HTTP/1.1

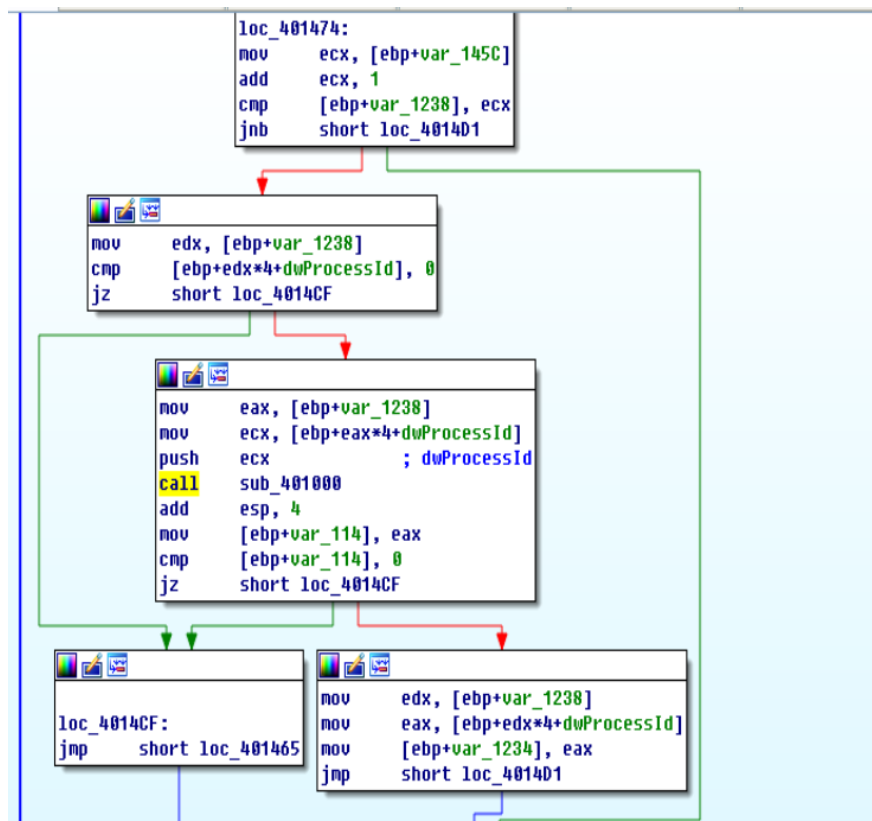
下面用 IDA 打开 Lab12-04.exe，从 main 函数开始分析。可以看到其通过 LoadLibraryA 和 GetProcAddress 解析三个函数，并将三个函数指针分别保存在 dword_40312c。

```
.text:00401396      mov     [ebp+var_1234], 0
.text:004013A0      mov     [ebp+var_122C], 0
.text:004013AA      push   offset ProcName ; "EnumProcessModules"
.text:004013AF      push   offset aPsapi_dll ; "psapi.dll"
.text:004013B4      call    ds:LoadLibraryA
.text:004013BA      push   eax ; hModule
.text:004013BB      call    ds:GetProcAddress
.text:004013C1      mov     dword_40312C, eax
.text:004013C6      push   offset aGetmodulebasen ; "GetModuleBaseNameA"
.text:004013CB      push   offset aPsapi_dll_0 ; "psapi.dll"
.text:004013D0      call    ds:LoadLibraryA
.text:004013D6      push   eax ; hModule
.text:004013D7      call    ds:GetProcAddress
.text:004013DD      mov     dword_403128, eax
.text:004013E2      push   offset aEnumprocesses ; "EnumProcesses"
.text:004013E7      push   offset aPsapi_dll_1 ; "psapi.dll"
.text:004013EC      call    ds:LoadLibraryA
.text:004013F2      push   eax ; hModule
.text:004013F3      call    ds:GetProcAddress
.text:004013F9      mov     dword_403124, eax
```

