

南開大學

恶意代码分析与防治技术课程实验报告

Lab13



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. 了解数据加密。
2. 进一步熟悉静态分析与动态分析的过程。
3. 学习使用 IDA、FindCrypt2、KANAL 等专业工具识别恶意软件采用的加密与编码技术。
4. 学会通过定位 XOR、Base64、AES 等加解密函数和数据段，逆向提取被隐藏的字符串、资源与网络通信信息。

二、实验原理

本实验的原理在于通过对恶意代码样本进行静态与动态分析，以揭示其所采用的加密、编码及混淆技术原理。

在静态分析中，研究者利用 IDA、FindCrypt2、KANAL 等工具对程序的汇编指令、字符串和加密函数调用进行剖析，发现其可能采用了如 XOR、Base64 及 AES 等加密手段，从而隐藏真实网络连接、命令以及数据传输意图。

同时，通过动态分析（如使用 ProcMon、Wireshark 和调试器），观察恶意程序在受控环境下的实际运行行为，验证其隐藏的数据在网络流量与文件资源中的具体表现。

三、实验过程

Lab13-1

分析恶意代码文件 Lab13-01.exe。

问题

1. 比较恶意代码中的字符串（字符串命令的输出）与动态分析提供的有用信息，基于这些比较，哪些元素可能被加密？

使用 String 查看程序，我们观察到了包含 'http' 的字符串，以及 '%s' 这一格式化占位符，后者通常用于标记字符串中的动态内容。

为了深入探究此程序的行为，我们在受控的虚拟机环境中执行了该程序，并对其进程进行了监控。监控过程揭示了该文件正在尝试访问特定的网络地址：www.practicalmalwareanalysis.com/aGFueHUtUEM=。这一发现是关键，因为

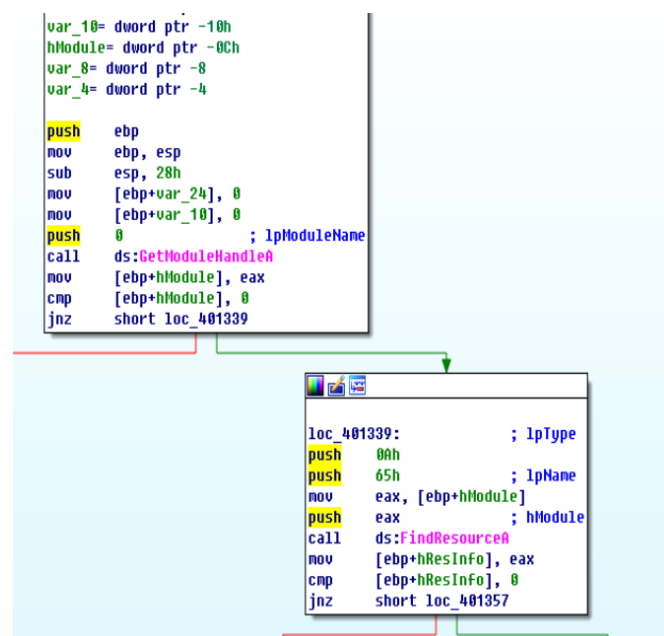
它暗示了两个' %s' 占位符可能分别对应于网址的两个部分:' www.practicalmalwareanalysis.com' 和' aGFueHUtUEM='。

然而, 在进一步检查名为' Lab13-01.exe' 的程序的字符串列表时, 我们并未发现明确包含上述两个部分的字符串。这种情况通常表明, 这些字符串可能已经被程序以某种方式加密或隐藏, 以避免直接检测。因此, 我们假设' www.practicalmalwareanalysis.com' 和' aGFueHUtUEM=' 这两个字符串在程序中以某种加密或编码的形式存在, 需要进一步的分析以揭示其真实形式。

```
C:\WINDOWS\system32\cmd.exe
WS2_32.dll
InternetReadFile
InternetCloseHandle
InternetOpenUrlA
InternetOpenA
WININET.dll
GetCommandLineA
GetVersion
ExitProcess
TerminateProcess
GetCurrentProcess
UnhandledExceptionFilter
GetModuleFileNameA
FreeEnvironmentStringsA
FreeEnvironmentStringsW
WideCharToMultiByte
GetEnvironmentStrings
GetEnvironmentStringsW
SetHandleCount
GetStdHandle
GetFileType
GetStartupInfoA
HeapDestroy
HeapCreate
VirtualFree
HeapFree
RtlUnwind
WriteFile
GetLastError
SetFilePointer
HeapAlloc
GetCPInfo
GetACP
GetOEMCP
VirtualAlloc
HeapReAlloc
GetProcAddress
LoadLibraryA
SetStdHandle
FlushFileBuffers
MultiByteToWideChar
LCMapStringA
LCMapStringW
GetStringTypeA
GetStringTypeW
CloseHandle
??"@
Mozilla/4.0
http://%s/%s/
Could not load exe.
Could not locate dialog box.
```

2. 使用 IDAPro 搜索恶意代码中字符串 'xor', 以此来查找潜在的加密, 你发现了哪些加密类型?

