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

实验四：IDAPython

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. 对教材第五章课上学习的 IDAPro 和 IDAPython 的知识进行回顾，在实验中加深对二者的理解和使用，精进使用 IDA 和 IDAPython 分析病毒的能力，熟悉 IDAPython 相关自带的 API 函数和功能模块等。
2. 完成教材 Lab5 的实验内容，编写能够检测 Lab5 病毒的 Python 分析脚本完成实验。
3. 在样本分析结果的基础上，编写样本的 Yara 检测规则。
4. 尝试编写 IDA Python 脚本来辅助样本分析。

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。

3 实验过程

3.1 实验环境及工具

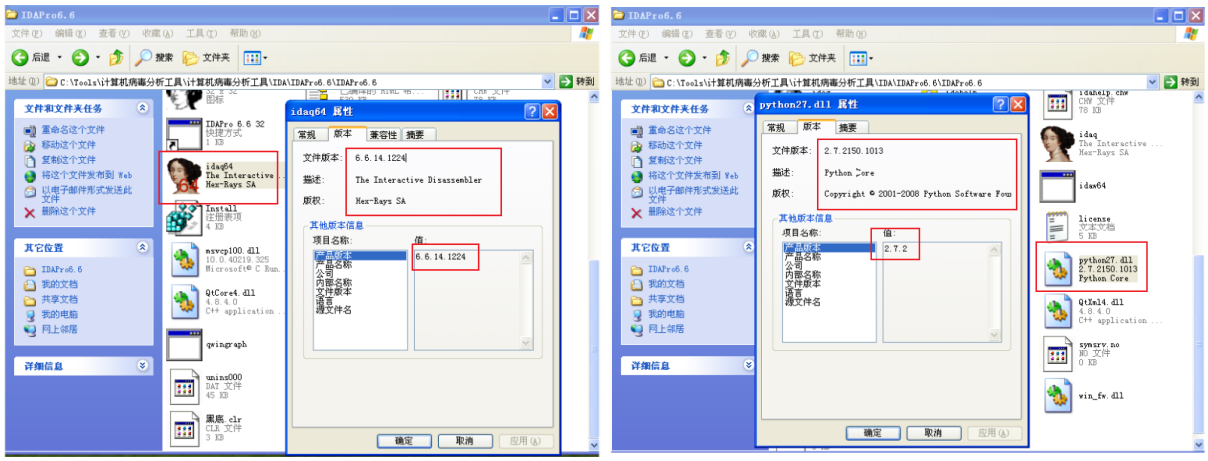
虚拟机软件	VMware Workstation 17 Pro
宿主机	Windows 11 家庭中文版
虚拟机操作系统	Windows XP Professional
实验工具	IDAPro 6.6.14.1224
配套工具	Python 2.7.2

表 1: 本次实验环境及工具

3.2 实验准备

3.2.1 IDA 及 Python 准备

本次实验直接使用课堂中实验老师提供的病毒分析工具集，其中包含了 IDA 以及 IDAPython 工具：



IDAPro 版本　　IDAPython 版本

图 3.1: 对应版本

图3.1展示了 IDAPro 工具版本为 **IDAPro 6.6.14.1224**，IDAPython 工具版本为 **Python 2.7.2**。这里由于已经自带了相关工具，就不再进行重新下载安装配置，但具体而言，应该通过如下方式配置 IDAPython 插件：

1. 到 googlecode 上面下载相应版本的 IDAPython。然后解压。注意 IDA 版本和 Python 版本都要和自己机器上安装的版本相对应。

2. 将 IDAPython 解压后的 Python 文件夹内的所有内容覆盖掉 IDA 原有 Python 文件夹（IDA 安装目录下）下面的内容。
3. 将 IDAPython 解压后的 Plugins 文件夹的所有内容拷贝到 IDA 原有 Plugins 文件夹（自定义，一般 IDA 安装目录下）下。
4. 将 IDAPython 解压后的 python.cfg 文件拷贝到 IDA 原有 cfg 文件夹（IDA 安装目录下）下。
5. 重启 IDA，就可以了。效果是：File 菜单下面会有 Python Command 选项，而且 Script files 选项中可以选择 py 文件。

这里直接以 Lab05-01.dll 为例先对 IDA 进行简单测试，使用 IDA 打开 Lab05-01.dll：

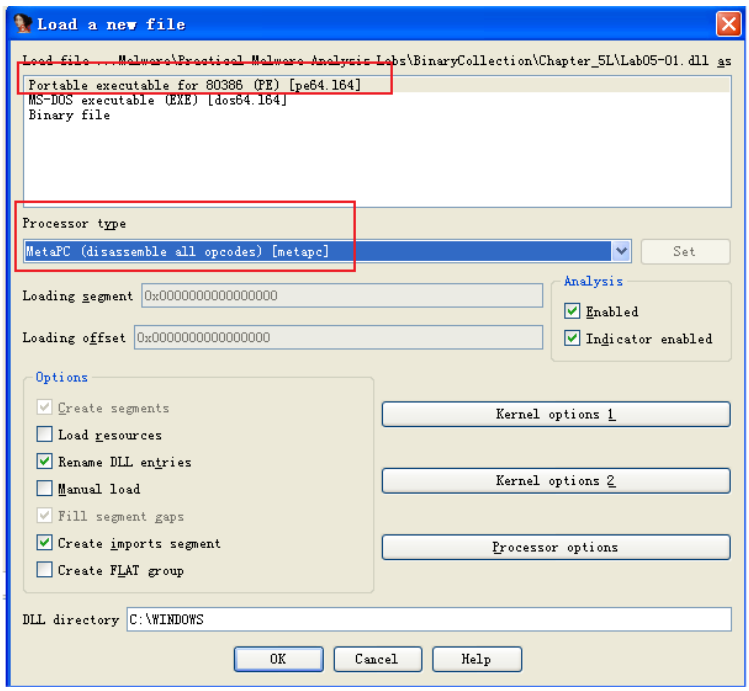


图 3.2: 加载选项

图3.2可以看到加载选项有如下两条值得注意的：

1. 加载文件类型为 Portable executable for 80386(PE) [pe64.l64]：
   - Portable executable (PE)：这是 Windows 操作系统中使用的可执行文件格式。它包括.EXE、.DLL、.SYS 等文件。**Lab05-01.dll 属于其中之一**。
   - for 80386：80386 是 Intel 的一个 32 位微处理器。这里的意思是该 PE 文件是为 32 位系统设计的。**Lab05-01.dll 确实是为 32 位**。
   - pe64.l64：这通常表示文件是 64 位的 PE 格式。这与前面的 “for 80386” 似乎有些矛盾，因为 80386 是 32 位的。**但是后面我加载后可以正确进行处理，证明这里也没有问题。因此推测这可能是 IDA 的一个描述错误或者是某种特定的命名约定导致了命名上的误导，有待进一步探究**
2. 处理器类型为 MetaPC(disassemble all opcodes)[metaPC]：

- MetaPC：这是 IDA 的一个特殊处理器模块，它可以处理多种指令集，包括 x86、x64 等。选择这个模块意味着 IDA 会尝试反汇编所有可能的操作码，而不仅仅是特定于某个处理器的操作码。
- disassemble all opcodes：这意味着 IDA 会尝试反汇编文件中的所有操作码，而不是只反汇编那些它认为是有效的操作码。

对于处理器类型，选择 MetaPC 通常是一个安全的选择，因为它可以处理多种指令集。但是如果知道 DLL 文件是为特定的处理器（如 x86 或 x64）编写的，那么选择该处理器可能会得到更准确的反汇编结果。

这里直接使用上述两个默认选项继续进行测试：

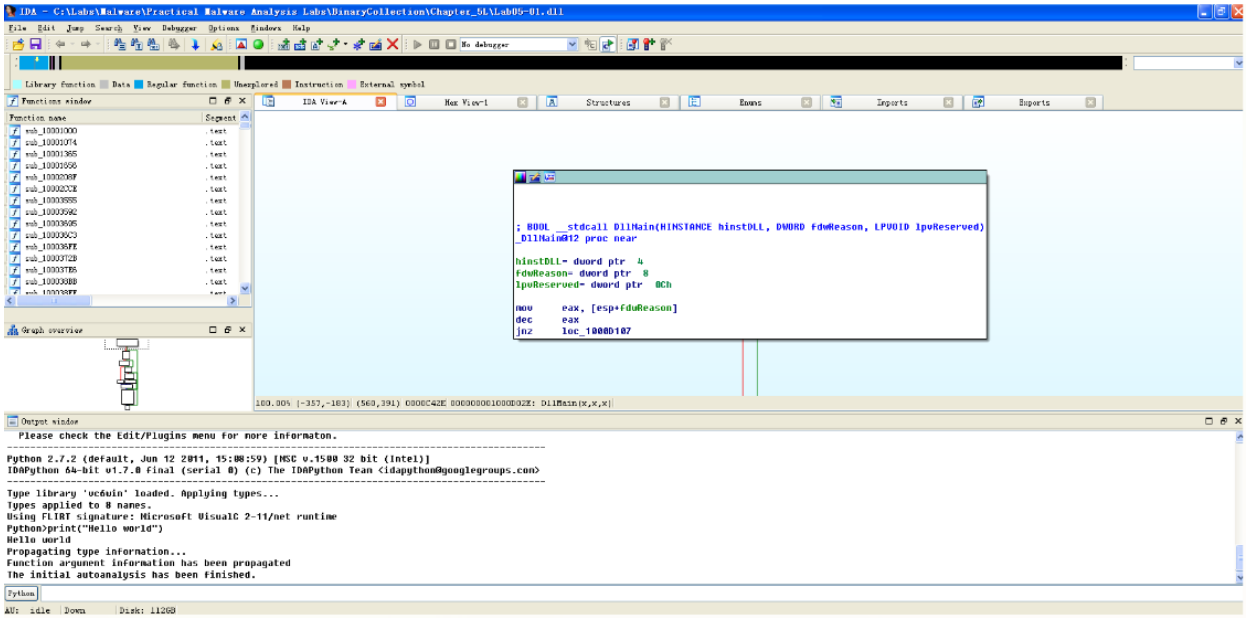


图 3.3: IDAPro 页面功能展示

图3.3可以看到加载了文件后正常显示了图形界面的反汇编，函数，结构体等，这些都与加载的文件密切相关。接下来通过简单的 HelloWorld 命令测试 IDAPython：

