



南開大學
Nankai University

网络空间安全学院
恶意代码分析与防治技术课程实验报告

实验七：OllyDBG

姓名：周钰宸
学号：2111408
专业：信息安全

2024 年 1 月 14 日

1 实验目的

1. 复习教材和课件内第八章和第九章的内容。
2. 完成课本 Lab9 的实验内容，编写 Yara 规则，并尝试 IDA Python 的自动化分析，在此提交实验报告。

2 实验原理

2.1 IDAPro

IDA 是一款广泛使用的交互式反汇编工具，它支持多种平台和文件格式。IDA 的主要功能是将机器代码转换为人类可读的汇编代码，从而帮助研究人员理解和分析程序的功能和行为。

2.1.1 特点及优势

1. **深入分析**：IDA 允许研究人员深入地分析恶意软件的内部工作原理，识别其行为和功能。
2. **多平台支持**：IDA 支持多种处理器架构，如 x86、ARM、MIPS 等。
3. **图形界面**：IDA 提供了一个图形界面，使得代码的流程和结构更加直观，有助于理解恶意代码的执行流程。
4. **高级分析**：IDA 可以识别函数、局部和全局变量、类和其他高级结构。加速了分析过程。
5. **交互式**：用户可以在 IDA 中手动更改、注释和重命名变量和函数，以帮助理解代码。
6. **插件支持**：IDA 支持插件，允许用户扩展其功能。**其中 IDAPython 就是一个十分有用的插件，也是本实验主要使用的插件工具。**

2.2 IDAPython

IDAPython 是一个 IDA 插件，允许用户使用 Python 语言来脚本化 IDA 的功能。这为自动化任务和复杂的分析提供了巨大的灵活性。可以调用 IDA 的所有 API 函数，使用 Python 的功能模块编写强大的自动分析脚本。

2.2.1 特点及优势

1. **自动化**：使用 Python 脚本可以自动化许多繁琐的任务，如标记特定的 API 调用、搜索特定的模式等。
2. **扩展性**：用户可以使用 Python 库来扩展 IDA 的功能，如进行数据分析、图形生成等。
3. **交互性**：IDAPython 允许用户在 IDA 的环境中交互式地运行 Python 代码，这对于快速测试和原型设计非常有用。
4. **快速响应**：当面对新的恶意软件样本时，分析师可以快速地使用 IDAPython 脚本来分析其行为，从而更快地响应威胁。
5. **定制化分析**：由于 IDAPython 的灵活性，分析师可以根据需要定制分析过程，以适应特定的恶意软件家族或攻击技术。

2.3 OllyDBG

OllyDbg 是一个 32 位的汇编级别的调试器，主要用于 Microsoft Windows。它是反向工程和软件分析中的一个流行工具。OllyDbg 的特点是其用户友好的界面、多窗口模式、直接修改代码、以及强大的插件支持。

2.3.1 主要特性

1. Intuitive UI: OllyDbg 的用户界面非常直观，允许用户轻松地查看寄存器、堆栈、内存等。
2. 代码分析：工具可以分析二进制代码，自动识别过程、循环、API 调用等。
3. 插件支持：OllyDbg 支持插件，这意味着您可以添加额外的功能。
4. 断点：支持硬件和软件断点。
5. 直接修改代码：可以直接在内存中修改代码，并即时看到效果。
6. 能够记录指令、寄存器值、API 调用等。

2.3.2 作为恶意代码分析工具的优势

1. 动态分析：使用 OllyDbg，分析师可以动态地执行恶意软件，观察其行为和执行流程。
2. API 调用跟踪：恶意代码通常会调用各种 API 来完成其任务，如文件操作、网络通信等。OllyDbg 允许用户追踪这些调用。
3. 代码逆向：可以深入研究恶意代码的内部逻辑和功能。
4. 解密和脱壳：许多恶意代码使用壳（packers）来避免被检测。OllyDbg 可以帮助分析师手动或自动脱壳。
5. 补丁或修改：分析师可以动态修改恶意代码的行为，例如更改某些条件跳转指令，以便观察不同的执行路径。以研究其不同的行为或用于创建恶意软件样本的变种。

尽管 OllyDbg 是一个强大的工具，但值得注意的是，它主要针对 32 位应用程序。另外，随着现代恶意软件的复杂性增加，分析师可能需要结合使用多种工具进行分析。**本次实验中也是主要使用 OllyDBG 进行对 Lab9 病毒的恶意目的分析。**

2.4 Yara

3 实验过程

3.1 实验环境及工具

虚拟机软件	VMware Workstation 17 Pro
宿主机	Windows 11 家庭中文版
虚拟机操作系统	Windows XP Professional
实验工具 1	OllyDBG 2.01
实验工具 2	IDAPro 6.6.14.1224
配套工具 2	Python 2.7.2

表 1: 本次实验环境及工具

3.2 Lab09-01

用 OllyDbg 和 IDA Pro 分析恶意代码文件 Lab09-01.exe，回答下列问题。在第 3 章中，我们使用基础的静态和动态分析技术，已经对这个恶意代码做了初步分析。

3.2.1 静态与动态分析

1. 基本静态分析

首先进行基本的静态分析，使用 PEiD 查看加壳和导入表情况：

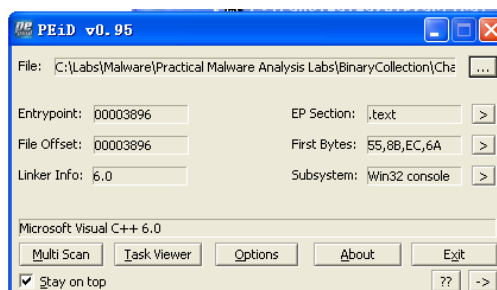


图 3.1: Lab09-01 加壳

图3.1可以看到没有进行任何加壳，由 VC6++ 编写，接下来查看导入表：

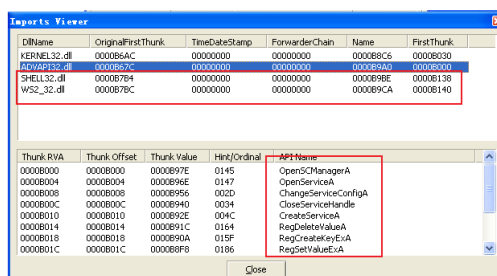


图 3.2: Lab09-01 导入表

图3.2可以看到有许多值得注意的地方：

- ADVAPI32.dll: 其中包含 OpenSCManager 等服务 API 和 RegCreateKeyValue 等注册表操作 API;
- SHELL32.dll: 其中就包含一个 ShellExecuteA, 代表着会使用命令行进行执行操作等;
- WS2_32.dll: 代表着有网络相关的操作, 是 WIN32 下进行 TCPsocket 编程的库。

因此初步推知, 病毒具有服务、注册表、命令行、网络相关的恶意行为。

接下来进一步使用 PEView 查看其字符串:

pFile	Raw Data	Value
0000C000	00 00 00 00 00 00 00 00 00 00 00 EF 42 40 00B@.
0000C010	6C 5A 40 00 00 00 00 00 00 00 00 94 43 40 00	IA@.....@.
0000C020	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0000C030	43 6F 6E 66 69 67 75 72 61 74 69 6F 6E 00 00 00	Configuration...
0000C040	53 4F 46 54 57 41 52 45 5C 4D 69 63 72 6F 73 6F	SOFTWARE\Microso
0000C050	66 74 20 5C 58 50 53 00 5C 6B 65 72 6E 65 6C 33	ft \XPS.\kernel3
0000C060	32 2E 64 6C 6C 00 00 00 00 0A 00 0A 00 00 00 00	2.dll.....
0000C070	20 48 54 54 50 2F 31 2E 30 0D 0A 0D 0A 00 00 00	HTTP/1.0.....
0000C080	47 45 54 20 00 00 00 00 27 60 27 60 27 00 00 00	GET.....
0000C090	60 27 60 27 60 00 00 00 4E 4F 54 48 49 4E 47 00NOTHING.
0000C0A0	72 62 00 00 60 00 00 00 43 4D 44 00 44 4F 57 4E	rb.....CMD: DOWN
0000C0B0	4C 4F 41 44 00 00 00 00 55 50 4C 4F 41 44 00 00	LOAD.....UPLOAD...
0000C0C0	20 00 00 00 53 4C 45 45 50 00 00 63 6D 64 2E	SLEEP.....cmd.
0000C0D0	66 78 65 00 20 3E 3E 20 4E 65 4C 00 2F 63 20 64	exe. >> NUL./c d
0000C0E0	66 6C 20 00 75 70 73 00 68 74 74 70 3A 2F 2F 77	el .ups.http://w
0000C0F0	77 77 2E 70 72 61 63 74 69 63 61 6C 6D 61 6C 77	ww.practicalmalw
0000C100	61 72 65 61 6E 61 6C 79 73 69 73 2E 63 6F 6D 00	areanalysis.com.
0000C110	38 30 00 00 36 30 00 00 20 4D 61 6E 61 67 65 72	80...6D... Manager
0000C120	20 53 65 72 78 69 63 65 00 00 00 2E 65 78 65	Service.....exe
0000C130	00 00 00 00 25 53 59 53 54 45 4D 52 4F 4F 54 25	...%SYSTEMROOT%
0000C140	5C 73 79 73 74 65 6D 33 32 5C 00 00 6B 3A 25 73	system32\...k:%s
0000C150	20 68 3A 25 73 70 70 3A 25 73 20 70 65 72 3A 25	h:%s p:%s per:%
0000C160	73 0A 00 00 2D 63 63 00 2D 63 00 00 2D 72 65 00	s....cc-c...re.
0000C170	20 69 6E 00 00 00 00 00 00 00 00 00 00 00 00	~in.....
0000C180	01 00 00 00 00 00 00 00 00 00 00 00 00 00 00@.....@
0000C190	BA 2D 40 00 01 00 00 00 00 82 40 00 F0 B1 40 00@.....@
0000C1A0	A0 F1 40 00 00 00 00 00 A0 F1 40 00 01 01 00 00@.....@
0000C1B0	00 00 00 00 00 00 00 00 00 10 00 00 00 00 00
0000C1C0	nn nn nn nn nn nn nn nn nn nn nn nn nn nn nn nn

图 3.3: Lab09-01 字符串

图3.3可以看到一些有意思的字符串, 具体而言:

- Configuration 和 SOFTWARE\...\kernel32.dll: 和之前分析导入表的结果相呼应, 这可能是注册表项, 证明了有注册表相关操作。
- NOTHING,...,UPLOAD 等: 可能是一些相关命令, 初步推测可能和导入表的命令行操作有关。
- 彩蛋网址和 80: 这可能和网络相关, 80 是 TCP 的端口。和导入表的 W32dll 相互照应。
- SYSTEMROOT 和 -cc 等: 这些 -cc 有点像命令行的参数类似于 git commit -m 这种, 前者不知道有何作用。

基本静态分析到此结束, 后面将重点结合是用 IDA 和 OllyDBG 分析。

2. 静态和动态结合分析

首先请根据之前在第三章所发现的可知: 病毒 Lab09-01 的 main 函数位于地址 0x402AF0 处, 而进入 main 函数是在 0x403945 处通过命令 CALL 00402AF0。因此进来后通过 step-over 的 F8 达到 0x403945 处:



图 3.6: IDA 自动识别结果

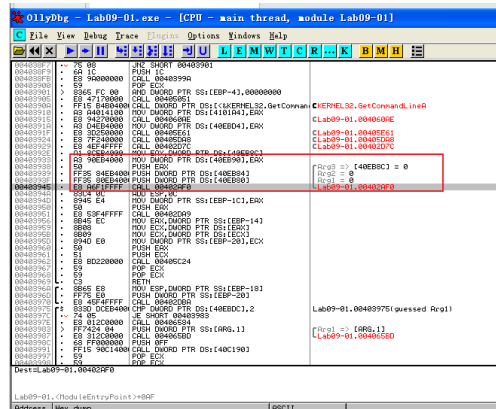


图 3.4: Lab09-01 的 main 函数调用点

然后 step-in 的 F7 进入到 main 函数即 0x402AF0 调用内部:

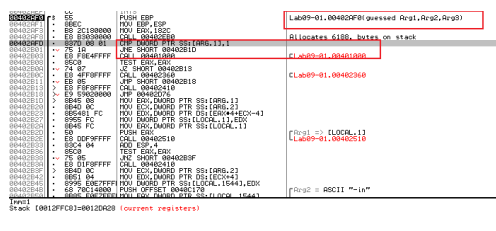


图 3.5: main 函数内部

图3.5可以看到 main 函数有着三个参数, 并且在其中进行了若干次的调用, 首先对调用关系进行一总览. 并使用 IDA 对部分函数查看是否已经有了自动识别的结果: 图??并没有显示任何识别结果, 接下来重点使用 OllyDBG 尝试:

回到图3.5中可以发现函数在 0x2AFD 处的语句: CMP DWORD PTR SS:[ARG,1],1, 具体而言:

- CMP: 用于比较两个操作数的大小。
- DWORD PTR: 指定操作数的大小为 32 位。
- SS: 段寄存器，表示使用栈段。

ARG,1 : 表示从栈中的第一个参数开始偏移 1 个单位的内存地址。

因此这个汇编语句的意思是比较从栈中的第一个参数开始偏移 1 个单位的内存地址中的值（即让这个恶意代码运行的参数个数 `argc` 的值）与 1 是否相等。

由于使用 OllyDBG 直接动态运行的该函数，因此没有显示输入任何参数，**但是函数的第一个参数通常是函数名或者函数指针**，因此这里比较会成功，下面的 JNE 00402B10 不会被成功执行，而会继续直接下面的语句 **call 00401000 直接运行到函数 401000 中**：

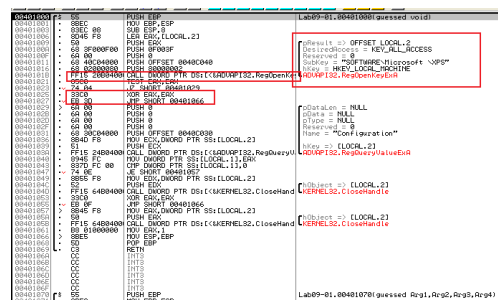


图 3.7: sub_401000

图3.7展示了运行到 sub 401000 中的代码，值得注意到其中的一些函数：

- **RegOpenKeyEx**: 打开一个指定的注册表键。它接受参数包括要打开的注册表的根键**在这里是 HKEY_LOCAL_MACHINE**、子键的名称**在这里是 SOFTWARE\Microsoft\XPS**、访问权限等。通过调用该函数，可以在注册表中定位和打开指定的键。即会打开注册表项。
- **RegQueryValueExA**: 从指定的注册表键中检索一个特定的值。**在这里是要查询 Configuration 即注册表项配置的值**
- **CloseHandle**: 关闭资源句柄。

由此可以初步推测,函数目的是打开注册表键 HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\X 并查询其中的配置值,可能是具有和注册表相关的恶意行为。

按着 F7 单步运行，发现函数提前返回，推测我使用的虚拟机 Windows XP 由于没有这个注册表项，sub_401000 的 RegOpenKeyEx 会打开错误，直接返回 0Retn，然后回到 main 函数中。

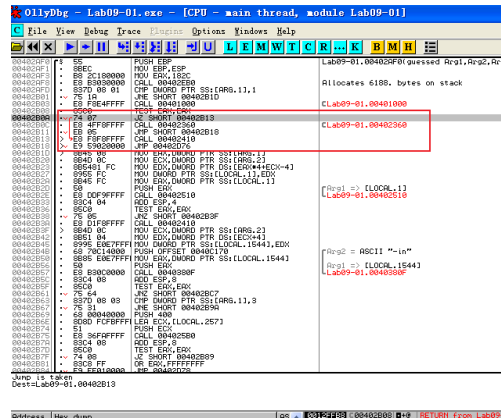


图 3.8: 回到 main

图3.8可以发现：

- test eax, eax: 函数返回 0，因此 eax 寄存器的值为 0，判断后 ZF=0；
- JZ 0x00402B13: 根据红色跳转箭头也可以注意到此时函数跳转到了 x 下面的 00402B13 处，而没有经过 call 00402360；
- call 0x00402410: 在 0x00402B13 处函数调用 sub_00402410。

因此可知，如果函数 sub_401000 函数没有成功打开注册表项，那么函数返回 0，就会跳转到 sub_00402410；若病毒成功打开注册表项。则会调用 sub_00402360。接下来进入 sub_00402410 内部：

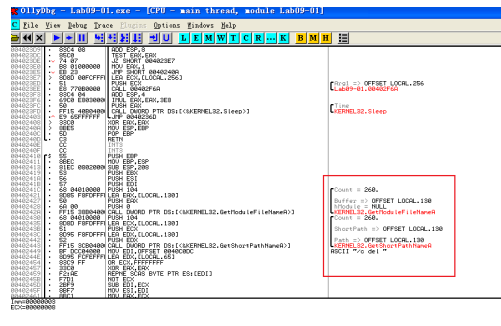


图 3.9: sub_00402410

图3.9可以看到：

- GetModuleFileNameA: 获取指定模块的完整路径名。这里传入的 OFFSET LOCAL.130 经过查阅就是当前恶意代码所在的绝对路径。缓冲区大小足够大以存储完整的路径名。
- GetShortPathNameA: 获取指定路径名的短路径名。返回路径名的长度。同样是绝对的本地地址。
- ASCII "/o del" 和 "»NUL" 等: ASCII 字符串，似乎在暗示什么。
- ShellExecuteA: 窗口命令的可执行函数。
- ASCII "cmd.exe": 代表着会对命令行有操作。

由此可以推断，这些函数可以用于获取文件路径和文件短路径名，以便在编程中进行文件操作。**病毒可能是为了获取自己所在的路径的路径名，然后进行类似删除的操作。**通过单步调试依次验证，运行到一定程度获得如下图所示的结果：

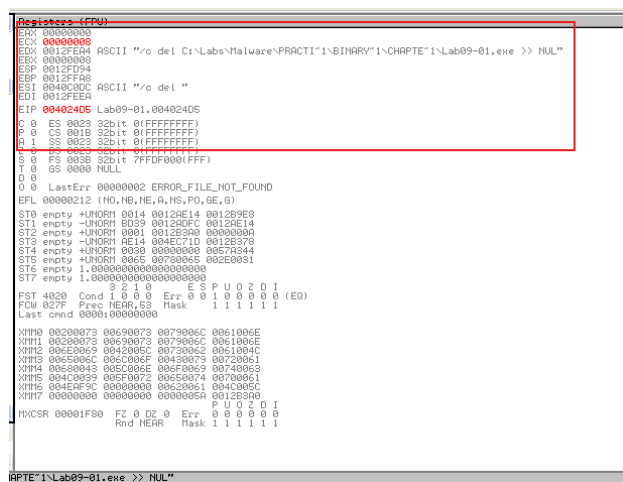


图 3.10: 字符串完整路径

图3.10查看寄存器面板窗口显示了完整的路径 ASCII 码结果：

1

ASCII "\c del C:\Labs\Malware\...\Lab09-01.exe >> NUL "

毫无疑问，这里 EDX 是在暗示病毒要删除自己。除此之外，结合之前发现的 cmd.exe 和 ShellExecuteA，我们可以知道病毒试图在电脑中删除自己。这与我们之前在 Lab3 中动态执行后病毒消失了相呼应，验证了分析正确性。

到此我们可知，病毒发现命令行参数只有一个，然后想打开的注册表项也不存在。因此试图删除自身结束了了。接下来我们要做的是不让恶意代码删除自身，而是强行让他继续运行，由此来观察它的恶意行为，一般有两种选择：

- 添加新的注册表项 HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\XPS 让其查询成功。或者修改路径。但通常来说修改路径可能会导致其利用不了新路径下的值仍然使得病毒运行失败；
- 添加命令行参数，令其在 CMP DWORD PTR SS:[ARG,1],1 中不会跳转进入 sub_401000。

使用第二种方式，加入新的命令行参数，这里选择-in。直接在 OllyDBG 中的 set new Arguments 中加入：

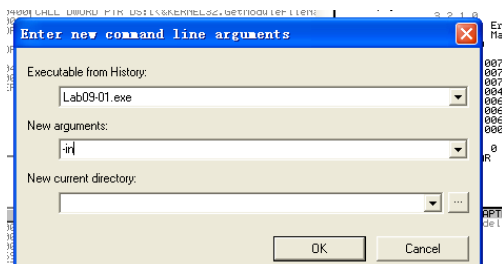


图 3.11: -in 新参数

此时可以让 00402AFD 处的比较不成立，即 ZF=0，JNE 满足，跳转到 00402B1D：

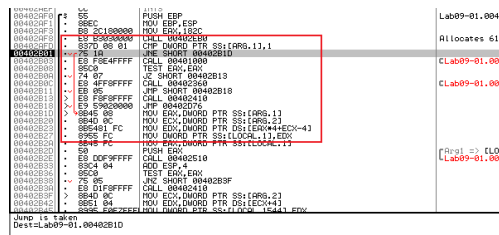


图 3.12: 跳转

图3.12可以看到根据红色箭头的跳转。但发现即使继续运行下去恶意代码函数还是 **u** 会最终在 00402B3A 处再次调用 00402410 删除自身。而唯一让不经过这条路径的方法，就是通过 call 00402510 返回值 eax，jnz 跳转到 00402B3F。

除此之外还注意到 00402510 调用前的参数：

- MOV EAX,...,[ARG,1] 其中我们知道 EAX 指向了 main 函数的第一个参数即 argc 参数个数。
- MOV ECX,...,[ARG,2] 其中 ECX 指向了 argv，即命令行参数的一个数组指针。
- MOV EDX, DWORD PTR DS:[EAX*4+ECX-4] 是在以 ECX 为基地址，一次移动四个 4 节的方式，移动命令函参数个数次，实际上就是移动到了最后一位。
- MOV EAX,[LOCAL,1],...,push eax: eax 被存入了最后一个参数，然和在函数调用前被压入栈顶。

因此我们为了让函数运行成功，去尝试查看 00402510。但我们没有发现明显的规律，也无法发现它究竟在干什么。不过很多 ADD,SUB,MUL,XOR 等指令和 ASCII 码的操作，推测其可能是在进行某种 ACI 的完整性检测。这提示我们可能是在检测输入的最后一个命令行参数是否正确，满足某种硬编码。而直接使用 OllyDBG 判断其行为较为困难，尝试使用 IDA 进行分析：

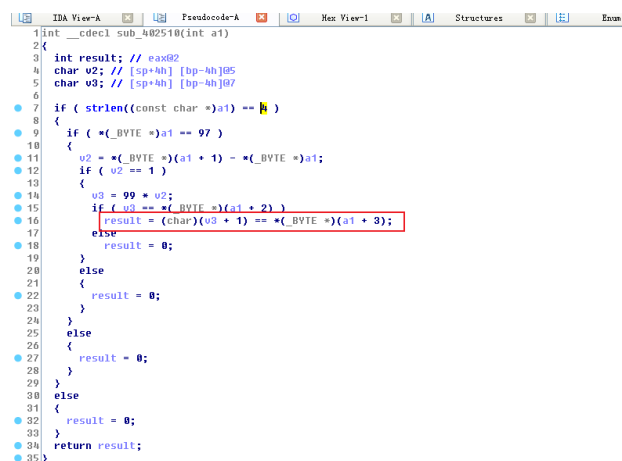


图 3.13: IDA 分析 sub_00402510

图3.13中显示了利用 IDA 将未修改的 Lab09-01.exe 的 sub_00402510 进行反编译的结果。可以发现一些关键的地方：

- 参数 a1: const char 类型的指针，实际上就是之前发分析到的命令行的最后一个参数，压入了栈中。
- ...*(a1)==97: 查看 a1 指针的第一个内容即头部的字符的 ASCII 码是否为 97，而 97 正好是 “a” 的 ASCII 值，即在判断命令行参数最后一个的第一个值是否为 “a”。
- v2: 将 a1 后移，计算与 a1 的头字母之间的 ASCII 码差；值为 1 才继续，即第二个字符串为 “b” 才继续；
- v3:v3 是 v2*99，即 v3 为 ASCII 码 99 的 “c”，然后判断 v3 是否是 a1 的第三个字符；
- result: 最后一部，函数将返回值 result 赋值，只有 a1 的第四个字符正好是 v3 的后一个才会为 1。

