Broadview
www.broadview.com.cn

安全技术大系

恶意代码分析实战

Michael Sikorski Andrew Honig 著
诸葛建伟 姜辉 张光凯 译

Richard Bejtlich
倾情作序

Practical Malware Analysis
The Hands-On Guide to
Dissecting Malicious Software

no starch press

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

内 容 简 介

本书是一本内容全面的恶意代码分析技术指南，其内容兼顾理论，重在实践，从不同方面为读者讲解恶意代码分析的实用技术方法。

本书分为 21 章，覆盖恶意代码行为、恶意代码静态分析方法、恶意代码动态分析方法、恶意代码对抗与反对抗方法等，并包含了 shellcode 分析，C++恶意代码分析，以及 64 位恶意代码分析方法的介绍。本书多个章节后面都配有实验并配有实验的详细讲解与分析。通过每章的介绍及章后的实验，本书一步一个台阶地帮助初学者从零开始建立起恶意代码分析的基本技能。

本书获得业界的一致好评，IDA Pro 的作者 Ilfak Guilfanov 这样评价本书：“一本恶意代码分析的实践入门指南，我把这本书推荐给所有希望解剖 Windows 恶意代码的读者”。

本书的读者群主要是网络与系统安全领域的技术爱好者与学生及恶意代码分析研究方面的安全从业人员。

版权贸易合同登记号　图字：01-2013-4025

图书在版编目（CIP）数据

恶意代码分析实战 /(美)斯科尔斯基(Sikorski,M.)，(美)哈尼克(Honig,A.)著；诸葛建伟，姜辉，张光凯译. —北京：电子工业出版社，2014.4
（安全技术大系）
书名原文：Practical Malware Analysis:The Hands-On Guide to Dissecting Malicious Software
ISBN 978-7-121-22468-3

Ⅰ. ①恶… Ⅱ. ①斯… ②哈… ③诸… ④姜… ⑤张… Ⅲ. ①电子计算机－安全技术－码 Ⅳ. ①TP309

中国版本图书馆 CIP 数据核字（2014）第 026844 号

策划编辑：刘　皎
责任编辑：贾　莉
印　　刷：北京中新伟业印刷有限公司
装　　订：三河市皇庄路通装订厂
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×980　1/16　　印张：45.75　　字数：1134 千字
印　　次：2014 年 4 月第 1 次印刷
印　　数：3000 册　　定价：128.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。
质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件到 dbqq@phei.com.cn。
服务热线：（010）88258888。

第 0 章　恶意代码分析技术入门

在我们学习恶意代码分析的具体技术之前，首先需要定义一些术语、了解一些常见的恶意代码类型，以及介绍恶意代码分析的基本方法。任何以某种方式对用户、计算机或网络造成破坏的软件，都可以被认为是恶意代码，包括计算机病毒、木马、蠕虫、内核套件、勒索软件、间谍软件，等等。尽管恶意代码以许多不同的形态出现，但分析恶意代码的技术是通用的。你选择使用哪项技术，将取决于你的目标。

0.1　恶意代码分析目标

恶意代码分析的目标，通常是为一起网络入侵事件的响应提供所需信息。因此，你的目标往往是确定到底发生了什么，并确保你能够定位出所有受感染的主机和文件。在分析可疑恶意代码时，你的目标通常是确定某一个特定的可疑二进制程序到底可以做什么，如何在网络上检测出它，以及如何衡量并消除它所带来的损害。

一旦你确定了哪些文件需要全面分析，便是时候来编写出相应的检测特征码了，以便在网络中检测出恶意代码感染的主机。你可以从本书中学到，恶意代码分析可以用来编写出基于主机的和基于网络的检测特征码。

基于主机的特征码，或称为感染迹象，用于在受感染主机上检测出恶意代码。这些迹象经常是恶意代码所创建或修改的文件，或是它们对注册表的特定修改。与反病毒软件所使用的病毒特征码不同，恶意代码感染迹象关注的是恶意代码对系统做了什么，而不是恶意代码本身的特性。这使得有时它们较反病毒软件特征码会更加有效，比如检测那些经常变化自身形态的多态性恶意代码，甚至是恶意代码已经将自身文件从硬盘中删除。

网络特征码是通过监测网络流量来检测恶意代码的。网络特征码可以在没有进行恶意代码分析时创建，但在恶意代码分析帮助下提取的特征码往往是更加有效的，可以提供更高的检测率和更少的误报。

在获得特征码之后，最终目标是要弄清楚究竟这些恶意代码是如何工作的。而这往往是高级管理人员经常提出的问题，他们希望得到一起重大入侵事件的详细解释。从本书中你所学习到的技术，

将允许你确定出恶意程序的目标与功能特性。

0.2 恶意代码分析技术

在大多数情况下，进行恶意代码分析时，你将只有恶意代码的可执行文件本身，而这些文件并不是人类可读的。为了了解这些文件的意义，你需要使用各种工具和技巧，而每种只能揭露出少量的信息。因此你需要综合使用各种工具，才能看到一个全貌。

恶意代码分析有两类基本的方法：静态分析与动态分析。静态分析方法是在没有运行恶意代码时对其进行分析的技术，而动态分析方法则需要运行恶意代码，而这两类技术又进一步分为基础技术和高级技术。

0.2.1 静态分析基础技术

静态分析基础技术包括检查可执行文件但不查看具体指令的一些技术。静态分析基础技术可以确认一个文件是否是恶意的，提供有关其功能的信息，有时还会提供一些信息让你能够生成简单的网络特征码。静态分析基础技术是非常简单，同时也可以非常快速应用的，但它在针对复杂的恶意代码时很大程度上是无效的，而且它可能会错过一些重要的行为。

0.2.2 动态分析基础技术

动态分析基础技术涉及运行恶意代码并观察系统上的行为，以移除感染，产生有效的检测特征码，或者两者。然而，在你可以安全运行恶意代码之前，你必须建立一个安全环境，能够让你在避免对你的系统与网络带来风险的前提下，研究运行的恶意代码。像静态分析基础技术一样，动态分析基础技术可以被大多数没有深厚编程知识的人所使用，但是它们并非对所有恶意代码都是有效的，也会错过一些重要功能。

0.2.3 静态分析高级技术

静态分析高级技术，主要是对恶意代码内部机制的逆向工程，通过将可执行文件装载到反汇编器中，查看程序指令，来发现恶意代码到底做了什么。这些指令是被CPU执行的，所以静态分析高级技术能够告诉你程序具体做了哪些事情。然而，静态分析高级技术较基础技术相比，有着较为陡峭的学习曲线，并且需要掌握汇编语言、代码结构、Windows操作系统概念等专业知识，而所有这些你都可以在本书中学到。

0.2.4　动态分析高级技术

动态分析高级技术则使用调试器来检查一个恶意可执行程序运行时刻的内部状态。动态分析高级技术提供了从可执行文件中抽取详细信息的另一条路径。

这些技术在你尝试获取采用其他技术难以得到的信息时是最有用的。在本书中，我们会向你展示如何结合使用动态分析高级技术和静态分析高级技术，更完备地分析可疑的恶意代码。

0.3　恶意代码类型

进行恶意代码分析时，你会发现一个非常有用的经验技巧，就是一旦你能够猜测出这个恶意代码样本在尝试做些什么，然后去验证这些猜想，就会加速你的分析过程。当然，如果你知道恶意代码通常会做哪些事情，那么你就能够做出更准确的猜测。到目前为止，绝大多数的恶意代码都可以被分到如下类别中。

后门：恶意代码将自身安装到一台计算机来允许攻击者访问。后门程序通常让攻击者只需很少认证甚至无须认证，便可连接到远程计算机上，并可以在本地系统执行命令。

僵尸网络：与后门类似，也允许攻击者访问系统。但是所有被同一个僵尸网络感染的计算机将会从一台控制命令服务器接收到相同的命令。

下载器：这是一类只是用来下载其他恶意代码的恶意代码。下载器通常是在攻击者获得系统的访问时首先进行安装的。下载器程序会下载和安装其他的恶意代码。

间谍软件：这是一类从受害计算机上收集信息并发送给攻击者的恶意代码。比如：嗅探器、密码哈希采集器、键盘记录器等。这类恶意代码通常用来获取E-mail、在线网银等账号的访问信息。

启动器：用来启动其他恶意程序的恶意代码。通常情况下，启动器使用一些非传统的技术，来启动其他恶意程序，以确保其隐蔽性，或者以更高权限访问系统。

内核套件：设计用来隐藏其他恶意代码的恶意代码。内核套件通常是与其他恶意代码（如后门）组合成工具套装，来允许为攻击者提供远程访问，并且使代码很难被受害者发现。

勒索软件：设计成吓唬受感染的用户，来勒索他们购买某些东西的恶意代码。这类软件通常有一个用户界面，使得它看起来像是一个杀毒软件或其他安全程序。它会通知用户系统中存在恶意代码，而唯一除掉它们的方法只有购买他们的“软件”，而事实上，他们所卖软件的全部功能只不过是将勒索软件进行移除而已。

发送垃圾邮件的恶意代码：这类恶意代码在感染用户计算机之后，便会使用系统与网络资源来

发送大量的垃圾邮件。这类恶意代码通过为攻击者出售垃圾邮件发送服务而获得收益。

蠕虫或计算机病毒：可以自我复制和感染其他计算机的恶意代码。

恶意代码还经常会跨越多个类别。例如，一个程序可能会有一个键盘记录器，来收集密码，而它可能同时有一个蠕虫组件，来通过发送邮件传播自身。所以不要太陷入根据恶意代码功能进行分类的误区。

恶意代码还可以根据攻击者的目标分成是大众性的还是针对性的两类。大众性的恶意代码，比如勒索软件，采用的是一种撒网捞鱼的方法，设计为影响到尽可能多的机器。在这两类恶意代码中，这类是最为普遍的，通常也不会太过复杂，而且是更容易被检测和防御的，因为安全软件以这类恶意代码作为防御目标。

而针对性恶意代码，比如特制后门，是针对特定组织而研制的。针对性恶意代码在网络上是比大众性恶意代码更大的安全威胁，因为它们不是广泛传播的，而你的安全产品很可能不会帮你防御它们。如果没有对针对性恶意代码的具体分析，你要保护你的网络免受这类恶意代码侵害或是移除感染，都几乎是不可能的。针对性恶意代码通常是非常复杂的，而你对它们的分析往往需要借助本书中提到的一些高级分析技巧。

0.4 恶意代码分析通用规则

在本章的最后，我们将介绍几个在进行恶意代码分析时需要牢记的通用规则。

首先，不要过于陷入细节。大多数恶意程序会是庞大而复杂的，你不可能了解每一个细节。你需要关注最关键的主要功能。当你遇到了一些困难和复杂的代码段后，你应该在进入到细节之前有一个概要性的了解。

其次，请记住对于不同的工作任务，可以使用不同的工具和方法。这里没有一种通吃的方法。每一种情况是不同的，而你将要学习的各种工具和技术将有类似的，有时甚至重叠的功能。如果你在使用一个工具的时候没有很好的运气，那么尝试另外一种。如果你在一个点上被卡住了，不要花太长时间在这个点上，尝试转移到其他问题。尝试从不同角度来分析恶意代码，或只是尝试不同的方法。

最后，请记住，恶意代码分析就像是猫抓老鼠的游戏。在新的恶意代码分析技术开发的同时，恶意代码编写者也在回应着可以挫败分析的新技术。作为一名恶意代码分析师，你如果想要取得成功，就必须能够认识、理解和战胜这些新技术，并能够快速地应对恶意代码分析艺术的新变化。

第 6 章　识别汇编中的 C 代码结构

在第4章中，我们讨论了x86体系结构以及它最常用的指令。但是成功的逆向工程师并不会单独评估每一条指令，除非到他们必须这么做的时候。这个过程实在是太乏味了，并且整个被反汇编程序中的指令数目可以达到上千甚至上百万。作为一名恶意代码分析师，你必须能通过技巧，来获得一个高层次的代码功能视图，这个技巧就是以组为单位来分析指令，只在需要时再将精力集中在单条指令上。这个技巧的使用需要时间积累。

作为开始，让我们先考虑一个恶意代码作者是如何开发代码的，从而确定该如何分组指令。恶意代码典型情况下都是采用高级语言开发的，大多数时候是C语言。一个代码结构是一段代码的抽象层，它定义了一个功能属性，而不是它的实现细节。代码结构的例子包括循环、`if`语句、链接表、`switch`语句，等等。程序可以被划分为单独的结构，当它们组合到一起时，才能实现程序的总体功能。

本章将通过讨论超过十种不同的C代码结构，领你入门。尽管本章的目的是帮助读者学习逆向分析，但我们还是会检查每一种代码结构的汇编语言代码，你作为恶意代码分析师的目标是将恶意代码程序反汇编后，再恢复到高级语言结构。而对于从高级语言结构到反汇编这个反方向的学习通常更简单，因为计算机程序员习惯阅读和理解高级语言源代码。

本章将精力集中在较常见的和有难度的结构上，比如循环和条件语句。当你构建了这些基础以后，你将学习如何快速地形成代码功能的高级视图。

除了讨论结构的区别外，我们也会验证编译器之间的区别，因为编译器版本和设置能影响一个特定结构在反汇编代码中的表现。我们会评估使用不同的编译器编译`switch`语句和函数调用的两种不同方式。本章将在C代码结构上进行更深层次的挖掘，所以总体而言，你理解的C和编程越多，你从中学到的东西就会越多。想在C语言上获得帮助，看一看由Brian Kernighan和Dennis Ritchie写的*The C Programming Language*（Prentice-Hall，1988）。多数恶意代码由C语言编写，尽管有时用Delphi和C++编写。C语言是一个与汇编有着紧密关系的简单语言，所以它是新手恶意代码分析师开始的最合理的地方。

阅读本章时，记住你的目标是理解一个程序的总体功能，而不是分析每一条指令。时刻记住这一点，不要在细节上陷入困境。将精力集中在程序整体上是如何工作的，而不是它们是如何做每一件特定事情的。

