第 19 章 分析对抗与检测逃避技术

恶意软件开发者与攻击者一定会想方设法应对分析人员，为了阻碍分析人员进行分析，他们会制造各种障碍。这就是所谓的分析对抗技术，这会使分析与检测恶意软件的过程变得更加困难。

恶意软件会使用各种各样的分析对抗与检测逃避技术。分析对抗技术通常旨在阻碍恶意软件分析，而检测逃避技术通常是为了规避反恶意软件工具。通常情况下，这两种技术并没有明确的界限，并且许多技术都是在两个领域内通用的。本章将会介绍恶意软件使用的各种分析对抗与检测逃避技术，这些技术能够阻碍恶意软件分析与调试。不仅如此，如何绕过这些对抗技术以便正确分析恶意软件样本文件也是本章讨论的重点。

反分析技术

反分析技术旨在阻碍恶意软件分析的过程。在前面的章节中曾经讨论过部分技术，但此前并未将它们归类为反分析技术。本章将会详细地介绍恶意软件使用的各种反分析技术。

反静态分析

静态分析是在不运行样本文件的情况下直接分析样本文件。正如第 12 章中阐述的，静态分析是通过查看字符串、API 名称、汇编代码与样本文件中的各种异常来进行分析的。

如果使用 C 语言开发，编译后生成的二进制文件与源代码并不完全相同。除非学习过逆向工程，否则通过这种程度的混淆就可以隐藏源代码的实际意图。而使用某些编程语言，如 VB 或者.NET，则能够在编译后的可执行文件中创建更高程度的混淆。即使具备逆向工程能力，也很难分析由这些编程语言创建的可执行文件中的汇编代码。

如果增加加密与压缩，就可以进一步隐藏文件的实际内容。之前介绍的加壳程序，就可以对原始可执行文件的内容进行混淆。想要查看 Payload 的实际内容，就需要实际执行文件或者进行动态分析。加壳程序是恶意软件用于对抗静态分析最有效的工具之一。使用动态分析可以用于找出静态分析无法发现的线索，但是恶意软件实际上也准备好了各种对抗动态分析的技术，后续将进行介绍。

反动态分析

如果样本文件经过加密或加壳，静态分析将会失效。因此，如第 13 章所述，恶意软件就只能通过在动态分析环境中执行样本文件来进行分析。如果恶意软件能通过一些手段检测到它正在被分析，就会尽量避免在执行期间表现出真实行为。这样就无法从动态分析中推断出任何信息。接下来会介绍。想要了解恶意软件如何确定自身正在被分析，就需要回到动态分析所需的设置上来。以下是设置动态分析环境并使用该环境进行分析的步骤：

1. 在虚拟化软件（又叫 Hypervisor，如 VMware、VirtualBox 与 Qemu）上安装虚拟机。
2. 在虚拟机中安装动态分析工具，如 ProcMon、Process Hakcer、Wireshark 等。
3. 分析正在执行的样本文件，启动分析工具观察恶意软件的行为。

以下两种情况中，哪种情况更容易被恶意软件检测到其正在被分析？

- 恶意软件可以使用非常简单的技术，例如识别 CPU 的数量、内存的大小等来确定其是否在分析环境中执行。
- 恶意软件可以尝试寻找分析过程中会使用的分析工具留下的痕迹。例如，虚拟化软件、动态分析工具等都会在分析虚拟机中留下一定的痕迹。恶意软件可以寻找这些痕迹的存在，例如文件、注册表项与进程，以确定其是否正在被分析。这些痕迹的存在通常表明样本文件正处于分析环境中，毕竟大多数实际用户并不需要安装虚拟机或者任何

分析工具。这些痕迹就足以让恶意软件确定它正在被分析。

恶意软件通常会搜索这些痕迹和线索以检测它是否正在被分析，不仅将这些特征列入黑名单，还要在这些特征存在时不显示实际的恶意行为。接下来会介绍部分技术，以及恶意软件会使用哪些技术来识别这些特征。

识别分析环境

大多数时候分析环境都会使用虚拟机，在创建虚拟机时通常会为之配置 CPU 数量、内存大小与硬盘空间大小。通常会选择单核的 CPU、1-2GB 的内存与 20-30GB 的硬盘空间，通常这就足以进行恶意软件分析了，但这个用于分析虚拟机的配置肯定比实际用户所使用的物理机配置要低。如今，大多数笔记本电脑与台式机都会配备四个核甚至更多核数的 CPU、8GB 以上的内存、1TB 的硬盘空间。但分析人员在配置分析虚拟机时通常不会如此“大方”，分配的资源通常远低于平均水平。

恶意软件可以使用这些属性来确定所在环境是否为一个分析虚拟机。例如在 CPU 核数很少、内存大小低于 4GB 或者硬盘大小低于 100GB 的情况下，恶意软件会更容易认为自己正在分析环境中执行。

实际用户使用的系统与分析环境在很多特征上都存在差异，以下是恶意软件常用的、用于确定其正在被分析的一些特征：

- 进程数：与分析环境相比，实际用户使用的系统中会包含很多进程，毕竟大多数用户会在系统中安装各种程序与工具。
- 软件类型：实际用户使用的系统中会安装 Skype、Microsoft Office、PDF 阅读软件（如 Adobe Reader）、媒体播放软件等，但这些软件通常不会安装在分析环境中。
- 登录时间：通常实际用户会登录系统较长时间后才开始执行一些动作，但是在分析环

境通常不会这样。在分析完成恶意软件后，就会迅速还原快照状态。

- 剪贴板数据：实际用户在系统上会存在大量复制、粘贴操作，但在分析环境中通常不会有此类操作，这也是表明处在一个分析环境的明显信号。
- 软件历史记录：实际用户会使用浏览器、办公软件或者各种其他软件，这些软件会留下相应的历史记录。但在分析环境中通常不会使用这些软件进行操作，所以历史记录中也是空白。恶意软件可以通过检查软件的历史记录，来确定其是否正在被分析。
- 存在文件：实际用户通常会在系统的不同文件夹中存放各种文件，包括媒体文件、图片文件、视频文件、.doc 文件、.pptx 文件、音频文件等，但在分析环境中通常不存在这些文件。恶意软件可以通过检查系统上是否缺少此类文件，来确定其是否正在被分析。