由此可知，函数只有整个命令行的最后一个参数正好为 abcd 时函数才会返回 1，前面任何一步不满足都会返回 0。由此可知，命令行的最后一个参数即某种硬编码的密码时 abcd。

如果恶意代码通过不了这里的验证（即不能通过 test eax, eax），那么就会回到 main 然后到 00402410 删除自身。但仔细一想，我们可以使用 OllyDBG 强行让其 eax 返回为 1！由于可以通过修改汇编指令的方式改变程序流向。

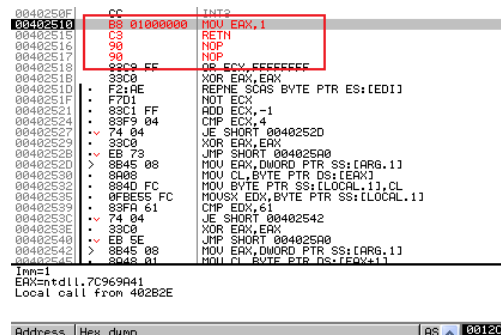


图 3.14: 修改汇编指令

图??可以看到我们直接通过替换指令为 mov eax, 0x1 和 ret 实现。将从 0x402510 开始的几个字节替换为 B8 01 00 00 00 C3 90 90。其中由于对应的二进制指令长于原来的代码，所以多出的位置使用空指令 nop 填充。

而后将修改复制到可执行文件后保存，将文件保存为新的名字”Lab09-01-modified.exe”，然后重新载入文件执行，令命令行参数依然为-in（其实这次什么都可以了），然后运行到下面代码：

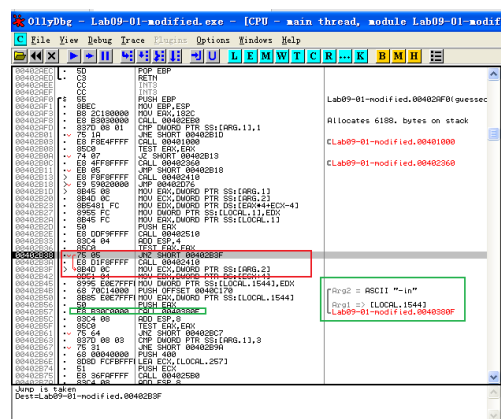


图 3.15: Lab09-01-modified 执行流程被改变

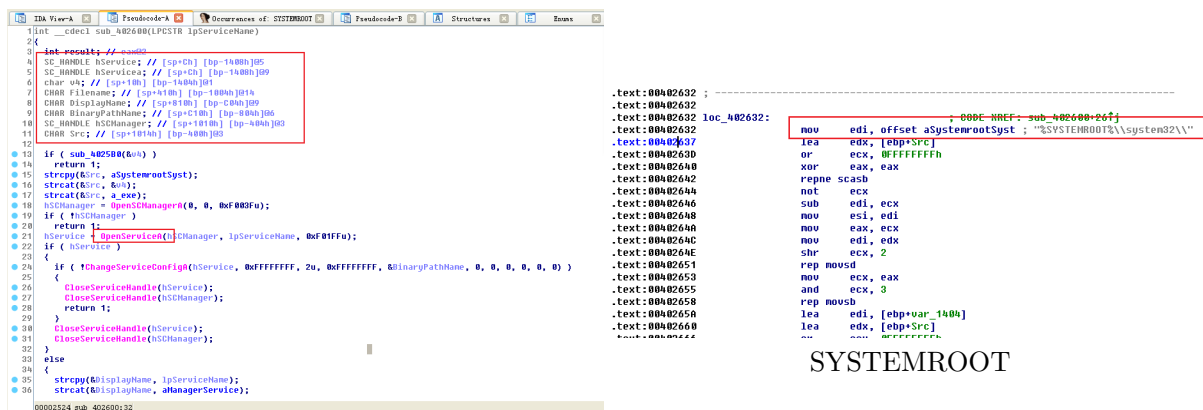
图3.15可以看到根据红色箭头，这次的命令执行过程被修改了！因为 `eax` 返回值为 1 了，`jnz` 发生跳转，跳过了删除自己的 00402410，调用函数 0040380F。直接查看这个函数内部没发现特别的点，想到 IDA 可能对其进行了标记，查看 IDA：

```
.text:0040380F __mbscmp __cdecl __mbscmp(const unsigned int8 * const unsigned int8 *)
.text:0040380F __mbscmp proc near
.text:0040380F ; CODE XREF: _main+677p
.text:0040380F ; _main+E7fp ...
.text:0040380F arg_0 = duword ptr 4
.text:0040380F arg_4 = duword ptr 8
.text:0040380F cmp dword 40E40, 0
.text:0040380F push ebx
.text:0040380F push esi
.text:0040380F push edi
.text:0040380F jnz short loc_40382C
.text:0040380F push [esp+0Ch+arg_4] ; char *
.text:0040380F push [esp+10h+arg_0] ; char *
.text:0040380F call _strcmp
.text:0040380F pop ecx
.text:0040380F pop ecx
.text:0040380F jmp short loc_403892
.text:0040382C ;
.text:0040382C loc_40382C: ; CODE XREF: __mbscmp+6fj
.text:0040382C mov esi, [esp+0Ch+arg_4]
.text:0040382C mov edi, [esp+10h+arg_0]
.text:00403834
```

图 3.16: `__mbscmp`

图3.16可以看到 0040380F 已经被标记为了 `__mbscmp` 函数，是一个字符串匹配函数，推测是根据检查列表支持的命令行参数选项，来判断恶意代码接下来的行为。

接下来继续使用 IDA 深度分析恶意代码的一些行为，发现了如下：



服务相关

图 3.17: 深度分析

图3.17可以看到 0x4026CC 处深度分析发现的一些东西：

- `OpenSCManager` 和 `OpenServiceA`：前者用于打开服务控制管理器数据库，它返回一个句柄，可以用于后续的操作，比如创建、删除和启动服务等；后者用于打开一个已存在的服务。由此可知病毒在尝试和服务有关的行为。
- 恶意代码使用一个与其可执行文件相同根路径 `basename`，`basename` 是去除目录路径和文件扩展名信息之后的文件名。如果服务不存在，则恶意代码以管理器服务 `basename` 作为名字，创建一个自启动的服务。
- 设置二进制路径为 `%SYSTEMROOT%\system32\filename>` 即 `aSystemrootSyst`。

然后还有：

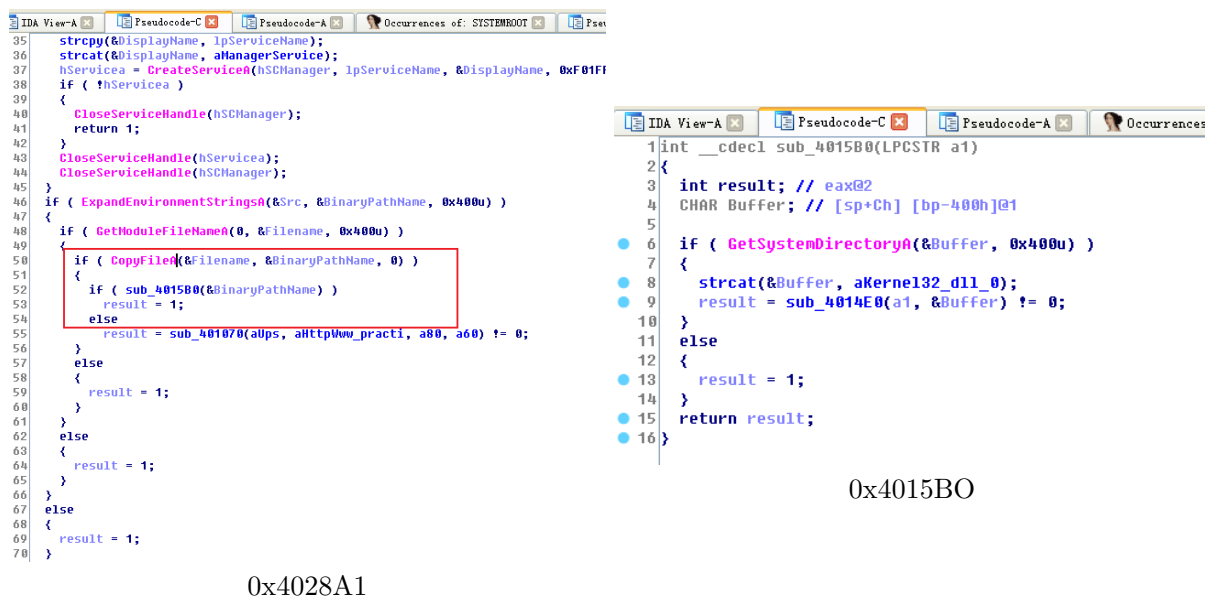


图 3.18: 其它操作

图3.18可以发现:

- 在地址 0x4028A1 处, 恶意代码通过 CopyFileA 将自己复制到%SYSTEMROOT%\system32 目录下。
- 在地址 0x4015BO 处, 恶意代码改变了复制文件的修改、访问和最后变化时间戳, 来与那些 kernel32dll 等系统文件保持一致。

接下来查看 sub_401070 处的反编译:

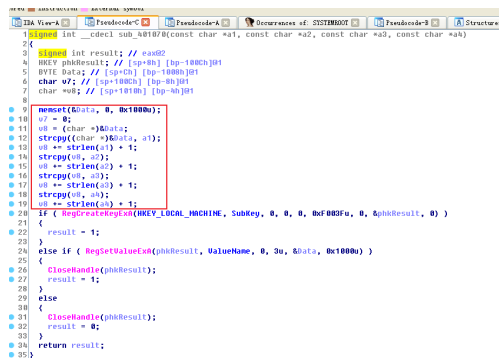


图 3.19: sub_401070

图3.19可以看到如下关键信息:

- Data: 对 Data 数组初始化并且 memset 设置为 0;
- v8 与 4 个 a: 将 v8 指向 Data 数组, 然后利用四个传递的参数 a1 到 a4, 以 _0 作为分割, 将参数复制到 v8 指向的 Data 数组中。即包含四个完整的 char 字符串。
- RegCreateKeyExA: 函数试图创建之前分析过的 HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\XPS 的注册表键, 如果已经存在, 直接打开; 如果都无法成功, result 为 1, 返回。

- RegSetValueExA：函数尝试设置这个注册表键下的 **Configuration** 值，其类型将其设置为表示多个完整的字符串的常量 **REG_MULTI_SZ**。其值为填好了参数的 **Data** 数组。同样如果过程出现失误比如设置值等发生错误，会关闭句柄后返回 1；
- 全部操作成功，函数返回 0。

由此可知，该函数作用为将参数的四个字符串保存到注册表中，并且正确设置其值的类型。其中值得注意的是 Microsoft 后面有一个空格，这个是病毒故意设置的，因此可以作为一个独特的感染主机标识。

但我们不知道参数即缓冲区中的值，通过 EDX 寄存器指向缓冲区内容的提示信息，在 0x4011BE 设置断点，运行到这个位置后发现了对应的 4 个 NULL 结束的字符串为：

- ups：代表不间断电源，是一种网络设备，用于提供在电力故障或停电情况下的电力保护，以确保网络设备的可靠性和连通性。
- <http://www.practicalmalwareanalysis.com>：经典彩蛋网址。
- 80：这是 TCP/IP 协议中用于 HTTP 通信的默认端口号。
- 60：同样可能是某个端口号。

因此可以推断出恶意代码项注册表项中加入的内容和网络相关，经典的网址，又是你。。

3. 命令行选项分析

这里最初分析的时候遇到了卡壳，查看书后参考答案后找到了解决方式，为了探索这个恶意代码所支持的全部命令行选项，可以考虑在 main 函数中找到所有的 __mbscmp 函数，这个函数按照之前分析的专门用来对比字符串即命令行是否符合要求的。

