

南开大学

恶意代码分析与防治技术实验报告

Lab13



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

本次实验的主要目的是分析和研究恶意软件，特别是恶意软件中的加密和隐藏技术。实验过程中，首先使用 IDA 工具对可疑程序的字符串列表进行审视，发现了疑似加密或隐藏的字符串。进一步监控程序执行过程揭示了其试图访问特定网络地址的行为。最终，通过对加密字符串的深入分析，推断出这些字符串在程序中可能以加密或编码的形式存在。实验中对 Base64 和 AES 加密技术进行了解密，Base64 解密结果为”dir”，指出攻击者可能尝试发送 shell 命令以列出目录内容，而 AES 解密揭示了加密内容实际上是 Microsoft Windows XP 操作系统的版权信息

关键字：恶意软件分析； 加密技术； 动静态分析

目录

一、 实验目的	1
二、 实验原理	1
三、 实验过程	1
(一) Lab13-01	1
(二) Lab13-02	6
(三) Lab13-01	11
四、 实验结论及心得体会	17

一、 实验目的

本实验旨在通过深入分析恶意软件，理解和掌握恶意代码的隐藏和加密机制。通过实际分析恶意软件的行为，提升恶意代码分析能力，并了解如何采取有效措施来防范和应对恶意软件的威胁

二、 实验原理

恶意软件分析主要包括静态分析和动态分析两部分，静态分析是不运行程序代码，通过分析程序的代码结构，寻找恶意特征；动态分析则是在受控环境中运行恶意程序，监视其行为。在本实验中，通过 IDA 工具和监控工具，深入探究了恶意软件的网络通信行为，以及使用的 Base64 和 AES 加密技术

三、 实验过程

(一) Lab13-01

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？

在进行恶意软件分析的初步阶段，我们采用了反汇编工具 IDA 以审视可疑程序的字符串列表。在这一过程中，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。

.rdata:00405714	0000000C	C	WININET.dll
.data:00406030	0000000C	C	Mozilla/4.0
.data:0040603C	0000000E	C	http://%s/%s/
.data:0040604C	00000014	C	Could not load exe.
.data:00406060	0000001D	C	Could not locate dialog box.
.data:00406080	0000001B	C	Could not load dialog box.
.data:0040609C	0000001B	C	Could not lock dialog box.
.data:004060B8	00000006	C	%02x

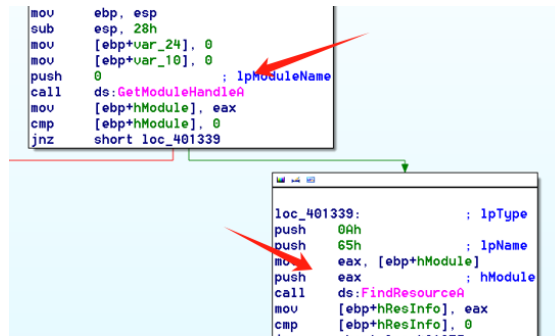
随后，为了深入探究此程序的行为，我们在受控的虚拟机环境中执行了该程序，并对其进程进行了监控。监控过程揭示了该文件正在尝试访问特定的网络地址：www.practicalmalwareanalysis.com/aGFueHUtUEM=。这一发现是关键的，因为它暗示了两个'%s'占位符可能分别对应于网址的两个部分：'www.practicalmalwareanalysis.com'和'aGFueHUtUEM='。

Lab13-01.exe 2080.2532 2080 NET_http www.practicalmalwareanalysis.com/aGFueHUtUEM=

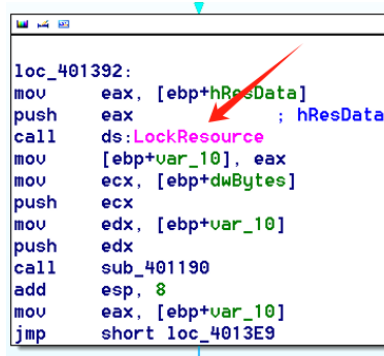
然而，在进一步检查名为'Lab13-01.exe'的程序的字符串列表时，我们并未发现明确包含上述两个部分的字符串。这种情况通常表明，这些字符串可能已经被程序以某种方式加密或隐藏，以避免直接检测。因此，我们假设'www.practicalmalwareanalysis.com'和'aGFueHUtUEM='这两个字符串在程序中以某种加密或编码的形式存在，需要进一步的分析以揭示其真实形式。