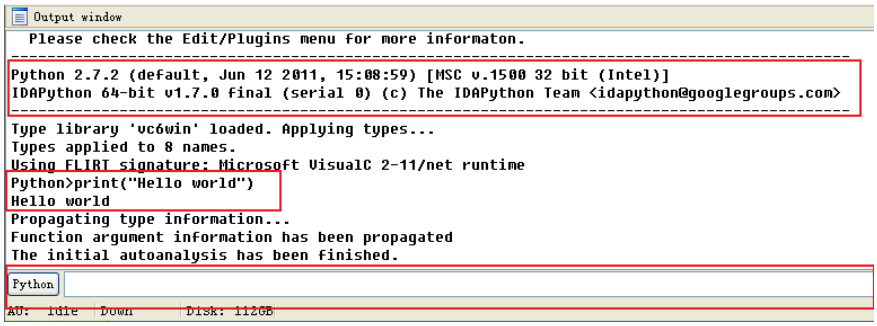


图 3.4: IDAPython 测试

图3.4可以看到一个在输出界面最底下的命令输入框，由 Python 命令的符号，还有对应的版本信息。也可以看到测试的过程和结果：

```
Python>print("Hello world")
Hello world
```

证明 IDA 和 IDAPython 均可以正常使用。所有准备工作完成，现在正式开始实验。

3.3 Lab05-01

由于已经正确加载 Lab05-01.dll 进入 IDAPro 中，并且检测 IDAPython 也运行正常，因此正式开始实验，并且依次回答实验当中的问题：

1. **Q1:DLLMain 的地址是什么？**

   回答：

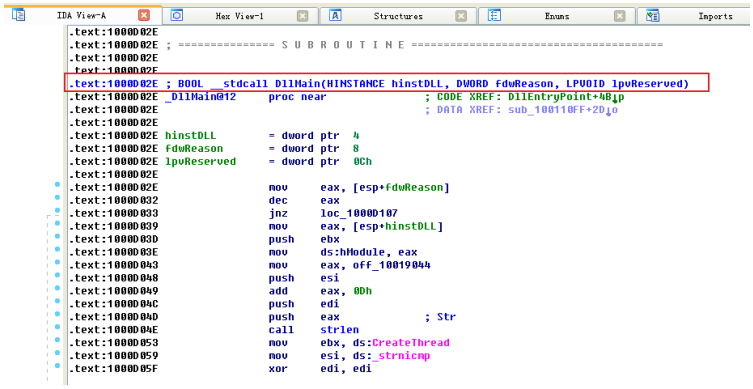


图 3.5: DllMain 函数位置

图3.5可以看到在 IDAPro 加载了恶意 DLL 后，直接就自动定位到了反汇编窗口的 DllMain 函数位置。根据旁边的地址信息我们可以得出 **DllMain 在.text 节的 0x1000D02E 处**。

DllMain 通常是开始分析的地方，因为 Dll 类型的恶意病毒往往也是经过了编译器的。而编译器的相关信息一般都会出现在 DllEntryPoint 到 DllMain 之间，这部分代码对于分析恶意代码的目的没有任何帮助和指示性作用。

2. **Q2: 使用 Imports 窗口并浏览到 gethostbyname，导入函数定位到什么地址？**

   回答：

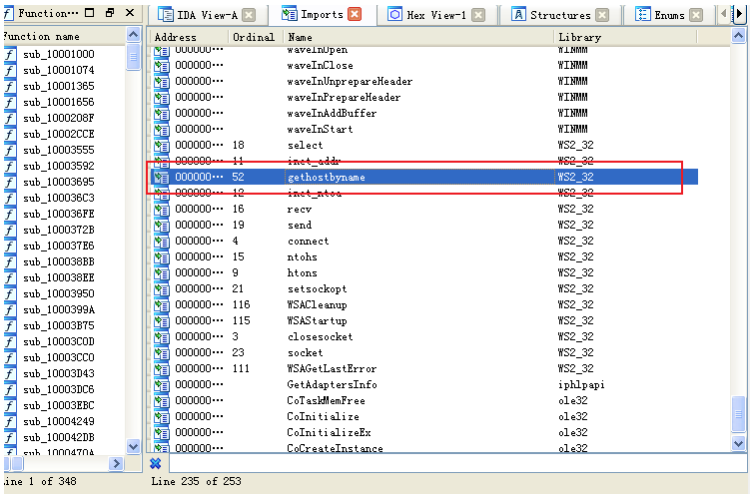


图 3.6: 定位导入表中的 gethostbyname

图3.6展示了第一步。首先通过菜单找到导入函数窗口 Imports，定位到该函数的所有导入函数，**在这之中首先定位 gethostbyname 函数。然后双击运行后查看其反汇编：**

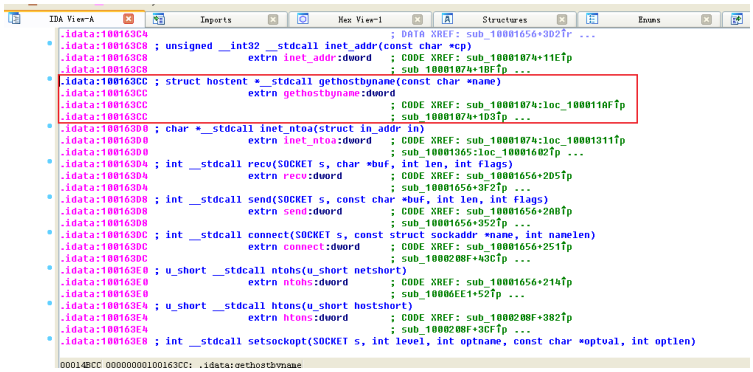


图 3.7: gethostbyname 反汇编

图3.7展示了双击进入查看的 gethostbyname 的反汇编地址，**在.idata 节的 0x100163CC 处。**

gethostbyname 函数一般用于将主机名解析为 IP 地址。它接受一个主机名作为参数，并返回一个指向 hostent 结构的指针，其中包含了与该主机名关联的 IP 地址信息。这个函数在早期的网络编程中被广泛使用，但现在已被弃用。**由此推测该病毒可能具有潜在的网络恶意行为。**

3. **Q3: 有多少函数调用了 gethostbyname?**

   回答：

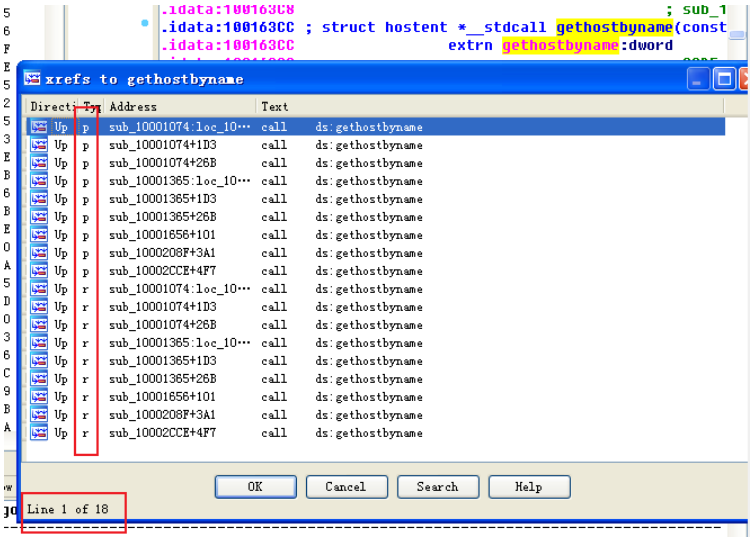


图 3.8: 查看 gethostbyname 的交叉引用情况

图3.8展示了定位到 gethostbyname 函数后使用 Cirl+X 查看器交叉应用情况，实际上就是查看其他函数对它的调用情况。

如图所示，可以以看到最底下的信息”Lines 1 of 18” 告诉我们了一些有用的信息。通过查阅资料我们发现，通常来说部分版本的 IDAPro 会计算两次交叉引用，分别是左侧的类型 p 代表的由于调用产生的引用，类型 r 代表被读取产生的引用。**由此可知，gethostbyname 一共被调用了 9 次**。

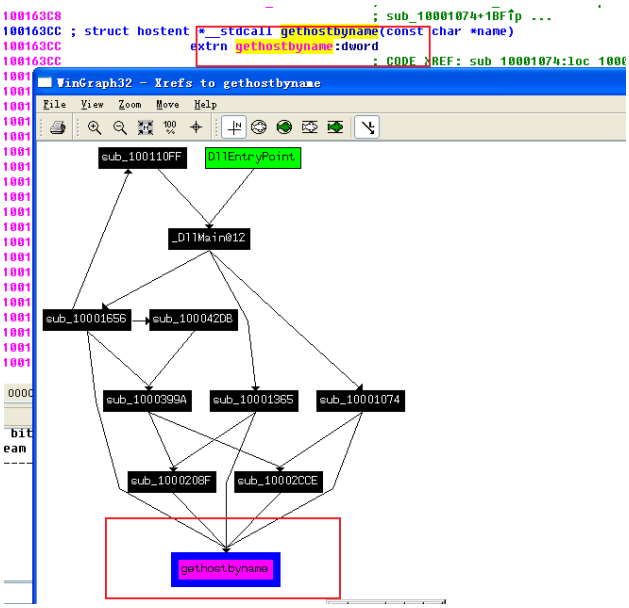


图 3.9: gethostbyname 的交叉引用图

图3.9展示了通过右键和查看交叉引用图的方式查看 gethostbyname 的交叉引用情况，**可以看清楚地看到对 gethostbyname 有五个来自其它函数的直接的引用**。也可以通过仔细核对前面的交叉引用表发现。

总的来说，在整个恶意代码中，gethostbyname 被 5 个不同的函数调用了 9 次

4. **Q4: 将精力集中在位于 0x1001757 处的对 gethostbyname 的调用，你能找出哪个 DNS 请求将被触发吗?**

   回答:

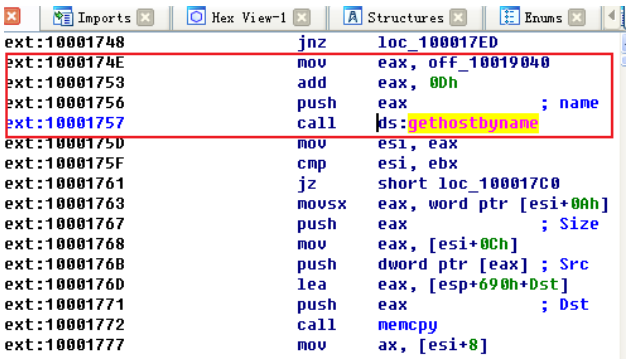


图 3.10: 定位到 0x10001757

图3.10展示了通过在3.8中的交叉引用表，双击来到 0x10001757 附近，可以看到**通过 call 命令实现了对 gethostbyname 的调用**。

仔细对这段代码进行分析，如果想要进一步得到它触发了什么样的 DNS 请求，合理地推测该 DNS 请求应该会把域名以参数的形式通过寄存器的方式传递给 gethostbyname。**而 eax 寄存器正是 x86 汇编中经常作为传递参数的容器**。也可以看到通过 push 的形式将 eax 的值压入了栈中，这就是参数入栈，即将开始函数调用的信号。

观察 eax，可以发现首先通过 mov 命令，将 off_10019040 偏移地址位置的值赋予了 eax，通过双击该地址达到对应位置:

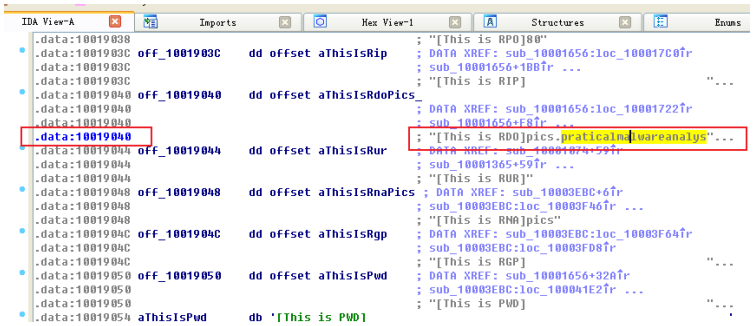


图 3.11: 查看 eax 的参数

图3.11可以看到来到该位置后发现了字符串”**[This is RDO]pics.praticalmalwareanalys**”。进一步思考，汇编代码将该字符串的起始位置交给 eax 后，又加了 0xD 字节的偏移，这正好是”[This is RDO]” 的长度，正好达到 pics.praticalmalwareanalys 的开头，是一个域名的指针。

因此可以得出结论:0x10001757 处对 gethostbyname 进行了调用，传入了 picspracticalmalwareanalysis.com 域名作为参数，帮助恶意代码发起对该网站的 DNS 请求。

5. **Q5:IDA Pro 识别了在 0x11656 处的子过程中的多少个局部变量?**

   回答: 在这里我通过左边的函数表直接找到了 sub_10001656 函数:

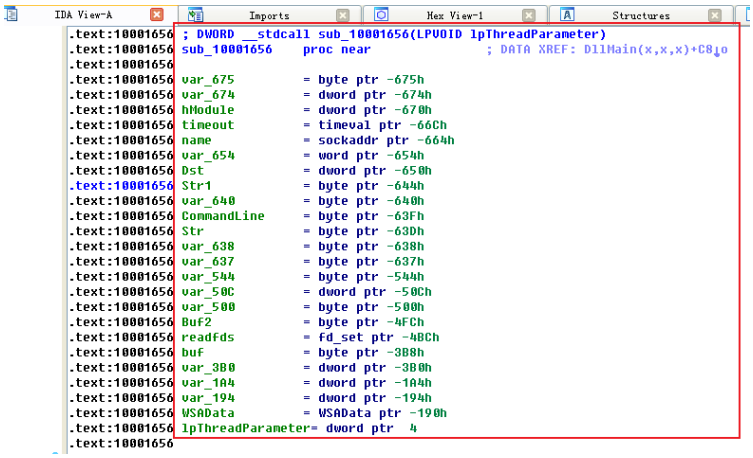


图 3.12: 查看 sub_10001656 处的函数布局

图3.12可以看到很多很多函数涉及的局部变量，这些都写在函数布局之中也就是函数的开始部分。总的来说，**可以看出这个函数似乎是一个线程函数，因为它的参数被命名为 lpThreadParameter，这是 Windows 线程函数的典型参数名**。

其中最后一行代码".text:10001656 lpThreadParameter= dword ptr 4" 定义了函数的参数 lpThreadParameter。它是一个双字大小的参数，位于基指针 ebp 的 4 偏移处。这是典型的 x86 调用约定，其中函数的参数是从右到左压入堆栈的，所以 dword ptr 4 是第一个参数。

这意味着在函数 sub_10001656 的执行过程中，可以使用 lpThreadParameter 来引用传递给该函数的参数。**除了该行和前面对函数原型的描述之外其他行每行都是局部变量，可以看到一共有 23 个局部变量。这些都被 IDAPro 识别了**

其中可以看到许多参数，以两个为例：

- var_675 = byte ptr -675h: 这定义了一个名为 var_675 的局部变量，它是一个字节大小的变量，位于基指针 ebp 的-675h 偏移处。
- hModule = dword ptr -670h: 这定义了一个名为 hModule 的局部变量，它是一个双字大小的变量，位于基指针 ebp 的-670h 偏移处。

6. **Q6:IDA Pro 识别了在 0x101656 处的子过程中的多少个参数?**

   回答：正如前面分析可知，我们通过发现最后一行代码找到了该函数其中的 lpThreadParameter。**因此，IDAPro 一共只识别了子过程中的一个参数，并命名为 lpThreadParameter**。

7. **Q7: 使用 Strings 窗口，来在反汇编中定位字符串\cmd.exe /c。它位于哪?**

   回答：首先使用 Strings 窗口，查看该 DLL 的所有字符串：

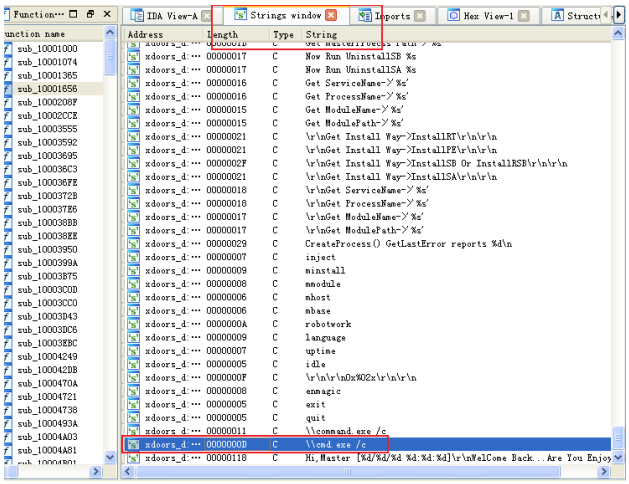


图 3.13: 查找 cmd.exe /c 字符串

图3.13显示了定位到 cmd.exe /c 字符串。然后双击查看它对应位置的代码信息:

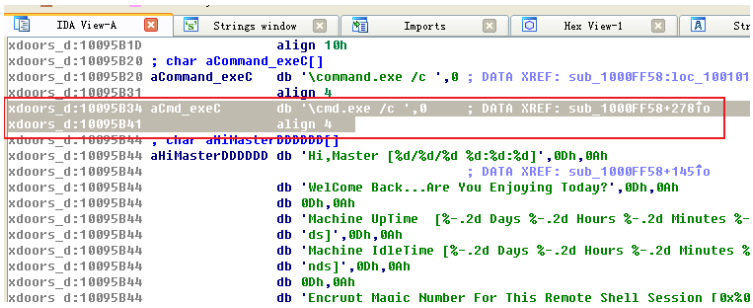


图 3.14: 查看 cmd.exe /c 的代码位置信息

图3.14可以看到”xdoors_d:10095B34 aCmd_exeC:”。这是一个标签,表示当前数据的地址是 10095B34,并且它位于 xdoors_d 段。因此可以得出 **cmd.exe /c 出现在 0x10095B34 处,位于 xdoors_d 段**。

8. **Q8: 在引用\cmd.exe /c 的代码所在的区域发生了什么?**

   回答: 接下来继续使用交叉引用的方式查看对该字符串的调用:

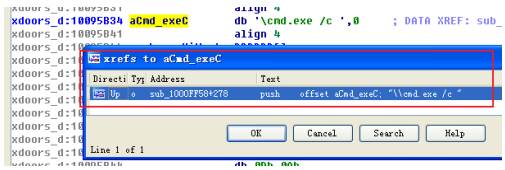


图 3.15: 查看字符串的交叉引用

图3.15可以看到只有一个对该字符串的引用, 双击进入该地址:

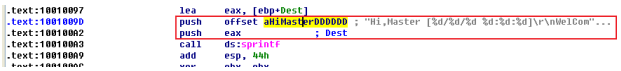


图 3.18: aHiMasterDDDDD

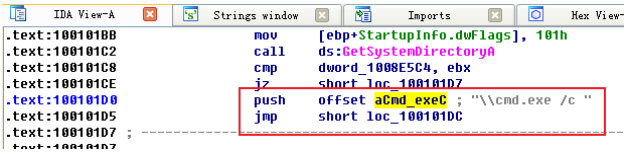
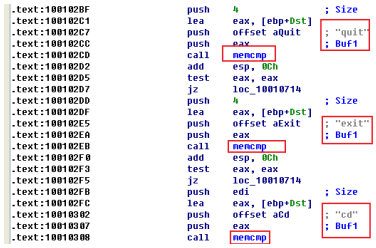


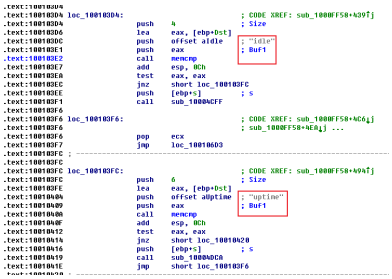
图 3.16: 引用位置代码

图3.16在引用该字符串的位置，可以看到一条命令:" push offset aCmd_exeC ; "\\cmd.exe /c "。这是把该字符串压入栈中作为函数参数。