众所周知，恶意软件会使用各种方式来区分分析环境与实际用户使用的设备，以上列表也并不完整。正如在第 2 章中所做的那样，调整分析虚拟机使其变得更像实际用户使用的设备非常重要。随着恶意软件使用的对抗技术越来越多，一定要使分析环境尽可能地像实际用户使用的设备，才能欺骗恶意软件。

分析工具识别

在分析虚拟机中安装的分析工具也会在操作系统上创建文件、注册表项等。此外，在运行这些分析工具时，它们也会作为进程运行。众所周知，恶意软件也会检查各种分析工具的存在，包括文件存在、进程存在等，以确定其正处于分析环境中。

例如，某些分析工具会安装在特定的位置，而使用独立可执行文件的分析工具则可以放置在任意位置。恶意软件会尝试查找安装在特定位置的分析工具，也会使用 FindFirstFile 与 FindNextFile 来查找系统中的文件，再与目标列表进行比对。代码 19- 1 显示了恶意软件

查找的两个文件夹名称，查看系统上是否存在这些分析工具。

表 19- 1

分析工具	默认位置
Wireshark	C:\Program Files\Wireshark
Sandboxie	C:\Program Files\Sandboxie

安装某些分析工具可能会在操作系统上创建注册表项，恶意软件也可以通过如 RegQueryValueExA 的 API 来查找此类注册表项以确认分析工具的存在。下面列出了部分恶意软件会查找的注册表项:

- SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall\Sandboxie
- SOFTWARE\SUPERAntiSpyware.com
- SOFTWARE\Microsoft\Windows\CurrentVersion\App Paths\wireshark.exe

不论分析工具的可执行文件位于系统上的什么位置，也不论分析工具会创建什么注册表项。在分析工具执行时，一定会产生进程。因此，恶意软件可以不对文件系统或者注册表来进行遍历与检索，也能够通过与之相关的进程来确定是否使用了分析工具。以下是部分恶意软件会查找的分析工具进程列表:

- SUPERAntiSpyware.exe
- SandboxieRpcSs.exe
- DrvLoader.exe
- ERUNT.exe
- SbieCtrl.exe
- SymRecv.exe
- irise.exe
- IrisSvc.exe

- apis32.exe
- wireshark.exe
- dumpcap.exe
- wspass.exe
- ZxSniffer.exe
- Aircrack-ng
- ollydbg.exe
- observer.exe
- tcpdump.exe
- windbg.exe
- WinDump.exe
- Regshot.exe
- PEBrowseDbg.exe
- ProcessHacker.exe
- procexp.exe

恶意软件可以使用名为 CreateToolhelp32Snapshot、Process32First、Process32Next 的 API 来遍历全部进程，然后利用 StrStrIA 等字符串比较的 API 将系统上的进程名称与分析工具的进程名称列表进行比对。作为练习，读者可以使用 IDA 打开样本文件 Sample-19-2 进行分析并跳转到地址 0x401056，如图 19- 1 所示。

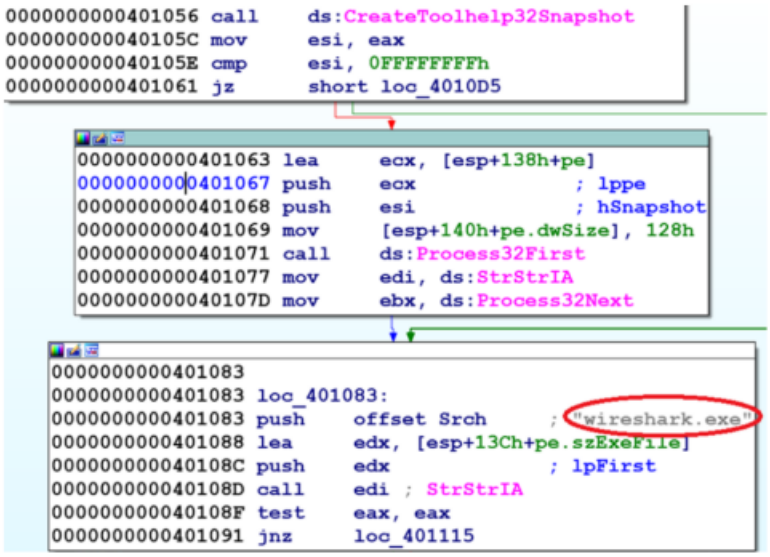

图 19- 1

根据图片可知，该样本文件使用了一种分析对抗技术："使用前文提到的那组 API 列出系统上运行的进程，再检查进程名是否与分析工具 wireshark.exe 的名称相匹配"。

为了绕过分析对抗技术，分析人员可以故意修改分析工具可执行文件的名称。这样当分析工具启动时，就不会显示出实际的分析工具名称，借此欺骗恶意软件。而恶意软件开发者为了再次绕过分析人员的分析，会尝试根据进程的窗口类而不是文件或进程的名称来进行判断。因为创建带有用户界面的程序时，会有相应的窗口类存在。利用名为 FindWindow 的 API，恶意软件就可以查看系统上进程是否拥有特定的窗口类。

作为练习，可以使用 IDA 或者 OllyDbg 打开样本文件 Sample-19-2 进行分析并跳转到地址 0x401022。如图 19- 2 所示，样本文件调用 Findwindow 来查看是否有进程拥有名为 OllyDbg 的窗口类，以此检查 OllyDbg 是否正在运行。