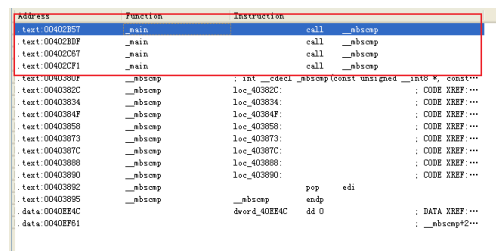


图 3.20: 查看 __mbscmp 调用位置

图3.51可以看到一共支持四个命令行选项。而这些 __mebscmp 其实是不同的 __mebscmp，对应四个命令行的检查。是我们之间分析字符串的静态分析结果之一。

命令行选项	调用代码地址	实现代码地址	对应行为
-in	0x00402B5F	0x402600	安装服务
-c	0x00402BDF	0x402900	设置注册表配置键
-cc	0x00402C67	0x401070	打印注册表配置键
-re	0x00402CF1	0x401280	卸载服务

表 2: Lab09-01 的恶意行为选项

表2可以看到这些是我通过查看 __mbscmp 出现位置总结的恶意代码支持的所有选项集合。现在使用之前已经修改过的 Lab09-01.exe 依次运行上述四个命令，结果如下图所示，其中有彩蛋哦！学姐别错过：

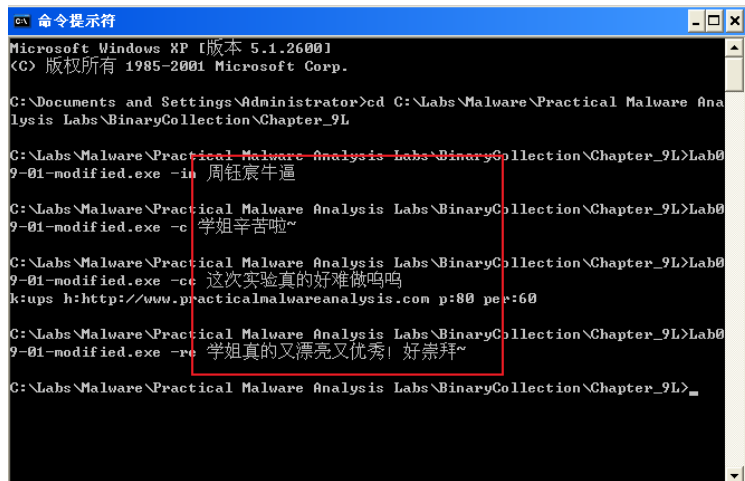


图 3.21: Lab09-01 病毒命令行选项尝试

图3.21可以看到我随便输入了一些中文还有四个命令行选项，其中输入-cc 后打印的内容是注册表键值即 k:ups h:http://www.practicalmalwareanalysis.com p:80 per:60。

值得一提的是，病毒如果被安装后没有提高任何命令行参数，恶意代码会获取当前注册表项的配置，调用一个函数，睡眠数秒，然后一直重复这个动作。

4. 后门分析

除此之外，这个恶意代码还存在着很多和后门相关的函数，比如 0x00402020 处的函数：

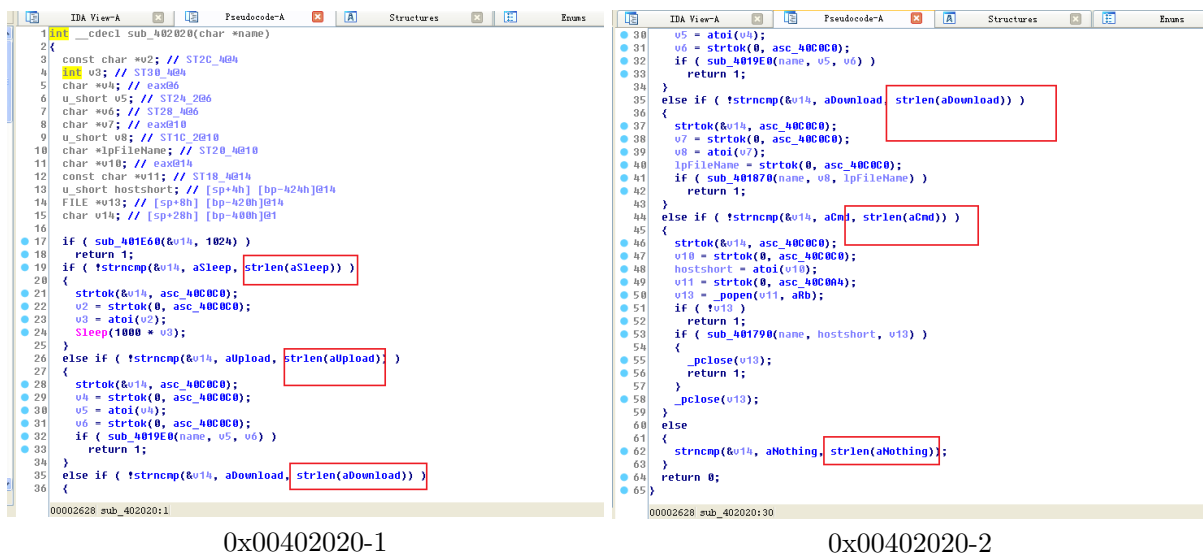


图 3.22: 0x00402020

图3.22可以发现此处会将 0x401E60 函数返回的值作为参数然后压入栈中。在函数内容会对字符串开头和很多硬编码的值进行挨个比较，包括 SLEEP,UPLOAD,DOWNLOAD,CMD 甚至是 NOTHING。只要满足其中任何之一除了 NOTHING 都会返回 1 代表成功，否则返回 0 代表失败，很类似对于命令行参数的解析过程。

具体而言，下面列出了全部可支持的命令，也是我们之间分析字符串的静态分析结果之一：

后门命令	函数地址	命令格式	对应行为
SLEEP	0x00402076	SLEEP seconds	休眠若干秒
UPLOAD	0x004019E0	UPLOAD port filename	通过端口 port 连接远程主机读取内容并创建本地文件
DOWNLOAD	0x00401870	DOWNLOAD port filename	通过端口 port 连接远程主机并发送本地文件
CMD	0x00402268	CMD port command	使用命令行运行 shell 的 commad 命令并发送远程主机
NOTHING	0x00402356	NOTHING	无操作

表 3: 恶意代码后门功能列表

5. 网络行为

由上面的分析可知, Lab09-01.exe 是一个后门, 可以运行任意的 shell 命令, 并且具备上传和下载功能。另外记得之前发现的彩蛋网页嘛? **证明病毒还存在着相关的网络行为, 不过由于篇幅原因, 这里不再详细阐述和放出我的分析图片等过程, 我直接说出我发现的结论:**

- 使用 OllyDbg 调试代码路径, 研究了从地址 0x401E60 开始的函数, 发现命令会从远程主机传回到恶意代码, 从而创建网络特征码。
- 0x401420 开始的函数将 ASCII 编码字符串写入缓冲区, 推测函数从 Windows 注册表获取特定的配置值。
- 0x401470 开始的函数返回数字 8, 与服务器相关的端口号。
- 0x401D80 开始的函数每次调用返回不同的包含随机字符的 ASCII 字符串。
- 恶意代码构造 HTTP/1.0 GET 请求并连接远程系统, 尝试连接注册表键值配置中记录的域名和端口。
- 这个病毒还会服务器返回的文档中搜索特殊的字符串, 并用它们来表示命令控制协议。

6. 病毒总结

基于以上的分析, 我们对病毒 Lab09-01 进行一个总结。它是一个具有反向连接功能的后门程序。在安装、配置和移除恶意代码时, 需要提供密码"abcd" 作为最后一个参数。恶意代码会将自身复制到%SYSTEMROOT%\WINDOWS\system32 目录, 并创建一个自启动服务来确保自身的安装。

一旦安装并运行, 恶意代码会使用注册表来获取服务器的配置信息, 并向远程系统发送一个 HTTP/1.0 的 GET 请求。命令控制协议被嵌入到响应文档中。该恶意代码能够识别 5 条命令, 其中包括执行指定的任意 shell 命令。

3.2.2 实验问题

完成了上面的分析后, 现在开始对书后的问题进行集中回答:

1. Q1: 如何让这个恶意代码安装自身?

回答: 需要提供一份密码 abcd (作为最后一个命令), 和-in 的命令行选项。即可让其安装自身而不是删除自己。

2. Q2: 这个恶意代码的命令行选项是什么？它要求的密码是什么？

回答：命令行选项包括四个：-in,-c,-cc 和-re，具体分析详见表2。密码是 abcd 字符串，且需要在命令行选项后面作为最后一个命令。默认情况下一旦安装，这个恶意代码将成为一个后门程序。

3. Q3: 如何利用 OllyDbg 永久修补这个恶意代码，使其不需要指定的命令行密码？

回答：通过更改地址 0x402510 处函数开头的几个字节的代码，即 MOV EAX, 0x1;RETN;, 对应十六进制序列为 B8 01 000000 C3，这样它就可以改变程序流。

4. Q4: 这个恶意代码基于系统的特征是什么？

回答：有三个，具体而言：

- :HKLM\Software\Microsoft \XPS\Configuration 注册表项：Microsof 后具有一个空格！由恶意代码创建。
- 任意名字的管理服务：由于命令行参数确定，者是恶意代码可执行文件的名字。
- 更改文件名：当恶意代码将自己复制到 Windows 系统目录时，它可能更改文件名为相匹配的服务名。

5. Q5: 这个恶意代码通过网络命令执行了哪些不同操作？

回答：借助于网络恶意代码会执行下面 5 个命令之:SLEEP、UPLOAD、DOWNLOAD、CMD 或 NOTHING。具体的分析详见表3。

6. Q6: 这个恶意代码是否有网络特征？

回答：恶意代码会发送资源的 HTTP/1.0 GET 的资源请求向远程主机,同时向 http://www.practicalmalwa 蛋网址发出标识信号。

资源内容可通过 xxxx/xxxx.xxx 的格式进行配置。

3.3 Lab09-02

用 OllyDbg 分析恶意代码文件 Lab09-02.exe，回答下列问题。

3.3.1 静态与动态分析

1. 基本静态分析

首先使用 PEiD 查看其加壳情况和导入表：

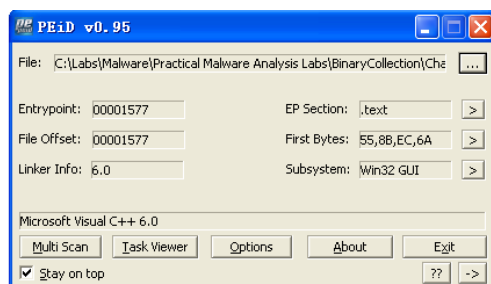


图 3.23: Lab09-02 加壳情况

图3.23可以看到没有任何加壳，同样 VC6++ 编写，接下来查看其导入表：

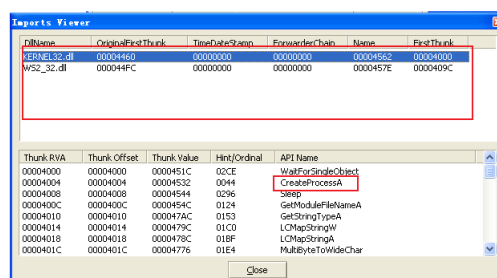


图 3.24: Lab09-02 导入表分析

图3.24分析了其导入表情况，有一些值得注意：

- KERNEL32.dll: 其中含有 CreateProcessA 用于创建进程；
- WS2_32.dll: 包含 WSAsocket, 推测会使用 socket 的 TCP 编程接口；

推测病毒可能具有连接远程主机，然后创建新的进程用来接收或者发送消息，这一点我们计算机网络第一次 socket 编程进行过实验，就是多线程编程。

接下来使用 PEView 查看下字符串：

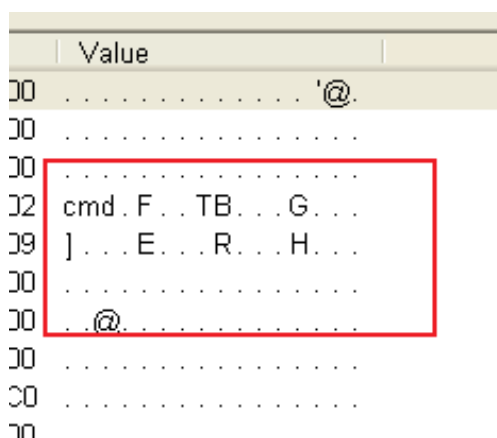


图 3.25: Lab09-02 字符串

图3.25看到了一个有意思的 cmd，推测也和操作命令行有关。还有一些导入函数。

基本静态分析到此结束，后面将重点结合是用 IDA 和 OllyDBG 分析。

2. 静态和动态结合分析

首先使用 IDA 查看其 main 函数：



图 3.26: IDA 查看 main 函数

图3.30可以明显看到一些值得注意的地方：

- strcpy: 创建了两个字符串 1qaz2wsx3edc 和 ocl.exe。推测前者是一个密码，后者是一个库函数调用。
- GetModuleFileNameA: 获取当前可执行文件即本病毒的路径；
- WSAStartup 和 WSASocketA, gethostbyname: 明显这些就在初始化 WSA 网络环境后初始化 socket 连接远程主机等。
- sub_401000: 这个函数内部做了一些事，然后就关闭 socket 了还进入了休眠。推测其为恶意行为的主函数，可能是在利用远程主机进行后门控制。