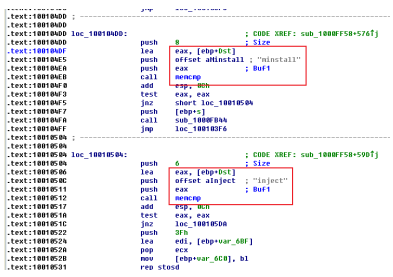
进一步通过图像模式查看附近的函数的目的，通过向下检测发现了如下的一系列函数信息:



((a)) quit,exit 与 cd



((b)) idle 与 uptime



((c)) minstall 与 inject

图 3.17: 查看函数线索

图3.17可以看到在这个字符串被引用的区域附近，出现了很多 **memcmp 函数用来比较 cd、exit、minstall、inject、idle、uptime、quit 等特殊的字符串。可以从这里初步推测这附近可能与一些系统命令相关**。接下来通过向上检测：图3.18可以看到在.text 0x1001009D 处有一个特殊的对某字符串的交叉引用。点进去查看后:

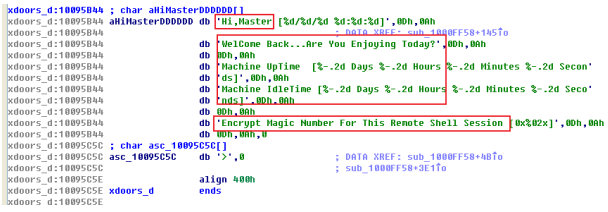


图 3.19: aHiMaterDDDDD 字符串内容

图3.19可以看到一些有趣的字符串，用于显示或记录某些信息。

(a) **'Hi,Master [%d/%d/%d %d:%d:%d]'**: 这个字符串似乎是用于显示或记录日期和时间的。'%d'是一个格式说明符，用于表示整数。从上下文来看，这可能是一个欢迎消息，显示当前的日期和时间。

(b) **'WelCome Back...Are You Enjoying Today?'**: 这是一个欢迎消息，可能在用户登录或启动某个应用程序时显示。

(c) **'Machine UpTime' 与'Machine IdleTime'**: 这两个字符串分别用于显示或记录机器的运行时间即从上次启动到现在的时间，以及显示或记录机器的空闲时间，即机器在没有用户活动的情况下的时间。

(d) **'Encrypt Magic Number For This Remote Shell Session [0x%02x]'**: 这个字符串似乎与远程 shell 会话的加密有关。它可能用于显示或记录用于加密的魔术数字。

除此之外再查看该函数以及在其内容进一步调用的子函数可以发现：

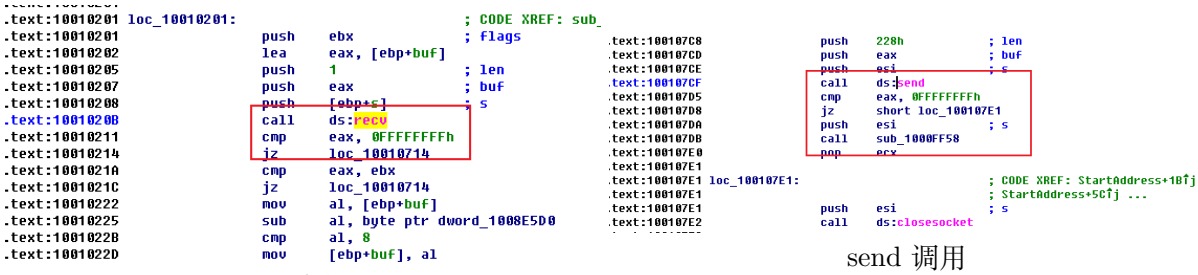


图 3.20: 对应版本

图3.20可以看到一系列 **resv 和 send 的调用，意味着程序正在进行内存分配和发送数据的操作。**

综合上述这些信息，我们可以推测这个恶意代码可能与远程访问或远程 shell 会话有关。用于远程控制或访问目标机器。当攻击者连接到远程 shell 时，它可能会显示这些消息，提供关于目标机器的状态和信息。

9. **Q9: 在同样的区域，在 0x100101C8 处看起来好像 dword _1008E5C4 是一个全局变量，它帮助决定走哪条路径。那恶意代码是如何设置 dword_108E5C4 的呢?**

回答:

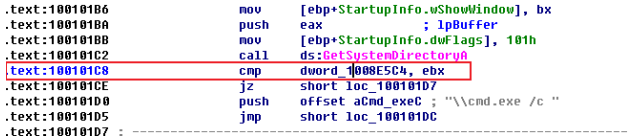


图 3.21: 来到 0x100101C8 处

图3.21可以看到此时在 0x100101C8 处通过 cmp 函数出现了一个变量 dword _1008E5C4, 并对其进行比较，根据比较结果来决定走的路径。双击进一步查看：

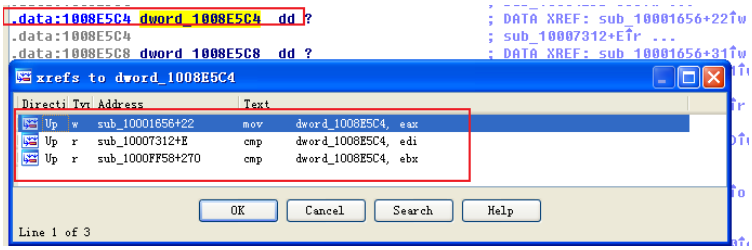


图 3.22: dword _1008E5C4 查看

图3.22可以看到 dword _1008E5C4 在 DLL 的.data 段被定义，而且是作为全局变量。**查看其交叉引用，来探索是否恶意代码对其进行了设置。发现一共具有三个交叉引用，经过查看后两者都是通过将其与 edi 和 ebx 进行比较来进行分支。而第一个交叉引用与其值的设置有关：**

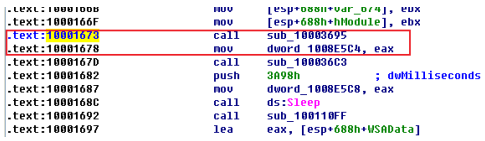


图 3.23: dword _1008E5C4 值的设定

在这个位置 0x10001673 处可以看到如下代码：

```
call sub_10003695
mov dword_1008E5C4, eax
```

由此可以看到恶意代码是**将 eax 的值赋给了全局变量 dword _1008E5C4**。而 eax 的值又正好在上一个子函数 sub_10003695 返回之后，将返回值寄存在了 eax 寄存器中（x86 架构约定）。因此为了能够进一步探索 word _1008E5C4 赋值的依据，来到子函数 sub_10003695 中：

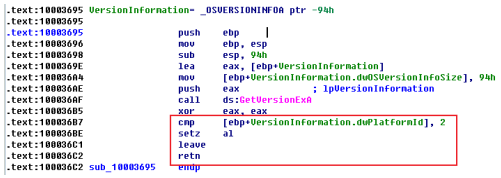


图 3.24: sub_10003695 函数

图3.24可以看到此时函数的汇编代码：

```
call   ds:GetVersionExA
xor    eax, eax
cmp    [ebp+VersionInformation.dwPlatformId], 2
setz   al
```

详细分析这段代码：

(a) **call ds:GetVersionExA**: 这行代码调用了 Windows API 函数 **GetVersionExA**，该函数用于检索关于当前操作系统的版本信息。

(b) **xor eax, eax**: 这行代码将 eax 寄存器的值设置为 0。这是一个常见的汇编技巧，用于快速清零寄存器。

(c) **cmp [ebp+VersionInformation.dwPlatformId], 2**: 这行代码比较 **VersionInformation.dwPlatformId**（一个结构中的成员，该结构由 **GetVersionExA** 填充）的值与 2。在 Windows API 中，**dwPlatformId** 的值为 2 表示操作系统是 Windows NT 系列。

(d) **setz al**: 如果上一行的比较结果为真（即 dwPlatformId 的值确实为 2），则这行代码将 al 寄存器（eax 的低 8 位）的值设置为 1；否则，它将保持为 0。

因此可以得出结论恶意代码判断当前操作系统是否 Windows 2000 或更高版本的 WindowsNT 系列，如果是则将全局变量 dword _1008E5C4 的值置为 1。即某种程序上算是保存了操作系统版本的一些信息

当然除非你遇到的是 Lab2 的我在使用 Windows3.2 进行实验，好好笑。(bushi)

10. **Q10: 在位于 0x1000FF58 处的子过程中的几百行指中，一系列使用 memcmp 来比较字符串的比较。如果对 robotwork 的字符串比较是成功的 (当 memcmp 返回 0)，会发生什么?**

    回答： 从之前的观察可以发现确实调用远程 shell 函数的部分会在 0x1000FF58 开始调用一系列的 memcmp 函数，找到 robotwork 的位置：

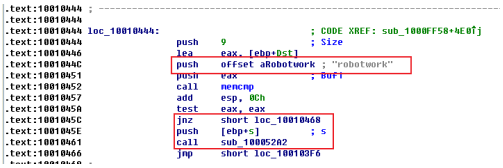


图 3.25: robotwork 部分代码

图3.25可以看到此时的先是对将 robotwork 字符串压入了栈中，而后将其与 eax 即某个缓冲区中的字符串进行比较。查看结果，若这个缓冲区中的字符串就是 robotwork，则不会触发 jnz 而跳转到 loc10010468 处。并且进一步调用 sub_100052A2。查看该函数目的：

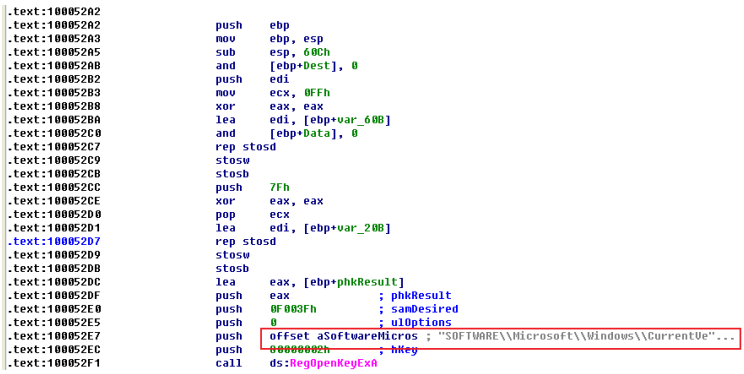


图 3.26: sub_100052A2 处

图3.26可以看到此处对注册表中的注册表项 HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion 的值进行了查询，而后通过 push [ebp+s] 的命令将其返回给 socket 套接字用于进一步通讯。