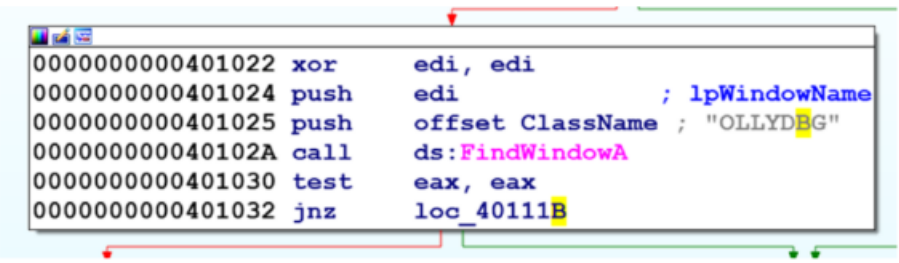

图 19- 2

在通过注册表项与进程检测到分析工具存在时，恶意软件便不会完整执行。并且在大多数情

况下，恶意软件反而会表现出良性行为或者自行退出。由于样本文件未展现出实际的行为，因此并不能断定该样本文件是否为恶意软件。这也是字符串分析能够起作用的地方，可以帮助分析人员确定样本文件中是否使用了反分析技术。通常来说，恶意软件要匹配的文件、注册表项与进程的列表都会出现在进程的内存中。如果样本并未加壳，甚至静态分析时即可获取这些字符串。分析人员可以检查进程的虚拟内存，防止动态分析使用 APIMiner 记录的 API 日志无法给出确定的结果。

为了进一步加强根据字符串表示对抗技术的判断，并且了解恶意软件样本文件对这些字符串的使用，分析人员可以在调试工具或者反汇编工具中查找对这些字符串的引用（XREF）来快速定位使用这些字符串的代码。

虚拟机识别

操作系统中的某些进程、文件与注册表项等特征，可以表明使用了特定类型的虚拟化软件。由于本书使用的分析虚拟机也是由管理平台创建的，因此恶意软件也可以通过与上一节中识别分析工具相同的方式识别虚拟机。以下是各种管理平台会在 Windows 操作系统中创建的文件、进程、注册表项与服务的列表。如表 19- 2 所示，对于 VMware 安装的 Windows 虚拟机，部分文件会位于 C:\windows\system32\drivers 中。

表 19- 2

Vmmouse.sys	vm3dver.dll	vmtray.dll
vm3dgl.dll	vmdum.dll	vmGuestLib.dll

如表 19- 3 所示，对于 VirtualBox 安装的 Windows 虚拟机，部分文件位于 C:\windows\system32。

表 19- 3

Vmmouse.sys	vm3dver.dll	vmtray.dll

vm3dgl.dll	vmdum.dll	vmGuestLib.dll

如所示，为在管理平台上运行的操作系统中创建的与虚拟化平台相关的进程。

表 19- 4

虚拟化软件	进程名
VMware	vmacthlp.exe
VMware	VGAuthService.exe
VMware	vmwaretray.exe
VMware	vmtoolsd.exe
VirtualBox	vboxtray.exe
VirtualBox	vboxcontrol.exe
VirtualBox	vboxservice.exe

与进程相似，在管理平台上运行的操作系统中也会创建一些与虚拟化平台相关的服务，如表 19- 5 所示。

表 19- 5

VMTools	Vmrawdsk	Vmware Tools
Vmxnet	vmx_svga	Vmmouse

在管理平台上运行的操作系统中创建的与虚拟化平台相关的注册表项，如表 19- 6 所示。

表 19- 6

虚拟化软件	注册表项
VMware	HKLM\SYSTEM\ControlSet001\Services\vmware
VMware	HKCU\SOFTWARE\VMware, Inc.\VMware Tools
VirtualBox	HKLM\SYSTEM\ControlSet001\Services\VBoxService.
VirtualBox	HKLM\SYSTEM\ControlSet001\Services\VBoxSF

VirtualBox	KLM\SOFTWARE\Oracle\VirtualBox Guest Additions
VirtualBox	HKLM\HARDWARE\ACPI\DSDT\VBOX__

检测模拟硬件

虚拟化就是在软件程序的帮助下模拟实际的硬件，包括 CPU、硬盘、网卡、内存甚至是支持的指令集。但这毕竟是底层虚拟化软件模拟出的硬件，尤其是在仿真模式下运行时，恶意软件是可以对此进行检测的。接下来分别介绍恶意软件如何识别此类模拟硬件。

检测处理器类型

通过CPUID指令可以用来识别使用的CPU。如果在EAX寄存器等于1的情况下执行CPUID指令，就会通过 EAX、EBX、ECX 与 EDX 寄存器返回 CPU 的各种数据。其中最为重要的是 ECX 寄存器中的第 31 位，如果该值为 0 则说明 CPU 是管理平台模拟的 CPU，如果该值为 1 则说明 CPU 是物理的 CPU。

示例代码如代码 19- 1 所示，该汇编代码可用于判断 CPU 是管理平台模拟的 CPU 还是实际的物理 CPU。

代码 19- 1

```
MOV EAX, 1 # assigns 1 to EAX
CPUID # gets cpu related features in EAX, EBX, ECX, EDX
BT ECX, 1F # BT is bit test instruction which copies
 # 31st(1F) bit to the Carry Flag(CF)
JC VmDetected # check if carry flag is 1 indicating it is a VM
```

与上述示例代码类似，如果在 EAX 寄存器等于 0X40000000 的情况下执行 CPUID 指令，会通过 EBX、ECX 与 EDX 寄存器返回管理平台的厂商名称。如代码 19- 2 所示，该汇编代码可用于获取厂商名称。

代码 19- 2

```
XOR EAX, EAX
```

```
MOV EAX, 0X40000000
CPUID
```

在 VMware 上运行的分析虚拟机中执行这些命令后，寄存器应设置为以下值。

```
代码 19- 3
ebx = 0x61774d56 # awMV
edx = 0x65726177 # erαw
ecx = 0x4d566572 # MVer
```

可以看出这些寄存器中的字符串是什么意思吗？如果将其反转并解码为等效的可打印 ASCII 字符，就可以看到字符串：VMwa waer reVM，这表明管理平台是 VMware。