2. 使用 IDA Pro 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？

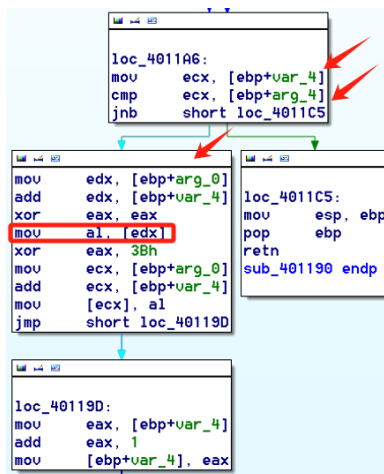
在进行恶意代码分析时，使用了反汇编工具 IDA 对特定软件进行深入探究。初步分析采用了 PEiD 来检测是否存在壳层保护，结果显示该软件未被加壳。通过进一步研究软件内的特定函数，特别关注了“sub_401300”的函数。



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。



继续在 IDA 中分析，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。最终，可以得出结论，地址“004011B8”处的 xor 指令是在“sub_401190”函数中执行的一个单字节 XOR 加密循环的一部分。

3. 恶意代码使用什么密钥加密，加密了什么内容？

在上一问的基础上,通过动态分析工具 OllyDbg 进一步探究了软件的行为。在地址 004013A7 设置断点并执行步过操作,可以观察到自解密过程的结果。



分析发现,栈中的内容变为了“www.practicalmalwareanalysis.com”,这表明经过“xor eax, 3Bh”操作解密后,原本加密的资源节内容就是这个网址。这一发现证实了所使用的加密方法为单字节 XOR 加密,且加密密钥为 0x3B。在此基础上,利用索引 101 对原始数据源进行 XOR 解密操作,可以还原出加密前的原始内容,即“www.practicalmalwareanalysis.com”。此种发现是动态分析的典型成果,通过在执行过程中检查和修改程序状态,可以揭示恶意软件的具体行为和隐藏的数据。

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。

```

push    1F4h
call     ds:Sleep
mov     edx, [ebp+var_19C]
push    edx
call     sub_4011C9
add     esp, 4
mov     [ebp+var_8], al
push    7530h
call     ds:Sleep

```

为了确定具体的加密方式,使用了 PEiD 的插件 KANAL 来进行查询。

```

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

```

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。

```

db 42h ; B
db 43h ; C
db 44h ; D
db 45h ; E
db 46h ; F
db 47h ; G
db 48h ; H
db 49h ; I
db 4Ah ; J
db 4Bh ; K
db 4Ch ; L

```

在确认了恶意代码使用 Base64 编码后，分析的重点将转移到如何解码这些数据，以及解码后的数据揭示了何种恶意活动或功能。Base64 编码通常用于网络传输中的数据编码，以避免传输过程中的字符解析问题，但在恶意软件中，它也经常用来隐藏和传输可执行代码或其他敏感数据。

5. 什么类型的加密被恶意代码用来发送部分网络流量？

进一步的 IDA 分析揭示了程序在 sub_4010B1 函数中调用了 sub_401000 函数，这是对 BASE64 编码的实际应用部分。在地址 00401201 处，程序获取了主机名，然后在 0040122A 处对其进行了加密。通过在 OllyDbg (OD) 等调试器中观察，可以比较加密前后的主机名变化。

```

call sub_4010B1
add esp, 8
mov byte ptr [ebp+va
lea ecx, [ebp+var_30
push ecx
mov edx, [ebp+arg_0]

```

加密前，主机名为“hanxu-PC”，而加密后，主机名变成了“aGFueHUtUEM=”。这个转换是标准的 Base64 编码过程，其中原始字符串被转换成了一串看似无规则的字符，但实际上可以被逆转回原始字符串。这种编码在网络通信中常用于表示二进制数据或在环境中传输数据，其中直接传输原始数据可能会遇到问题。

```

0012FD84 ASCII "hanxu-PC"
00000000
0012FD8C
7FFDF000
0012FD40
0012FD9C
00000000