6.1 全局与局部变量

全局变量可以被一个程序中的任意函数访问和使用。局部变量只能在它被定义的函数中访问。在C中全局和局部变量的声明方式是相似的，但是在汇编中看起来完全不同。

下面是全局变量和局部变量的两个C语言代码例子。全局变量的例子见代码清单6-1，在函数外面定义了x和y变量。局部变量的例子见代码清单6-2，这两个变量在函数中被定义。

代码清单6-1 有两个全局变量的简单程序例子

```
int x = 1;
int y = 2;

void main()
{
   x = x+y;
   printf("Total = %d\n", x);
}
```

代码清单6-2 有两个局部变量的简单程序例子

```
void main()
{
   int x = 1;
   int y = 2;

   x = x+y;
   printf("Total = %d\n", x);
}
```

在这两个C代码例子中，全局和局部变量的差别并不大，而且在这个例子中程序结果是一样的。但在反汇编代码中，如代码清单6-3和6-4中显示的，差别却很大。全局变量通过内存地址引用，而局部变量通过栈地址引用。

在代码清单6-3中，全局变量x通过dword_40CF60来标记，一个在0x40CF60处的内存位置。注意在❶处，eax的值被赋给dword_40CF60，所以x的值被修改了。所有后续使用这个变量的函数都会受影响。

代码清单6-3 代码清单6-1中的全局变量例子的汇编代码

```
00401003        mov     eax, dword_40CF60
00401008        add     eax, dword_40C000
0040100E        mov     dword_40CF60, eax ❶
00401013        mov     ecx, dword_40CF60
00401019        push    ecx
```

```
0040101A        push    offset aTotalD  ;"total = %d\n"
0040101F        call    printf
```

在代码清单6-4和6-5中，局部变量x位于栈上一个相对ebp的常量偏移处。在代码清单6-4中，内存位置[ebp-4]在整个函数中被一致地用来引用局部变量x。这告诉我们ebp-4是一个基于栈的局部变量，它只在被定义的那个函数中被引用。

代码清单6-4　代码清单6-2中局部变量例子的汇编代码（未经标记）

```
00401006        mov     dword ptr [ebp-4], 1
0040100D        mov     dword ptr [ebp-8], 2
00401014        mov     eax, [ebp-4]
00401017        add     eax, [ebp-8]
0040101A        mov     [ebp-4], eax
0040101D        mov     ecx, [ebp-4]
00401020        push    ecx
00401021        push    offset aTotalD  ; "total = %d\n"
00401026        call    printf
```

在代码清单6-5中，x已经被IDA Pro反汇编器用假名var_4漂亮地标记了。正如我们在第5章中讨论的，假名可以被重命名为反映它们功能的有意义的名字。将这个局部变量命名为var_4而不是-4简化了你的分析，因为一旦你重命名var_4为x，你就不需要贯穿整个函数中在你的头脑里跟踪偏移-4了。

代码清单6-5　代码清单6-2中显示的局部变量例子的汇编代码（经过标记）

```
00401006        mov     [ebp+var_4], 0
0040100D        mov     [ebp+var_8], 1
00401014        mov     eax, [ebp+var_4]
00401017        add     eax, [ebp+var_8]
0040101A        mov     [ebp+var_4], eax
0040101D        mov     ecx, [ebp+var_4]
00401020        push    ecx
00401021        push    offset aTotalD  ; "total = %d\n"
00401026        call    printf
```

6.2　反汇编算术操作

许多不同类型的算术操作可以用C编程来执行，我们在这节将展示这些操作的反汇编。

代码清单6-6显示了两个变量和一些算术操作的C代码。其中，两个是- -和++操作，它们分别被用来自减1和自增1。%操作在两个变量之间执行取模操作，这个操作是取得执行一次除法之后的余数。

代码清单6-6 两个变量和一些算术操作的C代码

```
int a = 0;
int b = 1;
a = a + 11;
a = a - b;
a--;
b++;
b = a % 3;
```

代码清单6-7显示了代码清单6-6中C语言代码的汇编，它们可以被翻译回C语言。

代码清单6-7 代码清单6-6中算术操作例子的汇编代码

```
00401006        mov     [ebp+var_4], 0
0040100D        mov     [ebp+var_8], 1
00401014        mov     eax, [ebp+var_4] ❶
00401017        add     eax, 0Bh
0040101A        mov     [ebp+var_4], eax
0040101D        mov     ecx, [ebp+var_4]
00401020        sub     ecx, [ebp+var_8] ❷
00401023        mov     [ebp+var_4], ecx
00401026        mov     edx, [ebp+var_4]
00401029        sub     edx, 1 ❸
0040102C        mov     [ebp+var_4], edx
0040102F        mov     eax, [ebp+var_8]
00401032        add     eax, 1 ❹
00401035        mov     [ebp+var_8], eax
00401038        mov     eax, [ebp+var_4]
0040103B        cdq
0040103C        mov     ecx, 3
00401041        idiv    ecx
00401043        mov     [ebp+var_8], edx ❺
```

在这个例子中，a和b是局部变量，因为它们被通过栈来引用。IDA Pro已经标记了a为var_4，将b标记为var_8。首先，var_4和var_8分别被初始化为0和1。a被移动到eax中❶，然后0x0b被加给eax，从而a增加了11，b然后被a减❷。（编译器决定使用sub和add指令❸和❹，而不是inc和dec指令。）

最后5条汇编指令实现了取模。当执行div或idiv指令时❺，你是在用edx:eax除操作数并将结果保存到eax中，余数保存到edx中。这就是为什么edx被移动到var_8中的原因❺。

6.3　识别if语句

程序员使用if语句来实现基于特定条件改变程序的执行。if语句在C代码和汇编代码中很常见。我们将在本节中检验基本的和嵌套的if语句。你的目标应该是学习如何识别不同类型的if语句。

代码清单6-8显示了一个简单C的if语句，这段代码对应的汇编代码在代码清单6-9中显示。注意在❷处的条件跳转jnz。对于一个if语句必定有一个条件跳转，但不是所有条件跳转都对应if语句。

代码清单6-8　if语句例子的C代码

```
int x = 1;
int y = 2;

if(x == y){
      printf("x equals y.\n");
}else{
      printf("x is not equal to y.\n");
}
```

代码清单6-9　代码清单6-8中if语句例子的汇编代码

```
00401006        mov     [ebp+var_8], 1
0040100D        mov     [ebp+var_4], 2
00401014        mov     eax, [ebp+var_8]
00401017        cmp     eax, [ebp+var_4] ❶
0040101A        jnz     short loc_40102B ❷
0040101C        push    offset aXEqualsY_ ; "x equals y.\n"
00401021        call    printf
00401026        add     esp, 4
00401029        jmp     short loc_401038 ❸
0040102B loc_40102B:
0040102B        push    offset aXIsNotEqualToY ; "x is not equal to y.\n"
00401030        call    printf
```

如你在代码清单6-9中看到的，代码清单6-8中在if语句内部的代码在被执行之前必须做一个决定。这个决定对应于❷处显示的条件跳转（jnz）。要跳转的决定是基于一个比较（cmp）语句来做的，这个比较语句检查在❶处var_4是否等于var_8（var_4和var_8对应我们源代码中的x和y）。如果这两个值不相等，这个跳转就会发生，并且这段代码打印“x is not equal to y.”；否则，代码继续执行并打印“x equals y.”。

同时也注意跳过在❸处else段的代码跳转（jmp）。你识别出这两个代码路径中只有一条会被执行，这是识别if语句的关键点。

6.3.1 用IDA Pro图形化分析函数

IDA Pro有一个在识别结构方面很有用的图形化工具，如图6-1所示。这个特性是分析函数的默认视图。

图6-1显示了代码清单6-9中汇编代码例子的控制流图。如你所见，两个不同代码执行的路径（❶和❷）都到达函数的末尾，并且每一条路径打印一个不同的字符串。代码路径❶会打印“`x equals y.`”，而❷会打印“`x is not equal to y.`”。

IDA Pro在代码框底部的决策点处添加了`false`❶和`true`❷标签。如你能想象到的，图形化表示一个函数，能极大地加速逆向工程的进程。

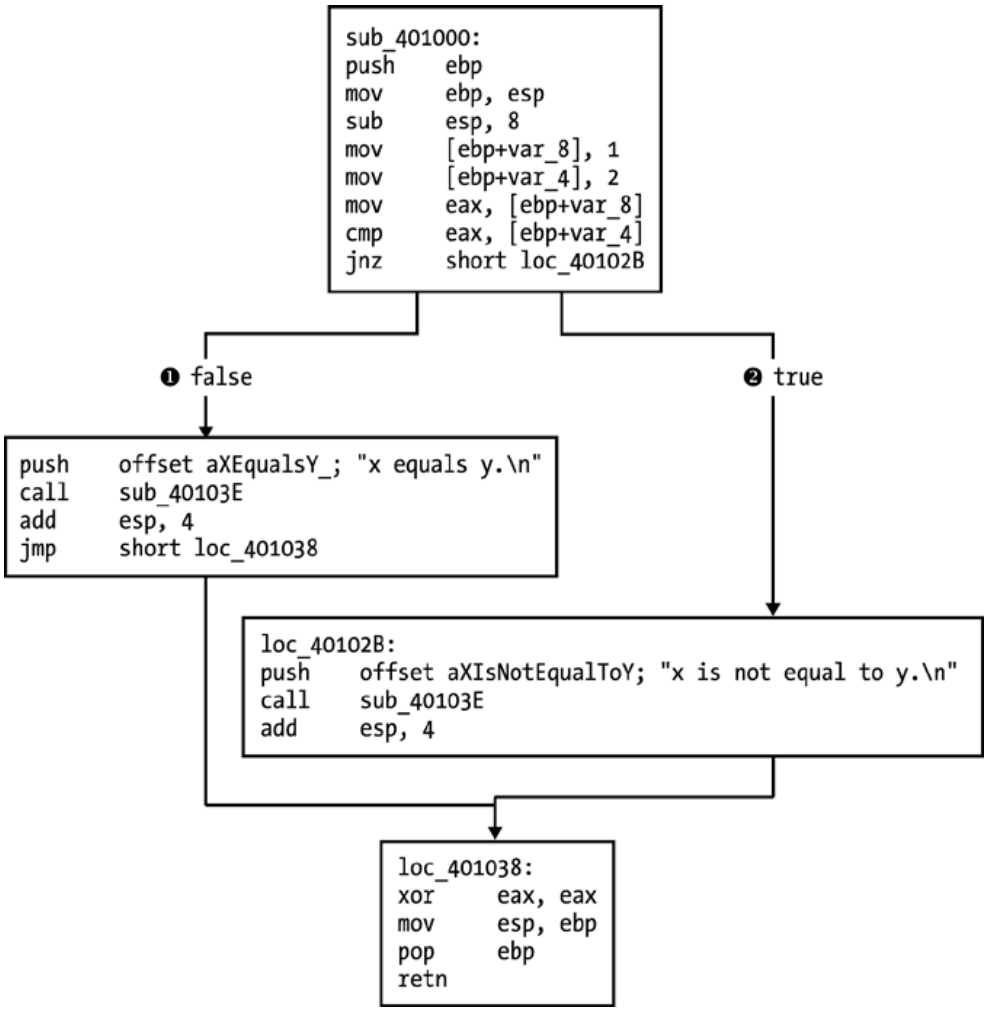

图6-1　在代码清单6-9中的if语句例子的反汇编图

6.3.2　识别嵌套的if语句

代码清单6-10显示了一个和代码清单6-8相似的嵌套`if`语句的C代码程序，除了两个附加的`if`语句被添加到原始的`if`语句中这点不太一样。这些附加语句通过测试，判断`z`是否等于0。

代码清单6-10　一个嵌套if语句的C代码

```
int x = 0;
int y = 1;
int z = 2;

if(x == y){
     if(z==0){
          printf("z is zero and x = y.\n");
     }else{
          printf("z is non-zero and x = y.\n");
     }
}else{
     if(z==0){
          printf("z zero and x != y.\n");
     }else{
          printf("z non-zero and x != y.\n");
     }
}
```

尽管C语言代码修改得很少，汇编代码却更加复杂，如代码清单6-11所示。

代码清单6-11　在代码清单6-10中的嵌套if语句例子的汇编代码

```
00401006        mov     [ebp+var_8], 0
0040100D        mov     [ebp+var_4], 1
00401014        mov     [ebp+var_C], 2
0040101B        mov     eax, [ebp+var_8]
0040101E        cmp     eax, [ebp+var_4]
00401021        jnz     short loc_401047 ❶
00401023        cmp     [ebp+var_C], 0
00401027        jnz     short loc_401038 ❷
00401029        push    offset aZIsZeroAndXY_ ; "z is zero and x = y.\n"
0040102E        call    printf
00401033        add     esp, 4
00401036        jmp     short loc_401045
00401038 loc_401038:
00401038        push    offset aZIsNonZeroAndX ; "z is non-zero and x = y.\n"
0040103D        call    printf
00401042        add     esp, 4
```

```
00401045 loc_401045:
00401045        jmp     short loc_401069
00401047 loc_401047:
00401047        cmp     [ebp+var_C], 0
0040104B        jnz     short loc_40105C ❸
0040104D        push    offset aZZeroAndXY_ ; "z zero and x != y.\n"
00401052        call    printf
00401057        add     esp, 4
0040105A        jmp     short loc_401069
0040105C loc_40105C:
0040105C        push    offset aZNonZeroAndXY_ ; "z non-zero and x != y.\n"
00401061        call    printf00401061
```

如你所见，3个不同的条件跳转发生了。第一个在❶处，如果var_4不等于var_8时发生。另外两个在❷和❸处，如果var_C不等于0时发生。

6.4 识别循环

循环与重复任务在所有软件中都很常见，你能够识别它们是很重要的。

6.4.1 找到for循环

for循环是一个C编程使用的基本循环机制。for循环总是有4个组件：初始化、比较、执行指令，以及递增或递减。

代码清单6-12显示了一个for循环的例子。

代码清单6-12 一个for循环的C代码

```
int i;

for(i=0; i<100; i++)
{
   printf("i equals %d\n", i);
}
```