然后使用 OllyDBG 分析 main 函数（在 0x00401128 处）：

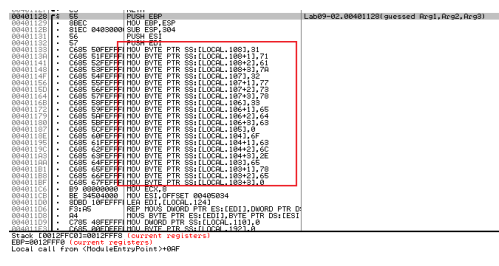


图 3.27: ODBG 查看 main

图3.25可以看到恶意代码在栈上创建了一大堆 ASCII 码并在结尾用 NULL 结束，即两个完整的字符串，并且每次向栈中移动一个字符，以此实现字符串混淆。将两个字符串解密后，就是之前 IDA 分析发现的 1qaz2wsx3edc 和 ocl.exe。

然后继续查看其 main 函数部分，发现其调用了 GetModuleFileNameA 和 sub_00401550 函数：

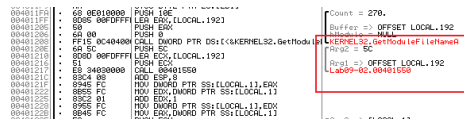


图 3.28: GetModuleFileNameA 和 sub_00401550

图??可以看到恶意代码调用的前者该函数通常用于获取当前进程或其他进程中的模块文件名，和之前静态分析导入表中提到过，然后查看其后者的汇编代码很难发现其是什么，于是直接查看其 IDA 的标记：

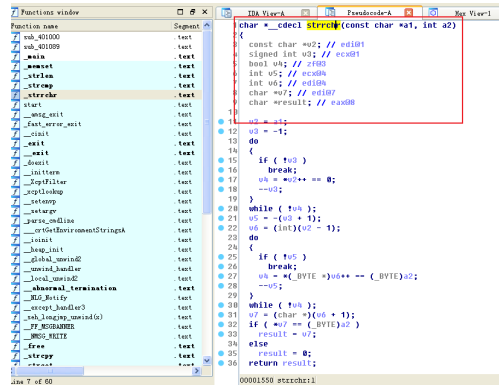


图 3.29: sub_00401550 在 IDA

图3.29可以看到这个函数被识别为了 __strchr，这个函数在字符串处理中经常被用来查找字符串中最后一个出现的某个字符，或者判断字符串中是否包含某个特定字符。因此判断其和字符串对比有关，很可能是之前的 1qaz2wsx3edc 和 ocl.exe 有关。

然后回到 OllyDBG 中发现接下来调用了 0x004014C0，IDA 将其识别为 strcmp：

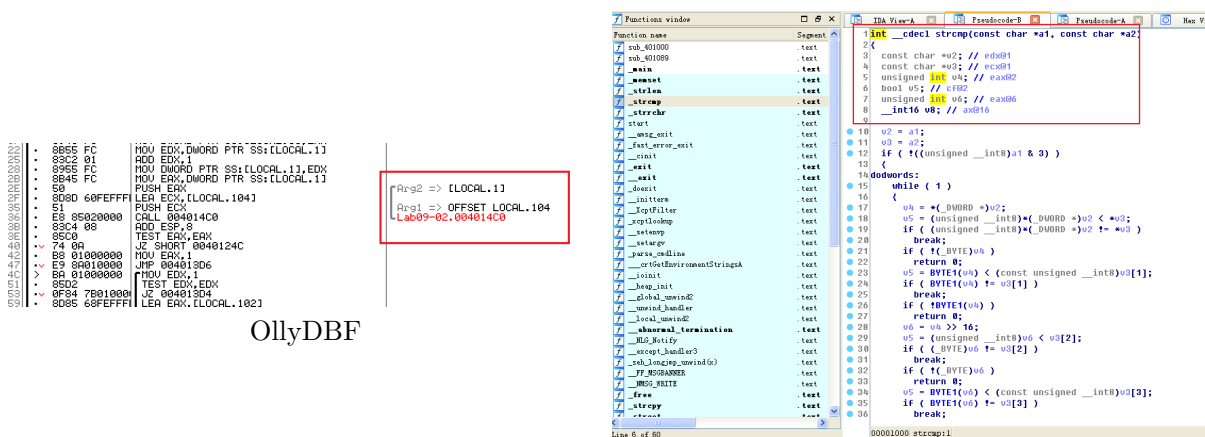


图 3.30: 0x004014C0

为了理清它这两个函数的调用逻辑,通过查看堆栈,了解程序的执行流程。直接 F4 运行到 0x004014C0 处后查看堆栈：

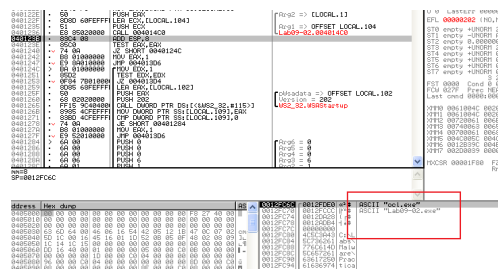


图 3.31: 0x004014C0 堆栈

图3.31可以看到正在试图将恶意代码的文件名即 **Lab09-02.exe** 和 **olc.exe** 进行字符串的比较。而后可以注意到如果比较结果不满足即不为 0，那么不会通过 JZ 达到 0040124C，而是通过 00401306 的跳转提前结束。因此为了进行后面的网络相关操作，这步比价必须成功，因此推出只有恶意代码被命名 **ocl.exe** 才能正常运行执行剩下正式的功能。

因此我们将病毒名字改为 **ocl** 后重新加载，然后发现这次可以正常运行。运行经过 WSA 环境和 socket 的初始化后达到 0x004012BD 处：

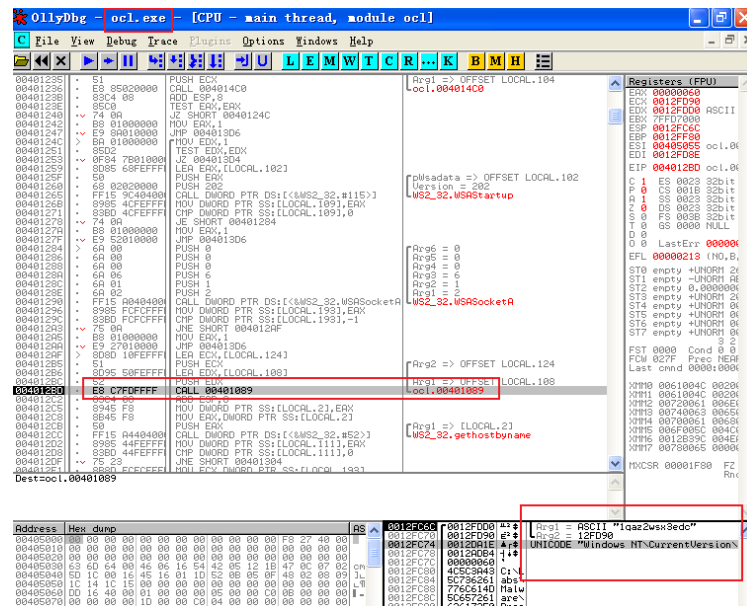


图 3.32: 0x004012BD

此处对 **sub_00401089** 这个子过程进行了调用,堆栈可以发现之前说过的密码 **1qaz2wsx3edc** 的 ASCII 串, 另一个 12FD90 的奇怪参数和一个路径。

然后使用 IDA 到 0x00401089 中查看其功能：

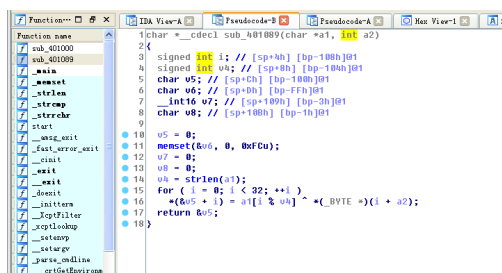


图 3.33: 0x00401089 功能

图3.33可以看到其通过进行多次字节异或和循环实际上在进行解密或者是某种程度上的反混淆。

使用 OllvDBG 运行通过该函数后查看结果堆栈:

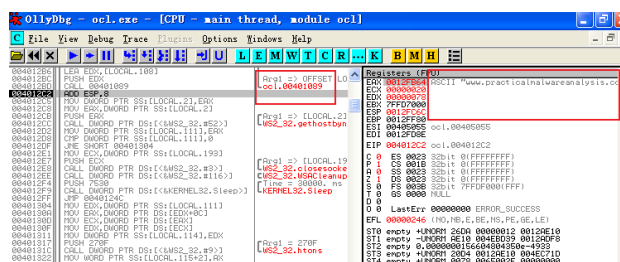


图 3.34: 解密结果

图3.34看到越过该函数后寄存器区显示了经典的彩蛋网址，因此可以得出结果函数 `0x00401089` 利用密钥对字符串进行反混淆，最终完整解密。

解密后的网址被传给 `gethostbyname` 函数作为参数。用于根据主机名获取主机的 IP 地址。它接受一个主机名作为参数，并返回与该主机名对应的 IP 地址。这个函数在网络编程中经常被用来进行主机名解析，以便与远程主机建立连接。由于利用返回的主机 IP 地址，来补充晚上 `sockaddr_in` 结构体，这也是 WinTCP 网络编程的一部分。

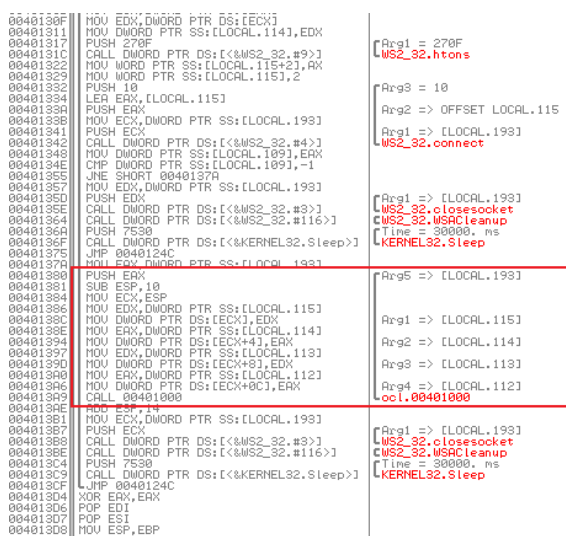


图 3.35: 而后 WinSocket

而后图3.36函数会经过一系列经典 Winsocket 编程操作, 设置服务器端口号 (9999), 设置结构体

属性 AF_INET 即使用 IPv4 地址, 使用 htonl 进行地址转换, 然后还有 connect 等 socket 函数, 值得注意的是, 只有连接成功那个彩蛋网址, 才会进入 0x401000 中, 否则会调用 Sleep 休眠 30 秒。