```

这个发现表明该恶意程序使用标准的 Base64 编码来创建 GET 请求的字符串。这可能是为了掩盖网络通信中的数据内容，使网络监控工具更难以识别真正的通信内容。

```

call Lab13-01.004010B1
add esp, 8
mov byte ptr ss:[ebp-0x20],0x0
lea ecx,[local.12]
push ecx
mov edx,[arg.1]
push edx
push Lab13-01.0040083C
lea eax,[local.213]
push eax
call Lab13-01.004015FE

```

6. Base64 编码函数在反汇编的何处？

```

text:004010B1
text:004010B1
text:004010B1
text:004010B1
text:004010B1
text:004010B1
text:004010B1
var_14 = dword ptr -14h
var_10 = byte ptr -10h
var_C = byte ptr -0Ch
var_8 = dword ptr -8
var_4 = dword ptr -4
arg_0 = dword ptr 8
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。这个信息是关键的，因为它表明恶意软件设计者在构造网络通信时的限制和意图。通过限制 GET 请求字符串的长度，恶意软件能够保证其网络通信不会因过长的 URL 而受到潜在的网络监控或防火墙拦截。

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。

```
while ( v10 < v9 )
{
    v3 = 0;
    for ( i = 0; i < 3; ++i )
    {
        v7[i] = a1[v10];
        result = v10;
        if ( v10 >= v9 )
        {
            result = i;
            v7[i] = 0;
        }
        else
        {
            ++v3;
            ++v10;
        }
    }
    if ( v3 )
    {
        result = (int)sub_401000((unsigned __int8 *)v7, v8, v3);
        for ( j = 0; j < 4; ++j )
        {
            result = j;
            *(_BYTE *)(v4++ + a2) = v8[j];
        }
    }
}
```

考虑到此情况，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？

该恶意程序首先获取了系统的主机名，例如“hanxu-PC”，然后使用标准的 Base64 编码对其进行加密。这种加密不是为了安全性，而是为了避免在网络传输中被轻易识别。加密后的主机名可能用作向某个外部服务器发送特定信号的一部分。这可能是一个 HTTP GET 请求，其中加密后的主机名作为请求参数的一部分，用于与远程服务器建立通信。恶意程序可能会等待来自该远程服务器的特定响应。这种响应可能是指令、确认信号或其他形式的数据，指导或触发程序的下一步行动。


```

hModule = GetModuleHandleA(0);
if ( hModule )
{
    hResInfo = FindResourceA(hModule, (LPCSTR)0x65, (LPCSTR)0xA);
    if ( hResInfo )
    {
        dwBytes = SizeofResource(hModule, hResInfo);
        GlobalAlloc(0x40u, dwBytes);
        hResData = LoadResource(hModule, hResInfo);
        if ( hResData )
        {
            v1 = LockResource(hResData);
            sub_401190((int)v1, dwBytes);
            result = v1;
        }
        else
        {
            result = 0;
        }
    }
    else
    {
        result = 0;
    }
}
else
{
    printf(aCouldNotLoadEx);
    result = 0;
}
}

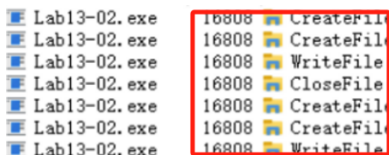
```

程序中可能设定了一个特定的退出条件，当接收到特定的响应后触发。这表明恶意程序的活动是有条件的，可能依赖于外部控制或某种形式的确认信号。

(二) Lab13-02

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，您已经揭示了一些关键的活动。ProcMon 是一个强大的工具，用于实时监控 Windows 操作系统中的进程活动，包括文件系统、注册表和网络活动。

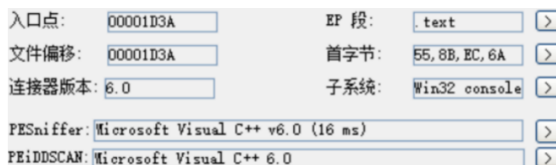


从您的描述中，该程序表现出了频繁的文件创建和写入操作。特别值得注意的是，在特定路径下出现了大量以“temp”开头、大小为 3073KB 的文件。此外，这些文件似乎在固定时间间隔内被创建，其内容的后半部分呈现出十六进制的随机数样式。



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

在这一阶段的分析中，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。



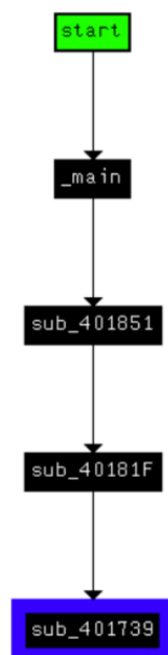
接着检查了是否有加密，同样没有发现明显的加密迹象。



随后，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。