在这个例子中，初始化设置i为0，并且这个比较检查i是否小于100。如果i小于100，printf指令会被执行，递增会给i加1，这个处理过程会检查i是否小于100。这些步骤会重复直到i大于或等于100。

在汇编代码中，for循环可以通过定位这4个组件识别出来——初始化、比较、执行指令以及递增/递减。例如，在代码清单6-13中，❶对应循环变量的初始化步骤，❸和❹之间的代码对应循环变量的递增，其最初会在❷处通过一个跳转指令而跳过。比较发生在❺处，循环决策在❻处被通过条件跳转指令而做出。如果跳转没有被执行，printf指令会被执行，并且通过❼处的一个无条件跳转，使

得循环变量进行递增。

代码清单6-13　代码清单6-12中for循环例子的汇编代码

```
00401004        mov     [ebp+var_4], 0 ❶
0040100B        jmp     short loc_401016 ❷
0040100D loc_40100D:
0040100D        mov     eax, [ebp+var_4] ❸
00401010        add     eax, 1
00401013        mov     [ebp+var_4], eax ❹
00401016 loc_401016:
00401016        cmp     [ebp+var_4], 64h ❺
0040101A        jge     short loc_40102F ❻
0040101C        mov     ecx, [ebp+var_4]
0040101F        push    ecx
00401020        push    offset aID  ; "i equals %d\n"
00401025        call    printf
0040102A        add     esp, 8
0040102D        jmp     short loc_40100D ❼
```

一个for循环可以使用IDA Pro的图形模式识别出来，如图6-2所示。

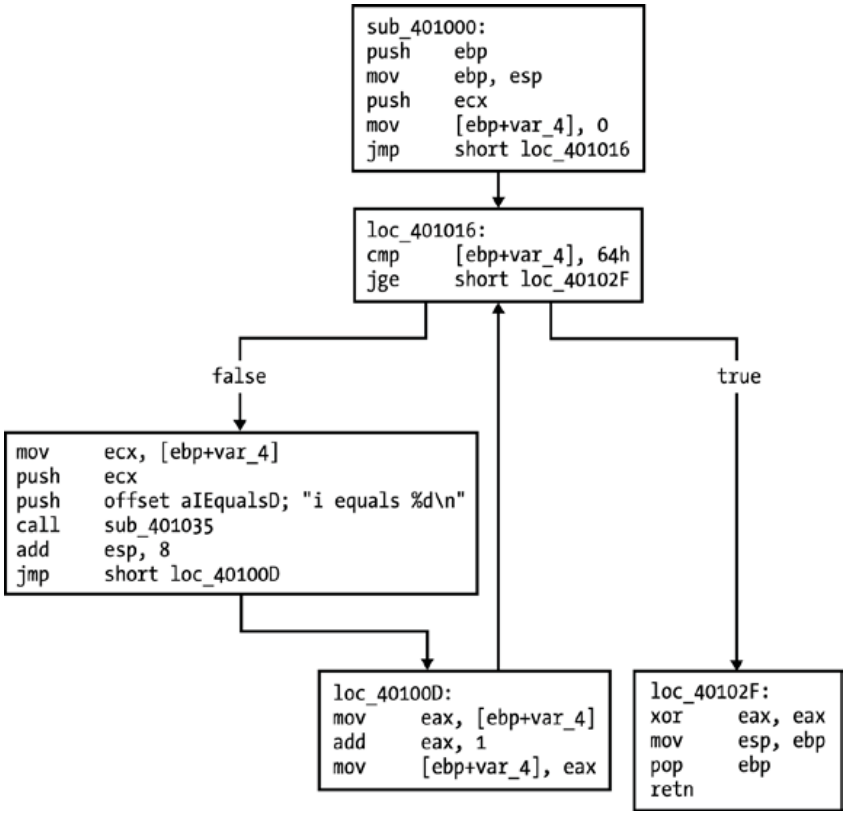

图6-2 代码清单6-13中for循环的反汇编图

在这个图示中，在递增代码之后向上指的箭头指示了一个循环结构。这些箭头使循环在图形视图中比在标准反汇编视图中更容易识别。这个图显示了5个框：顶部的4个是for循环的组件（初始化、比较、执行和循环递增）。右下部的框是函数结尾，我们在第4章中描述过，它作为一个函数的一段，负责清理栈并返回。

6.4.2 找到while循环

while循环频繁地被恶意代码作者使用，来通过循环知道一个条件是否被满足，比如接收到一个数据包或命令。while循环的汇编看起来和for循环类似，但是它们更容易理解。在代码清单6-14中的while循环会继续循环，直到从checkResult返回的状态是0。

代码清单6-14 一个while循环的C代码

```
int status=0;
int result = 0;

while(status == 0){
     result = performAction();
     status = checkResult(result);
}
```

在代码清单6-15中的汇编代码看来与for循环非常相似，唯一的区别在于它缺少一个递增。一个条件跳转发生在❶处，而一个无条件跳转发生在❷处，这段代码停止重复执行的唯一方式，就是那个期望发生的条件跳转。

代码清单6-15 代码清单6-14中while循环例子的汇编代码

```
00401036        mov     [ebp+var_4], 0
0040103D        mov     [ebp+var_8], 0
00401044 loc_401044:
00401044        cmp     [ebp+var_4], 0
00401048        jnz     short loc_401063 ❶
0040104A        call    performAction
0040104F        mov     [ebp+var_8], eax
00401052        mov     eax, [ebp+var_8]
00401055        push    eax
00401056        call    checkResult
0040105B        add     esp, 4
0040105E        mov     [ebp+var_4], eax
00401061        jmp     short loc_401044 ❷
```

6.5　理解函数调用约定

在第4章中，我们讨论了在进行函数调用时栈和call指令是如何使用的。函数调用在汇编代码中的表现可能不一样，而调用约定决定了函数调用发生的方式。这些约定包含了参数被放在栈上或寄存器中的次序，以及是由调用者还是被调函数（被调用者）负责在函数执行完成时清理栈。

调用约定的使用依赖于编译器，以及其他因素。通常编译器在如何实现这些约定方面，有一些细微差别。然而，当使用Windows API时你必须遵循特定约定，并且为了可兼容性方面的考虑，它们是由统一方式实现的（像在第7章讨论的）。

我们将使用代码清单6-16中的伪代码，来描述每一个调用约定。

代码清单6-16　一个函数调用的伪代码

```
int test(int x, int y, int z);
int a, b, c, ret;

ret = test(a, b, c);
```

你将遇到最常见的三个调用约定，分别是cdecl、stdcall以及fastcall。我们将在下面的章节中讨论它们的关键区别。

> **注意：同样的约定在编译器之间有实现上的差异，我们将注意力集中在使用它们的最常用方式上。**

6.5.1　cdecl

cdecl是最常用的约定之一，在第4章中我们介绍栈和函数调用时就描述过这种约定。在cdecl约定中，参数是从右到左按序被压入栈，当函数完成时由调用者清理栈，并且将返回值保存在EAX中。代码清单6-17显示了一个样例，这个例子展示了代码清单6-16中的程序被使用cdecl编译后的反汇编代码。

代码清单6-17　cdecl函数调用

```
push c
push b
push a
call test
add esp, 12
mov ret, eax
```

注意在加粗部分中，栈是由调用者清理的。在这个例子中，参数是按照从右到左的次序被压入

栈的，从c开始。

6.5.2 stdcall

stdcall约定除了被调用者在函数完成时清理栈之外，其他和cdecl非常类似。因此，如果stdcall约定被使用，在代码清单6-17中被加粗的add指令就不再需要了，因为被调函数将负责清理栈。

代码清单6-16中的test函数，在stdcall下编译出来就会不太一样，因为它必须关心栈的清理。它的结尾需要小心地进行清理。

stdcall是Windows API的标准调用约定。任何调用这些API的代码都不需要清理栈，因为清理栈的责任是由实现API函数代码的DLL程序所承担的。

6.5.3 fastcall

fastcall调用约定跨编译器时变化最多，但是它总体上在所有情况下的工作方式都是相似的。在fastcall中，前一些参数（典型的是前两个）被传到寄存器中，备用的寄存器是EDX和ECX（微软fastcall约定）。如果需要的话，剩下的参数再以从右到左的次序被加载到栈上。通常使用fastcall比其他约定更高效，因为代码不需要涉及过多的栈操作。

6.5.4 压栈与移动

除了到目前为止讨论的使用不同的调用约定外，编译器可能也会选择使用不同的指令来执行同样的操作，这通常发生在编译器决定移动而不是压栈的时候。代码清单6-18显示了一个函数调用的C代码例子。函数adder把两个参数相加并返回结果。main函数调用adder，并使用printf打印结果。

代码清单6-18　一个函数调用的C代码

```
int adder(int a, int b)
{
   return a+b;
}

void main()
{
   int x = 1;
   int y = 2;

   printf("the function returned the number %d\n", adder(x,y));
}
```

adder函数的汇编代码跨编译器编译后是一致的，如代码清单16-9中显示。如你所见，这段代码将arg_0和arg_4相加并把结果保存到EAX中。（如在第4章中讨论的，EAX保存返回值。）

代码清单6-19　在代码清单6-18中的adder函数的汇编代码

```
00401730        push    ebp
00401731        mov     ebp, esp
00401733        mov     eax, [ebp+arg_0]
00401736        add     eax, [ebp+arg_4]
00401739        pop     ebp
0040173A        retn
```

表6-1显示了由两种编译器使用的不同的调用约定：微软Visual Studio和GNU编译器集合（GCC）。在左侧，adder函数和printf的参数在调用前被压到栈上。而在右侧，参数在调用之前被移动到栈上。你应该为两种类型的调用约定做好准备，因为作为一名分析师，你对编译器并没有控制权。例如，左侧有一条指令不和右侧的任何指令对应。这个指令恢复栈指针，它在右侧并不是必需的，因为栈指针根本没有被修改。

> 注意：记住，即便使用的是同一个编译器，在调用约定方面也可能存在区别，这依赖于各种设置和选项。

表6-1　使用两种不同调用约定时一个函数调用的汇编代码

Visual Studio 版本			GCC 版本		
00401746	mov	[ebp+var_4], 1	00401085	mov	[ebp+var_4], 1
0040174D	mov	[ebp+var_8], 2	0040108C	mov	[ebp+var_8], 2
00401754	mov	eax, [ebp+var_8]	00401093	mov	eax, [ebp+var_8]
00401757	push	eax	00401096	mov	[esp+4], eax
00401758	mov	ecx, [ebp+var_4]	0040109A	mov	eax, [ebp+var_4]
0040175B	push	ecx	0040109D	mov	[esp], eax
0040175C	call	adder	004010A0	call	adder
00401761	**add**	**esp, 8**			
00401764	push	eax	004010A5	mov	[esp+4], eax
00401765	push	offset TheFunctionRet	004010A9	mov	[esp], offset TheFunctionRet
0040176A	call	ds:printf	004010B0	call	printf

6.6　分析switch语句

switch语句被程序员（和恶意代码作者）用来做一个基于字符或整数的决策。例如，后门通常使用单一的字节值从一系列动作中选择一个。switch语句通常以两种方式被编译：使用if样式或使用跳转表。

6.6.1 If样式

代码清单6-20显示了一个简单的switch语句，它使用变量i。依赖于i的值，和case值对应的代码将被执行。

代码清单6-20 一个有3个选项的switch语句C代码

```
switch(i)
{
   case 1:
      printf("i = %d", i+1);
      break;
   case 2:
      printf("i = %d", i+2);
      break;
   case 3:
      printf("i = %d", i+3);
      break;
   default:
      break;
}
```

这个switch语句已经被编译成在代码清单6-21中显示的汇编代码。它在❶和❷之间包含一系列条件跳转。条件跳转判定是通过在每一个跳转前面的比较直接进行的。

这个switch语句有3个选项，在❸、❹和❺处显示。由于在代码清单结尾处的无条件跳转，这些代码段相互之间是独立的。（你可能会发现，这个switch语句使用图6-3中显示的图形会更易于理解。）

代码清单6-21 在代码清单6-20中的switch语句例子的汇编代码

```
00401013        cmp     [ebp+var_8], 1
00401017        jz      short loc_401027 ❶
00401019        cmp     [ebp+var_8], 2
0040101D        jz      short loc_40103D
0040101F        cmp     [ebp+var_8], 3
00401023        jz      short loc_401053
00401025        jmp     short loc_401067 ❷
00401027 loc_401027:
00401027        mov     ecx, [ebp+var_4] ❸
0040102A        add     ecx, 1
0040102D        push    ecx
0040102E        push    offset unk_40C000 ; i = %d
00401033        call    printf
00401038        add     esp, 8
0040103B        jmp     short loc_401067
0040103D loc_40103D:
0040103D        mov     edx, [ebp+var_4] ❹
00401040        add     edx, 2
00401043        push    edx
00401044        push    offset unk_40C004 ; i = %d
00401049        call    printf
0040104E        add     esp, 8
00401051        jmp     short loc_401067
00401053 loc_401053:
00401053        mov     eax, [ebp+var_4] ❺
00401056        add     eax, 3
00401059        push    eax
0040105A        push    offset unk_40C008 ; i = %d
0040105F        call    printf
00401064        add     esp, 8
```

图6-3通过将代码分割成需要根据下一个决策才被执行的形式，将switch选项分开。在图中的3个框，被标记为❶、❷和❸，直接对应case语句的3个不同选项。注意所有这些框都在底部框终止，它是函数的末尾。你应该能够使用这个图看到，当`var_8`大于3时代码必须通过3个检查。

从这段反汇编代码，你很难知道原始代码是一个`switch`语句还是一个`if`语句序列，因为一个编译过的`switch`语句看起来就像一组`if`语句——两者都能包含一组`cmp`和`jcc`指令。当执行你的反汇编代码时，你并不是总能得到原始代码，因为可能有多种方式来代表同样的汇编语言结构的代码，所有这些都是有效并且相等的。