初步分析采用了 PEiD 来检测显示该软件未被加壳。通过进一步研究软件内的特定函数, 特别关注了 “sub_401300” 的函数。

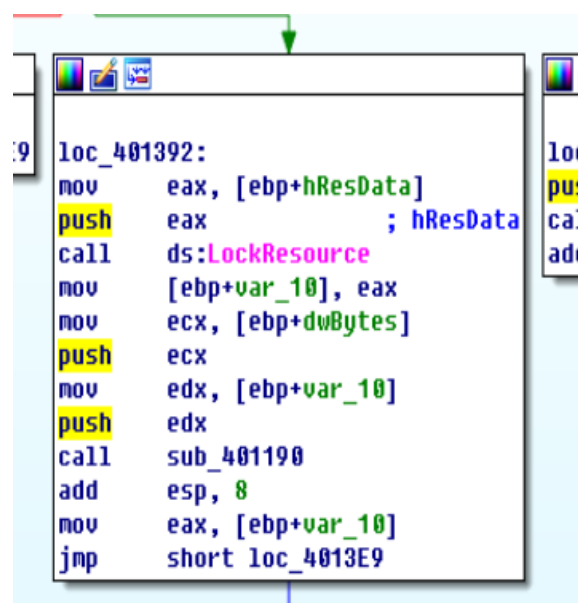


```
var_10= dword ptr -10h
hModule= dword ptr -0Ch
var_8= dword ptr -8
var_4= dword ptr -4

push    ebp
mov     ebp, esp
sub     esp, 28h
mov     [ebp+var_24], 0
mov     [ebp+var_10], 0
push    0 ; lpModuleName
call    ds:GetModuleHandleA
mov     [ebp+hModule], eax
cmp     [ebp+hModule], 0
jnz     short loc_401339

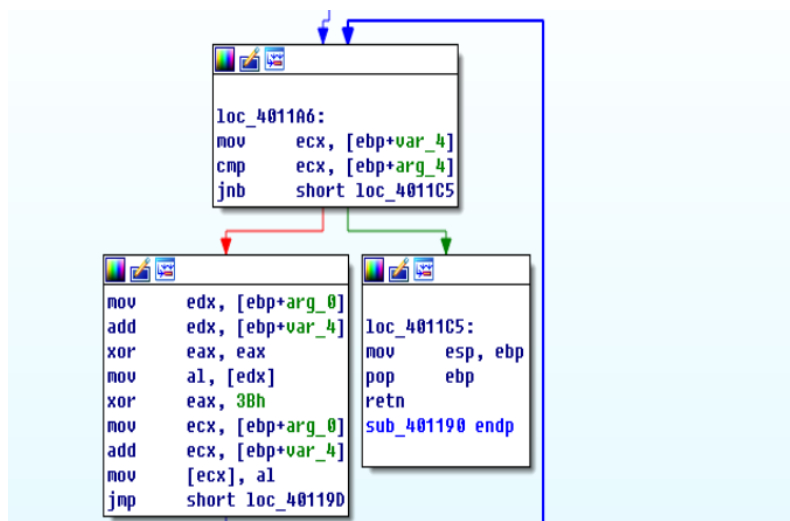
loc_401339: ; lpType
push    0Ah
push    65h ; lpName
mov     eax, [ebp+hModule]
push    eax ; hModule
call    ds:FindResourceA
mov     [ebp+hResInfo], eax
cmp     [ebp+hResInfo], 0
jnz     short loc_401357
```

在这个函数中, 使用了 GetModuleHandleA 来获取 “Lab13-01.exe” 的模块句柄, 而 FindResourceA 用于定位 “Lab13-01.exe” 内名为 65h 的资源节。这些发现暗示了 “Lab13-01.exe” 可能包含了一个资源节, 这一点通过使用 StudyPE 进行验证, 确实在资源节 0065 中发现了相关内容。然而, 这些内容表面上看起来并不包含明显的重要信息, 可能是因为进行了加密处理。



```
loc_401392:
mov     eax, [ebp+hResData]
push    eax ; hResData
call    ds:LockResource
mov     [ebp+var_10], eax
mov     ecx, [ebp+dwBytes]
push    ecx
mov     edx, [ebp+var_10]
push    edx
call    sub_401190
add     esp, 8
mov     eax, [ebp+var_10]
jmp     short loc_4013E9
```

发现在地址“loc_401357”处，相关资源已经被加载到了进程的内存中。进一步的分析揭示了 LockResource 函数返回了一个指向内存中资源的指针，这个指针以及资源的大小（由 SizeofResource 函数计算得到）被推入堆栈，随后调用了“sub_401190”函数。

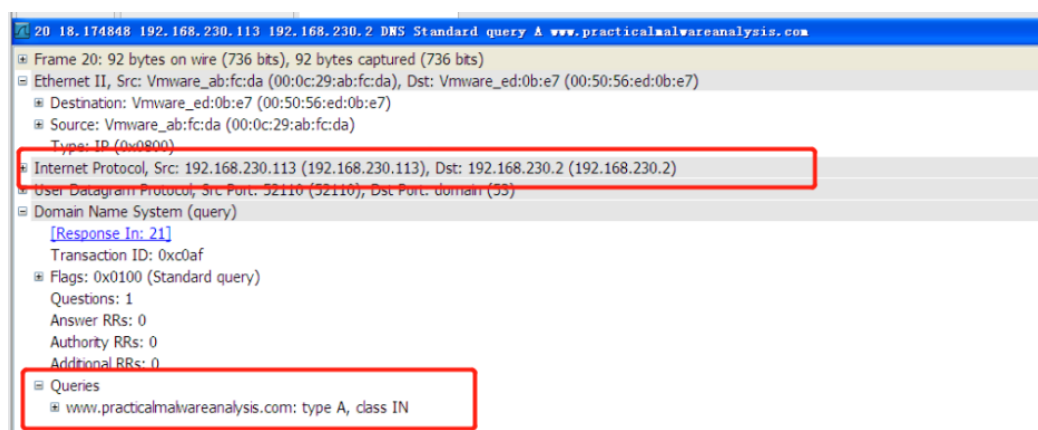


在“sub_401190”函数中，存在几个关键参数：’arg_0’ 为第一个参数，指向资源的指针；’arg_4’ 为第二个参数，表示资源的大小；’var_4’ 用作 for 循环的计数器。该函数通过执行“mov al, [edx]”获取资源内容，并利用“xor eax, 3Bh”进行资源解密。

地址 004011B8 处的 xor 指令是 sub_401190 函数中的一个单字节 XOR 加密循环的指令。

3. 恶意代码使用什么密钥加密, 加密了什么内容?