可以看到其使用 muEnumProcess 列出当前进程，返回 PID，保存到 dwProcessID 中。



接着为一个循环结构，遍历 PID，将其作为参数传递给 sub_401000。步入分析，可以看到 Str1 和 Str2 两个字符串。接着调用函数，将其返回值进行比较。



分析 sub_401174 函数，可以看到其调用 sub_4010FC，继续步入分析，可以看到其调用 lookupPrivilegeValueA 函数用于提升权限。

```

.text:00401174 ; ===== S U B R O U T I N E =====
.text:00401174
.text:00401174 ; Attributes: bp-based frame
.text:00401174
.text:00401174 ; int __cdecl sub_401174(DWORD dwProcessId)
.text:00401174 sub_401174 proc near ; CODE XREF: _main+19B↑p
.text:00401174
.text:00401174 var_C = dword ptr -0Ch
.text:00401174 hProcess = dword ptr -8
.text:00401174 var_4 = dword ptr -4
.text:00401174 dwProcessId = dword ptr 8
.text:00401174
.text:00401174 push ebp
.text:00401175 mov ebp, esp
.text:00401177 sub esp, 0Ch
.text:0040117A mov [ebp+var_4], 0
.text:00401181 mov [ebp+hProcess], 0
.text:00401188 mov [ebp+var_C], 0
.text:0040118F push offset aSeDebugprivile ; "SeDebugPrivilege"
.text:00401194 call sub_4010FC
.text:00401199 test eax, eax
.text:0040119B jz short loc_4011A1
.text:0040119D xor eax, eax
.text:0040119F jmp short loc_4011F8
.text:004011A1 ; -----

```

返回分析，可以看到其调用 LoadLibraryA 函数用于装载 sfc_os.dll，然后借助 GetProcAddress 函数获取其中编号为 2 的函数地址保存到 lpStartAddress 中，调用 OpenProcess 打开 winLogon.exe，将其句柄保存到 hProcess。

```

.text:004011A1 ; -----
.text:004011A1
.text:004011A1 loc_4011A1: ; CODE XREF: sub_401174+27↑j
.text:004011A1 push 2 ; lpProcName
.text:004011A3 push offset LibFileName ; "sfc_os.dll"
.text:004011A8 call ds:LoadLibraryA
.text:004011AE push eax ; hModule
.text:004011AF call ds:GetProcAddress
.text:004011B5 mov lpStartAddress, eax
.text:004011BA mov eax, [ebp+dwProcessId]
.text:004011BD push eax ; dwProcessId
.text:004011BE push 0 ; bInheritHandle
.text:004011C0 push 1F0FFFh ; dwDesiredAccess
.text:004011C5 call ds:OpenProcess
.text:004011CB mov [ebp+hProcess], eax
.text:004011CE cmp [ebp+hProcess], 0
.text:004011D2 jnz short loc_4011D8
.text:004011D4 xor eax, eax
.text:004011D6 jmp short loc_4011F8
.text:004011D8 ; -----

```

接着该调用 CreateRemoteThread 函数，其 hProcess 参数是 winlogon.exe 的句柄。004011de 处的 lpStartAddress 是 sfc_os.dll 中序号为 2 的函数的指针，负责向 winlogon.exe 注入一个线程，而该线程就是 sfc_os.dll 的序号为 2 的函数。


```

.text:004011D8 ; -----
.text:004011D8
.text:004011D8 loc_4011D8: ; CODE XREF: sub_401174+5E↑j
.text:004011D8      push    0 ; lpThreadId
.text:004011DA      push    0 ; dwCreationFlags
.text:004011DC      push    0 ; lpParameter
.text:004011DE      mov     ecx, lpStartAddress
.text:004011E4      push    ecx ; lpStartAddress
.text:004011E5      push    0 ; dwStackSize
.text:004011E7      push    0 ; lpThreadAttributes
.text:004011E9      mov     edx, [ebp+hProcess]
.text:004011EC      push    edx ; hProcess
.text:004011ED      call    ds:CreateRemoteThread
.text:004011F3      mov     eax, 1
.text:004011F8
.text:004011F8 loc_4011F8: ; CODE XREF: sub_401174+2B↑j
.text:004011F8      ; sub_401174+62↑j
.text:004011F8      mov     esp, ebp
.text:004011FA      pop     ebp
.text:004011FB      retn
.text:004011FB sub_401174      endp
.text:004011FB
.text:004011FC ; ===== S U B R O U T I N E =====