6.6.2　跳转表

下一个反汇编例子经常在庞大连续的`switch`语句中出现。编译器优化代码来避免需要做如此多

的比较。例如，若代码清单6-20中i的值是3，三个不同比较将会在第3个case被执行时发生。而在代码清单6-22中，我们添加一个case到代码清单6-20中（通过比较这个代码清单你可以看到），但是生成的汇编代码却大不一样了。

代码清单6-22　一个拥有4个选项的switch语句C代码

```
switch(i)
{
   case 1:
      printf("i = %d", i+1);
      break;
   case 2:
      printf("i = %d", i+2);
      break;
   case 3:
      printf("i = %d", i+3);
      break;
   case 4:
      printf("i = %d", i+3);
      break;
   default:
      break;
}
```

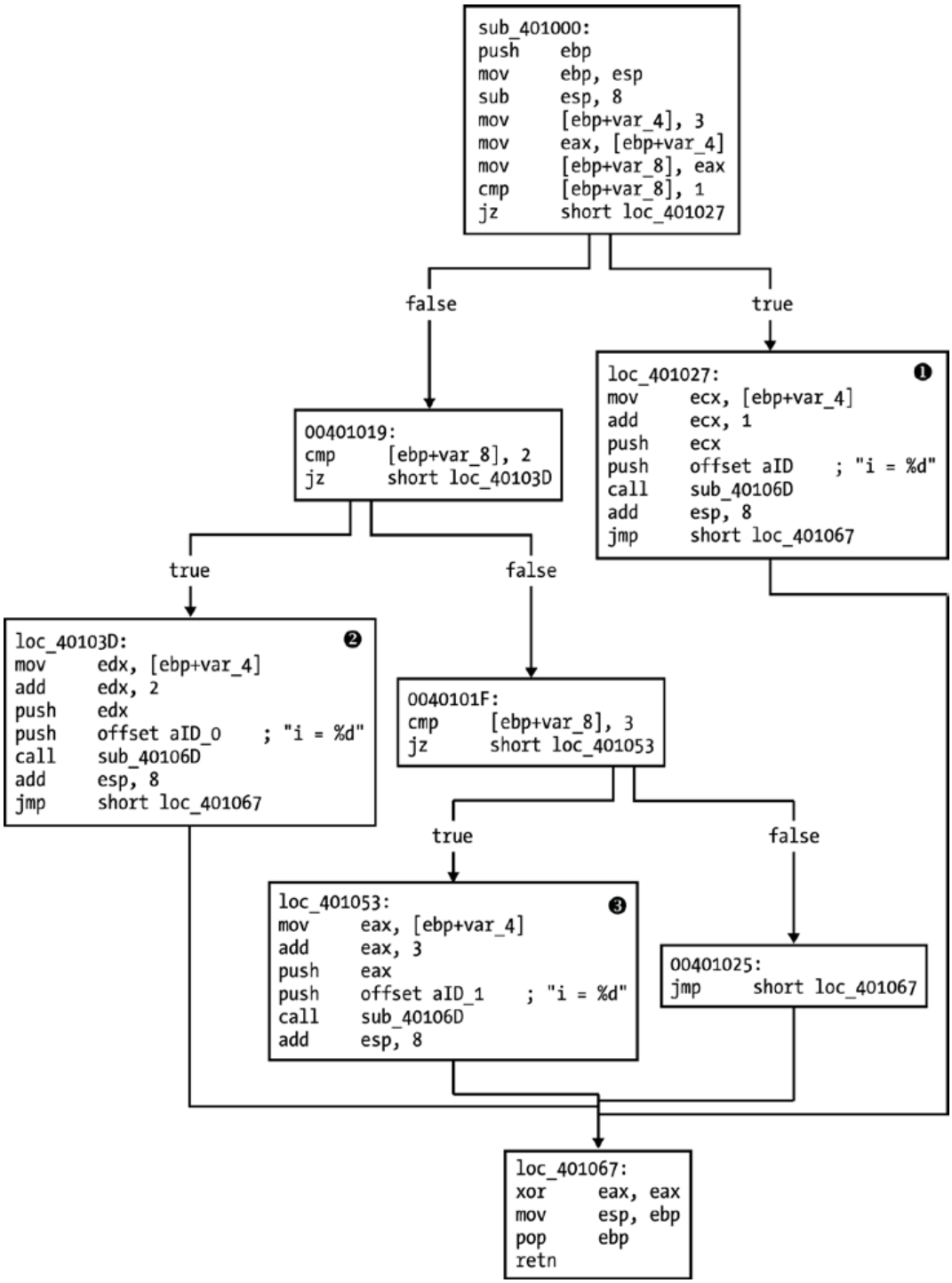

图6-3　代码清单6-21中if样式的switch语句例子的反汇编代码图

代码清单6-23中，使用了一个跳转表，来更加高效地运行汇编代码，显示在❷处，它定义了一些附加内存位置的偏移。这个switch变量被用来作为这个跳转表的一个索引。

在这个例子中，ecx包含这个switch变量，并且它在第一行被减1。在C代码中，switch表的范围从1到4，而汇编代码必须调整它从0到3，这样这个跳转表可以被合理地添加索引。在❶处的跳转指令是基于跳转表目标的位置。

在这个跳转指令中，edx被乘以4并且加上跳转表的基址（0x401088），来确定要跳转到哪个case代码块。它被乘以4是因为跳转表中的每一项是一个4字节大小的地址。

代码清单6-23　代码清单6-22中switch语句例子的汇编代码

```
00401016        sub     ecx, 1
00401019        mov     [ebp+var_8], ecx
0040101C        cmp     [ebp+var_8], 3
00401020        ja      short loc_401082
00401022        mov     edx, [ebp+var_8]
00401025        jmp     ds:off_401088[edx*4] ❶
0040102C    loc_40102C:
              ...
00401040        jmp     short loc_401082
00401042    loc_401042:
              ...
00401056        jmp     short loc_401082
00401058    loc_401058:
              ...
0040106C        jmp     short loc_401082
0040106E    loc_40106E:
              ...
00401082    loc_401082:
00401082        xor     eax, eax
00401084        mov     esp, ebp
00401086        pop     ebp
00401087        retn
00401087    _main   endp
00401088 ❷off_401088  dd offset loc_40102C
0040108C              dd offset loc_401042
00401090              dd offset loc_401058
00401094              dd offset loc_40106E
```

在图6-4中，这种类型switch语句的图形表示，比起标准的反汇编视图更加清晰。

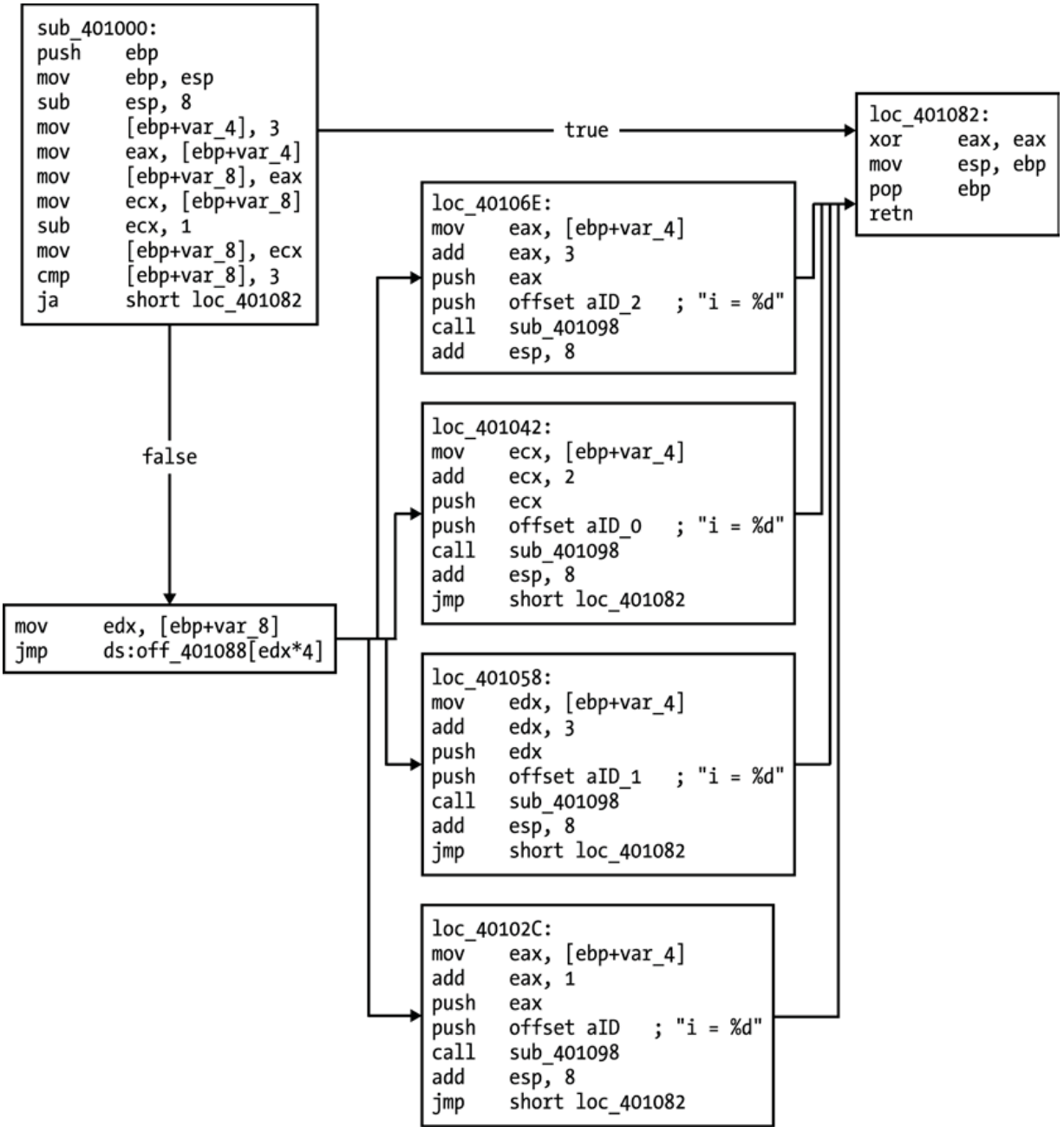

图6-4　跳转表switch语句例子的反汇编图

如你所见，四个case选项的每一个都被很清晰地分开成汇编代码块。这些代码块逐一排列在跳转表后的一列上，由跳转表决定使用哪一个代码块。注意所有这些代码框和初始框都在最右边的代码框中终止，而它就是这个函数的末尾。

6.7 反汇编数组

数组被程序员们用来定义一个相似数据项的有序集合。比如，恶意代码有时会使用一个指向字符串的指针数组，其中包含多个主机名，作为连接的选项。

代码清单6-24中显示了被一个程序使用的两个数组，两个数组都通过for循环进行迭代式设置。数组a是局部定义的，而数组b是全局定义的。这些定义将影响汇编代码。

代码清单6-24 一个数组的C代码例子

```
int b[5] = {123,87,487,7,978};
void main()
{
   int i;
   int a[5];

   for(i = 0; i<5; i++)
   {
      a[i] = i;
      b[i] = i;
   }
}
```

在汇编代码中，数组是通过使用一个基地址作为起始点来进行访问的。每一个元素的大小通常并不总是明显的，但是它可以通过看这个数组是如何被索引的来进行判断。代码清单6-25显示了代码清单6-24中C程序的汇编代码。

代码清单6-25 代码清单6-24中数组的汇编代码

```
00401006        mov     [ebp+var_18], 0
0040100D        jmp     short loc_401018
0040100F loc_40100F:
0040100F        mov     eax, [ebp+var_18]
00401012        add     eax, 1
00401015        mov     [ebp+var_18], eax
00401018 loc_401018:
00401018        cmp     [ebp+var_18], 5
0040101C        jge     short loc_401037
0040101E        mov     ecx, [ebp+var_18]
00401021        mov     edx, [ebp+var_18]
00401024        mov     [ebp+ecx*4+var_14], edx ❶
00401028        mov     eax, [ebp+var_18]
0040102B        mov     ecx, [ebp+var_18]
0040102E        mov     dword_40A000[ecx*4], eax ❷
00401035        jmp     short loc_40100F
```

在这个代码清单中，数组b的基地址对应dword_40A000，而数组a的基地址对应var_14。因为这两个都是整数的数组，每一个元素的大小是4，尽管访问两个数组的指令各不相同（在❶和❷处）。在两种情况下，ecx被作为索引使用，它被乘以4，来指明元素的大小，结果值与数组的基地址相加，来访问正确的数组元素。

6.8　识别结构体

结构体（简称结构）和数组相似，但是它们包括不同类型的元素。结构体通常被恶意代码作者用来组织信息。有时使用结构体比独立维护很多不同的变量要更简单，尤其是如果很多函数需要对同一组变量进行访问时。（Windows API函数通常使用必须由调用程序创建和维护的结构体。）

在代码清单6-26中，我们在❶处定义由一个整数数组、一个字符和一个双精度浮点数组成的结构体。在main中，我们为这个结构体分配内存，并将这个结构传递给test函数。在❷处被定义的结构gms则是一个全局变量。

代码清单6-26　一个结构例子的C代码

```
struct my_structure { ❶
     int x[5];
     char y;
     double z;
};

struct my_structure *gms; ❷

void test(struct my_structure *q)
{
     int i;
     q->y = 'a';
     q->z = 15.6;
     for(i = 0; i<5; i++){
          q->x[i] = i;
     }
}

void main()
{
     gms = (struct my_structure *) malloc(
     sizeof(struct my_structure));
     test(gms);
}
```

结构体（类似数组）通过一个作为起始指针的基地址来访问。要判定附近的数据字节类型是同

一个结构的组成部分，还是只是凑巧相互挨着是比较困难的，这依赖于这个结构体的上下文，你识别一个结构体的能力，对你的恶意代码分析产生很大影响。

代码清单6-27显示了来自代码清单6-26中被反汇编的main函数。因为结构gms是一个全局变量，它的基地址在代码清单6-27中显示的内存位置dword_40EA30。这个结构体的基地址被通过在❶处的push eax指令，传递给sub_401000（test）函数。