```

call    sub_401070
add     esp, 8
mov     edx, [ebp+nNumberOfBytesToWrite]
push    edx
mov     eax, [ebp+lpBuffer]
push    eax
call    sub_40181F
add     esp, 8
call    ds:GetTickCount
mov     [ebp+var_4], eax
mov     ecx, [ebp+var_4]
push    ecx
push    offset aTemp08x ; "temp%08x"
lea     edx, [ebp+FileName]
push    edx ; char *
call    _sprintf
add     esp, 0Ch
lea     eax, [ebp+FileName]
push    eax ; lpFileName
mov     ecx, [ebp+nNumberOfBytesToWrite]
push    ecx ; nNumberOfBytesToWrite
mov     edx, [ebp+lpBuffer]
push    edx ; lpBuffer
call    sub_401000
add     esp, 0Ch
mov     eax, [ebp+lpBuffer]
push    eax ; hMem

```

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 GetTickCount 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。

```

push    ecx
call    sub_401070
add     esp, 8
mov     edx, [ebp+nNumberOfBytesToWrite]
push    edx
mov     eax, [ebp+lpBuffer]
push    eax
call    sub_40181F
add     esp, 8
call    ds:GetTickCount
mov     [ebp+var_4], eax
mov     ecx, [ebp+var_4]
push    ecx
push    offset aTemp08x ; "temp%08x"
lea     edx, [ebp+FileName]
push    edx ; char *
call    _sprintf
add     esp, 0Ch
lea     eax, [ebp+FileName]
push    eax ; lpFileName
mov     ecx, [ebp+nNumberOfBytesToWrite]
push    ecx ; nNumberOfBytesToWrite
mov     edx, [ebp+lpBuffer]
push    edx ; lpBuffer
call    sub_401000
add     esp, 0Ch
mov     eax, [ebp+lpBuffer]

```

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?


基于对恶意程序的深入分析,特别关注了函数 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。

进一步审视 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, 可以明显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。再回头查看 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。