```

接着调用资源节，写入到 C:\windows\system32\wupdmgr.exe。

```

.text:0040126C      lea     eax, [ebp+Buffer]
.text:00401272      push    eax ; lpBuffer
.text:00401273      call    ds:GetWindowsDirectoryA
.text:00401279      push    offset aSystem32Wupdmgr ; "\\system32\wupdmgr.exe"
.text:0040127E      lea     ecx, [ebp+Buffer]
.text:00401284      push    ecx
.text:00401285      push    offset Format ; "%S%S"
.text:0040128A      push    10Eh ; Count
.text:0040128F      lea     edx, [ebp+Dest]
.text:00401295      push    edx ; Dest
.text:00401296      call    ds:_snprintf
.text:0040129C      add     esp, 14h
.text:0040129F      push    0 ; lpModuleName
.text:004012A1      call    ds:GetModuleHandleA
.text:004012A7      mov     [ebp+hModule], eax
.text:004012AA      push    offset Type ; "BIN"
.text:004012AF      push    offset Name ; "#101"
.text:004012B4      mov     eax, [ebp+hModule]
.text:004012B7      push    eax ; hModule
.text:004012B8      call    ds:FindResourceA
.text:004012BE      mov     [ebp+hResInfo], eax
.text:004012C4      mov     ecx, [ebp+hResInfo]
.text:004012CA      push    ecx ; hResInfo
.text:004012CB      mov     edx, [ebp+hModule]
.text:004012CE      push    edx ; hModule
.text:004012CF      call    ds:LoadResource
.text:004012D5      mov     [ebp+lpBuffer], eax
.text:004012D8      mov     eax, [ebp+hResInfo]
.text:004012DE      push    eax ; hResInfo
.text:004012DF      mov     ecx, [ebp+hModule]
.text:004012E2      push    ecx ; hModule
.text:004012E3      call    ds:SizeofResource
.text:004012E9      mov     [ebp+nNumberOfBytesToWrite], eax
.text:004012EF      push    0 ; hTemplateFile
.text:004012F1      push    0 ; dwFlagsAndAttributes
.text:004012F3      push    2 ; dwCreationDisposition
.text:004012F5      push    0 ; lpSecurityAttributes
.text:004012F7      push    1 ; dwShareMode
.text:004012F9      push    40000000h ; dwDesiredAccess
.text:004012FE      lea     edx, [ebp+Dest]
.text:00401304      push    edx ; lpFileName
.text:00401305      call    ds:CreateFileA
.text:00401308      mov     [ebp+hFile], eax
.text:00401311      push    0 ; lpOverlapped

```

可以看到恶意代码通过 WinExec 来启用已经被改写过的 wupdmgr.exe。在 0040133c 处可以看到 push 0 指令，将 0 作为 uCmdShow 参数值来启动，从而实现隐藏程序窗口的目的。

```
.text:00401335      push    ecx                ; hObject
.text:00401336      call   ds:CloseHandle
.text:0040133C      push    0                 ; uCmdShow
.text:0040133E      lea     edx, [ebp+Dest]
.text:00401344      push    edx                ; lpCmdLine
.text:00401345      call   ds:WinExec
```

问题

1. 位置 0x401000 的代码完成了什么功能？

该程序检查给定的 PID 是否为 winlogon.exe 进程。

2. 代码注入了哪个进程？

程序最终实现了向 winlogon.exe 进程的注入。

3. 使用 LoadLibraryA 装载了哪个 DLL 程序？

sfc_os.dll 被用于禁用 Windows 的文件保护机制。Windows 文件保护是 Microsoft Windows 操作系统的一个特性，旨在防止替换关键 Windows 系统文件。这个机制通常会阻止非授权的程序修改系统关键文件。在本题目中，恶意软件通过提升权限后，加载并利用 'sfc_os.dll 中的特定函数，目的是绕过这一保护机制。