代码清单6-27　在代码清单6-26结构体例子中的main函数汇编代码

```
00401050        push    ebp
00401051        mov     ebp, esp
00401053        push    20h
00401055        call    malloc
0040105A        add     esp, 4
0040105D        mov     dword_40EA30, eax
00401062        mov     eax, dword_40EA30
00401067        push    eax ❶
00401068        call    sub_401000
0040106D        add     esp, 4
00401070        xor     eax, eax
00401072        pop     ebp
00401073        retn
```

代码清单6-28显示了在代码清单6-26中被显示的test函数反汇编代码。arg_0是这个结构体的基地址。偏移0x14保存了结构中的字符，并且0x61对应ASCII中的字母a。

代码清单6-28　代码清单6-26结构体例子中test函数的汇编代码

```
00401000        push    ebp
00401001        mov     ebp, esp
00401003        push    ecx
00401004        mov     eax,[ebp+arg_0]
00401007        mov     byte ptr [eax+14h], 61h
0040100B        mov     ecx, [ebp+arg_0]
0040100E        fld     ds:dbl_40B120 ❶
00401014        fstp    qword ptr [ecx+18h]
00401017        mov     [ebp+var_4], 0
0040101E        jmp     short loc_401029
00401020 loc_401020:
00401020        mov     edx,[ebp+var_4]
00401023        add     edx, 1
00401026        mov     [ebp+var_4], edx
00401029 loc_401029:
00401029        cmp     [ebp+var_4], 5
0040102D        jge     short loc_40103D
0040102F        mov     eax,[ebp+var_4]
00401032        mov     ecx,[ebp+arg_0]
```

```
00401035        mov     edx,[ebp+var_4]
00401038        mov     [ecx+eax*4],edx ❷
0040103B        jmp     short loc_401020
0040103D loc_40103D:
0040103D        mov     esp, ebp
0040103F        pop     ebp
00401040        retn
```

我们能识别出偏移0x18是一个double变量，因为它被作为在❶处的一条浮点指令的参数被使用。我们也可以识别出一些整数数组元素被移动到偏移0，4，8，0xC以及0x10处，这是通过验证for循环以及这些偏移在❷处被访问得到的。我们可以从这个分析中推断出这个结构体的内容。

在IDA Pro中，你可以创建结构体，并使用热键T来将它们赋给内存引用。这样做会修改指令mov [eax+14h], 61h为mov [eax + my_structure.y], 61h。后者更容易阅读，而标识结构体通常能帮助你更快地理解反汇编，尤其是在你时刻查看被使用结构体的场景中。在这个例子中要想高效地使用热键T，你需要使用IDA Pro的结构体窗口，手动地创建my_structure结构体。这是一个乏味的过程，但是它可以对你频繁遇到的结构体有帮助。

6.9　分析链表遍历

一个链表是包含一个数据记录序列的数据结构，并且每一个记录都包括一个对序列中下一记录的引用（链接）域。使用一个链表，而不是一个数组，原则上的好处是被链接项的访问次序与数据项被保存在内存或磁盘上的次序可以不一样。

代码清单6-29显示了一个链表遍历的C代码例子。链表包含一系列名为pnode的节点结构体，并且它由两个循环来操作。在❶处的第一个循环创建10个节点，并用数据填充它们。第二个在❷处的循环迭代所有记录，并打印它们的内容。

代码清单6-29　一个链表遍历的C代码

```
struct node
{
   int x;
   struct node * next;
};

typedef struct node pnode;

void main()
{
   pnode * curr, * head;
   int i;
```

```
   head = NULL;

   for(i=1;i<=10;i++) ❶
   {
      curr = (pnode *)malloc(sizeof(pnode));
      curr->x = i;
      curr->next  = head;
      head = curr;
   }

   curr = head;

   while(curr) ❷
   {
      printf("%d\n", curr->x);
      curr = curr->next ;
   }
}
```

理解这段反汇编代码的最好方法，是识别在main函数中的两个代码结构。这当然也是本章的关键内容：你如果能识别这些结构，将会使分析更容易。

在代码清单6-30中，我们首先标识出for循环。var_C对应i，它是这个循环的计数。var_8对应head变量，而var_4对应curr变量。var_4是一个指向拥有两个被赋值变量结构体的指针（在❶和❷处显示）。

这个while循环（❸到❺）执行这个链表的迭代。在这个循环中，var_4被设置为这个链表中在❹处的下一记录。

代码清单6-30　代码清单6-29中链表遍历例子的汇编代码

```
0040106A        mov     [ebp+var_8], 0
00401071        mov     [ebp+var_C], 1
00401078
00401078 loc_401078:
00401078        cmp     [ebp+var_C], 0Ah
0040107C        jg      short loc_4010AB
0040107E        mov     [esp+18h+var_18], 8
00401085        call    malloc
0040108A        mov     [ebp+var_4], eax
0040108D        mov     edx, [ebp+var_4]
00401090        mov     eax, [ebp+var_C]
00401093        mov     [edx], eax ❶
00401095        mov     edx, [ebp+var_4]
00401098        mov     eax, [ebp+var_8]
0040109B        mov     [edx+4], eax ❷
0040109E        mov     eax, [ebp+var_4]
```

```
004010A1        mov     [ebp+var_8], eax
004010A4        lea     eax, [ebp+var_C]
004010A7        inc     dword ptr [eax]
004010A9        jmp     short loc_401078
004010AB loc_4010AB:
004010AB        mov     eax, [ebp+var_8]
004010AE        mov     [ebp+var_4], eax
004010B1
004010B1 loc_4010B1:
004010B1        cmp     [ebp+var_4], 0 ❸
004010B5        jz      short locret_4010D7
004010B7        mov     eax, [ebp+var_4]
004010BA        mov     eax, [eax]
004010BC        mov     [esp+18h+var_14], eax
004010C0        mov     [esp+18h+var_18], offset aD ; "%d\n"
004010C7        call    printf
004010CC        mov     eax, [ebp+var_4]
004010CF        mov     eax, [eax+4]
004010D2        mov     [ebp+var_4], eax ❹
004010D5        jmp     short loc_4010B1 ❺
```

要识别一个链表，你必须首先识别出一些包含指针的对象，而这些指针是指向同一类型的对象。对象的递归本性是使它形成链的原因，并且这就是你需要从反汇编代码中识别的关键。

在这个例子中，意识到在❹处，var_4被赋予eax，它来自[eax+4]，[eax+4]它自己来自var_4的前一个赋值。这就是说无论结构var_4是什么，它必须在其中包含一个4字节指针。这个指针指向另一个结构，并且那个结构也必须包含一个指向另一结构的4字节指针，如此往复。

6.10　小结

本章为你展示了一个在恶意代码分析中永恒不变的任务：从细节中进行抽象。不要在底层细节中停滞，而是应建立起能够识别代码在高层次上做什么的能力。

本章已经给你显示了每一个主要C编码结构的C代码和汇编代码，来帮助你快速识别在分析中遇到的最常见结构。本章也提供了几个例子，来显示编译器决定在什么地方做一些不同的事情，主要在结构和（当使用完全不同的编译器时）函数调用的例子中。建立这种洞察力，会帮助你走向在实战中遇到新结构时能够快速识别它们的理想状态。

6.11　实验

本章实验的目标是帮助你通过分析代码结构来理解一个程序的总体功能。每一个实验将指导你

发现与分析一个新的代码结构。每一个实验在前一个基础上构建，因此通过四种结构创建了一个复杂的恶意代码片段。一旦你完成了这个实验所需的工作，你应该能够当在恶意代码中遇到它们时，更容易地识别这些单独的结构。

Lab 6-1

在这个实验中，你将分析在文件Lab06-01.exe中发现的恶意代码。

问题

1. 由 `main` 函数调用的唯一子过程中发现的主要代码结构是什么？
2. 位于 0x40105F 的子过程是什么？
3. 这个程序的目的是什么？

Lab 6-2

分析在文件Lab06-02.exe中发现的恶意代码。

问题

1. `main` 函数调用的第一个子过程执行了什么操作？
2. 位于 0x40117F 的子过程是什么？
3. 被 `main` 函数调用的第二个子过程做了什么？
4. 在这个子过程中使用了什么类型的代码结构？
5. 在这个程序中有任何基于网络的指示吗？
6. 这个恶意代码的目的是什么？

Lab 6-3

在这个实验中，我们会分析在文件Lab06-03.exe中发现的恶意代码。

问题

1. 比较在 `main` 函数与实验 6-2 的 `main` 函数的调用。从 `main` 中调用的新的函数是什么？
2. 这个新的函数使用的参数是什么？

3. 这个函数包含的主要代码结构是什么？
4. 这个函数能够做什么？
5. 在这个恶意代码中有什么本地特征吗？
6. 这个恶意代码的目的是什么？

Lab 6-4

在这个实验中，我们会分析在文件Lab06-04.exe中发现的恶意代码。

问题

1. 在实验 6-3 和 6-4 的 `main` 函数中的调用之间的区别是什么？
2. 什么新的代码结构已经被添加到 `main` 中？
3. 这个实验的解析 HTML 的函数和前面实验中的那些有什么区别？
4. 这个程序会运行多久？（假设它已经连接到互联网。）
5. 在这个恶意代码中有什么新的基于网络的迹象吗？
6. 这个恶意代码的目的是什么？

第 6 篇

高级专题

第 19 章　shellcode 分析

shellcode是指一个原始可执行代码的有效载荷。shellcode这个名字来源于攻击者通常会使用这段代码来获得被攻陷系统上交互式shell的访问权限。然而，时过境迁，现在这个术语通常被用于描述一段自包含的可执行代码。

shellcode经常与漏洞利用并肩使用，来颠覆一个运行的程序，或是被恶意代码用于执行进程注入。漏洞利用与进程注入，在机制上与shellcode被植入到一个运行的程序并在进程启动后执行是相似的。

shellcode的编写者需要手动执行几个软件开发者从未意识到的动作。例如，shellcode代码不能依赖于Windows加载器在普通程序启动时所执行的如下动作：

- 将程序布置在它首选的内存位置。
- 如果它不能被加载到首选的内存位置，那么需要应用地址重定向。
- 加载需要的库，并解决外部依赖。
- 本章会向你介绍这些shellcode技术，并使用现实世界中的完整例子演示。

19.1　加载shellcode进行分析

在一个调试器中加载和运行shellcode是有问题的，因为shellcode通常只是一个二进制数据块，它不能像一个普通可执行文件一样运行。为了使事情更简单一些，我们将使用*shellcode_launcher.exe*（包括实验样本都可以在*http://www.practicalmalwareanalysis.com/*中获得），来加载并跳转到shellcode片段。

正如本书第5章讨论过的那样，加载shellcode到IDA Pro中进行静态分析是相对简单的，但用户必须在加载过程中提供输入，因为没有可执行文件格式来描述shellcode的内容。首先，你必须在加载进程对话框中选择正确的处理器类型。对于本章中的样本，你可以在提示时选择Intel 80x86 processors:metapc处理器类型，并选择32-bit disassembly。IDA Pro加载这段二进制代码，但不会执行自动分析过程（分析必须手动完成）。

19.2 位置无关代码

位置无关代码（PIC）是指不使用硬编码地址来寻址指令或数据的代码。shellcode就是位置无关代码。它不能假设自己在执行时会被加载到一个特定的内存位置，因为在运行时，一个脆弱程序的不同版本可能加载shellcode到不同内存位置。shellcode必须确定所有对代码和数据的内存访问都使用PIC技术。

表19-1为几种常见的x86代码与数据访问类型，以及它们是否是PIC代码。

表19-1 不同类型的x86代码和数据访问

Instruction mnemonics		Instruction bytes	Position-independent?
call	sub_401000	E8 C1 FF FF FF ❶	Yes
jnz	short loc_401044	75 0E ❷	Yes
mov	edx, dword_407030 ❸	8B 15 30 70 40 00	No
mov	eax, [ebp-4] ❹	8B 45 FC	Yes

在这个表中，`call`指令包含一个32位有符号相对位移值，通过将它和紧跟它后面的地址相加，计算出这个目标位置。因为表中显示的`call`指令位于0x0040103A，加上❶处偏移值0xFFFFFFC1到这个指令的位置，再加上`call`指令的大小（5字节），结果调用目标地址0x00401000。

这个表中的`jnz`指令除了仅使用一个8位的有符号相对位移值外和`call`非常类似。`jnz`指令位于0x0040103A处。将在指令中❷处（`0xe`）保存的偏移值与该指令大小（2字节）加到一起，产生的结果是跳转至目标地址0x00401044。

如你所见，像`call`和`jump`这样的控制流指令已经与位置无关了。这些指令的目标地址是通过将EIP寄存器指定的当前位置加上一个保存在指令中的相对偏移值来进行计算的。（某些形式的`call`和`jump`允许程序员使用绝对的或者非相对的不是位置无关的寻址方式。

❸处的`mov`指令显示了一个访问全局数据变量`dword_407030`的指令。这个指令中的后4个字节显示内存位置0x00407030。这个特别的指令并不是位置无关的，所以shellcode编写者必须避免使用它。

用❸处的`mov`指令与❹处的`mov`指令比较，它访问一个来自栈上的`DWORD`。这个指令使用EBP寄存器作为一个基值，并且包含一个有符号的相对偏移值：`0xFC(-4)`。这种类型的数据访问是位置无关的，并且是shellcode编写者必须对所有数据访问使用的模型：计算一个运行时地址，并仅通过使用相对这个位置的偏移值来引用数据。（在本书第19.3节中将讨论如何找到一个合适的运行时地址。）

19.3 识别执行位置

Shellcode在以位置无关的方式访问数据时，需要解引用一个基址指针。用这个基址指针加上或减去偏移值，将使它安全访问shellcode中包含的数据。因为x86指令集不提供相对EIP的数据访问寻址，而仅对控制流指令提供EIP相对寻址，所以，一个通用寄存器必须首先载入当前指令指针值，作为基址指针来使用。

获取当前指令指针值可能并不那么简单便捷，因为在x86系统上的指令指针不能被软件直接访问。事实上，没法汇编这条`mov eax, eip`指令，直接向一个通用寄存器中载入当前指令指针。然而，shellcode使用两种普遍的技术解决这个问题：`call/pop`指令和`fnstenv`指令。