```
hFile = CreateFileA(lpFileName, 0x40000000u, 0, 0, 2u, 0x80u, 0);
if ( hFile == (HANDLE)-1 )
    return 0;
v7 = WriteFile(hFile, lpBuffer, nNumberOfBytesToWrite, &nNumberOfBytesWritten, 0);
CloseHandle(hFile);
```



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。这样的操作模式在恶意软件中非常常见, 通常用于隐藏其行为的真实目的。

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:恶意程序中负责加密操作的函数是 sub_40181F。这一发现是在深入分析了程序的多个部分, 包括资源节的处理、Base64 编码应用, 以及网络通信相关操作之后得出的。

sub_40181F 函数的作用可能涉及到数据的加密处理, 这在恶意软件中是一个常见的策略, 用于隐藏其行为和传输的数据。例如, 它可能用于加密敏感信息 (如用户凭证、主机标识等), 或用于加密与控制服务器之间的通信内容, 以规避安全软件的检测。

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, 我们现在可以明确地得出以下结论: 该恶意程序具有捕获用户桌面屏幕截图的功能。程序的工作流程涉及调用系统 API 以获取用户的桌面画面, 创建 Bitmap 对象以捕获这些画面, 然后将捕获的屏幕截图加密并存放在用户的桌面上。

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？

根据之前的分析，这个恶意程序的行为包括定时截取屏幕内容，将截图数据存储在缓冲区中，然后对这些数据进行加密并写入文件。由于猜测生成的文件可能是图片格式，尝试将文件后缀改为 png，但由于加密，文件无法打开。由于所使用的加密算法是非标准的，因此难以直接识别和解密。为了进行解密，有几种可能的方法：

1. 通过对恶意软件加密过程的深入理解，可以尝试编写一个解密脚本来逆转加密过程。
2. 通过对加密流量的分析，可能可以找到解密的线索。
3. 在 OllyDbg 中载入样本，定位到加密函数 0040181F 的位置，并修改其行为，使其不执行加密过程，而是直接返回。这可以通过在函数头部添加 ret 指令来实现。

选择直接绕过加密函数是一种简洁有效的方法，因为它允许直接处理未加密的数据。通过这种方式修改恶意程序，执行完毕后，原本加密的文件现在应该保持为未加密状态，因此可以尝试直接打开这个被认为是图片的文件。

0040181F	C2	ret
00401820	8BEC	mov ebp,esp
00401822	83EC 44	sub esp,0x44
00401825	6A 44	push 0x44
00401827	6A 00	push 0x0
00401829	8D45 BC	lea eax,[local.17]
0040182C	50	push eax
0040182D	E8 5E040000	call Lab13-02.00401C98
00401832	83C4 0C	add esp,0xC
00401835	8B4D 0C	mov ecx,[arg.2]
00401838	51	push ecx
00401839	8B55 08	mov edx,[arg.1]
0040183C	52	push edx

如果成功，这种方法不仅能够避免复杂的解密过程，还能直接观察到恶意程序所捕获的屏幕内容，为进一步的分析和对策提供直接证据。

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，这些步骤都是为了准备加密文件的解密过程。详细步骤概述如下：

1. **设置断点：**在函数调用之前设置断点，确保在参数被压入堆栈之前中断程序的执行。这允许您观察和修改传递给加密函数的参数。
2. **文件操作：**open 函数用于打开文件系统中的已加密文件。之后的操作涉及读取这个文件的内容并计算其大小。
3. **内存分配：**使用 remoteVirtualAlloc 调用在当前执行的进程内存空间中创建一个足够大的缓冲区，用于存放即将解密的数据。
4. **数据复制：**writeMemory 操作用于将加密文件的内容复制到新创建的缓冲区中。
5. **变量替换：**利用 writeLong 调用来替换堆栈中加密缓冲区及其大小的变量。
6. **压栈操作：**将这些变量压入堆栈，以便后续的解密和写文件操作可以使用它们。
7. **运行脚本：**执行脚本将触发解密过程，并且可以观察到与之前屏幕截图中相同的解密效果。