直接分析这个函数不简单, 回去使用 IDA 查看其反编译:

```

0040138E  MOV EDI,DWORD PTR DS:[ECX]
00401391  MOV DWORD PTR SS:[LOCAL.114],EDI
00401394  PUSH 270F
00401397  CALL DWORD PTR DS:[<MS2_32.#9>]
0040139A  MOV WORD PTR SS:[LOCAL.115+2],AX
0040139D  MOV WORD PTR SS:[LOCAL.115],2
004013A0  PUSH 10
004013A3  LEA EDI,[LOCAL.115]
004013A6  PUSH EAX
004013A9  MOV ECX,DWORD PTR SS:[LOCAL.193]
004013AC  PUSH ECX
004013AF  CALL DWORD PTR DS:[<MS2_32.#4>]
004013B2  MOV DWORD PTR SS:[LOCAL.109],EAX
004013B5  CMP DWORD PTR SS:[LOCAL.109],-1
004013B8  JNE SHORT 0040137A
004013BB  MOV EDI,DWORD PTR SS:[LOCAL.193]
004013BE  PUSH EDI
004013C1  CALL DWORD PTR DS:[<MS2_32.#3>]
004013C4  CALL DWORD PTR DS:[<MS2_32.#116>]
004013C7  PUSH 7530
004013CA  CALL DWORD PTR DS:[<KERNEL32.Sleep>]
004013CD  JMP 0040124C
004013D0  MOV EDI,DWORD PTR SS:[LOCAL.193]
004013D3  PUSH EDI
004013D6  SUB ESP,10
004013D9  MOV ECX,ESP
004013DC  MOV EDI,DWORD PTR SS:[LOCAL.115]
004013DF  MOV DWORD PTR DS:[ECX],EDI
004013E2  MOV EAX,DWORD PTR SS:[LOCAL.114]
004013E5  MOV DWORD PTR DS:[ECX+4],EAX
004013E8  MOV EDI,DWORD PTR SS:[LOCAL.113]
004013EB  MOV DWORD PTR DS:[ECX+8],EDI
004013EE  MOV ECX,DWORD PTR SS:[LOCAL.112]
004013F1  MOV DWORD PTR DS:[ECX+0C],EAX
004013F4  CALL 00401000
004013F7  ADD ESP,DWORD PTR SS:[LOCAL.193]
004013FA  PUSH ECX
004013FD  CALL DWORD PTR DS:[<MS2_32.#3>]
00401400  CALL DWORD PTR DS:[<MS2_32.#116>]
00401403  PUSH 7530
00401406  CALL DWORD PTR DS:[<KERNEL32.Sleep>]
00401409  XOR EDI,EDI
0040140C  POP EDI
0040140F  POP ESI
00401412  MOV ESP,EBP
00401415  RETN
  
```

```

[Arg1 = 270F]
WS2_32.htons

[Arg3 = 10]
[Arg2 => OFFSET LOCAL.115]
[Arg1 => [LOCAL.193]]
WS2_32.connect

[Arg1 => [LOCAL.193]]
WS2_32.closesocket
cWS2_32.WSACleanup
[Time = 30000 ms]
[KERNEL32.Sleep]

[Arg5 => [LOCAL.193]]

[Arg1 => [LOCAL.115]]
[Arg2 => [LOCAL.114]]
[Arg3 => [LOCAL.113]]
[Arg4 => [LOCAL.112]]
ccl.00401000

[Arg1 => [LOCAL.193]]
WS2_32.closesocket
cWS2_32.WSACleanup
[Time = 30000 ms]
[KERNEL32.Sleep]
  
```

图 3.36: 0x401000 反编译

图3.36可以看到一些重点的函数:

- **StartupInfo**: 结构体初始化, 其中值得注意的是 **wShowWindow = 0** 字段代表着窗口不可见。根据参数 **a5** 即套接字的句柄来控制 **Input**、**Output** 和 **Error**。
- **ProcessInformation**: **memset** 为全 0
- **CreateProcessA**: 使用 **StartupInfo** 和 **ProcessInformation** 作为参数, 创建并运行一个新的命令行进程。
- **WaitForSingleObject**: 阻塞当前线程, 直到指定的进程终止或超过指定的等待时间。**hProcess**, 即新创建的进程的句柄。0xFFFFFFFF, 表示无限等待, 即一直等待直到进程终止。作用是等待新创建的进程的终止, 并且会阻塞当前线程直到进程终止。

综上所述, **sub_401000** 函数会在 **main** 函数创建网络行为并且连接成功后使用, 然后阻塞当前进程, 创建并运行一个新的不可见的命令行进程。并且该进程实际上会通过传入的套接字 **a5** 于远程主机进行性能通信, 从而允许服务器能够远程发送命令并在本地执行。这是一种反向 Shell 的恶意行为。

3.3.2 实验问题

完成了整体对病毒 Lab09-02 的分析后, 来集中回答问题:

1. Q1: 在二进制文件中, 你看到的静态字符串是什么?

回答: 导入函数和字符串 **cmd**。

2. Q2: 当你运行这个二进制文件时, 会发生什么?

回答: 如果直接运行这个函数会直接终止。

3. Q3: 怎样让恶意代码的攻击负载 (payload) 获得运行?

回答: 将其名字改为 `ocl.exe`。那么它将会和远程服务器建立一个连接, 并通过一个反向 Shell 让远程主机对被感染的主机进行远程命令的操纵。其中若连接断开, 则会等待 30 秒后重试。值得注意的是不论是否运行成功, 都不会显示什么东西, 因为创建进程的窗口是不可见的。

但由于它其实没法翻墙上外网, 永远连接不会成功 (乐)。

4. Q4: 在地址 0x00401133 处发生了什么?

回答: 字符串密码 `1qaz2wsx3edc` 在栈上被创建, 其目的是混淆静态分析技术和字符串工具中的字符串。让单纯的静态分析失效。

5. Q5: 传递给子例程 (函数) 0x00401089 的参数是什么?

回答: 字符串 `1gaz2wsx3edc` 和一个数据缓冲区。

6. Q6: 恶意代码使用的域名是什么?

回答: 经典彩蛋网址 `practicalmalwareanalysis.com`

7. Q7: 恶意代码使用什么编码函数来混淆域名?

回答: 将字符串 `1qaz2wsx3edc` 循环混淆, 异或加密的 DNS 名来解密域名

8. Q8: 恶意代码在 0x0040106E 处调用 `CreateProcessA` 函数的意义是什么?

回答: 重定向 `stdout` 和 `stderr` 的句柄到 `socket` 句柄。用 `cmd` 作为 `CreateProcessA` 的参数。来实现反向 Shell 的傀儡操作。

3.4 Lab09-03

使用 OllyDbg 和 IDA Pro 分析恶意代码文件 `Lab09-03.exe`。这个恶意代码加载 3 个自带的 DLL (`DLL1.dll`、`DLL2.dll`、`DLL3.dll`), 它们在编译时请求相同的内存加载位置。因此, 在 OllyDbg 中对照 IDA Pro 浏览这些 DLL 可以发现, 相同代码可能会出现在不同的内存位置。这个实验的目的是在使用 OllyDbg 看代码时可以轻松地在 IDA Pro 里找到它对应的位置。

3.4.1 静态与动态分析

1. 基本静态分析

首先使用 PEiD 查看其加壳情况和导入表, 依然没有加壳 (懒得展示了), 但是导入表出现了:

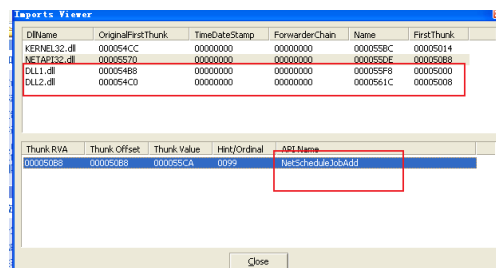


图 3.37: Lab09-03 导入表

图3.37看到:

- NetScheduleJobAdd: 推测其可能有网络相关行为;
- DLL1.dll 和 Dll2.dll: 我说这俩哪来的呢... 应该是病毒对这两个文件进行了导入。
- user32.dll

接下来使用 PEView 查看其字符串:

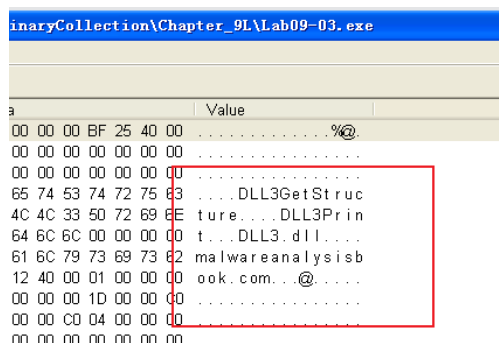


图 3.38: Lab09-03 字符串

图3.25可以看到字符串:

- DLL3.dll: 这个没在导入表中出现, 却在字符串中出现了因此推测病毒可能对 DLL3 进行了动态导入。
- malwareanalysisbook.com: 本书的彩蛋网址。

基本静态分析到此结束, 后面将重点结合是用 IDA 和 OllyDBG 分析。

2. 静态和动态分析结合

首先使用 IDA 查看其 main 函数的反编译结果:

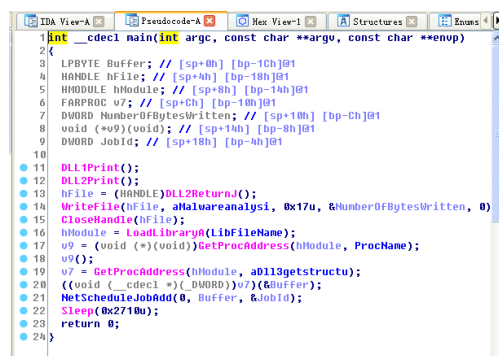


图 3.39: Lab09-03 的 main 函数

图3.39可以看到一些地方:

- 调用 DLL1Print 函数和 DLL2Print 函数, 这两个函数可能是导入到的 DLL1 和 DLL2 的打印函数, 用于输出一些信息。
- 调用 DLL2ReturnJ 函数, 获取一个句柄 hFile。而后使用 WriteFile 函数将字符串"aMalwareanalysisi"即彩蛋网址对应的 ASCII 串写入 hFile 句柄对应的文件中。然后关闭句柄。

- 使用 GetProcAddress 函数获取动态链接库中名为 aDll3getstructu 的函数的地址，即加载 DLL3.dll 的 DLL，然后打印函数 DLL3Print。并将其赋值给函数指针 v7。
- 调用 v7 函数指针所指向的函数，将 Buffer 作为参数传入。使用 NetScheduleJobAdd 函数将 Buffer 传递给调度任务，并将任务 ID 保存在 JobId 中。由此实现利用 DLL3 和缓冲区控制网络。
- NetScheduleJobAdd：添加网络计划任务的代理调用，直接调用了另一个同名的 API。其实现的是将缓冲区内容作为某种任务发送给远程主机。
- 使用 Sleep 函数暂停程序执行，延迟 10000 毫秒即 10 秒。(0x2710u)。

由此可知，IDA 告诉我们该病毒调用已经导入的 DLL1 和 DLL2 打印一些信息，而后写入字符串到某文件中。之后加载 DLL3，获取某缓冲区中的内容来执行网络相关的任务。最后会休眠 10 秒。

由此我们会想知道究竟打印了哪些信息，还干了哪些内容，因此有必要去查看 DLL1 和 DLL2 的打印函数以及 DLL2Return。J



图 3.40: 三个函数

首先查看 DLL1 的 DLL1Print，为此使用 IDA 加载 DLL1，然后直接搜索 DLL1Print 的 text，发现：

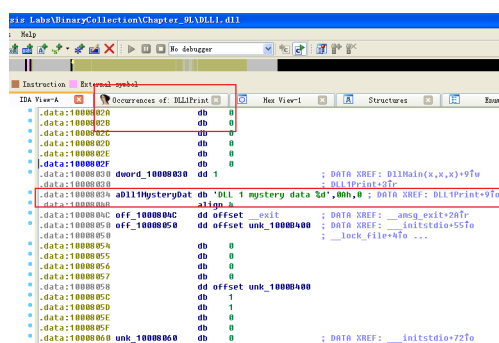


图 3.41: DLL1Print

图3.41可以看到其打印了内容 DLL 1 mystery data %d \n。其中占位符%d 的位置是 dword_1008030，因此再去搜索照这个东西：

```

.text:10001000 lpvReserved = dword ptr 10h
.text:10001000
.text:10001000
.text:10001001
.text:10001003
.text:10001009
    call ds:GetCurrentProcessId
    mov dword_1000B030, eax
.text:1000100E
    mov al, 4
.text:10001010
    pop ebp
.text:10001011
    retn 0Ch
.text:10001011 _DllMain@12
    endp