19.3.1 使用call/pop指令

当一个`call`指令被执行时，处理器将`call`后面的指令的地址压到栈上，然后转到被请求的位置进行执行。这个函数执行完后，会执行一个`ret`指令，将返回地址弹出到栈的顶部，并将它载入指令指针寄存器中。这样做的结果是执行刚好返回到`call`后面的指令。

shellcode可以通过在一个`call`指令后面立刻执行`pop`指令滥用这种通常约定，这会将紧跟`call`后面的地址载入指定寄存器中。代码清单19-1为一个使用这种技术的简单Hello World程序。

代码清单19-1 `call/pop`技术入门示例程序

```
Bytes           Disassembly
83 EC 20        sub     esp, 20h
31 D2           xor     edx, edx
E8 0D 00 00 00  call    sub_17 ❶
48 65 6C 6C 6F  db 'Hello World!',0 ❷
20 57 6F 72 6C
64 21 00

sub_17:
5F              pop     edi ❸           ; edi gets string pointer
52              push    edx             ; uType: MB_OK
57              push    edi             ; lpCaption
57              push    edi             ; lpText
52              push    edx             ; hWnd: NULL
B8 EA 07 45 7E  mov     eax, 7E4507EAh  ; MessageBoxA
FF D0           call    eax ❹
52              push    edx             ; uExitCode
B8 FA CA 81 7C  mov     eax, 7C81CAFAh  ; ExitProcess
FF D0           call    eax ❺
```

❶处的`call`指令将控制转移到❸处的`sub_17`函数处。这是一个PIC指令，因为`call`指令使用一个相对EIP的值（`0x0000000D`）来计算`call`的目标。❸处的`pop`指令将保存在栈顶的地址载入EDI中。

请记住，这个由call指令保存的EIP值指向紧跟这个call的位置，所以在pop指令执行后，EDI将包含一个指向❷处声明的db的指针。这个db声明是一段汇编语言语法，创建一个字节序列，用来拼写出字符串Hello World!。在❸处的pop执行后，EDI将指向这个Hello World!字符串。

这种混合代码和数据的方法对shellcode来说很普遍，但是它能很容易地使那些试图将call指令后面的数据作为代码进行解析的反汇编器困惑，结果要么是解析出来的反汇编代码很混乱，要么是遇到无效的opcode组合时停止反汇编过程。如在第15章中所见，使用call/pop指令组合获取指向数据的指针，可能被作为一种额外的对抗逆向工程技术应用于大型程序中。

剩下的代码调用❹处的MessageBoxA来显示这个“Hello World!”消息，然后调用❺处的ExitProcess完全退出。这个样本对两个函数调用都使用了硬编码的位置，因为在shellcode中的导入函数无法被加载器自动解析，但硬编码的位置使这段代码很脆弱。（这些地址来自于一个Windows XP SP3机器，并且可能和你的并不一样。）

要通过OllyDbg找到这些函数地址，需要打开任意一个进程，并按Ctrl+G组合键，弹出Enter Expression to Follow对话框。在对话框中输入MessageBoxA，然后按回车键。只要被调试的进程加载了导出这个函数的库（*user32.dll*），调试器就应该显示这个函数的位置。

要使用*shellcode_launcher.exe*加载，并单步执行这个例子程序，应在命令行输入下面内容：

```
shellcode_launcher.exe -i helloworld.bin -bp -L user32
```

需要-L user32选项，因为shellcode不调用LoadLibraryA，所以*shellcode_ launcher.exe*必须确认这个库被加载。-bp选项恰好在跳转到由-i选项指定shellcode二进制之前插入一个断点指令。回忆一下，调试器可以被注册成即时调试器，并可以在一个程序遇到一个断点时自动启动（或弹出）。如果类似OllyDbg的调试器已经被注册成一个即时调试器，它会打开并附加到那个遇到断点的进程。这种方法允许你跳过*shellcode_launcher.exe*程序的内容，并从shellcode二进制开始执行。

你可以通过选择Options→Just-in-time Debugging→Make OllyDbg Just-in-time Debugger选项，设置OllyDbg作为你的即时调试器。

> 提示：那些希望执行这个例子的读者可能需要修改硬编码的 MessageBoxA 和 ExitProcess 的函数位置。它们的地址可以通过文本中描述的方法找到。找到这些地址后，你就可以在 OllyDbg 中，找到加载硬编码的函数位置到 EAX 寄存器中的指令，通过将光标放在这个指令上，然后按空格键，来给 helloworld.bin 打补丁。这会使 OllyDbg 弹出 Assemble At 对话框，这个对话框允许你输入自己的汇编代码。你输入的代码将被 OllyDbg 汇编并覆盖当前的指令。简单地用你机器上的正确值来替换 7E4507EAh 值，并且 OllyDbg 会在内存中给这个程序打补丁，允许 shellcode 正确执行。

19.3.2 使用fnstenv指令

x87浮点单元（FPU）在普通x86架构中提供了一个隔离的执行环境。它包含一个单独的专用寄存器集合，当一个进程正在使用FPU执行浮点运算时，这些寄存器需要由操作系统在上下文切换时保存。代码清单19-2为被`fstenv`指令与`fnstenv`指令使用的28字节结构体，这个结构体在32位保护模式中执行时被用来保存FPU状态到内存中。

代码清单19-2 FpuSaveState结构体的定义

```
struct FpuSaveState {
    uint32_t    control_word;
    uint32_t    status_word;
    uint32_t    tag_word;
    uint32_t    fpu_instruction_pointer;
    uint16_t    fpu_instruction_selector;
    uint16_t    fpu_opcode;
    uint32_t    fpu_operand_pointer;
    uint16_t    fpu_operand_selector;
    uint16_t    reserved;
};
```

这里唯一影响使用的域是在字节偏移量12处的`fpu_instruction_pointer`。它将保留被FPU使用的最后一条CPU指令的地址，并为异常处理器标识哪条FPU指令可能导致错误上下文信息。需要这个域是因为FPU是与CPU并行运行的。如果FPU产生了一个异常，异常处理器不能简单地通过参照中断返回地址来标识导致这个错误的指令。

代码清单19-3为另外一个使用`fnstenv`获取EIP的Hello World程序反汇编代码。

代码清单19-3 使用fnstenv的Hello World例子

```
Bytes           Disassembly
83 EC 20        sub     esp, 20h
31 D2           xor     edx, edx
EB 15           jmp     short loc_1C
EA 07 45 7E     dd 7E4507EAh            ; MessageBoxA
FA CA 81 7C     dd 7C81CAFAh            ; ExitProcess
48 65 6C 6C 6F  db 'Hello World!',0
20 57 6F 72 6C
64 21 00

loc_1C:

D9 EE           fldz ❶
D9 74 24 F4     fnstenv byte ptr [esp-0Ch] ❷
5B              pop     ebx ❸           ; ebx points to fldz
```

```
8D 7B F3        lea     edi, [ebx-0Dh] ❹  ; load HelloWorld pointer
52              push    edx               ; uType: MB_OK
57              push    edi               ; lpCaption
57              push    edi               ; lpText
52              push    edx               ; hWnd: NULL
8B 43 EB        mov     eax, [ebx-15h] ❺  ; load MessageBoxA
FF D0           call    eax               ; call MessageBoxA
52              push    edx               ; uExitCode
8B 43 EF        mov     eax, [ebx-11h] ❻  ; load ExitProcess
FF D0           call    eax               ; call ExitProcess
```

❶处的fldz指令将浮点数0.0压到FPU栈上。fpu_instruction_pointer的值在FPU中被更新成指向fldz指令。

执行在❷处的fnstenv指令，将FpuSaveState结构体保存到栈上的[esp-0ch]处，这允许shellcode在❸处执行一个pop，将fpu_instruction_pointer的值载入EBX中。一旦这个pop执行，EBX会包含一个值，这个值指向这个内存中fldz指令的位置。然后shellcode开始使用EBX作为一个基址寄存器访问嵌入到代码中的数据。

与使用call/pop技术的Hello World例子一样，这段代码使用硬编码的位置调用MessageBoxA和ExitProcess函数，但在这里，这些函数的位置与ASCII字符串一起作为数据保存。❹处的lea指令将Hello World!字符串的地址载入，这个地址是从保存在EBX中的fldz指令地址减去0x0d偏移值得到的。❺处的mov指令载入MessageBoxA函数的位置，❻处的mov指令载入ExitProcess函数的位置。

> **提示：代码清单 19-3 是一个人为设计的例子，但它的 shellcode 保存或创建函数指针数组机制很常见。我们在这个例子中使用了 fldz 指令，但是任何非控制类的 FPU 指令都可以被使用。**

可以使用shellcode_launcher.exe通过下面的命令来执行这个例子。

```
shellcode_launcher.exe -i hellofstenv.bin -bp -L user32
```

19.4 手动符号解析

shellcode作为一个获得执行的二进制代码块存在。一旦它获得执行，它必须做些有用的事情，这通常意味着通过API与系统进行交互。

记住，shellcode不能使用Windows加载器来确保所有需要的库被加载并可用，同时也不能确认所有的外部符号依赖都被解决。相反，它必须自己找到这个符号。前面的例子中shellcode使用硬编码地址来找符号，这是非常脆弱的方法，因为它只在操作系统的一个特殊版本与补丁包中工作。shellcode必须动态定位这些函数，以确保它们在不同环境中都能可靠地工作。为了完成这个任务，

shellcode经常使用LoadLibraryA和GetProcAddress函数。

LoadLibraryA加载指定的库，并返回一个句柄。GetProcAddress函数在库的导出表中查找给定的符号名或序号。如果shellcode有这两个函数的访问权限，它可以加载任何库到系统中并找到导出符号，这时它就可以完整地访问API了。

两个函数都是从*kernel32.dll*中导出的，所以shellcode必须做如下事情：

- 在内存中找到*kernel32.dll*。
- 解析*kernel32.dll*的PE文件，并搜索导出函数LoadLibraryA和GetProcAddress。

19.4.1 在内存中找到kernel32.dll

为了定位*kernel32.dll*，我们会跟踪一系列未文档化的Windows结构体。这些结构体中的一个包含着*kernel32.dll*的加载地址。

> **提示：大多数 Windows 结构体都在微软开发者网络（MSDN）上列出，但它们没有完全被文档化。其中的许多结构体包含名为 Reserved 的字节数组，并且还有一个警告“在 Windows 将来版本中这个结构可能被修改。”要想查看一个完整的结构体清单，请参考 *http://undocumented.ntinternals.net/*。**

要找到*kernel32.dll*的基地址，我们需要跟踪图19-1所示的一些数据结构（在每一个结构体中只显示了相关域与偏移值）。

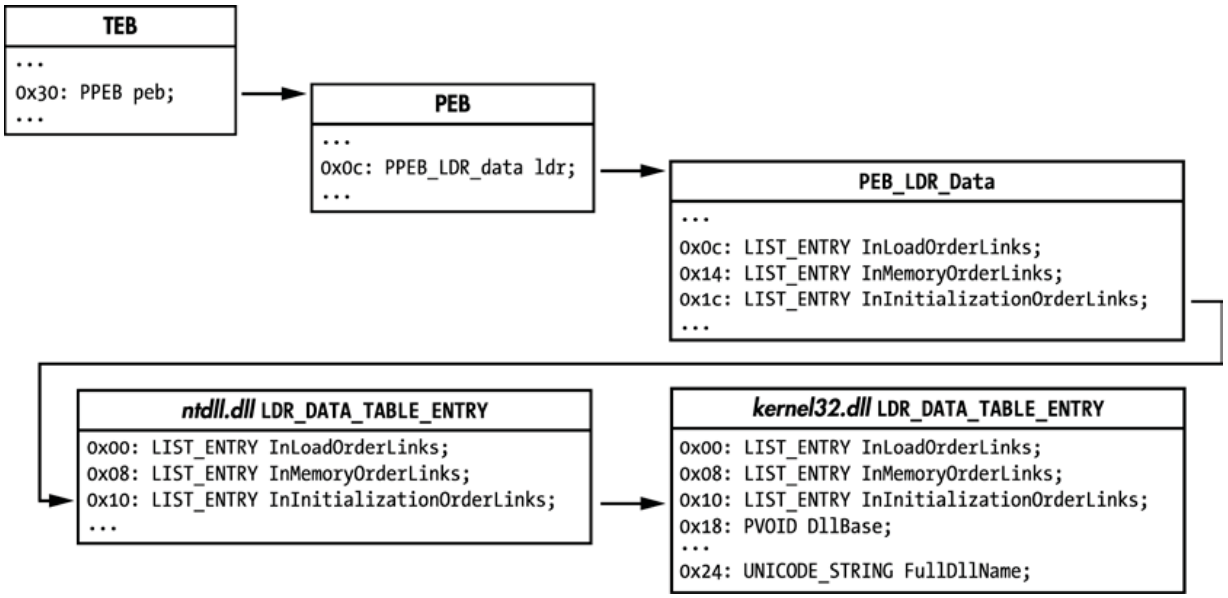

图19-1 遍历结构体来找到kernel32.dll的DllBase

进程从TEB结构体开始，其地址可以从FS段寄存器中访问到。TEB中偏移0x30是指向PEB的指针。PEB中偏移0xc是指向PEB_LDR_DATA结构体的指针，它是包含三个链接LDR_DATA_TABLE结构的双向链表——每个被加载的模块都有一个。在*kernel32.dll*项中的DllBase域就是我们正在查找的值。

三个LIST_ENTRY结构体以不同次序，按名字将LDR_DATA_TABLE项链接到一起。通常，InInitializationOrderLinks链表项会被shellcode跟踪。从Windows 2000到Vista，*kernel32.dll*一直是第二个被初始化的DLL，就在*ntdll.dll*后面，这意味着InInitializationOrderLinks链表结构体中的第二项应该属于*kernel32.dll*。然而，从Windows 7开始，kernel32.dll不再是第二个被初始化的模块了，所以这个简单的算法已不再工作。可移植的shellcode会转而去检查这个UNICODE_STRING类型的FullDllName域，来确认它就是kernel32.dll。