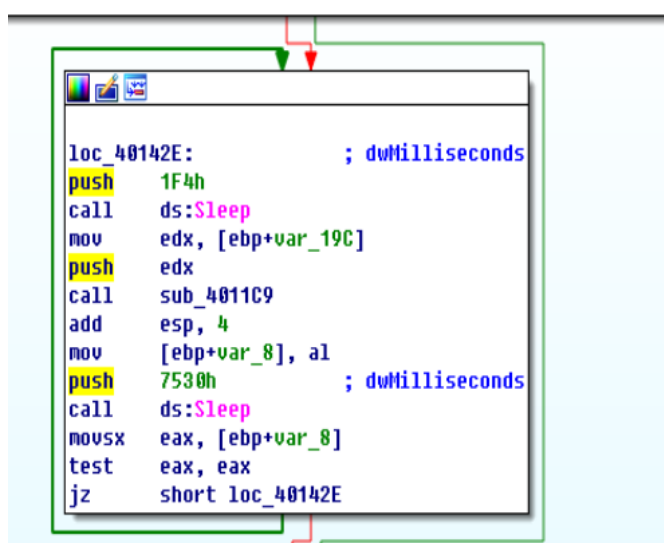
单字节 XOR 加密使用字节 0x3B。首先我们可以运行 Lab13-01.exe 来分析其具体的行为，并通过 wireshark 进行抓包分析。我们可以通过结果简单发现他请求了一个网站 www.practicalmalwareanalysis.com。



加密了 www.practicalmalwareanalysis.com。

4. 使用静态工具 FindCrypt2、Krypto ANALyzer (KANAL) 以及 IDA 熵插件识别一些其他类型的加密机制, 你发现了什么?

分析进一步深入后, 注意到在执行 Sleep 函数暂停程序后, 恶意代码将一个变量 (标记为 var_19C) 压入堆栈。在地址 00401402 处的判断逻辑表明, 这个 var_19C 变量实际上是对先前提到的资源节内容进行解密的结果。



使用了 PEiD 的插件 KANAL 来进行查询。KANAL 的分析结果表明, 所使用的加密方法是 BASE64 编码。随后, 在 IDA 中转向地址 4050E8 进行进一步的分析, 确认了恶意代码确实使用了标准的 Base64 编码方式。这种编码包括了字符集: ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/, 这是 Base64 编码的典型字符集, 用于将二进制数据转换成文本格式。

```

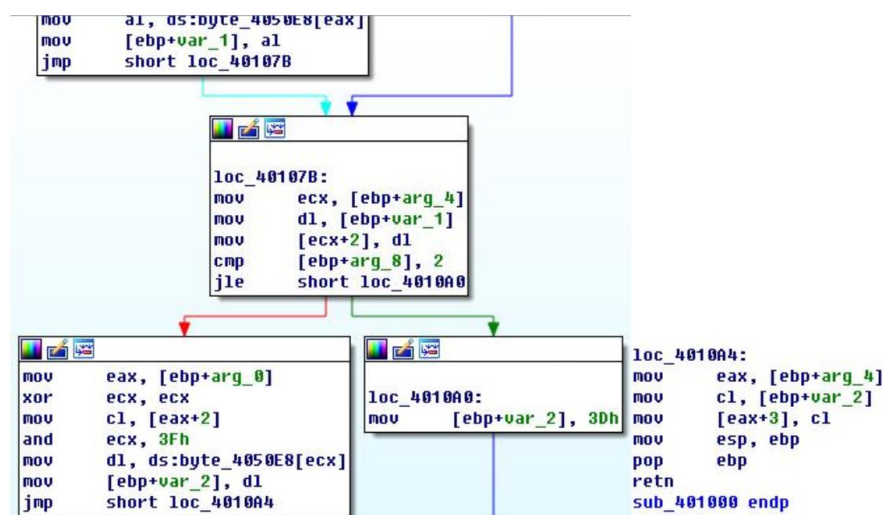
BASE64 table :: 000050E8 :: 004050E8
.... Referenced at 00401013
.... Referenced at 0040103E
.... Referenced at 0040106E
.... Referenced at 00401097

.rdata:004050E9      db  42h ; B
.rdata:004050EA      db  43h ; C
.rdata:004050EB      db  44h ; D
.rdata:004050EC      db  45h ; E
.rdata:004050ED      db  46h ; F
.rdata:004050EE      db  47h ; G
.rdata:004050EF      db  48h ; H
.rdata:004050F0      db  49h ; I
.rdata:004050F1      db  4Ah ; J
.rdata:004050F2      db  4Bh ; K
.rdata:004050F3      db  4Ch ; L
.rdata:004050F4      db  4Dh ; M
.rdata:004050F5      db  4Eh ; N
.rdata:004050F6      db  4Fh ; O
.rdata:004050F7      db  50h ; P
.rdata:004050F8      db  51h ; Q
.rdata:004050F9      db  52h ; R
.rdata:004050FA      db  53h ; S
.rdata:004050FB      db  54h ; T
.rdata:004050FC      db  55h ; U
.rdata:004050FD      db  56h ; V
.rdata:004050FE      db  57h ; W
.rdata:004050FF      db  58h ; X
.rdata:00405100      db  59h ; Y
.rdata:00405101      db  5Ah ; Z
.rdata:00405102      db  61h ; a
.rdata:00405103      db  62h ; b
.rdata:00405104      db  63h ; c
.rdata:00405105      db  64h ; d
.rdata:00405106      db  65h ; e
.rdata:00405107      db  66h ; f

```

5. 什么类型的加密被恶意代码用来发送部分网络流量？

这里一共进行了三次的清空寄存器和存放的操作，很明显这里就是对“GET”字样进行一个解密。标准的 Base64 编码用来创建 GET 请求字符串。



6. Base64 编码函数在反汇编的何处？

.text:004010B1 var_1C	= dword ptr -1Ch
.text:004010B1 var_18	= dword ptr -18h
.text:004010B1 var_14	= dword ptr -14h
.text:004010B1 var_10	= byte ptr -10h
.text:004010B1 var_C	= byte ptr -0Ch
.text:004010B1 var_8	= dword ptr -8
.text:004010B1 var_4	= dword ptr -4
.text:004010B1 arg_0	= dword ptr 8
.text:004010B1 arg_4	= dword ptr 0Ch

由之前的分析可知，Base64 编码函数是从 0x004010B1 开始的。

7. 恶意代码发送的 Base64 加密数据的最大长度是什么?加密了什么内容?