```

图 3.42: dword_1000B030

图3.42可以看到在 DLLMain 中，其通过 GetCurrentProcessID 将当运行的进程即 DLL1 的 PID 的值赋给了 dword_1000B030，然后由 DLL1Print 打印。故第一个秘密数据破解成功！就是 DLL1 的 PID。然后重新加载 DLL2，这次直接反编译其代码，首先是 main 函数：

```

.text:10001000 ; 800L_stdcall @DllMain(HINSTANCE hInstDLL, DWORD fdwReason, LPVOID lpvReserved)
.text:10001000 _DllMain@12 proc near ; CODE XREF: DllEntryPoint+40Jp
.text:10001000
.text:10001000 hInstDLL = dword ptr 8
.text:10001000 fdwReason = dword ptr 0Ch
.text:10001000 lpvReserved = dword ptr 10h
.text:10001000
.text:10001000
    push ebp
    mov ebp, esp
    push 0 ; hTemplateFile
    push 0 ; dwFlagsAndAttributes
    push 2 ; dwCreationDisposition
    push 0 ; lpSecurityAttributes
    push 0 ; dwShareMode
    push 0 ; dwDesiredAccess
    push offset FileName ; "temp.txt"
    call ds:CreateFileA
    mov dword_1000B078, eax
    mov al, 1
    pop ebp
    retn 0Ch
.text:10001028 _DllMain@12
    endp

```

图 3.43: DLL2 的 main

图3.43可以看到 DLLMain 使用 CreateFileA 创建了文件”temp.txt” 的句柄，这个句柄的值被赋给了 dword_1000B078。然后查看那两个 DLL2Print 和 DLL2ReturnJ：

```

int DLL2Print()
{
    return sub_1000105A(aDll2MysteryDat, dword_1000B078);
}

```

DLL2Print

```

int DLL2ReturnJ()
{
    return dword_1000B078;
}

```

DLL2ReturnJ

图 3.44: DLL2 的两个函数

图3.44看到 DLL2Print 照样打印 DLL 2 mystery data，然后这个秘密数值是那个文件句柄的值；同样的 DLL2ReturnJ 是直接获取这个句柄。

其作用就是在 Lab09-03 的 main 函数，获取 temp.txt 的文件句柄，然后将 aMalwareanalysis 即彩蛋网址对应的 ASCII 串写入句柄对应的文件中。

然后如果继续使用静态分析，无法验证其对 DLL3 的导入，因此转而使用 OllyDBG 来进行动态分析，直接来到加载 DLL3 的位置：

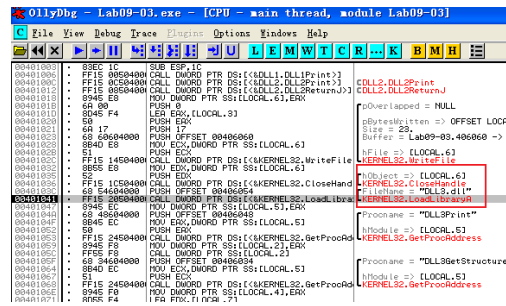


图 3.45: 加载 DLL3

图3.45可以看到病毒通过将”DLL3.dll”的 ASCII 串文件名作为参数来调用 LoadLibraryA 来动态导入，这验证了我们静态分析的猜测。

然后查看 Memory Map 发现：

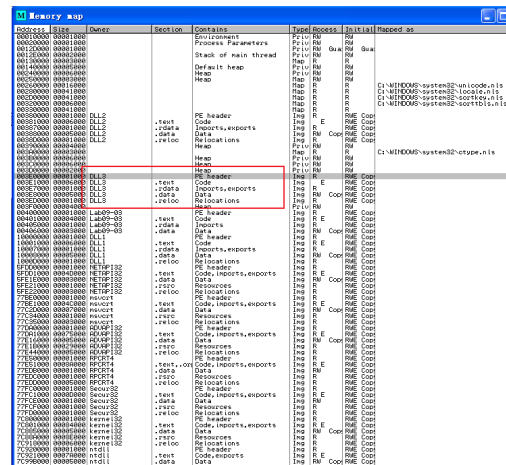


图 3.46: DLL3 已经被加载

图3.46可以看到 DLL3 已经被加载到了内存的 0x003E0000 的位置。另外 DLL1 在 0x10000000，DLL2 在 0x00380000。接下来回到 IDA 中加载 DLL3 并分析其功能，首先是 main 函数：

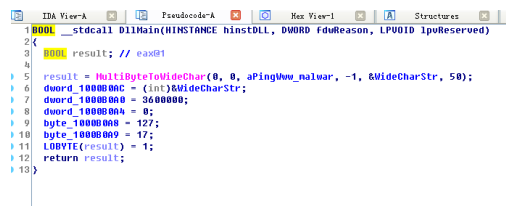


图 3.47: DLL3 的 main

图3.47发现其用 MultiByteToWideChar 函数将一个字符串从多字节编码转换为 Unicode 编码。而 aPingWww__malwar 这个还未知，不急。然后将其内存位置存放在 WideCharStr 中。

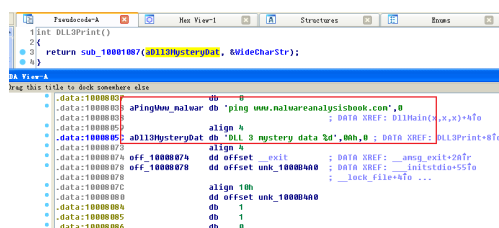


图 3.48: DLL3Print

图3.48看到 Print 又是打印了 DLL 3 mystery data, 然后这个秘密数值是那个 WideCharStr 即 aPingWww_malwar 的内存位置。

而 aPingWww_malwar 对应的就是 ping 操作：ping 彩蛋书网址。

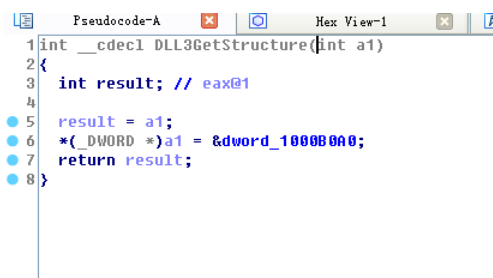


图 3.49: DLL3GetStructure

图3.49看到其返回一个结构体，其中包含 dword_1000B0A0 的值。

最后回到 IDA 加载 Lab09-03，查看最后一部分的工作即 `NetScheduleJobAdd`。正好在调用 `DLL3GetStruture` 后。

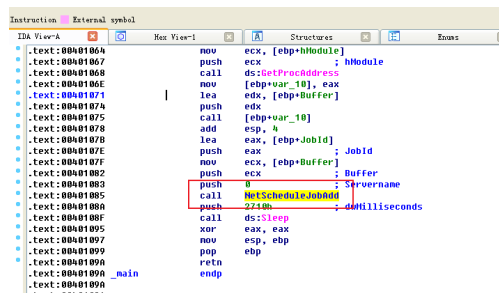


图 3.50: NetSchedule.JobAdd

调用的结果是由 Buffer 指向的一个结构体。这个结构体被传递给了 NetScheduleJobAdd 函数。通过查阅资料，Buffer 是一个指向 AT_INFO 结构体的指针。然后查看这个结构体：

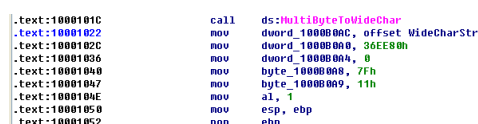


图 3.51: AT INFO 结构体

通过这个结构体，我们可以看到计划任务被设置为每周的任意一天，在凌晨 1 点执行 ping 命令来访问 malwareanalysisbook.com 网站。

总结来说，调用的结果是一个 `AT_INFO` 结构体，用于设置计划任务的详细信息，包括执行时间和执行的命令。

3.4.2 实验问题

1. Q1: Lab09-03.exe 导入了哪些 DLL?

回答：静态分析中发现有 `kernel32.dll`、`NetAP32.dll`、`DLL1.dll` 和 `DLL2.dll`。另外还有动态加载的 `user32.dll` 和 `DLL3.dll`。

2. Q2: DLL1.dll、DLL2.dll、DLL3.dll 要求的基地址是多少?

回答：补充使用 PEiD 查看一下 `ImageBase`：

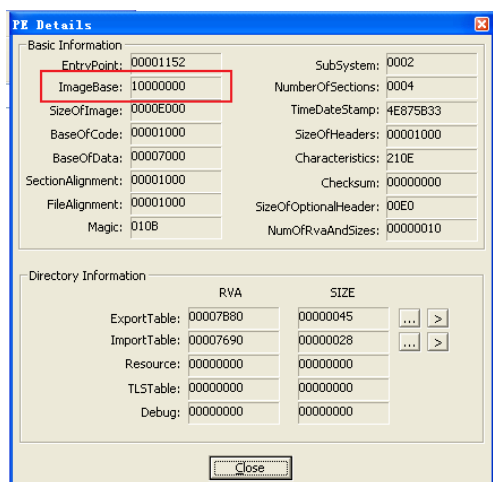


图 3.52: DLL1 的基地址

图3.52看到 DLL1 要求的基地址为 `0x10000000`，同样查看另外两个后发现都是 `0x10000000`。

3. Q3: 当使用 OllyDbg 调试 Lab09-03.exe 时，为 DLL1.dll，DLL2.dll，DLL3.dll 分配的基地址是什么?

回答：DLL3 被加载到内存的 `0x003E0000` 的位置。另外 DLL1 在 `0x10000000`，DLL2 在 `0x00380000`。即它们最终被分配的基地址。这点和书中答案不同，也是体现了不同机器的差异。

4. Q4: 当 Lab09-03.exe 调用 DLL1.dll 中的一个导入函数时，这个导入函数都做了些什么?

回答：格式化字符串打印出 “DLL1mystery data”，然后占位符是当前进程的 PID。

5. Q5: 当 Lab09-03.exe 调用 `writeFile` 函数时，它写入的文件名是什么?

回答：是 `DLL2ReturnJ` 返回的文件名 `temp.txt`。

6. Q6: 当 Lab09-03.exe 使用 `NetScheduleJobAdd` 创建一个 job 时，从哪里获取第二个参数的数据?

回答：从 `DLL3GetStructure` 中获取 `NetScheduleJobAdd` 调用的缓冲区，动态地获取数据，然后作为第二个参数。Lab09-03.exe

7. Q7: 在运行或调试 Lab09-03.exe 时会看到 Lab09-03.exe 打印出三块神秘数据。DLL1 的神秘数据、DLL2 的神秘数据，DLL3 的神秘数据分别是什么？

回答：当前进程的 PID，emp.txt 的句柄值和 ping www.malwareanalysisbook.com 在内存中的位置。

8. Q8: 如何将 DLL2.dll 加载到 IDAPro 中，使得它与 OllyDbg 使用的加载地址匹配？

回答：使用 IDA Pro 加载 DLL 时选择手动加载，然后输入新的 ImageBase 即 0x0038000。

3.5 Yara 检测

3.5.1 Sample 提取

利用课程中老师提供的 Scan.py 程序，将电脑中所有的 PE 格式文件全部扫描，提取后打开 sample 文件夹查看相关信息：

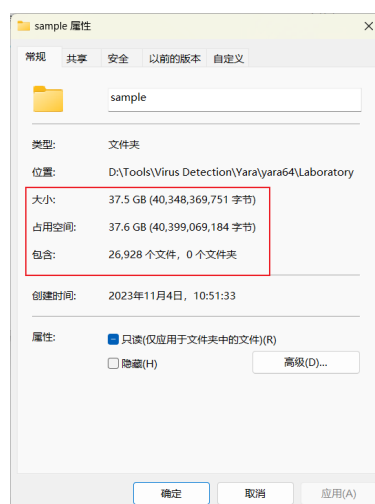


图 3.53: Sample 信息

图3.53可以看到从电脑中提取了所有 PE 格式文件后的文件及 sample 大小为 37.6GB，包含一共 26928 个文件。Yara 编写的规则将目标从 sample 中识别成功检测出本次 La9 的全部三个恶意代码。