```

1  import immllib
2
3  def main():
4      imm = immllib.Debugger()
5
6      # 设置第一个断点
7      imm.setBreakpoint(0x00401875)
8      imm.Run()
9
10     # 打开文件并读取内容
11     cfile = open("C:\\temp\\062da212", "rb")
12     buffer = cfile.read()
13     sz = len(buffer)
14
15     # 在内存中分配空间并写入文件内容
16     membuf = imm.remoteVirtualAlloc(sz)
17     imm.writeMemory(membuf, buffer)
18
19     # 获取寄存器值并修改
20     regs = imm.getRegs()
21     imm.writeLong(regs['EBP'] - 12, membuf)
22     imm.writeLong(regs['EBP'] - 8, sz)
23
24     # 设置第二个断点并运行
25     imm.setBreakpoint(0x0040190A)
26     imm.Run()

```

(三) Lab13-01

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？

通过 Wireshark 抓包得到的结果表明，恶意代码试图通过 DNS 解析域名“www.practicalmalwareanalysis.com”。此外，观察到了 TCP 的三次握手和四次挥手操作，这是网络通信中建立和终止连接的标准过程。这些发现表明恶意代码不仅试图解析特定域名，还试图与远程主机建立 TCP 连接，这通常是为了数据传输或接收指令。在 IDA 中查看字符串信息时，除了明确的域名信息外，其余字符串大多是 API 函数名和看似无意义的乱码。这些乱码可能是加密后的数据或代码。

data:00412208	00000011	C	ijklmnopqrstuvwxyz
data:0041221C	00000021	C	www.practicalmalwareanalysis.com
data:0041224C	00000017	C	Object not Initialized
data:00412264	00000020	C	Data not multiple of Block Size
data:00412284	0000000A	C	Empty key
data:00412290	00000015	C	Incorrect key length
data:004122A8	00000017	C	Incorrect block length
data:004122C8	00000010	C	?.AVexception@@
data:004122E8	00000013	C	?.AVios_base@std@@
data:00412308	0000002E	C	?.AV?\$basic_ios@DU?\$char_traits@D@std@@@std@@
data:00412340	00000032	C	?.AV?\$basic_istream@DU?\$char_traits@D@std@@@std@@
data:00412380	00000032	C	?.AV?\$basic_ostream@DU?\$char_traits@D@std@@@std@@
data:004123C0	00000034	C	?.AV?\$basic_streambuf@DU?\$char_traits@D@std@@@std@@

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？

在 IDA 中搜索 xor 指令的结果显示大量的使用情况。由于 xor 指令在汇编语言中有多种用途，包括寄存器清零和标准库函数操作，因此需要仔细筛选与加密直接相关的用法。在排除了这

些常规用途后，可以确定 6 个可能与加密操作相关的独立函数。

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

由于 xor 指令的简洁性和灵活性，它常被用于实现各种加密算法，特别是在流密码和某些形式的块密码中。在恶意软件的上下文中，xor 通常被用于简单的加密操作，例如单字节 XOR 加密，这是一种基本的加密方法，其中每个字节的数据都与一个固定的字节进行 XOR 操作。别这些函数中使用的具体加密类型需要进一步的分析。

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“krypto analyzer”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。

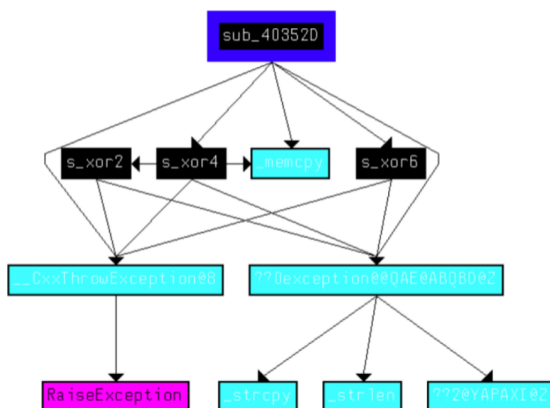
Directi	Ty	Address	Text
Up	r	s_xor2+243	mov edx, ds:Rijndael_Te0[ecx*4]
Up	r	s_xor2+293	mov eax, ds:Rijndael_Te0[ecx*4]
Up	r	s_xor2+2E4	mov ecx, ds:Rijndael_Te0[edx*4]
Up	r	s_xor2+334	mov edx, ds:Rijndael_Te0[ecx*4]
Up	r	s_xor4+1ED	mov ecx, ds:Rijndael_Te0[ecx*4]

Directi	Ty	Address	Text
Up	r	s_xor3+248	mov edx, ds:Rijndael_Td0[ecx*4]
Up	r	s_xor3+298	mov eax, ds:Rijndael_Td0[ecx*4]
Up	r	s_xor3+2E9	mov ecx, ds:Rijndael_Td0[edx*4]
Up	r	s_xor3+339	mov edx, ds:Rijndael_Td0[ecx*4]
Up	r	s_xor5+1F0	mov ecx, ds:Rijndael_Td0[ecx*4]

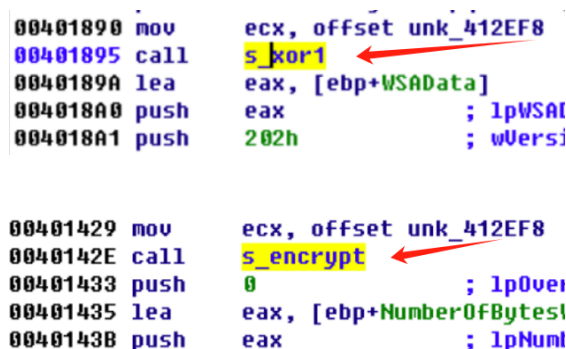
此外，我们还分析了 s_xor6 函数，它涉及到一个循环过程，其中使用了 XOR 指令来执行加密操作。该函数接受两个指针参数，一个指向原始数据缓冲区，另一个指向异或操作的原始数据缓冲区。



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。



综合以上分析，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？

结合前一问的分析，确实可以确认恶意程序使用了 AES 加密以及 Base64 编码技术。AES（高级加密标准）是一种广泛使用的对称加密算法，它提供了强大的加密能力，适用于保护数据的安全性。在这个特定案例中，AES 可能被用来加密敏感数据或通信内容，而 Base64 编码则用于确保加密数据能够在各种网络协议中安全传输。

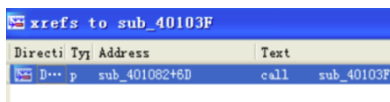
由于 AES 是一种对称加密算法，这意味着它使用相同的密钥进行加密和解密操作。这种特性在某些情况下可能是一个安全隐患，特别是如果密钥管理不当或密钥在不安全的通道中传输。因此，在恶意软件分析和反恶意软件工具开发中，理解和识别这种加密技术的使用是非常重要的。

5. 对于每一种加密技术，它们的密钥是什么？

了解到 AES 加密使用的密钥为“ijklmnopqrstuvwxyz”，以及自定义的 Base64 加密所采用的索引字符串为“CDEFGHIJKLMNOPQRSTUVWXYZABcdefghijklmnopqrstuvwxyzab0123456789+/-”。

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？

继续对恶意软件进行深入分析，我们关注到了之前发现的字符串的引用。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。



xrefs to sub_40103F			
Directi	Ty	Address	Text
00401082+6D	call	sub_40103F	

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。



xrefs to sub_401082			
Directi	Ty	Address	Text
StartAddress+02	call	sub_401082	

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。



xrefs to sub_40132B			
Directi	Ty	Address	Text
sub_4015B7+208	push	offset sub_40132B:lpStarAddress	

在深入分析 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。