从以上分析中得知，Lab13-01.exe 在执行 Base64 加密前，首先从系统中获取主机名，并且复制其最多 12 个字节的内容。由于 Base64 编码的特性，每 3 个字节的原始数据会被编码成 4 个字节的输出。因此，如果原始数据是最大 12 个字节(如主机名)，编码后的 Base64 字符串的长度将最多是 16 个字符。

```
int __cdecl base64(int a1, int a2, signed int a3)
{
    int result; // eax@7
    char v4; // [sp+0h] [bp-2h]@5
    char v5; // [sp+1h] [bp-1h]@2

    *(_BYTE *)a2 = byte_4050E8[(signed int)*(_BYTE *)a1 >> 2];
    *(_BYTE *)a2 + 1 = byte_4050E8[((*(_BYTE *)a1 + 1) & 0xF0) >> 4] | 16 * ((*(_BYTE *)a1 & 3)];
    if ( a3 <= 1 )
        v5 = 61;
    else
        v5 = byte_4050E8[((*(_BYTE *)a1 + 2) & 0xC0) >> 6] | 4 * ((*(_BYTE *)a1 + 1) & 0xF);
    *(_BYTE *)a2 + 2 = v5;
    if ( a3 <= 2 )
        v4 = 61;
    else
        v4 = byte_4050E8[(a1 + 2) & 0x3F];
    result = a2;
    *(_BYTE *)a2 + 3 = v4;
    return result;
}
```

8. 恶意代码中，你是否在 Base64 加密数据中看到了填充字符(=或者==)?

在 Base64 编码中，如果待编码的数据长度不能被 3 整除，通常会使用填充字符（通常是 '='）来确保编码后的字符串长度是 4 的倍数。这是因为 Base64 编码将输入数据划分为 3 字节（24 位）的组，并将每个组进一步分为 4 个 6 位的单元，每个单元对应于 Base64 字符集中的一个字符。

如果主机名的长度小于 12 个字节且不能被 3 整除，这意味着在 Base64 编码过程中必须使用填充字符来达到所需的长度。例如，对于之前提到的主机名“hanxu-PC”，它的长度为 8 个字节，不是 3 的倍数。因此，在 Base64 编码过程中，使用了一个 '=' 字符作为填充，得到最终的编码结果“aGFueHUtUEM=”。

9. 这个恶意代码做了什么?

通过上述分析可知，Lab13-01.exe 用加密的主机名发送一个特定信号，直到接收特定的回应后退出。

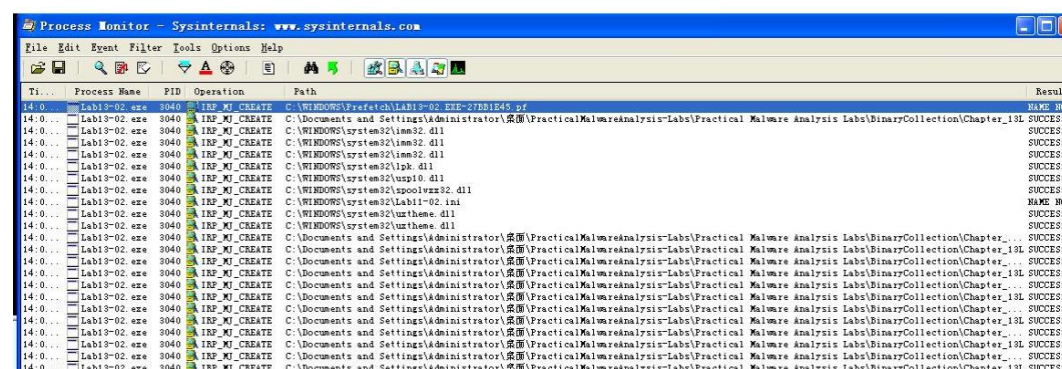
Lab 13-2

分析恶意代码文件 Lab13-02.exe。

问题

1. 使用动态分析, 确定恶意代码创建了什么?

通过使用 Process Monitor (ProcMon) 来监控恶意程序的行为, 该程序表现出了频繁的文件创建和写入操作。特别值得注意的是, 在特定路径下出现了大量以 “temp” 开头、大小为 3073KB 的文件。此外, 这些文件似乎在固定时间间隔内被创建, 其内容的后半部分呈现出十六进制的随机数样式。



2. 使用静态分析技术, 例如 xor 指令搜索、FindCrypt2、KANAL 以及 IDA 插件, 查找潜在的加密, 你发现了什么?

首先使用了 PEiD 来检查恶意软件是否加壳, 结果显示软件没有加壳, 接着检查了是否有加密, 同样没有发现明显的加密迹象。

在 IDA 中使用搜索功能查找 “xor” 操作, 这是因为 xor (异或) 操作常用于简单的加解密过程。

sub_401000	xor	eax, eax
sub_40128D	xor	eax, [ebp+var_10]
	xor	eax, [esi+edx*4]
sub_401739	xor	edx, [ecx]
sub_401739	xor	edx, ecx
sub_401739	xor	edx, ecx
sub_401739	xor	eax, [edx+8]
sub_401739	xor	eax, edx
sub_401739	xor	eax, edx
sub_401739	xor	ecx, [eax+10h]
sub_401739	xor	ecx, eax
sub_401739	xor	ecx, eax
sub_401739	xor	edx, [ecx+18h]
sub_401739	xor	edx, ecx
sub_401739	xor	edx, ecx
_main	xor	eax, eax

通过这种方法发现了一个关键函数 sub_401739。观察调用关系图，推测 sub_401851 与加密过程可能相关。



进一步检查 main 函数，发现其中存在调用 sub_401851 的 call 指令。对 sub_401851 的分析揭示了它调用了几个 API，并且涉及了 sub_401070 和 sub_40181F 两个函数。

```

.text:00401870      call     sub_401070
.text:00401875      add      esp, 8
.text:00401878      mov      edx, [ebp+nNumberOfBytesToWrite]
.text:0040187B      push     edx
.text:0040187C      mov      eax, [ebp+hMen]
.text:0040187F      push     eax
.text:00401880      call     sub_40181F
.text:00401885      add      esp, 8
.text:00401888      call     ds:GetTickCount
.text:0040188E      mov      [ebp+var_4], eax
.text:00401891      mov      ecx, [ebp+var_4]
.text:00401894      push     ecx
.text:00401895      push     offset aTemp00x ; "temp%08x"
.text:0040189A      lea      edx, [ebp+FileName]
.text:004018A0      push     edx ; char *
.text:004018A1      call     _sprintf
.text:004018A6      add      esp, 0Ch
.text:004018A9      lea      eax, [ebp+FileName]
.text:004018AF      push     eax ; lpFileName
.text:004018B0      mov      ecx, [ebp+nNumberOfBytesToWrite]
.text:004018B3      push     ecx ; nNumberOfBytesToWrite
.text:004018B4      mov      edx, [ebp+hMen]
.text:004018B7      push     edx ; lpBuffer
.text:004018B8      call     sub_401000
.text:004018BD      add      esp, 0Ch
.text:004018C0      mov      eax, [ebp+hMen]
.text:004018C3      push     eax ; hMen
.text:004018C4      call     ds:GlobalUnlock
.text:004018CA      mov      ecx, [ebp+hMen]
.text:004018CD      push     ecx ; hMen
.text:004018CE      call     ds:GlobalFree