检测网络设备

通常来说 VMware 与 VirtualBox 为操作系统提供的网卡是模拟的，当然用户也可以使其直接使用底层的实际物理网卡。与物理网卡相同，这些模拟网卡也需要 MAC 地址来进行唯一标识。MAC 地址为六个字节，格式为 xx:xx:xx:xx:xx:xx。管理平台在为模拟网卡生成 MAC 地址时，前三个字节会使用特定的字节序列。例如 VMware 所使用的 MAC 地址会以 00:0C:29、00:1C:14、00:05:69 和 00:50:56 开头，而 VirtualBox 所使用的 MAC 地址会以 08:00:27 开头。除此之外，不同的管理平台可能会使用类似的特定字节序列开头。因此恶意软件也会获取系统上网卡的 MAC 地址，检查其是否与任何已知的、属于管理平台为网卡分配的 MAC 地址能够匹配，以判断恶意软件是否正处于分析环境中。

通讯端口

IN 指令是 ring0 级的指令，该指令只有在真正的物理 CPU 才能在内核态下执行。如果指令是从用户态应用程序执行的，在真正的物理 CPU 下会引发异常。但如果指令是从 VMware 虚拟机中用户态应用程序执行的，则会在 EBX 寄存器中返回 VMware 的 Magic Number。在调用 IN 指令前，需要将 EAX 寄存器设置为 VMXh、将 ECX 寄存器设置为 0xA、将 DX

寄存器设置为 VX。代码 19- 4 为在分析虚拟机中运行的汇编代码片段，该段代码使用 IN 指令来识别底层平台正在使用 VMware。

代码 19- 4

```
MOV EAX, 0X564D5868 # “VMXh” VMWare Magic
MOV ECX, 0xA # 0xA commands gets VMWare version
MOV DX, 'VX’ # Vmware port (0X5658)
MOV EBX, 0
IN EAX, DX # Read port
```

在 VMware 虚拟机中执行指令，EBX 寄存器将会被设置为 0x564D5868（即 VMXh）。这就是 VMware 返回的 Magic Number，可以通过它来确认正处于 VMware 环境中。

现在已经介绍了几种可用于检测虚拟机以及在虚拟机中安装的分析工具的技术，恶意软件经常使用这些技术来检测分析环境。

恶意软件使用的分析对抗技术严重破坏了动态分析能力，包括沙盒等各种依赖动态分析样本文件的反恶意软件产品。但由于恶意软件使用了这些分析对抗技术，可以通过动态分析提取的恶意行为就被恶意软件所篡改了。

当通过动态分析无法得出结论时，就可以转为手动调试样本并对其进行逆向工程。但恶意软件开发者也会使用反调试技术，接下来将会介绍恶意软件用于阻止分析人员逆向工程的部分反调试技术。

反调试

在分析恶意软件时，通常需要对其进行调试。例如想要了解恶意软件在代码层面是如何实现的，就需要对其进行调试。如果一个可疑的可执行文件在动态分析过程中只表现出了极少的行为，也需要对其进行调试。就像正常程序开发一样，当程序没有按照预期执行时就需要进行调试。

同样的，恶意软件开发者也不希望分析人员发现恶意软件中的“秘密”。因此，他们也在恶

意软件中大量应用反调试技术。大多数反调试技术本身就是为了保护软件本身免遭破解。大多数情况下，恶意软件开发者不需要重复发明轮子，只需要使用现有的反调试技术即可保护恶意软件。

接下来介绍一些常见的反调试技术。反调试技术大体上可以分为两类：一类检测调试器，发现后不执行恶意行为的代码；另一类不论是否检测到调试器的存在，都是使用代码混淆来对抗调试器。

使用调试器检测进行反调试

恶意软件到底可以通过哪些技术来发现其是否被调试？当样本文件被调试时，调试器通常会修改被调试进程的一些数据结构与代码。进程环境块（PEB）就是这样的数据结构之一，接下来介绍恶意软件如何使用 PEB 来确定其是否被调试。

基于 PEB 的调试检测

当进程被调试时，进程相关的数据结构会被改变。PEB 是进程最重要的数据结构之一，包含了有关进程的各种信息。下面列出了 PEB 中可用于识别进程是否被调试的重要字段：

- 距 PEB 起始位置 0x2 字节处的 BeingDebugged
- 距 PEB 起始位置 0x68 字节处的 NtGlobalFlags
- 距 PEB 起始位置 0x18 字节处的 ProcessHeap

可以使用 FS 段寄存器访问正在运行的进程的 PEB 数据结构，如代码 19- 5 所示的指令将 PEB 的地址读入 EAX 寄存器。

代码 19- 5

```
MOV EAX, FS:[30] # EAX has address of PEB structure
```

而使用线程环境块（TEB）数据结构地址访问 PEB 的指令如代码 19- 6 所示：

代码 19- 6

```
MOV EAX, FS:[18] # EAX how holds address of TEB structure
MOV EAX, DS:[EAX+30] # EAX will now hold the address of PEB
```

获取了 PEB 后，就可以访问其所属的各种字段，利用这些字段就可以判断该进程是否正在被调试。如前所述，PEB 中的 BeingDebugged 字段位于距起始位置 2 个字节的位置。如果该字段为 1，则表示该进程正在被调试。使用 PEB 中的 BeingDebugged 字段来检测进程是否正在被调试的汇编代码如代码 19- 7 所示:

代码 19- 7

```
XOR EAX, EAX # set all bytes to EAX to 0
MOV EAX, FS:[0x30] # get PEB in EAX
MOVZX EAX, BYTE [EAX+0x2] # EAX= PEB.BeingDebugged to EAX
TEST EAX,EAX # EAX = 1 means debugger is present
 # and the TEST would set ZF to 0
JNE ProcessIsBeingDebugged # Jumps if ZF is 0
```