4. 传递给 CreateRemoteThread 调用的第 4 个参数是什么？

CreateRemoteThread 函数主要用于创建一个线程，在这个案例中，它被用来在 winlogon.exe 进程中创建一个新线程。该函数的第四个参数，即 lpStartAddress，是一个函数指针。

5. 二进制主程序释放出了哪个恶意代码？

恶意代码释放文件，将复制 wupdmgr.exe 到 %TEMP% 目录，然后覆盖原来的 wupdmgr.exe。

6. 释放出恶意代码的目的是什么？

恶意软件向 winlogon.exe 注入一个远程线程，并调用 sfc_os.dll 中的一个导出函数（序号为 2 的 SfcTerminateWatcherThread），在下次系统启动前暂时禁用 Windows 的文件保护机制。由于这个函数必须在 winlogon.ex

e 进程中运行，因此 CreateRemoteThread 调用是必要的。此外，恶意软件还通过下载新的二进制文件来更新自身。

• Yara 规则

1. 编写依据

通过 Strings 工具观察字符串，结合该.dll 和功能性函数以及文件大小和 PE 文件特征进行综合编写 Yara 检测规则。

2. yara 规则

```
rule lab1201exe{
strings:
    $dll1 = "Lab12-01.dll"
    $string1 = "GetModuleBaseNameA"
    $dll2 = "psapi.dll"
    $string2 = "EnumProcessModules"
condition:
    filesize < 200KB and uint16(0) == 0x5A4D and uint16(uint16(0x3C)) == 0x00004550 and
all of them
}

rule lab1202exe{
strings:
    $reg1 = "AAAqAAApAAAsAAArAAAUAAAtAAAwAAAvAAAYAAAXAAA"
    $dll1 = "spoolvxx32.dll"
    $exe1 = "svchost.exe"
    $string = "NtUnmapViewOfSection"
condition:
    filesize < 200KB and uint16(0) == 0x5A4D and uint16(uint16(0x3C)) == 0x00004550 and
3 of them
}

rule lab1203exe{
strings:
    $log = "practicalmalwareanalysis.log1"
    $func = "VirtualAlloc"
    $string1 = "TerminateProcess"
    $string2 = "[Window:"
condition:
    filesize < 200KB and uint16(0) == 0x5A4D and uint16(uint16(0x3C)) == 0x00004550 and
2 of them
}
```

```

rule lab1204exe{
  strings:
    $log = "http://www.practicalmalwareanalysis.com//updater.exe"
    $exe1 = "wupdmgrd.exe"
    $exe2 = "winup.exe"
    $string1 = "<SHIFT>"
    $string2 = "%s%s"
  condition:
    filesize < 200KB and uint16(0) == 0x5A4D and uint16(uint16(0x3C)) == 0x00004550 and
    2 of them
}