因此可以得出，注册表项 HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion 的值会被查询，并通过远程 shell 连接发送出去。

11. **Q11:PSLIST 导出函数做了什么?**

    回答：

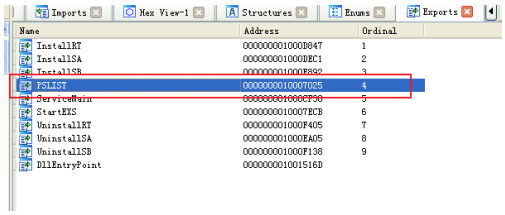


图 3.27: 查看 DLL 导出表

图3.27可以看到此时通过查看 DLL 的导出表，能够找到 PSLIST 函数。双击进一步查看其代码：

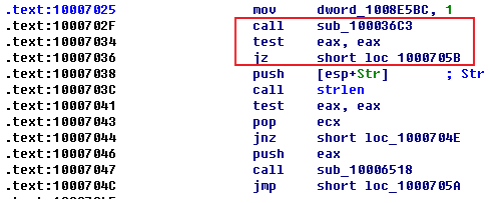


图 3.28: 查看 PSLIST 函数代码

图3.28可以看到通过双击来到了 0x10007025 处，此处的代码会进行一个分支，而分支取决于比较 “test eax,eax” 的结果，而 eax 和 eax 比较结果取决于 sub_100036C3 的返回值 eax。故进一步查看该函数：

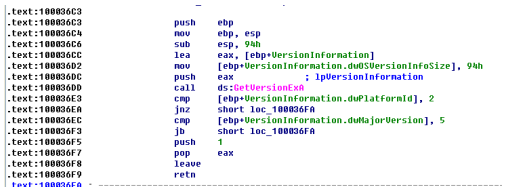


图 3.29: sub_100036C3 代码

图3.29这段代码可以看到通过对操作系统的版本进行检查，判断其是否是 Windows Vista/7 或者是 XP/2003/2000。

再回到刚才的两条路径，深入分析两条路径，以其中一条为例，会到达：

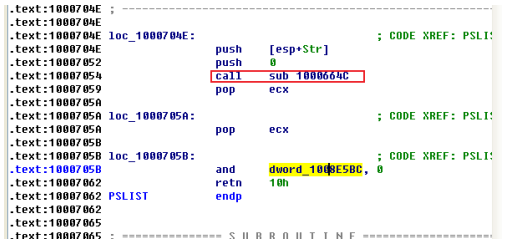


图 3.30: 深入分析

图3.30显示进一步跳转到了 sub_1000664C，查看：

```
.text:1000664C                push    ebp
.text:1000664D                mov     ebp, esp
.text:1000664F                mov     eax, 1634h
.text:10006654                call    __alloca_probe
.text:10006659                and     [ebp+Dest], 0
.text:10006660                push    ebx
.text:10006661                push    edi
.text:10006662                mov     ecx, 0FFh
.text:10006667                xor     eax, eax
.text:10006669                lea     edi, [ebp+var_633]
.text:1000666F                rep stosd
.text:10006671                stosw
.text:10006673                stosb
.text:10006674                push    49h
.text:10006676                xor     ebx, ebx
.text:10006678                pop     ecx
.text:10006679                xor     eax, eax
.text:1000667B                lea     edi, [ebp+pe.cntUsage]
.text:10006681                mov     [ebp+pe.dwSize], ebx
.text:10006687                rep stosd
.text:10006689                mov     ecx, 3FFh
.text:1000668E                lea     edi, [ebp+var_1630]
.text:10006694                mov     [ebp+hModule], ebx
.text:1000669A                push    ebx             ; th32ProcessID
.text:1000669B                rep stosd
.text:1000669D                push    2               ; dwFlags
.text:1000669F                call    CreateToolhelp32Snapshot
.text:100066A4                cmp     eax, 0FFFFFFFFh
```

图 3.31: 发现 CreateToolhelp32Snapshot

图3.31可以发现一个函数 **CreateToolhelp32Snapshot**，继续探索：

```
.idata:100160E8 ; HANDLE __stdcall CreateToolhelp32Snapshot(DWORD dwFlags, DWORD th32ProcessID)
.idata:100160E8                 extrn __imp_CreateToolhelp32Snapshot:dword
.idata:100160E8                                         ; DATA XREF: CreateToolhelp32Snapshot↑r
```

图 3.32: CreateToolhelp32Snapshot

图3.32可以看到该函数应与进程 Process 有关，接受进程作为参数，因此推测其可能用来获取进程列表。

综上所述：使用 PSLIST 导出项可以通过网络发送进程列表，或者搜索该列表以获取特定进程名称并提取相关信息。

12. **Q12: 使用图模式来绘制出对 sub_10004E79 的交叉引用图。当进入这个函数时，哪个 API 函数可能被调用？仅仅基于这些 API 函数，你会如何重命名这个函数？**

回答：

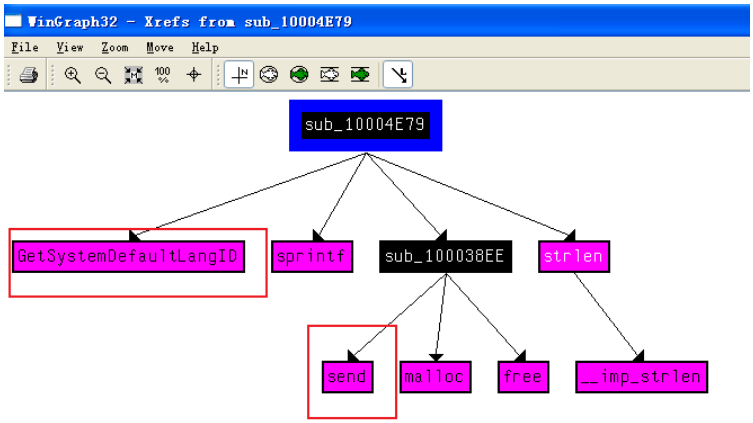


图 3.33: 查看 sub_10004E79 的交叉引用图

图3.33展示了先通过左侧函数找到 sub_10004E79，再打开其交叉引用模式。**但是这次与之前不同，需要打开的是其调用别的函数的情况。**

可以看到其调用了 GetSystemDefaultLangID、send 以及 sprintf 三个函数 API。**胡推测其可能是传递语言标志信息，通过套接字 socket。故这里将其重新命名为：send_LangID_byxiaoyuan，能够更有意义的指向性。**

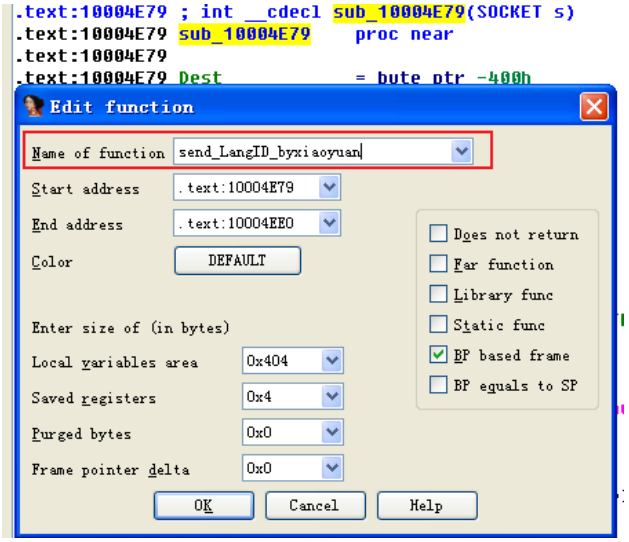


图 3.34: 修改函数 sub_10004E79 函数名

13. **Q13:D11Main 直接调用了多少个 Windows API？多少个在深度为 2 时被调用？**

回答：这里为了查看 DLLMain 调用的函数 API，可以采用交叉引用图的方式。首先先要为交叉引用图调整设置：

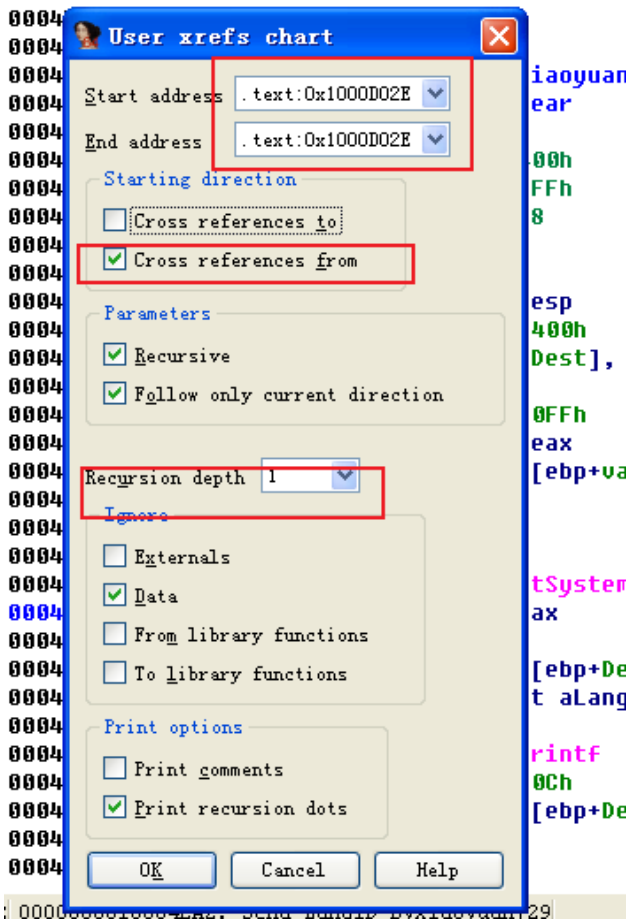


图 3.35: 交叉引用图设置

图3.35显示了使用设置对话框，查找指定函数的调用关系。

- 由于要找的是 DllMain 函数其内部的函数调用。因此首先先将起始和结束地址都设为 DllMain 的起始地址，即 0x1000D02E。
- 只关心调用别的函数，即只选上 Cross references from
- 只关心直接调用，因此设置递归深度为 1。