```

深入分析 sub_401851，可以发现该函数涉及到生成文件名的部分，其中使用了 GetTick-Count 函数来获取操作系统启动后经过的毫秒数，并将其作为后续函数的参数。此外，还调用了 sub_401000 函数。

对 sub_401070 的分析表明，这部分功能似乎与截屏有关。通过完整分析可以得知，程序具备获取用户桌面、创建位图对象并将其加密后放置在桌面的能力。

综上所述，与 xor 相关的加密功能可能与 sub_401739 和 sub_401851 两个函数相关联。这表明恶意软件可能使用了简单的 xor 加密方法来隐藏或保护其收集的信息（如截屏数据），或者在执行其它恶意活动时隐藏其真实意图。xor 加密由于其简单性，在恶意软件中被频繁使用，尤其是在需要快速且不太复杂的加密场景中。

3. 基于问题 1 的回答, 哪些导入函数将是寻找加密函数比较好的一个证据?

函数 sub_40181F。在该函数中，发现调用了 sub_401739，这指向了更深层次的操作。进一步看 sub_401739，它随后调用了 sub_4012DD。在这之后，观察到了一系列的异或（XOR）操作。这种操作在计算机编程中常用于加密，因为它可以通过再次应用相同的异或操作来逆转加密，从而实现数据的加密和解密。

```

xor     edx, [ecx]
mov     eax, [ebp+arg_0]
mov     ecx, [eax+14h]
shr     ecx, 10h
xor     edx, ecx
mov     eax, [ebp+arg_0]
mov     ecx, [eax+0Ch]
shl     ecx, 10h
xor     edx, ecx
mov     eax, [ebp+arg_8]
mov     [eax], edx
mov     ecx, [ebp+arg_4]
mov     edx, [ebp+arg_0]
mov     eax, [ecx+4]
xor     eax, [edx+8]

```

继续分析 sub_4012DD，可以看出这里涉及了加密操作。这个函数可能涉及将数据转换为一种不易被直接识别的格式，从而在不暴露其内容的情况下进行数据存储或传输。再查看 sub_401000，这里涉及了一些文件操作，特别是 WriteFile 的调用。在调用 WriteFile 之前，可能会执行加密函数，这里特别提到的加密函数是 sub_40181F。

```

.idata:00406048 ; DATA XREF: ...
.idata:0040604C ; HANDLE __stdcall CreateFileA(LPCSTR lpFileName, DWORD dwDesiredA
.idata:0040604C         extrn CreateFileA:dword ; CODE XREF: sub_401000+31
.idata:0040604C         ; DATA XREF: sub_401000+31
.idata:00406050 ; BOOL __stdcall WriteFile(HANDLE hFile, LPCVOID lpBuffer, DWORD n
.idata:00406050         extrn WriteFile:dword ; CODE XREF: sub_401000+56
.idata:00406050         ; __NMSG_WRITE+14A1p ...
.idata:00406054 ; BOOL __stdcall GetStringTypeA(LCID Locale, DWORD dwInfoType, LPC
.idata:00406054         extrn GetStringTypeA:dword

```

综上所述，这些函数的分析指向了一个加密和文件操作的复杂过程。sub_40181F 和 sub_401739 可能负责执行加密操作，而 sub_4012DD 可能是实施这种加密的具体手段。

4. 加密函数在反汇编的何处？

结合第二问和第三问的分析，可以得出结论加密函数是 sub_40181F。

5. 从加密函数追溯原始的加密内容，原始加密内容是什么？

结合第二问和第三问的分析，可以知道原始加密内容是屏幕截图。

6. 你是否能够找到加密算法？如果没有，你如何解密这些内容？

加密算法是不标准算法，并且不容易识别，最简单的方法是通过解密工具解密流量。

7. 使用解密工具, 你是否能够恢复加密文件中的一个文件到原始文件?

使用 Immunity Debugger 的脚本功能来解密这些文件。设置断点、读取文件内容、分配内存缓冲区、以及将数据复制到新创建的缓冲区中。

Lab 13-3

分析恶意代码文件 Lab13-03.exe。

问题

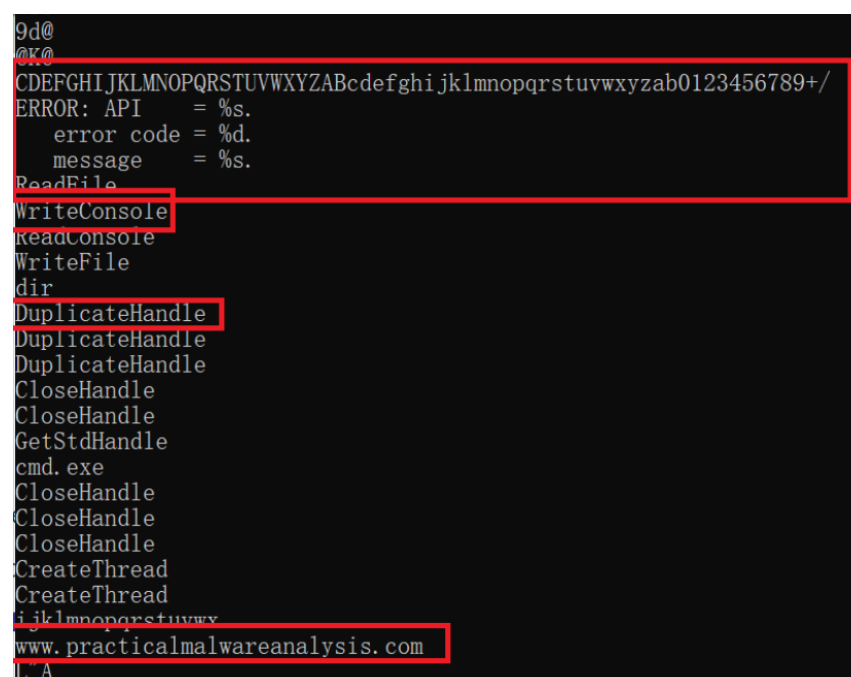
1. 比较恶意代码的输出字符串和动态分析提供的信息, 通过这些比较, 你发现哪些元素可能被加密?