当遍历这个LIST_ENTRY结构体时，Flink指针指向下一个LDR_DATA_TABLE结构体中的LIST_ENTRY结构，Blink指针指向上一个LDR_DATA_TABLE结构体中的LIST_ENTRY结构，意识到这一点很重要。这意味着当跟踪InInitializationOrderLinks获取kernel32.dll的LDR_DATA_TABLE_ENTRY时，你只需要加八个字节到这个指针就可以得到DllBase，而不是加0x18，如果这个指针指向这个结构体的开头，你就必须这样做了。

代码清单19-4包含了找到*kernel32.dll*基地址的样本汇编代码。

代码清单19-4 findKernel32Base实现

```
; __stdcall DWORD findKernel32Base(void);
findKernel32Base:
    push    esi
    xor     eax, eax
    mov     eax, [fs:eax+0x30] ❶ ; eax gets pointer to PEB
    test    eax, eax             ; if high bit set: Win9x
    js      .kernel32_9x ❷
    mov     eax, [eax + 0x0c] ❹  ; eax gets pointer to PEB_LDR_DATA
    ;esi gets pointer to 1st
    ;LDR_DATA_TABLE_ENTRY.InInitializationOrderLinks.Flink
    mov     esi, [eax + 0x1c]
    ;eax gets pointer to 2nd
    ;LDR_DATA_TABLE_ENTRY.InInitializationOrderLinks.Flink
    lodsd ❺
    mov     eax, [eax + 8]       ; eax gets LDR_DATA_TABLE_ENTRY.DllBase
    jmp     near .finished
.kernel32_9x:
    jmp     near .kernel32_9x ❸  ; Win9x not supported: infinite loop
.finished:
    pop     esi
    ret
```

这个代码清单在❶处使用FS段寄存器访问TEB，获取指向PEB的指针。为了区别Win9x和WinNT

系统，❷处的js（如果有符号就跳转）指令用来测试PEB指针的最高有效位是否被设置。在WinNT（包括Windows 2000、XP，以及Vista）中，PEB指针的最高有效位通常从未被设置，因为高端内存地址是为操作系统保留的。使用符号位来识别操作系统家族的方法，在使用/3GB引导选项的系统中会失败，这个选项使用户层/内核层内存划分发生在0xC0000000而不是0x8000000，本例忽略这种情况。这段shellcode选择不支持Win9x，所以如果检测到Win9x，它将在❸处进入无限循环。

这段shellcode继续进行到在❹处的PEB_LDR_DATA，它假设自己正运行在Windows Vista或更早的版本中，所以它可以简单地获取❺处InInitializationOrderLinks链表中的第二个LDR_DATA_TABLE_ENTRY，并返回它的DllBase域。

19.4.2 解析PE文件导出数据

一旦你找到*kernel32.dll*的基地址，你必须解析它来找到导出的符号。和找到*kernel32.dll*的位置一样，这个过程涉及跟踪内存中的几个结构体。

当定义一个文件内的位置时，PE文件使用相对虚拟地址（RVA）。这些地址可以被认为是相对内存中PE映像的偏移值，所以PE映像基地址必须被加到每一个RVA中，来将得到的结果转换为一个有效指针。

导出数据被保存在IMAGE_EXPORT_DIRECTORY中。一个相对IMAGE_EXPORT_DIRECTORY的RVA被保存在IMAGE_OPTIONAL_HEADER末尾部分的IMAGE_DATA_DIRECTORY结构体数组中。IMAGE_DATA_DIRECTORY数组的具体位置取决于这个PE文件是32位的应用程序还是64位的。通常，shellcode假设自己正运行在一个32位平台上，所以它知道在编译时从PE特征值域到这个目录数组的正确偏移，计算方法如下：

```
sizeof(PE_Signature)+sizeof(IMAGE_FILE_HEADER)+sizeof(IMAGE_OPTIONAL_HEADER) = 120字节
```

图19-2为IMAGE_EXPORT_DIRECTORY结构体中的相对域。AddressOfFunctions是一个RVA的数组，指向实际导出的函数。它由一个导出序号来索引（另外一种查找导出符号的方法）。

为了使用这个数组，这段shellcode需要映射导出函数名到这个序号，并且shellcode的确是使用AddressOfNames和AddressOfNameOrdinals来这样做的。这两个数组同时存在，它们有着相同数目的项，而且显而易见的是这两个数组是直接相关的。AddressOfNames是一个32位RVA的数组，指向符号名的字符串。AddressOfNameOrdinals是一个16位序号的数组。对于一个给定的数组索引idx，AddressOfNames[idx]处的符号对应的导出序号就在AddressOfNameOrdinals[idx]处。AddressOfNames数组是按字母顺序排列的，这样一个二分搜索可以快速地找到一个指定字符串，尽管如此，大多数shellcode仍然是简单地从数组的开头执行线性查找。

按照如下步骤可以找到一个符号的导出地址：

1. 迭代 AddressOfNames 数组查看每一个 char *项，然后和需要的符号执行一个字符串比较，直

到找到一个匹配的项。我们将这个 AddressOfNames 的索引称为AddressOfNames iName。

2. 在 AddressOfNameOrdinals 数组中使用 iName 的索引。获取的值就是 iOrdinal 值。
3. 使用 iOrdinal 索引到 AddressOfFunctions 数组。获取到的值是被导出的符号的 RVA。最后将这个值返回给请求者。

本章后面会将这个算法的一个简单实现作为完整Hello World程序例子的一部分显示。

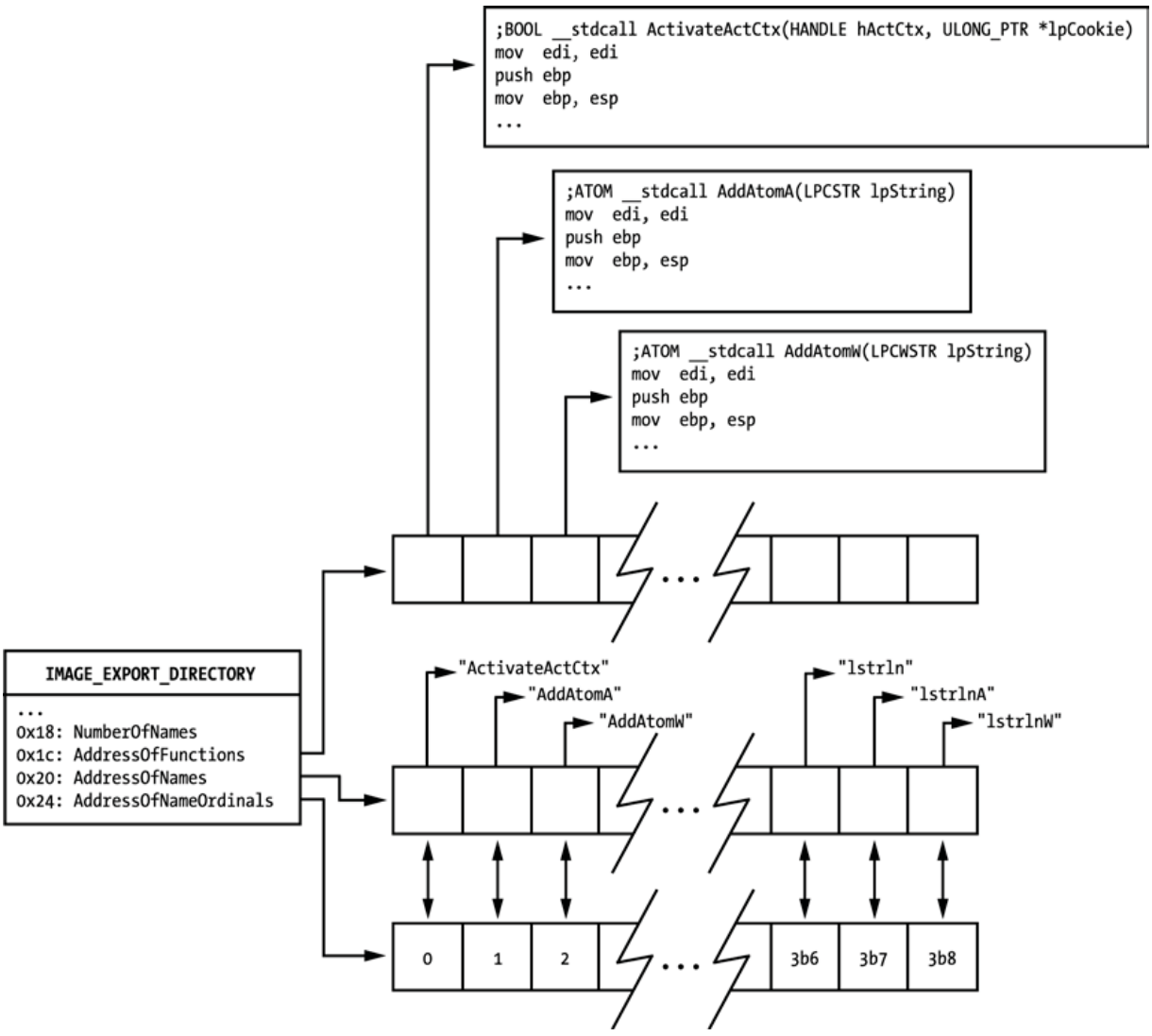

图19-2 kernel32.dll的IMAGE_EXPORT_DIRECTORY

一旦shellcode找到LoadLibraryA，它就可以加载任意库。LoadLibraryA的返回值被作为一个

Win32 API中的句柄看待。检查这个句柄值，显示它确实是一个指向被加载库dllBase的32位指针，这意味着这个shellcode可以跳过GetProcAddress，继续使用它自己的PE解析代码，来解析从LoadLibraryA返回的dllBase指针（这样做在导出符号名被散列时也是有效的，如本书第19.4.3节所讨论的）。

19.4.3 使用散列过的导出符号名

刚刚讨论过的算法有一个弱点：它对每一个导出名字执行strcmp，直到它找到正确的那个为止。这要求shellcode所使用的每一个API函数的全名，以ASCII字符串形式被包含进来。当这段shellcode的大小受限时，这些字符串可能使这段shellcode的大小超过限制。

解决这个问题的方法是计算出每个符号字符串的散列值，并用这个结果与保存在shellcode中的预先计算的值进行比较。散列函数不需要很复杂；只需要保证在每个被shellcode使用的DLL中，这些散列值是独一无二的就可以了。在不同DLL的符号之间及shellcode不使用的符号之间的散列冲突是可以接受的。

最常用的散列函数是32位旋转向右累加散列，如代码清单19-5所示。

代码清单19-5 hashString实现

```
; __stdcall DWORD hashString(char* symbol);
hashString:
    push    esi
    push    edi
    mov     esi, dword [esp+0x0c]   ; load function argument in esi
.calc_hash:
    xor     edi, edi ❶
    cld
.hash_iter:
    xor     eax, eax
    lodsb ❷                         ; load next byte of input string
    cmp     al, ah
    je      .hash_done              ; check if at end of symbol
    ror     edi, 0x0d ❸             ; rotate right 13 (0x0d)
    add     edi, eax
    jmp     near .hash_iter
.hash_done:
    mov     eax, edi
    pop     edi
    pop     esi
    retn    4
```

这个函数计算字符串指针参数的一个32位DWORD散列值。EDI寄存器被作为当前散列值对待，并

且在❶处被初始化为零。输入的字符串的每一个字节通过❷处的lodsb指令被加载。如果这个字节不是NULL，当前的散列值在❸处循环右移13位（0x0d），并且当前字节被加到这个散列中。这个散列在EAX中返回，这样它的调用者可以将这个结果与编译到代码中的预先计算的值比较。

> **提示：由于代码清单 19-5 中的特定算法被列入 Metasploit 中，已经成为了普遍使用的算法，但有时还是可以看见使用不同循环移位值和散列大小的变种。**

19.5 一个完整的Hello World例子

代码清单19-6为一个完整的findSymbolByHash函数实现，可以用它从已经被加载的DLL中找到导出符号。

代码清单19-6 findSymbolByHash实现

```
; __stdcall DWORD findSymbolByHash(DWORD dllBase, DWORD symHash);
findSymbolByHash:
    pushad
    mov     ebp, [esp + 0x24]       ; load 1st arg: dllBase
    mov     eax, [ebp + 0x3c] ❶     ; get offset to PE signature
    ; load edx w/ DataDirectories array: assumes PE32
    mov     edx, [ebp + eax + 4+20+96] ❷
    add     edx, ebp                ; edx:= addr IMAGE_EXPORT_DIRECTORY
    mov     ecx, [edx + 0x18] ❸     ; ecx:= NumberOfNames
    mov     ebx, [edx + 0x20]       ; ebx:= RVA of AddressOfNames
    add     ebx, ebp                ; rva->va
.search_loop:
    jecxz   .error_done             ; if at end of array, jmp to done
    dec     ecx                     ; dec loop counter
    ; esi:= next name, uses ecx*4 because each pointer is 4 bytes
    mov     esi, [ebx+ecx*4]
    add     esi, ebp                ; rva->va
    push    esi
    call    hashString ❹            ; hash the current string
    ; check hash result against arg #2 on stack: symHash
    cmp     eax, [esp + 0x28] ❺
    jnz     .search_loop
    ; at this point we found the string in AddressOfNames
    mov     ebx, [edx+0x24]         ; ebx:= ordinal table rva
    add     ebx, ebp                ; rva->va
    ; turn cx into ordinal from name index.
    ; use ecx*2: each value is 2 bytes
    mov     cx, [ebx+ecx*2] ❻
    mov     ebx, [edx+0x1c]         ; ebx:= RVA of AddressOfFunctions
    add     ebx, ebp                ; rva->va
    ; eax:= Export function rva. Use ecx*4: each value is 4 bytes
    mov     eax, [ebx+ecx*4] ❼
    add     eax, ebp                ; rva->va
    jmp     near .done
.error_done:
    xor     eax, eax                ; clear eax on error
.done:
    mov     [esp + 0x1c], eax ❽     ; overwrite eax saved on stack
    popad
    retn    8
```