```

00401823
00401823 loc_401823:
00401823 mov     eax, [ebp+var_18]
00401826 mov     [ebp+var_58], eax
00401829 mov     ecx, [ebp+arg_10]
0040182C mov     [ebp+var_54], ecx
0040182F mov     edx, dword_41336C
00401835 mov     [ebp+var_50], edx
00401838 lea     eax, [ebp+var_3C]
0040183B push    eax             ; lpThreadId
0040183C push    0                ; dwCreationFlags
0040183E lea     ecx, [ebp+var_58]
00401841 push    ecx             ; lpParameter
00401842 push    offset sub_401320 ; lpStartAddress
00401847 push    0                ; dwStackSize
00401849 push    0                ; lpThreadAttributes
0040184B call    ds:CreateThread
00401851 mov     [ebp+var_20], eax
00401854 cmp     [ebp+var_20], 0
00401858 jnz     short loc_401867

```

```

00401823 mov     eax, [ebp+var_18]
00401826 mov     [ebp+var_58], eax
00401829 mov     ecx, [ebp+arg_10]
0040182C mov     [ebp+var_54], ecx
0040182F mov     edx, dword_41336C
00401835 mov     [ebp+var_50], edx
00401838 lea     eax, [ebp+var_3C]
0040183B push    eax             ; lpThreadId
0040183C push    0                ; dwCreationFlags
0040183E lea     ecx, [ebp+var_58]
00401841 push    ecx             ; lpParameter
00401842 push    offset aes_thread ; lpStartAddress
00401847 push    0                ; dwStackSize
00401849 push    0                ; lpThreadAttributes
0040184B call    ds:CreateThread
00401851 mov     [ebp+var_20], eax
00401854 cmp     [ebp+var_20], 0

```

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。

```

1443 lea     edx, [ebp+var_FE8]
1449 push    edx             ; lp
144A mov     eax, [ebp+var_BE0]
1450 mov     ecx, [eax+4]
1453 push    ecx             ; h
1454 call    ds:WriteFile

```

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。

```

lea     ecx, [ebp+var_18]
push    ecx             ; lpTarget
call    ds:GetCurrentProcess
push    eax             ; hTarget
mov     edx, [ebp+hReadPipe]
push    edx             ; hSource
call    ds:GetCurrentProcess
push    eax             ; hSource
call    ds:DuplicateHandle

```

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。