通过 String 工具检查字符串, 可以发现很多格式字符串以及和 Base64 相关的字符串, 这说明有一些内容很有可能被加密了, 但是网站的域名却没有被加密, 同时我们观察这个很像 Base64 加密编码的字符序列, 发现他并不是最常见的编码方式, 因为 AB 和 ab 都被放在了最后, 而他们却是以 C 和 c 开始的, 并且, 这里有一些比较新颖的 win 系统函数, 我们可以通过查阅官方文档来了解这些函数的

WriteConsole: 从当前光标位置开始, 将字符串写入控制台屏幕缓冲区

ReadConsole: 从控制台输入缓冲区读取字符输入, 将其从缓冲区中删除。

DuplicateHandle: 复制对象句柄



```
9d@
@k@
CDEFGHIJKLMNOPQRSTUVWXYZAbcdefghijklmnopqrstuvwxyzab0123456789+/
ERROR: API      = %s.
      error code = %d.
      message   = %s.
ReadFile
WriteConsole
ReadConsole
WriteFile
dir
DuplicateHandle
DuplicateHandle
DuplicateHandle
CloseHandle
CloseHandle
GetStdHandle
cmd.exe
CloseHandle
CloseHandle
CloseHandle
CreateThread
CreateThread
i j k l m n o p q r s t u v w x y z
www.practicalmalwareanalysis.com
L A
```


2. 使用静态分析搜索字符串 xor 来查找潜在的加密。通过这种方法,你发现什么类型的加密?

使用 IDA 查找 xor 指令以后可以发现非常多的地方都使用了 xor。在 IDA 中搜索 xor 指令的结果显示大量的使用情况。在排除常规用途后,可以确定 6 个可能与加密操作相关的独立函数。

Address	Function	Instruction
.text:00401135	sub_401082	xor eax, eax ; jumtable 0040112E case 0
.text:0040123C	sub_401082	xor eax, eax
.text:00401310	sub_4012E9	xor ecx, ecx
.text:00401341	sub_40132B	xor eax, eax
.text:00401357	sub_40132B	xor eax, eax
.text:0040136D	sub_40132B	xor eax, eax
.text:004014A5	StartAddress	xor eax, eax
.text:004014BB	StartAddress	xor eax, eax
.text:00401873	sub_4015B7	xor eax, eax
.text:004019A5	main	xor eax, eax
.text:00401A53	sub_401A50	xor eax, eax
.text:00401D51	sub_401AC2	xor edx, edx
.text:00401D69	sub_401AC2	xor eax, eax
.text:00401D88	sub_401AC2	xor eax, eax
.text:00401DA7	sub_401AC2	xor eax, eax
.text:00401EBD	sub_401AC2	xor eax, edx
.text:00401ED8	sub_401AC2	xor eax, edx
.text:00401EF3	sub_401AC2	xor eax, edx
.text:00401F08	sub_401AC2	xor eax, edx
.text:00401F13	sub_401AC2	xor edx, eax
.text:00401F56	sub_401AC2	xor eax, [esi+edx*4+414h]
.text:00401FB1	sub_401AC2	xor edx, [esi+ecx*4+414h]
.text:00402028	sub_401AC2	xor edx, ecx
.text:00402046	sub_401AC2	xor edx, ecx
.text:00402064	sub_401AC2	xor edx, ecx
.text:00402070	sub_401AC2	xor ecx, edx
.text:004020B3	sub_401AC2	xor edx, [esi+ecx*4+414h]
.text:004021E3	sub_401AC2	xor ecx, ds:dword_40EF08[eax*4]
.text:004021F6	sub_401AC2	xor ecx, ds:dword_40F308[edx*4]
.text:00402205	sub_401AC2	xor ecx, ds:dword_40F708[eax*4]
.text:00402246	sub_40223A	xor ecx, ecx
.text:00402276	sub_40223A	xor edx, edx
.text:0040228C	sub_40223A	xor edx, edx
.text:004022A7	sub_40223A	xor eax, eax

3. 使用静态工具,如 FindCrypt2、KANAL 以及 IDA 熵插件识别一些其他类型的加密机制。发现的结果与搜索字符 XOR 结果比较如何?

通过对恶意软件的进一步分析,我们发现了其使用了 Rijndael 算法,这实际上是高级加密标准(AES)的另一种说法。这一发现是通过使用 IDA 的“find crypt”插件以及 PEiD 的“kryptoanalyzer”插件得出的。这两个插件识别出了加密算法中的 S 盒结构,这是许多加密算法的关键组成部分。

```
Found const array Rijndael_Te1 (used in Rijndael)
Found const array Rijndael_Te2 (used in Rijndael)
Found const array Rijndael_Te3 (used in Rijndael)
Found const array Rijndael_Td0 (used in Rijndael)
```

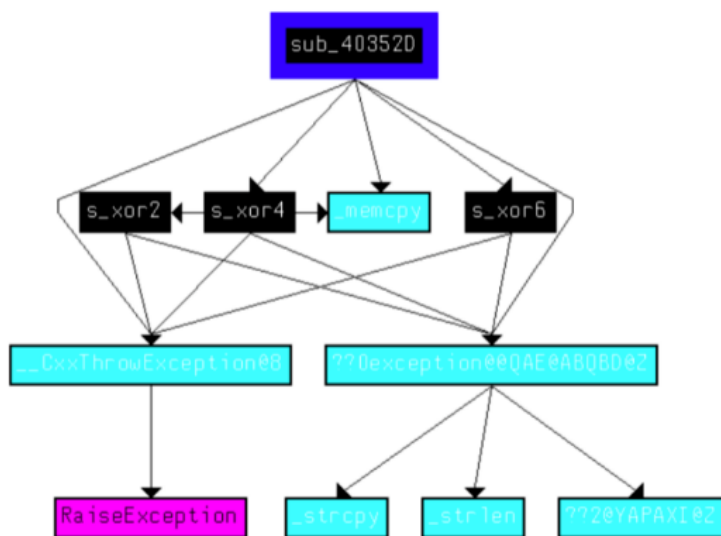
在这个案例中,我们发现了几个与 AES 加密和解密过程直接相关的函数。特别地,s_xor2 和 s_xor4 与 AES 加密过程中使用的加密常量 Te0 有关,而

s_xor3 和 s_xor5 则与解密常量 Td0 有关。这些发现表明恶意程序中包含了复杂的加密和解密机制。

我还分析了 s_xor6 函数，它涉及到一个循环过程，其中使用了 XOR 指令来执行加密操作。该函数接受两个指针参数，一个指向原始数据缓冲区，另一个指向异或操作的原始数据缓冲区。