这个函数使用一个指向DLL的基址指针和一个32位的与要查找符号相对应的散列值作为参数。它用寄存器EAX返回被请求的函数指针。请记住PE文件中的所有地址都是作为RVA保存的，所以代码需要不断地将`dllBase`值（在这个例子中保存在寄存器EBP中）与从PE结构体中获取的RVA相加，从而创建实际可以使用的指针。

这段代码在❶处开始解析这个PE文件，并获取指向PE特征域的指针。假设这是一个32位PE文件，在❷处，通过加上正确的偏移值，可以创建一个指向IMAGE_EXPORT_DIRECTORY的指针。这段代码开始解析❸处的IMAGE_EXPORT_DIRECTORY结构体，加载NumberOfNames值及AddressOfNames指针。AddressOfNames中的每个字符串指针都被传递到❹处的hashString函数中，然后将这个计算结果和在❺处传递给这个函数的参数进行比较。

一旦AddressOfNames的正确索引被找到，它将被作为位置❻处的AddressOfNameOrdinals数组的索引，来获得对应的序号数值，这个值被当作❼处的AddressOfFunctions数组的索引使用。这是用户想要的值，所以它被写在❽处的栈上，覆盖通过pushad指令保存的EAX值，以便这个值被后面的popad指令保存。

代码清单19-7为一个完整的Hello World shellcode例子，它使用前面定义的findKernel32Base和findSymbolByHash函数，而不依赖于硬编码的API位置。

代码清单19-7 位置无关的Hello World程序

```
    mov     ebp, esp
    sub     esp, 24h
    call    sub_A0  ❶           ; call to real start of code
    db 'user32',0   ❷
    db 'Hello World!!!!',0
sub_A0:
    pop     ebx                 ; ebx gets pointer to data
    call    findKernel32Base ❸
    mov     [ebp-4], eax        ; store kernel32 base address
    push    0EC0E4E8Eh          ; LoadLibraryA hash
    push    dword ptr [ebp-4]
    call    findSymbolByHash ❹
    mov     [ebp-14h], eax      ; store LoadLibraryA location
```

```
lea     eax, [ebx] ❺        ; eax points to "user32"
push    eax
call    dword ptr [ebp-14h] ; LoadLibraryA
mov     [ebp-8], eax        ; store user32 base address
push    0BC4DA2A8h ❻        ; MessageBoxA hash
push    dword ptr [ebp-8]   ; user32 dll location
call    findSymbolByHash
mov     [ebp-0Ch], eax      ; store MessageBoxA location
push    73E2D87Eh           ; ExitProcess hash
push    dword ptr [ebp-4]   ; kernel32 dll location
call    findSymbolByHash
mov     [ebp-10h], eax      ; store ExitProcess location
xor     eax, eax
lea     edi, [ebx+7]        ; edi:= "Hello World!!!!" pointer
push    eax                 ; uType: MB_OK
push    edi                 ; lpCaption
push    edi                 ; lpText
push    eax                 ; hWnd: NULL
call    dword ptr [ebp-0Ch] ; call MessageBoxA
xor     eax, eax
push    eax                 ; uExitCode
call    dword ptr [ebp-10h] ; call ExitProcess
```

这段代码在❶处通过`call/pop`调用获取指向❷处的数据头部的指针。然后在❸处调用`findKernel32Base`找到*kernel32.dll*，并在❹处调用`findSymbolByHash`，找到在*kernel32.dll*中散列为0xEC0E4E8E的导出符号。这是字符串`LoadLibraryA`的ror-13-additive散列。当这个函数返回EAX时，它指向`LoadLibraryA`的实际内存位置。

这段代码在❺处加载一个指向字符串`user32`的指针，并调用`LoadLibraryA`函数。然后它找到在❻处的导出函数`MessageBoxA`，并调用这个函数来显示“Hello World!!!!”消息。最终，它调用`ExitProcess`来完全退出。

> **提示：使用 shellcode 的 PE 解析能力，而不是 `GetProcAddress`，一个额外好处就是让逆向工程 shellcode 的难度变得更大，以免被随意的检查发现 API 调用。**

19.6 shellcode编码

为了在shellcode被触发时能够被执行，它被放置到程序地址空间中的某个位置。当与一个漏洞利用组合使用时，意味着shellcode必须在漏洞利用发生前存在，或是与漏洞利用数据一起被传递给程序。例如，如果一个程序正在对输入数据执行一些基本的过滤，那么这个shellocde必须绕过这个过滤，否则它就不会存在于这个脆弱进程的内存空间中。这意味着shellcode通常必须看起来像是合法数据，这样才能被一个脆弱程序所接受。

一个例子是程序使用不安全的字符串函数strcpy和strcat，这两个函数都没有对要写的数据设置最大长度。如果一个程序使用这些函数中的某个读取或复制恶意数据到一个长度固定的缓冲区中，这些数据能够轻易地超过缓冲区的大小，并导致一个缓冲区溢出攻击。这些函数将字符串作为以NULL（0x00）字节结尾的字符数组看待。一个攻击者想要复制到这个缓冲区的shellcode必须看起来像是有效数据，这意味着它不能在中间有任何NULL字节，这个NULL字节会提前终止这个字符串复制操作。

代码清单19-8中显示了一小段被用来访问注册表的反汇编代码，里面有7个NULL字节。这段代码一般不能用在shellcode中。

代码清单19-8 将NULL字节高亮后的典型代码

```
57                  push    edi
50                  push    eax             ; phkResult
6A 01               push    1               ; samDesired
8D 8B D0 13 00 00   lea     ecx, [ebx+13D0h]
6A 00               push    0               ; ulOptions
51                  push    ecx             ; lpSubKey
68 02 00 00 80      push    80000002h       ; hKey: HKEY_LOCAL_MACHINE
FF 15 20 00 42 00   call    ds:RegOpenKeyExA
```

程序可能对shellcode必须传递给它的数据执行额外的正确性检查，比如下面这些：

- 所有字节都是可打印的（小于0x80）ASCII字节。
- 所有字节都是字母数字组合的（A到Z，a到z，或0到9）。

要克服漏洞程序的过滤限制，几乎所有的shellcode都需要对主要载荷进行编码，从而绕过漏洞程序的过滤，然后再通过插入一个解码器将编码过的载荷转换为可执行代码。只有小段的解码器部分必须仔细地编写，使它的指令字节能够满足严格的过滤要求，载荷的其他部分可以在编译时进行编码，绕过过滤。如果这段shellcode将解码后的字节写回到编码字节位置（通常都是这样的），那么这个shellcode就是自修改的代码。当解码完成时，解码器转移控制到主要载荷上进行执行。

下面是常用的编码技术：

- 用常量字节掩码来XOR所有载荷字节。记住对所有拥有同样大小的值a、b，都有（a XOR b）XOR b ＝ a。
- 使用一种字母变换，有效载荷的每个字节被分割成两个4比特，然后与一个可打印的ASCII字符（比如A或a）相加。

shellcode编码对攻击者来说有额外的好处，主要体现在他们可以通过隐藏诸如URL或IP地址之类的人类可读的字符串，使分析更困难。此外，编码还可以帮助shellcode躲避网络入侵检测系统。

19.7 空指令雪橇

一个空指令雪橇（NOP sled）（也被称为空指令滑行区）是在shellcode之前的一段很长的指令序列，如图19-3所示。空指令雪橇并不是shellcode所必需的，但是它们经常被包含到一次漏洞利用中，以增加这个漏洞利用成功的可能性。shellcode编写者往往可以通过在shellcode后创建一大段空指令雪橇实现这一点。只要代码执行到这个空指令雪橇中的某处，shellcode最终都会运行。

图19-3 空指令雪橇和shellcode布局

传统的空指令雪橇由一长段NOP（`0x90`）指令序列组成，但是漏洞利用的编写者会用很多创新来避免检测。其他常用的空指令操作码在`0x40`至`0x4f`范围内。这些操作码是单字节指令，用于对通用寄存器的递增或递减。这个操作码字节范围也包含了可打印ASCII字符。这通常是有用的，因为空指令雪橇在解码器运行之前执行，所以它必须与shellcode的其余部分一样通过过滤。

19.8 找到shellcode

可以在很多资源中找到shellcode，包括网络流量、网页、媒体文件及恶意代码。因为不可能总创建拥有漏洞程序正确版本且能够触发漏洞利用的环境，所以恶意代码分析者必须尝试仅仅通过静态分析技术来逆向工程shellcode。

通常恶意网页使用JavaScript探测一个用户的系统轮廓，并检查存在漏洞的浏览器版本及已安装的插件。通常，JavaScript中的`unescape`函数也被用来将编码过的shellcode文本转换为可执行的二进制代码。shellcode经常作为一个编码后的文本字符串，包含在能够触发漏洞的渗透脚本中。

`unescape`的编码方式会将文本`%u`*XXYY*视作一个编码后大端Unicode字符，这里的*XX*和*YY*是十六进制值。在小端的机器（比如x86）上，字节序*YY XX*是被解码后的结果。例如，考虑如下文本字符串：

```
%u1122%u3344%u5566%u7788%u99aa%ubbcc%uddee
```

它将被解码为下面的二进制字节序列：

```
22 11 44 33 66 55 88 77 aa 99 cc bb ee dd
```

一个后面没有紧跟字母u的%符号，会被作为一个单独编码后的十六进制字节对待。例如，文本字符串%41%42%43%44会被解码为二进制字节序列41 42 43 44。

> **提示：单字节与双字节字符编码可以被同时用在同一个文本字符串中。这在使用 JavaScript 语言的地方（包括 PDF 文档中）是非常普遍的编码混淆技术。**

在恶意可执行文件中使用的shellcode通常是容易识别的，因为整个程序会使用混淆shellcode技术编写，或是一段shellcode有效负载被保存在恶意代码中，并被注入到另外一个进程进行执行。

通常可以通过查找一些典型的进程注入API调用来找shellcode有效载荷，在本书第12章中讨论过相关调用，它包括`VirtualAllocEx`、`WriteProcessMemory`及`CreateRemoteThread`。如果恶意代码启动一个远程线程，但没有应用重定位修正或解除外部依赖，那么被写入其他进程的缓冲区中很可能包含shellcode。这种操作对恶意代码编写者来说可能很方便，因为shellcode可以引导自身，不需要原先恶意代码的帮助来执行。

有时shellcode在一个媒体文件中以未编码的形式保存。诸如IDA Pro这样的反汇编器可以加载任意二进制文件，包括那些被怀疑包含shellcode的文件。然而，IDA Pro即使加载这个文件，也可能不分析这段shellcode，因为它不知道哪些字节是有效的代码。

寻找shellcode的一般方法是搜索很可能在shellcode起始位置上存在的初始解码器。表19-2中列举了可以用来搜索的一些重要操作码。

表19-2 一些要搜索的操作码字节

Instruction type	Common opcodes
Call	0xe8
Unconditional jumps	0xeb, 0xe9
Loops	0xe0, 0xe1, 0xe2
Short conditional jumps	0x70 through 0x7f

试着在已经加载的文件中反汇编表19-2中列举的操作码的每一个实例。所有有效代码应立即变得很明显。记住有效载荷可能被编码过，所以只有解码器是最先可见的。

如果这些搜索都找不到任何shellcode的迹象，仍然可能存在嵌入的shellcode，因为有些文件格式允许加入编码过的嵌入数据。例如，针对Adobe Reader软件CVE-2010-0188高危安全漏洞的渗透利用代码，使用了恶意构造的TIFF图像，进行了zlib压缩，并以Base64编码成字符串，然后被包含在PDF文件中。当你遭遇到一些特定的文件格式时，你需要熟悉这种格式及其包含的数据类型，以便查找恶意内容。

19.9 小结

shellcode编写者必须采用一些技术，来解决执行shellcode运行时环境中各种奇怪的固有局限性。这包括标识出这段shellcode会在内存中的什么位置运行，以及手动解除shellcode的所有外部依赖，以便它能够和系统交互。为了节省空间，这些依赖项通常使用哈希值而不是ASCII函数名，这同时也带来了一定的混淆效果。将几乎整个shellcode进行编码，绕过任何目标进程的数据过滤，也是非常普遍的技术。所有这些技术会轻易地让新入门的分析师感觉到沮丧，但是本章中的内容应该能帮助你识别这些常见的活动，使你能聚焦在理解这些shellcode的主要功能上。

19.10 实验

在这些实验中，我们将使用第19章涵盖的内容，来分析包含真正野外shellcode的样本。因为一个调试器不能轻易地直接加载与运行shellcode，我们将使用一个叫做*shellcode_launcher.exe*的工具，来动态分析shellcode二进制代码。你可以在第19章中找到关于如何使用这个工具的指导，以及在附录C中找到关于它的详细分析。

Lab 19-1

使用*shellcode_launcher.exe*，分析文件Lab19-01.bin。

问题

1. 这段 shellcode 是如何编码的？
2. 这段 shellcode 手动导入了哪个函数？
3. 这段 shellcode 和哪个网络主机通信？
4. 这段 shellcode 在文件系统上留下了什么迹象？
5. 这段 shellcode 做了什么？

Lab 19-2

文件Lab19-02.exe包含一段shellcode，这段shellcode会被注入到另外一个进程并运行，请分析这个文件。

问题

1. 这段 shellcode 被注入到什么进程中？

2. 这段 shellcode 位于哪里？
3. 这段 shellcode 是如何被编码的？
4. 这段 shellcode 手动导入了哪个函数？
5. 这段 shellcode 和什么网络主机进行通信？
6. 这段 shellcode 做了什么？

Lab 19-3

分析文件Lab19-03.pdf。如果你被卡住了，并且无法找到这段shellcode，那就跳过这个实验的前半部分，使用*shellcode_launcher.exe*工具分析文件Lab19-03_sc.bin。

问题

1. 这个 PDF 中使用了什么漏洞？
2. 这段 shellcode 是如何编码的？
3. 这段 shellcode 手动导入了哪个函数？
4. 这段 shellcode 在文件系统上留下了什么迹象？
5. 这段 shellcode 做了什么？