这次比上次大了，是因为为了做计网第二次实验，装了个 WireShark。

3.5.2 Yara 规则编写

本次 Yara 规则的编写基于上述的病毒分析和实验问题，主要是基于静态分析的 Strings 字符串和 IDA 分析结果以及 OllyDBG 的分析。为了能够更好地进行 Yara 规则的编写，首先对之前分析内容进行回归。分别对 Lab09-01, 02, 03.exe 可以利用的病毒特征进行分条总结如下：

1. Lab09-01.exe:

- **SOFTWARE\\Microsoft \\XPS:** 静态分析查看字符串发现的，就是一个自己新安装的注册表键，注意 Microsoft 后有空格。
- **UPLOAD:** 命令之一。

- **-in**: 安装的命令行选项之一。

2. Lab09-02.exe:

- cmd: 命令行操作;
- Sleep: 休眠操作;
- WSASocket: 初始化网络环境;
- WS2_32.dll: 网络编程库。

3. Lab09-03.exe:

- DLL1Print: DLL1 的打印函数。
- DLL2Print: DLL2 的打印函数。
- **DLL2ReturnJ**: DLL2 返回句柄的函数
- **malwareanalysisbook.com**: 彩蛋网址。

因此加上必要的一些修饰符如 wide、ascii 以及 nocase 后, 编写如下 Yara 规则:

```
1 rule Lab09_01
2 {
3   meta:
4     description = "Lab09_01:Yara Rules"
5     date = "2023/11/4"
6     author = "ErwinZhou"
7   strings:
8     $clue1 = "SOFTWARE\\Microsoft \\XPS" wide ascii
9     $clue2 = "UPLOAD" wide ascii
10    $clue3 = "-in" wide ascii
11
12   condition:
13     all of them //Lab09-01.exe
14 }
15
16
17 rule Lab09_02
18 {
19   meta:
20     description = "Lab09_02:Yara Rules"
21     date = "2023/11/4"
22     author = "ErwinZhou"
23
24   strings:
25     $clue1 = "cmd" wide ascii nocase
26     $clue2 = "Sleep" wide ascii
```



```
27 $clue3 = "WSASocket" wide ascii
28 $clue4 = "WS2_32.dll" wide ascii nocase
29
30 condition:
31     all of them //Lab09-02.exe
32 }
33
34
35 rule Lab09_03
36 {
37 meta:
38     description = "Lab09_03:Yara Rules"
39     date = "2023/11/4"
40     author = "ErwinZhou"
41
42 strings:
43     $clue1 = "DLL1Print" wide ascii
44     $clue2 = "DLL2Print" wide ascii
45     $clue3 = "DLL2ReturnJ" wide ascii
46     $clue4 = "malwareanalysisbook.com" wide ascii nocase
47
48 condition:
49     all of them //Lab09-03.exe
50 }
```

然后使用如下 Python 代码进行对 sample 的扫描：

```
1 import os
2 import yara
3 import time
4
5 # 加载YARA规则
6 rules = yara.compile('D:\Tools\Virus Detection\Yara\yara64\Lab9.yar')
7
8 # 初始化计数器
9 total_files_scanned = 0
10 total_files_matched = 0
11
12 def scan_folder(folder_path):
13     global total_files_scanned
14     global total_files_matched
15
16     # 检查文件夹是否存在
```

```
17 if os.path.exists(folder_path) and os.path.isdir(folder_path):
18     # 遍历文件夹内的文件和子文件夹
19     for root, dirs, files in os.walk(folder_path):
20         for filename in files:
21             total_files_scanned += 1
22             file_path = os.path.join(root, filename)
23             with open(file_path, 'rb') as file:
24                 data = file.read()
25                 # 扫描数据
26                 matches = rules.match(data=data)
27                 # 处理匹配结果
28                 if matches:
29                     total_files_matched += 1
30                     print(f"File '{filename}' in path '{root}' matched YARA
31                           rule(s):")
32                     for match in matches:
33                         print(f"Rule: {match.rule}")
34             else:
35                 print(f'The folder at {folder_path} does not exist or is not a folder.')
36 # 文件夹路径
37 folder_path = 'D:\Tools\Virus Detection\Yara\yara64\Laboratory\sample'
38 # 记录开始时间
39 start_time = time.time()
40 # 递归地扫描文件夹
41 scan_folder(folder_path)
42 # 记录结束时间
43 end_time = time.time()
44 # 计算运行时间
45 runtime = end_time - start_time
46
47 print(f"Program runtime: {runtime} seconds.")
48 print(f"Total files scanned: {total_files_scanned}")
49 print(f"Total files matched: {total_files_matched}")
```

3.5.3 Yara 规则执行效率测试

扫描结果如下图所示：

```
File 'Lab03-04.exe' in path 'D:\Tools\Virus Detection\Yara\yara64\Laboratory\sample' m
atched YARA rule(s):
Rule: Lab09_01
File 'Lab09-01.exe' in path 'D:\Tools\Virus Detection\Yara\yara64\Laboratory\sample' m
atched YARA rule(s):
Rule: Lab09_01
File 'Lab09-02.exe' in path 'D:\Tools\Virus Detection\Yara\yara64\Laboratory\sample' m
atched YARA rule(s):
Rule: Lab09_02
File 'Lab09-03.exe' in path 'D:\Tools\Virus Detection\Yara\yara64\Laboratory\sample' m
atched YARA rule(s):
Rule: Lab09_03
File 'Lab16-01.exe' in path 'D:\Tools\Virus Detection\Yara\yara64\Laboratory\sample' m
atched YARA rule(s):
Rule: Lab09_01
File 'Lab16-03.exe' in path 'D:\Tools\Virus Detection\Yara\yara64\Laboratory\sample' m
atched YARA rule(s):
Rule: Lab09_02
File 'Lab21-01.exe' in path 'D:\Tools\Virus Detection\Yara\yara64\Laboratory\sample' m
atched YARA rule(s):
Rule: Lab09_02
```

图 3.54: Yara 检测结果

图3.54可以看到能够成功地从 26928 个文件中扫描和检测出 Lab9 的三个病毒。先不提 Lab09-02 的规则，另外的 Lab09-01 和 03 的规则也有别的 Lab 病毒被检测出了出现了，推测像是 Lab03-04 这种不过是用重复的样本病毒，换了个名字。

```
File '全球学术快报.exe' in path 'D:\Tools\Virus Detection\Yara\yara64\Laboratory\sa
mple' matched YARA rule(s):
Rule: Lab09_02
Program runtime: 131.7245535850525 seconds.
Total files scanned: 26928
Total files matched: 273
```

图 3.55: Yara 检测时间

但是也可以注意到有一些其它文件也被识别了出来，有些甚至不是 Lab 中的，可能是当初我们只进行了静态分析也没有使用 IDA 工具没有发现他们实际上具有这么多特殊的功能，同样也是因为 Lab09-02.exe 并没有什么特殊的字符串，因此即使是正常的文件也容易被匹配。但所有的检测结果都聚集在病毒样本 Lab 中，并且仅用时 131.72 秒，时间性能较快。总的来说，Yara 规则编写和检测较为成功。

3.6 IDAPython 辅助样本分析

根据要求，分别选择病毒分析过程中一些系统的过程编写如下的 Python 脚本：

3.6.1 自动化反汇编

目的是打印当前选中函数的所有汇编指令。它首先确定函数的起始和结束地址，然后逐个遍历这些地址，打印每个地址及其对应的汇编指令。

```
1 # 获取当前选中的地址
2 ea = here()
3 # 使用 IDA API 获取该地址所在的函数信息
4 func = idaapi.get_func(ea)
5 # 打印函数的起始和结束地址
6 print "Start: 0x%x, End 0x%x" % (func.startEA, func.endEA)
```

3.6.2 寻找 socket 函数

目的是在给定的二进制中找出调用网络相关函数（如 send,recv,bind,connect, accept 等）的汇编指令。这是因为本次实验中有许多 Windows socket 编程的相关内容，为了是网络初始化和连接远程主机。本次实验中是 Lab09-02 的 connect 函数调用。

```
1 # 遍历IDA中已知的所有函数
2 for func in idautils.Functions():
3     # 获取函数的属性标志
4     flags = idc.get_func_attr(func, FUNCATTR_FLAGS)
5     # 如果函数是库函数或者是跳板函数，则跳过
6     if flags & FUNC_LIB or flags & FUNC_THUNK:
7         continue
8     # 获取函数中所有指令的地址
9     dism_addr = list(idautils.FuncItems(func))
10    # 遍历函数中的每个指令地址
11    for line in dism_addr:
12        # 获取当前指令的助记符（如MOV, ADD等）
13        m = idc.print_insn_mnem(line)
14        # 检查助记符是否是我们关心的网络函数
15        if m == "connect":
16            # 获取指令的第一个操作数的类型
17            op = idc.get_operand_type(line, 0)
18            # 如果操作数是一个寄存器，打印指令的地址和汇编行
19            if op == o_reg:
20                print("0x%x %s" % (line, idc.generate_disasm_line(line, 0)))
```

3.6.3 寻找特定字符串

目的是搜索在当前函数内包含特定文本（在这里是"olc.exe" 和密码"1qaz2wsx3edc"）的地址，并打印这些地址。这有助于我们在一些病毒比如本次的 Lab09-02.exe 中快速定位所需的内容。

```
1 # 初始化当前地址为函数的起始地址（这部分似乎没有在提供的代码中明确给出）
2 cur_addr = start_addr
3 # 保证当前地址小于函数的结束地址
4 while cur_addr < end_addr:
5     # 在当前地址下方查找包含文本“olc.exe”的地址
6     cur_addr = idc.find_text(cur_addr, SEARCH_DOWN, 0, 0, "olc.exe") # FindText
7     # 注意：第五个参数（文本）需要直接写成字符串形式。使用字符串变量可能会导致错误。
8     # 如果没有找到文本，则cur_addr会被设置为BADADDR
9     if cur_addr == BADADDR:
10         continue
11     else:
```

```
12     # 打印包含“olc.exe”文本的地址
13     print(cur_addr)
14     # 移动到下一个指令的地址以继续搜索
15     cur_addr = idc.next_head(cur_addr)
```

3.6.4 定位函数

来遍历当前段内的所有函数，并打印它们的名称。目的是快速定位特定函数。

```
1 # 获取当前数据库的开始地址
2 ea = BeginEA()
3 # 遍历从当前段的开始地址到结束地址的所有函数
4 for funcea in Functions(SegStart(ea), SegEnd(ea)):
5     # 获取函数的名称
6     functionName = GetFunctionName(funcea)
7     # 打印函数名称
8     print(functionName)
```

使用上述 Python 脚本在 IDAPro 中便可以辅助进行分析。经过测试全部与使用 IDA 动态分析的结果相同，编写较为成功。

4 实验结论及心得体会

4.1 实验结论

本次实验，通过结合使用静态分析工具和动态分析方法，对 Lab9 的三个恶意代码进行了全面的分析，并依次回答了书中的问题。并在其中第一次使用了 OllyDBG。然后结合之前的分析和 IDA 的 String 模块编写了 Yara 规则，对病毒样本进行了成功检测并且时间性能较强。

最后为本次实验中编写了对应可以辅助分析的 IDAPython 脚本，并且成功实现了辅助功能。总的来说，实验非常成功。

4.2 心得体会

本次实验，我收获颇丰，这不仅是课堂中的知识，更是我解决许多问题的能力，具体来说：

1. 首先我进一步熟练掌握到了将课堂中学习的病毒分析工具 IDAPro，用于更全面的静态分析；
2. 然后我还第一次使用了 OllyDBG 在调试的过程中进行了深度的动态分析，这不仅加强了我使用了 OllyDBG 的能力，更让我具备了深度动态分析的能力；
3. 其次我还多次交叉使用 IDA 和 OllyDBG，学习到了使用不同工具结合破解病毒的能力。
4. 除此之外我还精进了我编写更为高效的 Yara 规则的能力，更好地帮助我识别和检测病毒；
5. 我还精进了使用 IDAPython 脚本对病毒进行辅助分析。

总的来说，本次通过亲自实验让我感受到了很多包括 IDA 和 IDAPython 最重要的是 OllyDBG，也加深了我对病毒分析综合使用静态和动态分析的能力，尤其是动态的深度分析调试的能力。培养了我对病毒分析安全领域的兴趣。我会努力学习更多的知识，辅助我进行更好的病毒分析。

感谢助教学姐审阅:)