PEB 中可以利用的另一个字段为 NtGlobalFlags，该字段位于距起始位置 0x68 字节处。NtGlobalFlags 字段用于表示堆在程序中的分配方式，是否存在调试器对分配方式有所影响。存在调试器时，字段中的 FLG_HEAP_ENABLE_TAIL_CHECK 、FLG_HEAP_ENABLE_FREE_CHECK 与 FLG_HEAP_VALIDATE_PARAMETERS 标志位将会被设置为 1。如代码 19- 8 所示，可以检查 NtGlobalFlags 字段中的标志位是否为 1。

代码 19- 8

```
MOV EAX, FS:[0x30] # EAX=address of PEB
MOV EAX, [EAX+0x68] # EAX = PEB.NtGlobalFlags
AND EAX, 0X70 # Checks if the three flags we mentioned
 # in the above para are set
TEST EAX, EAX # EAX = 1 means debugger is present
 # and the TEST would set ZF to 0
JNE ProcessBeingDebugged # Jumps if ZF is 0
```

PEB 中有另一个字段 ProcessHeap 也可用于识别进程是否在被调试。ProcessHeap 字段以及另外两个子字段 Flags 与 ForceFlags，都能够用于识别进程是否在被调试。Flags 字段位于 ProcessHeap 字段内的 0xC 偏移处，而 ForceFlags 字段位于 ProcessHeap 字段内的 0x10 偏移处。如果该进程正在被调试，这两个字段都会被修改。

基于 EPROCESS 的调试检测

EPROCESS 是内核中代表进程的数据结构，且其中的 DebugPort 字段可用于识别进程是否正在调试。如果该字段设置为非零值，则表示该进程正在被调试。恶意软件可以使用名为 NtQueryInformationProcess 的 API 来访问 EPROCESS 结构中的 DebugPort 字段。如果第二个参数 ProcessInformationClass 被设置为 0x7 时调用该 API，这表示进程的 DebugPort 是一个非零值。如果进程正在被调试，第三个参数将会返回非零值。如果未被调试，则会返回 0。代码 19- 9 为示例 C 语言代码，该代码通过前述 API 检测其是否正在被调试。

代码 19- 9

```
DWORD retVal;
NtQueryInformationProcess(-1, 7, retVal, 4, NULL)
if (retVal != 0) {
 ;// Process is being debugged
}
```

使用 Windows API 检测调试器

Windows 操作系统也提供了可以直接访问 PEB 的 API，能够使调用者确认进程是否正在被调试。IsDebuggerPresent 就是典型的 API，很多恶意软件都会使用该 API 来检测是否正在被调试。如果通过 IsDebuggerPresent 检测到进程正在被调试，则会返回 1。代码 19- 10 中的伪代码显示了恶意软件如何使用这个 API。

代码 19- 10

```
int debugger = IsDebuggerPresent();
if (debugger == 1)
 # exit program
else
 # do malicious activity
```

代码 19- 11 为示例 C 代码，对应编译后的样本文件为 Sample-19-1。该程序会检查自身是否正在被调试，并且根据调试状态对应调用不同的代码分支。

代码 19- 11

```
#include <stdio.h>
#include <windows.h>
int main()
{
 int is_being_debugged;
 is_being_debugged = IsDebuggerPresent();
 if (is_being_debugged == 1)
 printf("YES, process is being debugged!\n");
 else
 printf("NO, process is not being debugged!\n");
}
```

如果使用命令提示符独立执行样本文件 Sample-19-1，就可以看到样本文件确认自身未被调试并且打印 else 分支，如图 19- 3 所示。

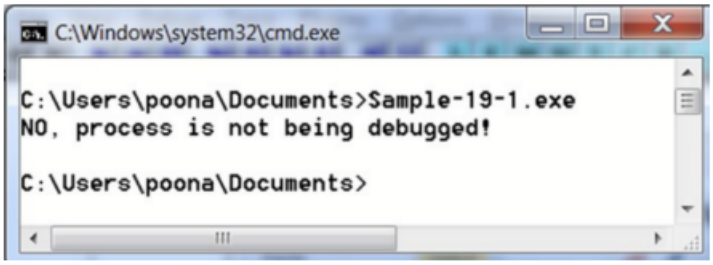

图 19- 3

使用 OllyDbg 打开相同的样本文件并运行，就可以看到样本文件执行了 if 分支。这表明它确实正在被调试，如图 19- 4 所示。

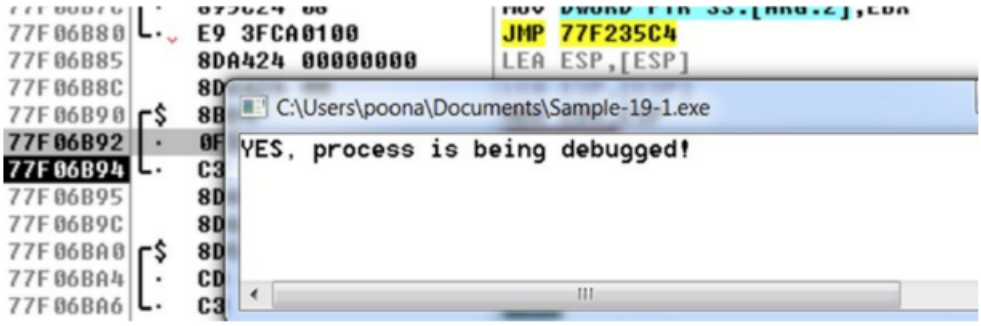

图 19- 4

以下列出了其他可用于识别进程是否在调试的 API:

- CheckRemoteDebuggerPresent
- OutputDebugString
- FindWindow

通过识别断点检测调试

在分析应用程序时，通常会设置不同类型的断点，如软件断点、硬件断点与内存断点。调试器会通过 INT 3 指令替换设置断点的指令，以此在指令处设置软件断点。该指令的操作码为 CC 或 CD 03。恶意软件可以检索所有代码块中的 CC 或 CD 03 字节，以此识别软件断点的存在。当然，必须使用过滤条件来确保其不会将其他用途下的相同字节检出。这些字节通常不是调试器插入的，而是编译器为了填充插入的。

x86 架构通过 DR0-DR7 寄存器来设置硬件断点。而名为 GetThreadContext 的 API 可用于检索 CONTEXT 数据结构中线程的状态。CONTEXT 数据结构中包含与线程状态相关的信息，其中就包括调试寄存器 DR0-DR7。如果恶意软件想要确定自身是否正在被调试，可以通过 API 获取 CONTEXT 并检查调试寄存器的状态。如果寄存器的值不为零，则可认为已设置硬件断点。如代码 19- 12 所示，为检测硬件断点的伪代码。

代码 19- 12

```
CONTEXT Context;
HANDLE hThread;
# get handle to current thread
hThread = GetCurrentThread();
GetThreadContext(hThread, &Context);
if (Context.Dr0 != 0 || Context.Dr1 !=0 ||
 Context.Dr2 != 0 || Context.Dr3!=0 )
{
 //Debugger detected
} else
{
 //Debugger NOT detected
}
```

通过识别指令执行间隔检测调试

在手动调试反汇编代码时，两条指令的执行间隔存在一定程度的滞后。如果在单步执行指令，执行间隔的滞后会更加夸张。恶意软件可以通过比较指令执行的间隔时间来判断其是否正在

被调试。

RTDSC（读取时间戳计数器）是恶意软件用于检测指令执行间隔的最常见指令之一，该指令主要用于获取自系统启动以来的总时间，指令执行的结果返回存储在 EAX 寄存器中。如代码 19- 13 所示，为通过 RTDSC 指令检测进程是否正在被调试的代码。

代码 19- 13

```
XOR EBX, EBX # Clears EBX. Basically sets EBX=0
RTDSC # Retrieves system time in EAX. Call this Time1
MOV EBX, EAX # EBX = Time1
..... other instructions.....
RTDSC # Retrieves system time in EAX. Call this Time2
SUB EAX, EBX # Time2 - Time1
CMP EAX, threshold_lag
```

代码将两个 RTDSC 指令执行的时间作差，并将该值与阈值进行比较。如果两个指令的执行间隔超过阈值，则表示进程正在被调试。请注意，也可以通过 GetTickCount()、QeryPerformanceCounter()等 API 获取执行间隔。

另一种检测调试时指令执行间隔的常用方法是使用 EFLAGS 寄存器中的 trap 标志位。调试器通过代码将 trap 标志位设置为单步执行后，每条指令执行后都会触发由调试器处理的异常。为了检测进程是否正在被调试，恶意软件可以设置自己提供的异常处理程序，然后在 EFLAGS 寄存器中设置 trap 标志位，以触发异常处理。在进一步执行代码时，如果调用了恶意软件自己提供的异常处理程序，则表明此时恶意软件样本文件并未被调试。但如果没有调用恶意软件自己提供的异常处理程序，则表明其他程序（如调试器）接管了异常处理，也就确定了本身正在被调试。

其他反调试技术

恶意软件使用的各种反调试技术层出不穷。一个典型的例子是当在调试器中打开样本文件时，调试器会成为该进程的父进程。恶意软件可以检查其是否存在父进程，如果存在可以进一步

验证父进程是否为调试器。另一个经常被恶意软件用于检测其是否正在被调试的技术是调用像 INT 2D 与 INT 3D 这样的中断，这些中断是被调试器保留使用的。如果中断在被调试的进程中执行，则中断的方式会存在差异，与在未被调试的进程中执行产生相反的结果。

可能有许多类似的反调试技术都是针对特定调试器的，由于调试器的不同实现而各不相同。同样的，可能也已经存在某些滥用调试器中错误与漏洞的反调试技术。作为练习，读者可以尝试通过网络或者其他途径搜索恶意软件使用的各种反调试技术。将这些技术整理起来，不仅要编写对应的示例程序来进行测试，还要注意收集相关的恶意样本。恶意软件开发者会不断使用新技术来保护自身，了解恶意软件使用的最新技术是十分重要的。

使用垃圾代码进行反汇编

恶意软件开发者可以在开发恶意软件时，在有效代码之间掺杂各种垃圾代码。这些插入的垃圾代码并不会影响有效代码的执行，也不会影响恶意软件的正常功能。如代码 19- 14 所示，为一段能够充当垃圾代码的指令。这些指令的执行对程序的功能没有任何影响。

代码 19- 14

```
PUSH EAX
POP EAX
NOP
PUSH EAX
NOP
NOP
POP EAX
```

垃圾代码可能会创建程序中从未使用过的变量，并且修改那些从未使用过的内存区域。这种垃圾代码的存在使得反汇编代码更加不可读，使逆向工程变得更加困难。作为一名逆向分析人员，必须要过滤掉所有垃圾代码，找出包含恶意软件真正功能的有效代码。到目前为止，已经介绍了数种使恶意软件分析变得困难的技术。接下来的几节中，将会介绍恶意软件如何应用使反病毒产品更加难以检测的技术。

规避技术

安全软件与反恶意软件工具试图保护系统免受恶意软件的侵害。反病毒软件用于检测主机上的恶意软件，而入侵检测系统与网络防火墙通过拦截恶意软件在网络上的通信来阻止恶意软件。本书的第六部分（VI）在介绍检测工程时将会详细介绍这些安全产品的工作原理，本章接下来将会先介绍恶意软件常用的规避技术。

反病毒规避

大多数反病毒检测规则都是根据恶意软件样本中的特定模式创建的。之前已经介绍过如何通过加壳来隐藏恶意文件的实际内容，加壳是对抗反病毒检测规则最有效的工具。此外，一些恶意软件会寻找系统上是否存在反病毒软件进程并将其终止。恶意软件可以使用本章前述用于进程遍历的相同 API，来在系统上检索反病毒软件进程。如表 19- 7 所示，恶意软件经常会在系统上检索这些反病毒软件的进程以便将其终止。