进一步的交叉引用分析显示 s_xor6 与 sub_40352d（被重命名为 s_encrypt）以及 s_xor1（密钥初始化代码）有关。这表明 s_xor1 负责初始化 AES 密钥，而 s_encrypt 负责执行加密操作。



在主函数中，s_xor1 的调用前有对 unk_412ef8 的引用，这表明这是一个与 AES 加密器相关的 C++ 对象，并且 s_xor1 是其初始化函数。在分析中发现，s_xor1 的一个参数是一个字符串，推测这个字符串被用作 AES 加密的密钥。

<pre> .text:00401882 .text:00401884 .text:00401886 .text:00401888 .text:00401890 .text:00401895 .text:0040189A .text:004018A0 .text:004018A1 .text:004018A6 .text:004018AC .text:004018B2 .text:004018B9 .text:004018BB .text:004018C0 .text:004018C5 ; ----- </pre>	<pre> push 10h ; int push 10h ; int push offset unk_413374 ; void * push offset aIjklmnopqrstuv ; "ijklmnopqrstuvwx" mov ecx, offset unk_412EF8 call sub_401AC2 lea eax, [ebp+WSAData] push eax ; lpWSAData push 202h ; wVersionRequested call ds:WSAStartup mov [ebp+var_194], eax cmp [ebp+var_194], 0 jz short loc_4018C5 mov eax, 1 jmp loc_4019A7 </pre>
--	---

我们可以得出结论，该恶意软件使用了高级加密标准 AES 算法（Rijndael 算法），并涉及到多个与 AES 加密和解密相关的 XOR 函数。同时，还存在 BASE64 编码的使用。这些发现表明了恶意软件在设计上的高度复杂性，以及在隐藏其行为和通信内容方面的高级技术应用。

4. 恶意代码使用哪两种加密技术？

结合之前的分析可知，恶意代码使用 AES 和自定义的 Base64 加密。

5. 对于每一种加密技术，它们的密钥是什么？

AES 密钥是：ijklmnopqrstuvwx

自定义加密的索引字符串是: CDEFGHIJKLMNOPQRSTUVWXYZABcdefghijklmno
pqrstuvwxyzab0123456789+/-

6. 对于加密算法, 它的密钥足够可靠吗? 另外你必须知道什么?

在跟踪 sub_40103f 函数的交叉引用时, 我们发现该函数进一步引用了 sub_401082。进入 sub_401082 进行交叉引用分析, 我们注意到这个函数在 ReadFile 和 WriteFile 之间被调用, 这与我们之前对 AES 加密的分析相似。进一步的分析表明, s_encrypt 函数在 sub_40132b 处被调用。通过查看 sub_40132b 的交叉引用, 我们来到了 sub_4015b7。这里, 我们发现 sub_40132b 实际上是一个新线程的起始点, 该线程通过 CreateThread 创建。因此, 我们可以将 sub_40132b 重命名为 aes_thread。

分析 aes_thread 时, 我们关注到传递给线程的参数被保存在 lpParameter, 也就是 var_58 中。在地址 00401826, var_18 移入 var_58; 在 0040182c, arg_10 移入 var_54; 而在 00401835, dword_41336c 移入 var_50。随后, 我们继续跟进 aes_thread, 分析这些参数在函数中的流程。

```
.text:00401820 ; -----  
.text:00401820             add     esp, 4  
.text:00401823  
.text:00401823 loc_401823:             ; CODE XREF: sub_4015B7+25D↑j  
.text:00401823             mov     eax, [ebp+var_18]  
.text:00401826             mov     [ebp+var_58], eax  
.text:00401829             mov     ecx, [ebp+arg_10]  
.text:0040182C             mov     [ebp+var_54], ecx  
.text:0040182F             mov     edx, dword_41336C  
.text:00401835             mov     [ebp+var_50], edx  
.text:00401838             lea     eax, [ebp+var_3C]  
.text:0040183B             push    eax             ; lpThreadId  
.text:0040183C             push    0             ; dwCreationFlags  
.text:0040183E             lea     ecx, [ebp+var_58]  
.text:00401841             push    ecx             ; lpParameter  
.text:00401842             push    offset sub_40132B ; lpStartAddress  
.text:00401847             push    0             ; dwStackSize  
.text:00401849             push    0             ; lpThreadAttributes  
.text:0040184B             call    ds:CreateThread  
.text:00401851             mov     [ebp+var_20], eax  
.text:00401854             cmp     [ebp+var_20], 0  
.text:00401858             jnz     short loc_401867  
.text:0040185A             push    offset aCreatethread_0 ; "CreateThread"  
.text:0040185F             call    sub_401256  
.text:00401864 ; -----
```

对 ReadFile 的分析显示, 其参数 hFile 来自 var_BE0。回溯发现, 这个参数实际上来自函数的唯一参数。而在分析 WriteFile 时, 我们发现参数 hFile 来自 var_BE0+4, 也就是 var_54, 或者说是 arg_10。

```

.text:00401442      push     ecx                ; nNumberOfBytesToWrite
.text:00401443      lea     edx, [ebp+var_FE8]
.text:00401449      push     edx                ; lpBuffer
.text:0040144A      mov     eax, [ebp+var_BE0]
.text:00401450      mov     ecx, [eax+4]
.text:00401453      push     ecx                ; hFile
.text:00401454      call    ds:WriteFile
.text:0040145A      test    eax, eax
.text:0040145C      jnz     short loc_40146B
.text:0040145E      push    offset aWriteconsole ; "WriteConsole"
.text:00401463      call    sub_401256