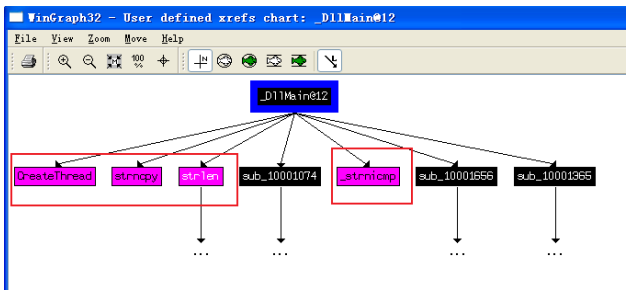


图 3.36: C 显示 DllMain 直接调用的 API 函数

图3.36可以看到 **D11Main 直接调用了 strncpy、strnicmp、CreateThread 和 strlen 这些 API**。接下来设置递归深度为 2：

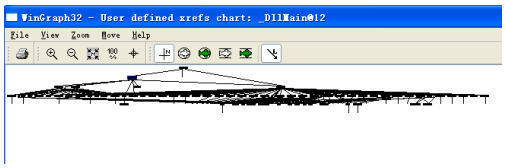
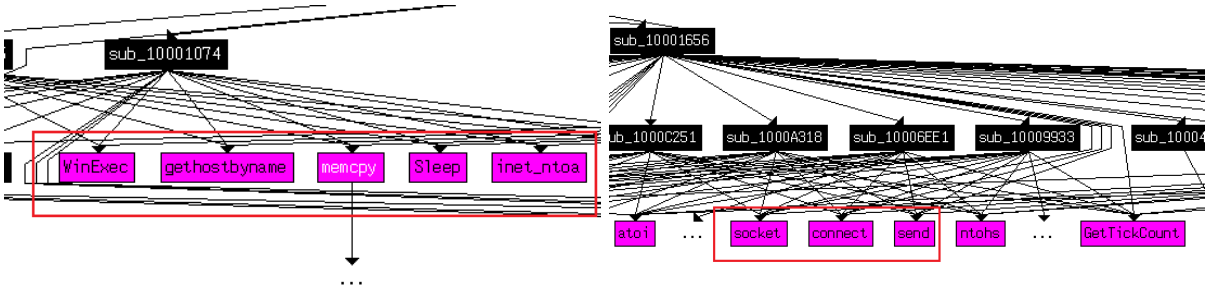


图 3.37: 深度为 2

图3.37可以看到此时调用关系相当复杂，放大来看：



调用关系 1 调用关系 2

图 3.38: 深度为 2 调用

图3.38可以**看到非常多的 API，包括 Sleep、WinExec、gethostbyname，以及许多其他网络函数调用比如 socket,send,connect 等**。

14. **Q14: 在 0x101358 处有一个对 sleep(一个使用一个包含要睡眠的毫秒数的参数的 API 函数)的调用。顺着代码向后看，如果这段代码执行，这个程序会睡眠多久？**

回答：

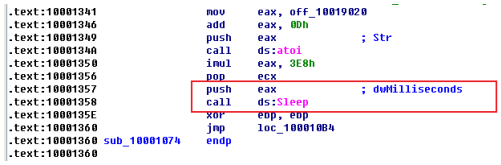


图 3.39: Sleep 函数调用

图3.39可以看到

- 在 0x10001358 处有一个对 sleep 的调用，并且 sleep 使用一个参数，也就是要休眠的毫秒数 dwMilliseconds。这个参数被传入了 eax 中压入栈中供 sleep 使用。
- imul eax,3E8h:eax 被乘了 0x3E8(十进制是 1000)。而 eax 正好是前面 atoi 函数调用的结果返回值。对 atoi 调用的结果被乘以 1000，得到要休眠的毫秒数。
- mov eax, off_10019020：off 100192 被赋给 EAX。双击查看：

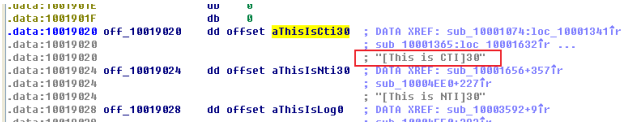


图 3.40: 查看 off_10019020

图3.40可以看到它指向了一个字符串 [This is CTI]30。

再次回答代码位置，其中 add eax, 0Dh：0xD 被加到 EAX 上作为偏移。因此，通过将 EAX 指向 30 来调用 atoi 函数，将字符串"30" 转换为数字 30。然后将 30 乘以 1000，得到 30,000 毫秒（即 30 秒），这就是程序休眠的时间。

故恶意代码会休眠 30 秒。

15. **Q15: 在 0x10001701 处是一个对 socket 的调用。它的 3 个参数是什么?**

回答:

```
.text:100016F5 loc_100016F5:                           ; CODE XREF: sub_10001656+7C↑j
.text:100016F5                                         ; sub_10001656+8F↑j
.text:100016F5                 mov     ebp, ds:closesocket
.text:100016FB
.text:100016FB loc_100016FB:                           ; CODE XREF: sub_10001656+374↓j
.text:100016FB                                         ; sub_10001656+A09↓j
.text:100016FB                 push    6               ; protocol
.text:100016FD                 push    1               ; type
.text:100016FF                 push    2               ; af
.text:10001701                 call    ds:socket
.text:10001707                 mov     edi, eax
.text:10001709                 cmp     edi, 0FFFFFFFFh
.text:1000170C                 jnz     short loc_10001722
.text:1000170E                 call    ds:WSAGetLastError
.text:10001714                 push    eax
.text:10001715                 push    offset aSocketGetlaste ; "socket() GetLastError reports %d\n"
.text:1000171A                 call    ds:__imp_printf
.text:10001720                 pop     ecx
.text:10001721                 pop     ecx
```

图 3.41: 0x10001701 处

图3.41显示了直接通过 jump 到地址 0x10001701 处，可以看到在 socket 函数调用前，三个参数入栈，因此**它的三个参数为 6、1 和 2。**

16. **Q16: 使用 MSDN 页面的 socket 和 IDA Pro 中的命名符号常量你能使参数更加有意义吗? 在你应用了修改以后，参数是什么?**

回答:

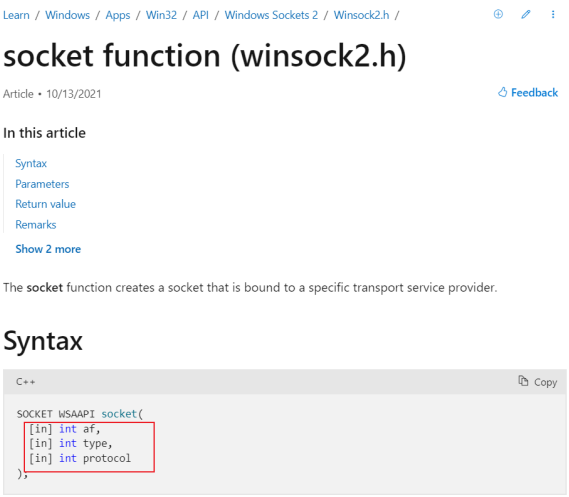


图 3.42: MSDN 对 socket 描述

首先来到 MSDN 页面找到对 socket 的描述，函数原型为：

```
SOCKET WSAAPI socket(
    int af,
    int type,
    int protocol
```

```
5       );
```

其中，**af**、**type** 和 **protocol** 是该函数的参数。

- **af**：指定地址族。常见的地址族有 AF_INET（IPv4）、AF_INET6（IPv6）等。**2 指的是 AF_INET。用来设置 IPv4 socket**。
- **type**：指定套接字的类型。常见的类型有 SOCK_STREAM、SOCK_DGRAM 等。**1 指的是 SOCK_STREAM**。
- **protocol**：指定协议。**6 指的是 IPPROTO_TCP 表示 TCP 协议，这个 socket 会被配置为基于 IPv4 的 TCP 连接在 HTTP 中**。

因此得出结论它们对应的 **3 个符号常量分别是 IPPROTO_TCP、SOCK_STREAM 和 AF_INET**。修改后结果如下图所示：

```
.text:100016FB                  ; sub_1000165
.text:100016FB                  push    IPPROTO_TCP     ; protocol
.text:100016FD                  push    SOCK_STREAM     ; type
.text:100016FF                  push    AF_INET         ; af
.text:10001701                  call    ds:socket
.text:10001707                  mov     edi, eax
.text:10001709                  cmp     edi, 0FFFFFFFFh
.text:1000170C                  jnz     short loc_10001722
.text:1000170E                  call    ds:WSAGetLastError
.text:10001714                  push    eax
.text:10001715                  push    offset aSocketGetlaste ; "soc
.text:1000171A                  call    ds:__imp_printf
.text:10001720                  pop     ecx
.text:10001721                  pop     ecx
```

图 3.43: 修改符号常量后

17. **Q17: 搜索 in 指令（opcode @xED）的使用。这个值和一个魔术字符串 VMXh 用来进行 VMware 检测这在这个恶意代码中被使用了吗？使用对执行 in 指函数的交叉引用，能发现其进一步检测 VMware 的证据吗？**

回答：

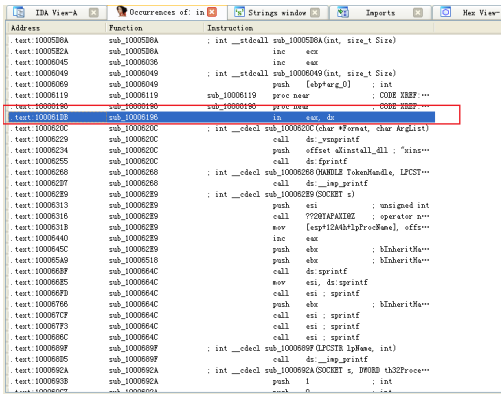


图 3.44: 搜索 in 指令

图3.44先搜索 in 指令，使用 Find All Occurences 显示了所有匹配。其中只有 0x100061DB 是符合条件的，进一步查看：

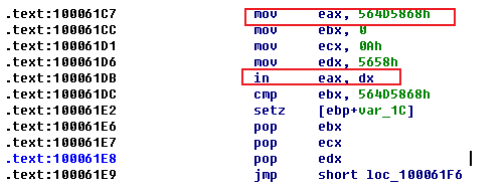


图 3.45: 查看 in 指令

图3.45可以看到此时前面将一个字符串 0x564D5868 赋给 eax，查看该字符串：

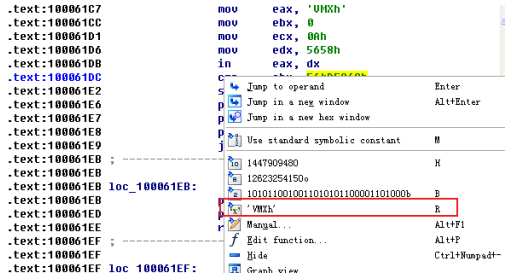


图 3.46: 查看字符串 0x564D5868

图3.46可以看到 **0x564D5868 对应 ASCII 字符"VMXH"。因此证明恶意代码可能使用了反虚拟机的方法来进行自我保护**。

图 3.47: 查看交叉引用

图3.47能够发现其被一个地方引用，来到该地方**看到了字符串 "Found Virtual Machines, Install Cancel"。这正是恶意代码发现自己在虚拟机后放弃安装选择隐藏的进一步检测 VMware 的证据**。