表 19- 7

反病毒软件厂商	进程名
McAfee	mcshield.exe
Kaspersky	kav.exe
AVG	avgcc.exe
Symantec	Navw32.exe
ESET	nod32cc.exe
BitDefender	bdss.exe

恶意软件还可以通过阻止反病毒软件访问更新站点来禁用反病毒软件的更新，一般是通过在 host 文件中将安全软件更新站点的域名修改为指向本地实现的。这些域名通过 DNS 解析会

解析为本地主机，从而导致更新失败。Windows 系统上的 host 文件位于 C:\windows\system32\drivers\etc\路径下，如代码 19- 15 所示，为主机的 host 文件被恶意软件篡改的示例。

代码 19- 15

```
www.symantec.com 127.0.0.1
www.sophos.com 127.0.0.1
www.mcafee.com 127.0.0.1
```

网络安全产品规避

业界常用防火墙、入侵检测系统（IDS）与入侵防御系统（IPS）等网络安全产品来阻止恶意软件的网络活动。这些网络安全产品主要依赖于通过分析网络流量、恶意域名与 IP 地址创建的检测规则，但这些规则很容易被恶意软件通过修改流量模式而规避。此外，目前的恶意软件正在从基于 HTTP 的非加密流量向基于 HTTPS 的加密流量转变，仅此就足以绕过部分网络安全产品。

现在也有网络安全产品通过中间人（MITM）模式对加密通信进行解密，以便能够检查通信的实际内容。但并不是所有位置都能够部署防火墙，也不是所有防火墙都能拦截与解密加密流量，传统的 IDS 或者 IPS 仍然会遗漏恶意软件的加密 C&C 流量。

沙盒规避

攻击者可以通过沙盒中使用的各种组件来进行识别。由于沙盒主要用于进行动态分析，因此在沙盒中通常会安装各种动态分析工具。最重要的是，沙盒通常是作为虚拟机进行安装部署的。因此各种通过文件、注册表、虚拟机组件、进程名称来识别虚拟机与分析工具的反分析技术，也都适用于此。通过这些特征，恶意软件可以确定自己正在被分析。除了前述这些反分析技术，恶意软件也可以使用特定的反分析技术来进行沙盒检测。这些技术不仅可以使恶

意软件确定其正在被分析，而且是在沙盒中被分析的。

用户交互

沙盒是自动化分析系统，无需用户的干预即可对恶意软件进行分析。恶意软件则可以利用这一点："尽量自动化且没有用户的干预"，来作为针对沙盒的特定反分析技术。恶意软件开发者可以将恶意软件设计为必须有用户干预才能执行，以此来确保不被沙盒分析。为了实现这一点，恶意软件可能会提示输入部分文本信息或者要要求用鼠标点击消息框，以此来确保系统中确实存在实际用户。沙盒通常无法提供此类行为，恶意软件检测到没有实际用户存在，也不会表现出真实的恶意行为。

恶意软件识别实际用户存在的另一种方式就是检查鼠标移动行为。利用名为 GetCursorPos 的 API 可以检索鼠标的光标位置，恶意软件可以使用此类 API 计算出鼠标移动。如果光标位置在不同的时间点是不同的，则可以假定系统中存在用户活动。同样的逻辑也可以适用于键盘按键，如果恶意软件没有在系统上检测到键盘按键行为，则可以假定其是在没有实际用户的系统中执行的，例如沙盒。

检测知名沙盒

业内有很多沙盒，例如 Cuckoo、Joe Sandbox、Hybrid Analysis 与 CWSandbox。其中部分沙盒是免费的，也有部分沙盒是需要商业付费的。不仅可以通过通用的反分析技术来实现对沙盒的检测，也可以通过沙盒独有的特征来进行识别。例如，沙盒在获取样本文件后会将其传给 Guest 分析虚拟机，但在运行样本之前可能会将其重命名为特定的文件名（sample、virus、malware、application 等），这些文件名可能是与特定的沙盒绑定的。部分沙盒也有特定的用户，例如 John、sandbox-user 等。有些沙盒会在执行文件前，将文件复制到

系统的特定文件夹中。众所周知，恶意软件也会检索这些特定文件、属性，以判断其运行在沙盒中。

部分商业付费的沙盒会使用特定产品密钥的 Windows 操作系统来进行分析，可以通过注册表中的 HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\ProductID 来查询所使用的产品密钥。例如 Anubis 沙盒的 Windows 产品密钥为 76487-337-8429955-22614，而 Joe Sandbox 的产品密钥为 55274-640-2673064-23950。对恶意软件进行分析可知，攻击者也在检查这些特定的产品密钥，以确定恶意软件正在这些特定的沙盒中进行分析。

检测 Agent

对任何沙盒来说，最重要的组件之一就是注入到正在分析的可执行样本中的 Agent DLL 文件。Agent 的任务是记录正在分析的样本所使用的 API，它通过 Hook 样本文件导入的 API 来实现这一点。由于 Agent 通常为 DLL 文件实现，必须先注入样本文件的内存空间后才能对样本所使用的任意 API 进行 Hook。这些 DLL 文件可能会在内存中保存一些字符串或者其他痕迹，而这些可以唯一标识该沙盒 Agent 的供应商。恶意软件可以利用这一点来判断其是否正在沙盒中进行分析。恶意软件可以扫描所有加载的模块与 DLL 文件列表，并分析其中是否存在属于沙盒 Agent 的模块或者 DLL 文件。众所周知，恶意软件会扫描其内存空间以检查是否存在 API Hook。此外，恶意软件也会在扫描内存空间时检查与特定沙盒 Agent 供应商有关的模块或 DLL 文件中的字符串。

计时攻击

使用前述技术就可以识别沙盒环境，如果恶意软件检测到在沙盒中执行，可能会修改执行流。另外也有很多恶意软件会采用无需检测沙盒环境即可绕过沙盒的技术，最常见的就是计时攻