```

00401944 lea     edx, [ebp+name]
00401948 push    edx                ; name
00401948 mov     eax, [ebp+S]
00401951 push    eax                ; S
00401952 call   ds:connect
00401958 mov     [ebp+var_194], eax
0040195E cmp     [ebp+var_194], 0FFFFFFFh
00401965 jnz     short loc_40196E

```

在 main 中，参数来自使用 connect 调用创建的网络套接字。因此，我们可以得出结论，aes_thread 用于读取 shell 命令的输出结果，并在将其写入网络套接字之前对其进行加密。同时，基于 Base64 加密函数也在一个由宿主进程启动的函数中使用，我们推测 Base64 线程可能会读取远程套接字的内容作为输入，经过解密后，再将结果发送作为命令 shell 的输入。关于密钥可靠性问题的结论是，对于自定义 Base64 加密的实现，索引字符串已经足够。但对于 AES，实现解密可能需要除了密钥之外的其他变量。如果使用密钥生成算法，这可能包括密钥生成算法、密钥大小、操作模式，必要时还包括初始化向量等。

7. 恶意代码做了什么？

结合上述所有分析，这里可以给出结论：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。

8. 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？

首先对 Base64 加密的数据进行解密，Base64 解密是一个标准过程，可以轻松实现。

```

1 import string
2 import base64
3
4 s = ""
5 tab = "CDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"
6 b64 = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/"
7
8 ciphertext = 'BInaEi=='
9
10 for ch in ciphertext:
11     if (ch in tab):
12         s += b64[string.find(tab, str(ch))]
13     elif (ch == '='):
14         s += '='
15
16 print(s)

```

Base64 解密的结果是“dir”，这表明攻击者可能试图发送一个 shell 命令“dir”，用于列出目录的内容。

接下来，还需要进行 AES 解密。AES（高级加密标准）是一种更为复杂的加密方法，需要特定的密钥才能成功解密。需要指出的是，在 IDA 中找到的密钥，这是进行 AES 解密的关键部分。结合之前的 wireshark 捕获到的加密内容，制定如下的解密脚本：

```

1 from Crypto.Cipher import AES
2 import binascii
3
4 raw = '37 f3 1f 04 51 20 e0 b5 86 ac b6 0f 65 20 89 92' \
5       '4f af 98 a4 c8 76 98 a6 4d d5 51 8f a5 cb 51 c5' \
6       'c8 86 11 0d c5 35 38 5c 9c c5 ab 66 78 40 10 df' \
7       '4a 53 f0 11 0f 57 6d 4f b7 c9 c8 bf 29 79 2f c1' \

```

```
8      'ec 60 b2 23 00 7b b8 fa 4d c1 7b 81 93 bb ca 9e' \
9      'bb 27 dd 47 b6 be 0b 0f 66 10 95 17 9e d7 c4 8d' \
10     'ee 11 09 99 20 49 3b df de be 6e ef 6a 12 db bd' \
11     'a6 76 b0 22 13 ee a9 38 2d 2f 56 06 78 cb 2f 91' \
12     'af 64 af a6 d1 43 f1 f5 47 f6 c2 c8 6f 00 49 39'
13
14     ciphertext = binascii.unhexlify(raw.replace(' ', ''))
15
16     obj = AES.new('ijklmnopqrstuvwxyz', AES.MODE_CBC, 'AAAABBBBCCCCDDDD')
17     print('Plaintext is:\n' + obj.decrypt(ciphertext))
```

得到的解密结果为：

```
1  Microsoft Windows XP [Version 5.1.2600] (C) Copyright 1985-2001 Microsoft Corp.
```

四、 实验结论及心得体会

通过这次恶意代码分析与防治技术的实验，我深感恶意软件分析的难度和挑战性，同时也体会到了在网络安全领域深入钻研的重要性。在实验中，我使用了 IDA 等静态分析工具来检查可疑程序的字符串列表，并观察到了一些可能被加密的字符串。这些发现使我认识到，恶意软件设计者通常会使用复杂的方法来隐藏其代码的真实意图，以避免安全软件的检测。此外，我在受控的环境中执行了恶意程序，并对其进行了动态监控。通过监控，我揭示了恶意程序试图访问特定网络地址的行为，并根据动态分析提供的信息推断出了加密字符串的实际内容。这一过程不仅锻炼了我的动态分析技能，而且增强了我对恶意软件行为模式的理解。在实验中，我还实践了对 Base64 和 AES 加密内容的解密过程，其中 Base64 的解密相对简单，但 AES 解密则需要正确的密钥和加密模式。在获得解密结果后，我发现加密内容实际上是操作系统的版权信息。这次实验教会了我，即使是最看似简单的数据，也可能隐藏着恶意软件的线索。

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.