18. **Q18: 将你的光标跳转到 0x1001D988 处，你发现了什么?**

    回答:

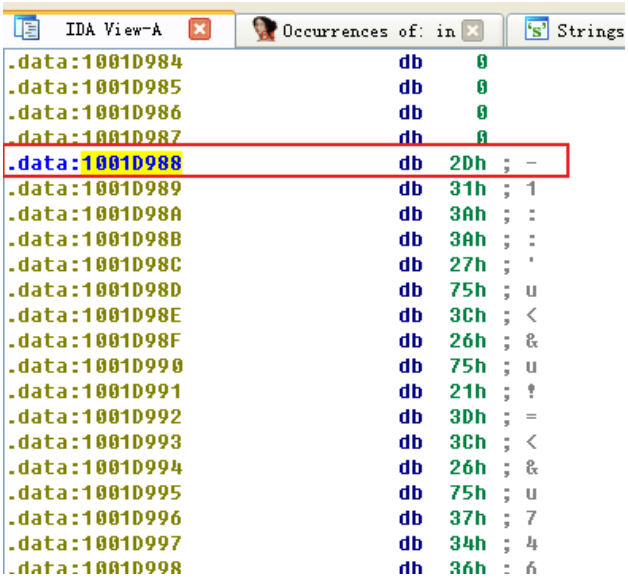


图 3.48: 查看 0x1001D988

图3.48处**位于.data 节，可以看到一些看起来随机的数据**。

19. **Q19: 如果你安装了 IDA Python 插件 (包括 IDA Pro 的商业版本的插件，运行 Lab05-01.py, 一个本书中随恶意代码提供的 IDA Pro Python 脚本，(确定光标是在 0x1001D988 处。) 在你运行这个脚本后发生了什么?**

回答：使用该脚本文件运行，运行结果为：

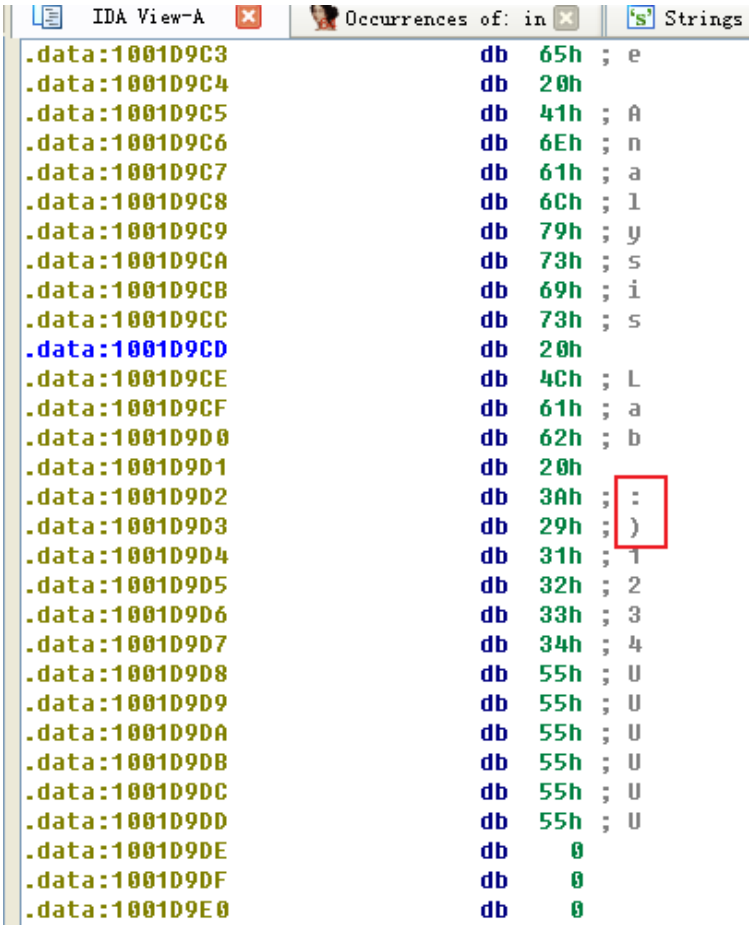


图 3.49: py 脚本运行结果

图3.49可以看到**此时上面的数据被反混淆得到一个 ASCII 字符串并且具有一定含义**。

20. **Q20: 将光标放在同一位置，你如何将这个数据转成一个单一的 ASCII 字符串?**

    回答: 从上往下读有些别扭，但是连起来看是: 是 xdoor is this backdoorstring decoded for Practical Malware Analysis Lab :)1234。

    我真的很喜欢这个非常古老的笑脸哈哈，可能是那个作者年代互联网还和流行的东西。有点可爱，给作者加个心:)。

21. **Q21: 使用一个文本编辑器打开这个脚本。它是如何工作的?**

    回答:

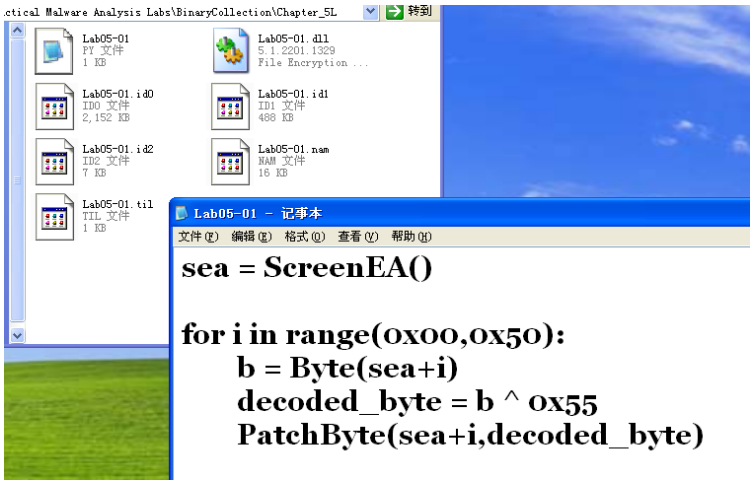


图 3.50: 查看 Python 脚本

图3.50可以看到打开的 Python 脚本 Lab05-01.py，先解释下代码含义：

(a) 使用 **ScreenEA()** 函数获取当前屏幕上的有效地址，将其赋值给变量 **sea**。

(b) 通过一个范围为 **0x00** 到 **0x50** 的循环，对每个字节进行以下操作：

- 使用 **Byte(sea+i)** 函数获取地址 **sea+i** 处的字节值，并将其赋值给变量 **b**。
- 使用 XOR 操作(异或操作)对该字节与 **0x55** 进行解码,得到解码后的字节 **decoded_byte**。
- 使用 **PatchByte(sea+i,decoded_byte)** 函数将地址 **sea+i** 处的字节替换为解码后的字节。

总之该脚本的目的是对 DLL 文件中的某个数据段进行 XOR 解码，可能用于解密那些使用 0x55 作为密钥的数据。这最后还会写回 IDAPro 的显示，而不会修改源文件。

3.4 Yara 检测

3.4.1 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：

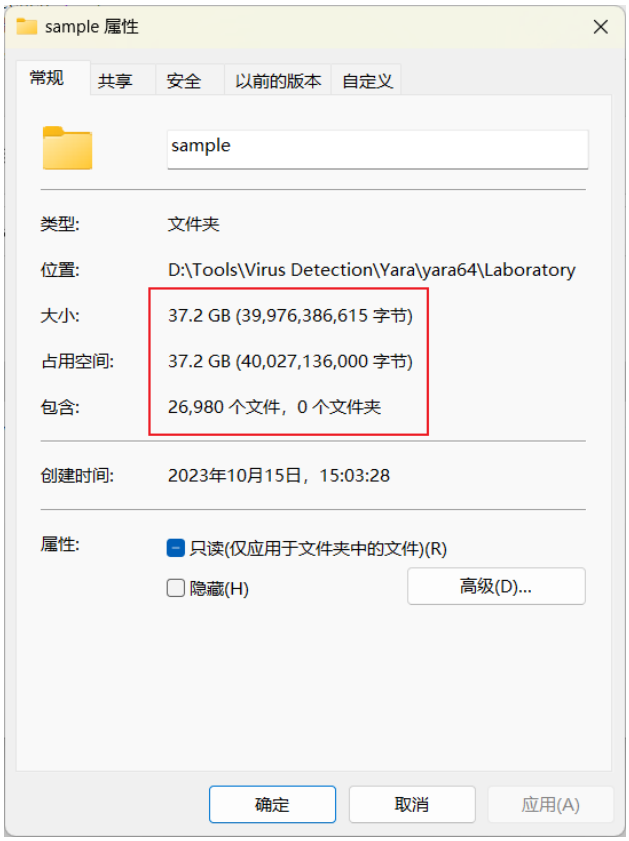


图 3.51: Sample 信息

图3.51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 37.2GB，包含一共 26890 个文件。Yara 编写的规则将目标从 sample 中识别成功检测出 Lab05-01.dll**。

文件比上次大了一些，是因为为了写计算机网络下了个 Qt。

3.4.2 Yara 规则编写

本次 Yara 规则的编写基于上述的病毒分析以及对应的 20 个问题。为了能够更好地进行 Yara 规则的编写，再次使用 IDAPro 的 Strings 窗口直接查看可疑的字符串：