```

我们发现 var_58 和 var_18 持有一个管道的句柄，这个管道与一个 shell 命令的输出相连接。通过 DuplicateHandle，命令 hSourceHandle 复制到 shell 命令的标准输出和标准错误。这条 shell 命令由 CreateProcess 启动。

```

.text:00401686 ; -----
.text:00401686      add     esp, 4
.text:00401689
.text:00401689 loc_401689:      ; CODE XREF: sub_4015B7+C3↑j
.text:00401689      push    2                ; dwOptions
.text:0040168B      push    0                ; bInheritHandle
.text:0040168D      push    0                ; dwDesiredAccess
.text:0040168F      lea     eax, [ebp+var_4C]
.text:00401692      push    eax                ; lpTargetHandle
.text:00401693      call    ds:GetCurrentProcess
.text:00401699      push    eax                ; hTargetProcessHandle
.text:0040169A      mov     ecx, [ebp+hSourceHandle]
.text:0040169D      push    ecx                ; hSourceHandle
.text:0040169E      call    ds:GetCurrentProcess
.text:004016A4      push    eax                ; hSourceProcessHandle
.text:004016A5      call    ds:DuplicateHandle
.text:004016AB      test    eax, eax
.text:004016AD      jnz     short loc_4016BC
.text:004016AF      push    offset aDuplicatehan_1 ; "DuplicateHandle"
.text:004016B4      call    sub_401256

```

进一步回溯 var_54 或 arg_10，我们了解到它们源自 sub_4015b7 的唯一参数。在查看交叉引用时，我们来到了 main 函数。

```

.text:00401914 ; -----
.text:00401914
.text:00401914 loc_401914: ; CODE XREF: _main+8F↑j
.text:00401914      mov     ecx, [ebp+var_198]
.text:0040191A      mov     edx, [ecx+0Ch]
.text:0040191D      mov     eax, [edx]
.text:0040191F      mov     ecx, [eax]
.text:00401921      mov     dword ptr [ebp+name.sa_data+2], ecx
.text:00401927      push    22CEh ; hostshort
.text:0040192C      call    ds:htons
.text:00401932      mov     word ptr [ebp+name.sa_data], ax
.text:00401939      mov     [ebp+name.sa_family], 2
.text:00401942      push    10h ; namelen
.text:00401944      lea     edx, [ebp+name]
.text:0040194A      push    edx ; name
.text:0040194B      mov     eax, [ebp+s]
.text:00401951      push    eax ; s
.text:00401952      call    ds:connect
.text:00401958      mov     [ebp+var_194], eax
.text:0040195E      cmp     [ebp+var_194], 0FFFFFFFh
.text:00401965      jnz     short loc_40196E
.text:00401967      mov     eax, 1
.text:0040196C      jmp     short loc_4019A7
.text:0040196E ; -----

```

对于 AES 算法，解密还需要密钥之外的变量，包括密钥生成算法、密钥大小、操作模式，以及一些常量的初始化等；而对于这个自定义的 Base64 加密，当前已知的索引字符串已经足够了。

7. 恶意代码做了什么？

恶意代码使用以自定义 Base64 加密算法加密传入命令和以 AES 加密传出 shell 命令响应来建立反连命令 shell。

7. 构造代码来解密动态分析过程中生成的一些内容，解密后的内容是什么？

具体的 Base64 算法为：

```

1  import string
2  import base64
3
4  s=""
5  tab="CDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"
6  b64='ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/
7  ciphertext= 'BInaEi=='
8  for ch in ciphertext:
9      if (ch in tab):
10         s+= b64[string.find(tab,chr(ch))]
11     elif(ch=='='):
12         s+='-'
13 print base64.decodestring(s)

```

具体的 AES 的解密算法为：

```

from Crypto.Cipher import AES
import binascii

raw = ' 37 f3 1f 04 51 20 e0 b5 86 ac b6 0f 65 20 89 92 ' + \
' 4f af 98 a4 c8 76 98 a6 4d d5 51 8f a5 cb 51 c5 ' + \
' cf 86 11 0d c5 35 38 5c 9c c5 ab 66 78 40 1d df ' + \
' 4a 53 f0 11 0f 57 6d 4f b7 c9 c8 bf 29 79 2f c1 ' + \
' ec 60 b2 23 00 7b 28 fa 4d c1 7b 81 93 bb ca 9e ' + \
' bb 27 dd 47 b6 be 0b 0f 66 10 95 17 9e d7 c4 8d ' + \
' ee 11 09 99 20 49 3b df de be 6e ef 6a 12 db bd ' + \
' a6 76 b0 22 13 ee a9 38 2d 2f 56 06 78 cb 2f 91 ' + \
' af 64 af a6 d1 43 f1 f5 47 f6 c2 c8 6f 00 49 39 ' ❶

ciphertext = binascii.unhexlify(raw.replace(' ', '')) ❷
obj = AES.new('ijklmnopqrstuvwxyz', AES.MODE_CBC) ❸
print 'Plaintext is:\n' + obj.decrypt(ciphertext) ❹

```

四、实验结论及心得体会

通过此次实验，我更加深刻地理解了恶意代码在对抗分析上的技巧和手段。实验展示了从简单的单字节 XOR 与 Base64 编码，到更为复杂的 AES 加密等多层次的混淆与防护措施。这些手段并非独立存在，而是常常配合资源隐藏、数据加密和动态生成等策略，意在增加逆向工程和静态分析的难度。借助 IDA、KANAL、FindCrypt2 以及熵分析插件等专业工具，我得以更系统地探索恶意代码的内部逻辑、加密方法与密钥位置，进而对其网络通信、资源访问和数据处理过程有了更清晰的理解。通过对函数调用、XOR 指令和加密常量的分析，我不仅学到了加解密算法在实战中的应用，也掌握了识别与还原被加密数据的有效方法。

这次实践让我意识到，在面对复杂威胁时，安全分析者需要灵活运用多种技术手段和分析思路，快速定位问题关键点。能够利用动态与静态分析相结合的方法，从内存加载、资源段提取、网络流量解码，到 AES、Base64 等加密数据的还原，都是提升分析效率与准确度的关键。同时，本次实验也强化了我对多层加密和混淆策略的认知。