```

3. 运行结果

```

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 .
\12.y .\Chapter_12L\
lab1203exe .\Chapter_12L\Lab12-01.dll
lab1201exe .\Chapter_12L\Lab12-01.exe
lab1203exe .\Chapter_12L\Lab12-01.exe
lab1204exe .\Chapter_12L\Lab12-04.exe
lab1203exe .\Chapter_12L\Lab12-03.exe
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

获取光标所在函数的函数名、开始地址和结束地址，分析函数 FUNC_FAR、FUNC_USERFAR、FUNC_LIB（库代码）、FUNC_STATIC（静态函数）、FUNC_FRAME、FUNC_BOTTOMBP、FUNC_HIDDEN 和 FUNC_THUNK 标志，获取当前函数中 jmp 或者 call 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"

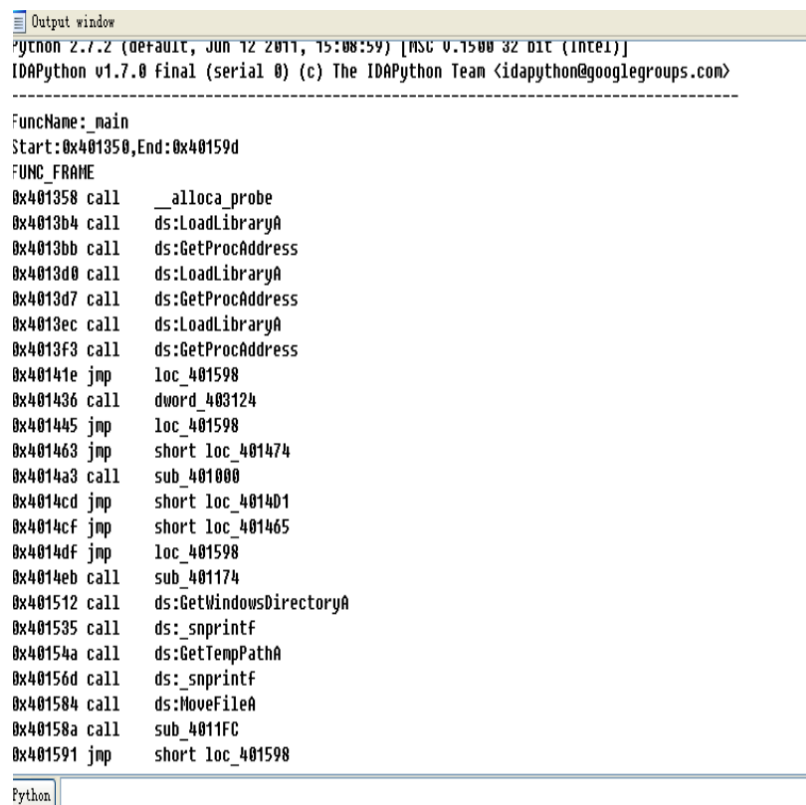
```

```

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



```

Python 2.7.2 (default, JUN 12 2011, 15:08:59) [MSC v.1500 32 bit (Intel)]
IDAPython v1.7.0 final (serial 0) (c) The IDAPython Team <idapython@googlegroups.com>
-----
FuncName: main
Start:0x401350,End:0x40159d
FUNC_FRAME
0x401358 call    __alloca_probe
0x4013b4 call    ds:LoadLibraryA
0x4013bb call    ds:GetProcAddress
0x4013d0 call    ds:LoadLibraryA
0x4013d7 call    ds:GetProcAddress
0x4013ec call    ds:LoadLibraryA
0x4013f3 call    ds:GetProcAddress
0x40141e jmp     loc_401598
0x401436 call    dword_403124
0x401445 jmp     loc_401598
0x401463 jmp     short loc_401474
0x4014a3 call    sub_401000
0x4014cd jmp     short loc_401401
0x4014cf jmp     short loc_401465
0x4014df jmp     loc_401598
0x4014eb call    sub_401174
0x401512 call    ds:GetWindowsDirectoryA
0x401535 call    ds:_snprintf
0x40154a call    ds:GetTempPathA
0x40156d call    ds:_snprintf
0x401584 call    ds:MoveFileA
0x40158a call    sub_4011FC
0x401591 jmp     short loc_401598

```

四、实验结论及心得体会

通过完成 Lab12 实验，我深入学习了恶意代码分析的核心技术，特别是 DLL 注入、进程替换、键盘记录和远程线程创建等技术的工作原理和应用方式。在实验中，我结合动态和静态分析方法，使用了 IDA Pro、Process Monitor、WinHe

x 等工具，逐步揭示恶意代码的行为逻辑，掌握了从行为观察到代码逆向的完整分析过程。此外，通过编写 Yara 规则和 IDA Python 脚本，我初步尝试了自动化检测和分析的实践，显著提升了分析效率。

实验让我深刻认识到恶意代码如何利用隐蔽技术规避检测，例如进程注入、加密保护负载和挂钩键盘事件等，同时也让我对系统安全防护的改进方向有了更多思考。