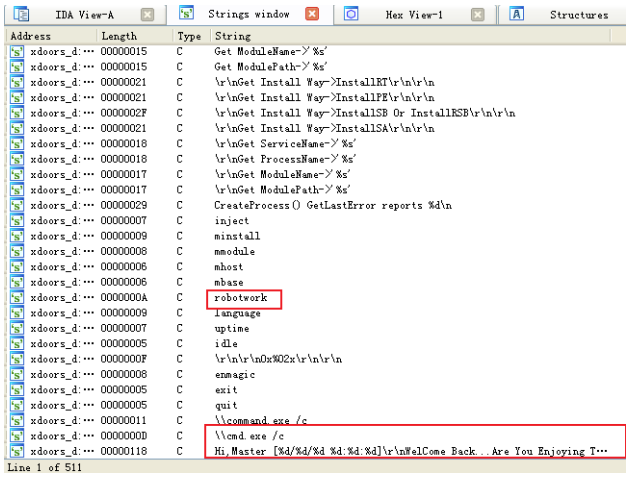
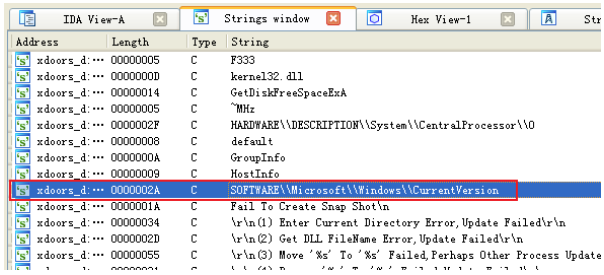
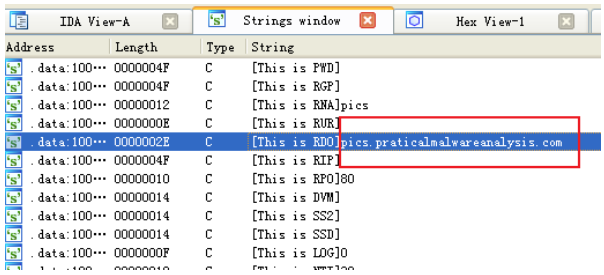


图 3.52: 可疑 Strings1

图3.52可以看到有 **\\cmd.exe /c 以及 robotwork，还有最有标志性的"Hi,Master"**。继续探索：



可疑 Strings2



可疑 Strings3

图 3.53: 继续探索

图3.53可以看到此时又出现了一些前面分析也提到的标志性字符串 **SOFTWARE...CurrentVersion 以及本书彩蛋的 pics.praticalmalwareanalysis.com 再次出现**。

综上所述，结合下面字符串信息编写 Yara 规则：

- **\\cmd.exe /c**：标志病毒调用远程 Shell 窗口会话使用的命令行。
- **robotwork**：与网络相关。
- **Hi,Master**：**最有标志性的字符串**，代表着与远程 Shell 窗口交互连接时，对方对本方客户的问好。
- **SOFTWARE...CurrentVersion**：病毒对宿主机版本的判断。
- **pics.praticalmalwareanalysis.com**：经典彩蛋网址。

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：

```
rule Lab5
{
meta:
   description = "Lab5:Yara Rules"
   date = "2023/10/15"
   author = "ErwinZhou"
strings:
   $clue1 = "pics.praticalmalwareanalysis.com" wide ascii nocase
   $clue2 = "\\cmd.exe /c" wide ascii nocase
   $clue3 = "Hi,Master" wide ascii
   $clue4 = "robotwork" wide ascii nocase
   $clue5 = "SOFTWARE\\Microsoft\\Windows\\CurrentVersion" wide ascii
condition:
   all of them //Lab05-01.dll
}
```

然后使用如下 Python 代码进行对 sample 的扫描：

```python
import os
import yara
import time

# 加载YARA规则
rules = yara.compile('D:\Tools\Virus Detection\Yara\yara64\Lab5.yar')

# 初始化计数器
total_files_scanned = 0
total_files_matched = 0

def scan_folder(folder_path):
    global total_files_scanned
    global total_files_matched

    # 检查文件夹是否存在
    if os.path.exists(folder_path) and os.path.isdir(folder_path):
        # 遍历文件夹内的文件和子文件夹
        for root, dirs, files in os.walk(folder_path):
            for filename in files:
                total_files_scanned += 1
                file_path = os.path.join(root, filename)
                with open(file_path, 'rb') as file:
                    data = file.read()
                    # 扫描数据
                    matches = rules.match(data=data)
                    # 处理匹配结果
                    if matches:
                        total_files_matched += 1
                        print(f"File '{filename}' in path '{root}' matched YARA rule(s):")
                        for match in matches:
                            print(f"Rule: {match.rule}")
    else:
        print(f'The folder at {folder_path} does not exist or is not a folder.')

# 文件夹路径
folder_path = 'D:\Tools\Virus Detection\Yara\yara64\Laboratory\sample'
# 记录开始时间
start_time = time.time()
# 递归地扫描文件夹
```

```
scan_folder(folder_path)
# 记录结束时间
end_time = time.time()
# 计算运行时间
runtime = end_time - start_time

print(f"Program runtime: {runtime} seconds.")
print(f"Total files scanned: {total_files_scanned}")
print(f"Total files matched: {total_files_matched}")
```

3.4.3 Yara 规则执行效率测试

扫描结果如下图所示：

图 3.54: Yara 检测结果

图3.54可以看到**能够唯一地从 26980 个文件中识别检测到 Lab05-01.dll，并且用时 133.80 秒，时间性能较快。总的来说，Yara 规则编写和检测较为成功**。

3.5 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：

3.5.1 Q5: 来到指定地址，并统计参数个数

```
# 查找指定地址的参数个数
# 设置要分析的函数的有效地址
ea = 0x10001656
# 使用idaapi.get_arg_addrs方法获取该函数的参数地址
# ea: 函数的有效地址
# 返回值: 一个包含函数参数地址的列表
arg_addrs = idaapi.get_arg_addrs(ea)
# 打印获取到的参数地址
print(arg_addrs)
```

3.5.2 Q10: 找到指定函数，打印输出其汇编语句

```python
# 获取当前光标所在位置的地址
here_addr = idaapi.here()
# 使用idautils.funcItems方法获取当前函数的所有指令地址
# here_addr: 当前光标所在位置的地址
# 返回值: 一个包含当前函数所有指令地址的列表
dism_addr = list(idautils.funcItems(here_addr))
# 遍历所有的指令地址
for line in dism_addr:
    # 使用idc.generate_disasm_line方法为每个地址生成对应的汇编指令
    # line: 指令地址
    # 0: 生成汇编指令时不包含指令的地址
    # 返回值: 对应的汇编指令
    disasm_line = idc.generate_disasm_line(line, 0)
    # 打印指令地址和对应的汇编指令
    print("0x%x: %s" % (line, disasm_line))
```

3.5.3 Q9 和 Q17: 使用交叉引用查找函数或字符串调用关系

```python
# 查找名称为'in'的函数的地址并打印其汇编指令
# 之后查找并打印所有引用该函数的代码地址及其对应的汇编指令

# 使用idc.get_name_ea_simple方法获取名称为'in'的函数的地址
wf_addr = idc.get_name_ea_simple("in")
# 使用idc.generate_disasm_line方法为该地址生成对应的汇编指令
# wf_addr: 函数地址
# 0: 生成汇编指令时不包含指令的地址
# 返回值: 对应的汇编指令
disasm_line = idc.generate_disasm_line(wf_addr, 0)
# 打印函数地址和对应的汇编指令
print("0x%x: %s" % (wf_addr, disasm_line))
# 使用idautils.CodeRefsTo方法查找所有引用该函数的代码地址
# wf_addr: 函数地址
# 0: 查找直接引用该地址的代码
for addr in idautils.CodeRefsTo(wf_addr, 0):
    # 使用idc.generate_disasm_line方法为每个引用地址生成对应的汇编指令
    ref_disasm_line = idc.generate_disasm_line(addr, 0)
    # 打印引用地址和对应的汇编指令
    print("0x%x: %s" % (addr, ref_disasm_line))
```

3.5.4 Q7 和 Q14: 搜索与遍历字符串

```python
# 搜索当前二进制文件中的前五个包含"sleep"文本的指令
# 对于每个找到的地址，打印该地址及其对应的汇编指令

# 获取当前二进制文件的最小有效地址
cur_addr = idc.get_inf_attr(INF_MIN_EA)
# 循环五次，搜索五个包含"sleep"文本的指令
for x in range(0, 5):
    # 使用ida.search.find_text方法搜索包含"sleep"文本的下一个指令
    # cur_addr: 当前搜索的起始地址
    # 0, 0: 搜索的列范围 (0表示搜索整行)
    # "sleep": 要搜索的文本
    # ida_search.SEARCH_DOWN: 向下搜索
    cur_addr = ida.search.find_text(cur_addr, 0, 0, "sleep", ida_search.SEARCH_DOWN)
    # 如果搜索失败，ida.search.find_text会返回idc.BADADDR
    if cur_addr == idc.BADADDR:
        break
    # 使用idc.generate_disasm_line方法为找到的地址生成对应的汇编指令
    disasm_line = idc.generate_disasm_line(cur_addr, 0)
    # 打印找到的地址和对应的汇编指令
    print("0x%x: %s" % (cur_addr, disasm_line))
    # 使用idc.next_head方法获取下一个指令的地址，为下一次搜索做准备
    cur_addr = idc.next_head(cur_addr)
```

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 静态分析的结果相同，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，通过使用 IDAPro 和 IDAPython，对 Lab5 中的病毒进行了全面的分析，并回答了课本中的 21 个问题。并其中发现了很多有意思的东西。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高。最后为本次实验中部分问题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. 首先我学习到了将课堂中学习的动态分析工具 IDAPro 真正使用来分析一个恶意病毒，在这个过程中学习到了查看字符串、交叉引用、修改常量名等多种操作。

2. 其次我学会了自己搜集资料和探索，跟着实验的问题，我逐渐瓦解和刨析了病毒的特点，发现了许许多多潜在的恶意目的。
3. 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。
4. 我还学会了使用 IDAPython 脚本对病毒进行辅助分析甚至解密和反混淆。
5. **最后我还发现了作者真的很可爱。再夸夸作者:)**

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，也加深了我对病毒分析的能力，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:)