击。计时攻击就像定时炸弹一样，恶意软件在特定日期或者特定时间后才会表现出恶意行为。恶意软件在沙盒中休眠一段时间，等待分析虚拟机的快照恢复，沙盒就不能实际运行样本文件。恶意软件可以利用这一事实，使恶意软件在固定时间配额内不表现出任何恶意行为，以此来逃避沙盒检测。为此，恶意软件经常使用 Sleep 在程序启动后立即进入休眠状态。例如，Sleep(10000)可以使恶意软件在唤醒并继续执行之前先休眠 10000 毫秒。如果恶意软件在实际用户的系统上执行，恶意软件最终会从休眠状态中恢复过来，并且执行真实的恶意行为。但如果恶意软件在沙盒中执行，沙盒会在一段时间后重置虚拟机的状态，恶意软件就不会表现出真实的恶意行为，沙盒便也永远不会发现真实的恶意行为。

除了 Sleep，以下 API 也可以用于延迟执行：

- CreateWaitableTimer
- SetWaitableTimer
- WaitForSingleObject
- WaitForMultipleObject
- NtDelayExecution

如今，沙盒试图通过 Hook 这些 API 并且使超时的 Sleep 值无效来规避基于睡眠或是延迟的对抗措施。恶意软件也可以使用其他方式来实现睡眠，例如通过大量的循环与特殊指令达成延迟，这样就可以绕过那些会被沙盒 Hook 的 API。

愚弄恶意软件装甲

许多分析对抗技术都是使用原始汇编指令与 API 实现的，例如识别虚拟机的 CPUID 指令与 IN 指令、检测样本是否处于调试状态的 IsDebuggerPresent()。分析人员可以在代码中寻找这些特殊指令与 API，来发现恶意软件所使用的反分析技术。识别反分析技术的另一种简

单方法是通过字符串分析，在恶意软件样本文件的内存中通常可以找到与虚拟机检测或分析工具检测相关的字符串。

从安全产品的角度来看，加强产品抵抗恶意软件使用的分析对抗与检测规避技术的能力是极其重要的。在配置沙盒时，尽可能确保分析虚拟机与实际用户系统相似，可以使用 Pafish 这样的分析工具来检查分析环境隐藏的效果。

从逆向分析的角度来看，在识别出恶意软件使用的分析对抗技术的代码、指令与 API 后，就可以对正在调试进程的指令与寄存器进行修正。对进程修正后就可以改变代码执行流程来绕过任意分析对抗手段，分析人员就可以查看恶意软件的实际代码以及其执行的行为。

在第 16 章中，介绍了如何对正在运行的进程进行修正以改变其代码执行流程。以此为例，读者可以尝试修正样本文件 Sample-19-1 中使用 IsDebuggerPresent()作为分析对抗手段的代码流程。在使用调试器对样本文件进行调试时，使其执行 else 分支，如代码 19- 12 所示。

开源项目

业界已经进行了大量的研究来总结恶意软件使用的各种对抗技术，并且编写了对应的分析工具来检测底层环境的各个方面，确定其是否正在被分析。最广为人知的是 Al-Khaser 与 Pafish，如图 19- 5 所示即为运行 Pafish 的截图。从运行的结果来看，Pafish 能够检测到正在 VMware 虚拟机中运行。

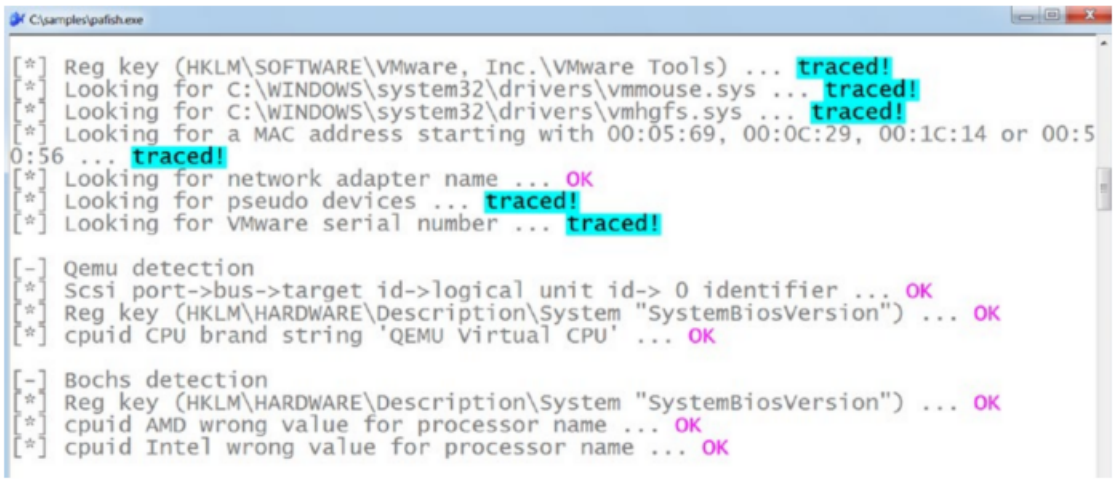

图 19- 5

这两个项目旨在对反恶意软件解决方案进行评估，以便确定目前存在的缺陷，这样就可以针对性改进以加强抵御恶意软件对抗技术。二者都是开源的，可以在 GitHub 上下载 Al-Khaser（https://github.com/LordNoteworthy/al-khaser）与 Pafish（https://github.com/a0rtega/pafish）。读者可以利用这些工具检测是否处于分析环境中，进行练习尝试。

总结

恶意软件会出于各种原因使用各种对抗技术，例如反分析、反逆向与检测逃避等，对抗技术几乎已经成为所有恶意软件功能库的一部分。本章介绍了恶意软件用于识别其正在被分析或调试的各种对抗技术，以便能对抗任何形式的静态分析与动态分析。此外，还介绍了各种恶意软件常用的对抗调试的技术与各种规避技术，借此使分析人员难以进行调试或者安全软件难以进行检测。本章的结尾介绍了作为恶意软件分析人员如何来规避这些对抗技术，以便能够使恶意软件表露出实际的恶意行为。