第二章　逆向与反汇编工具

知道了一些反汇编的背景知识后，在开始深入学习 IDA pro 之前，了解一些其他用于逆向工程的工具知识也会非常有用。这些工具大部分都要早于 IDA 出生并且依然是很好的快速分析二进制文件的工具，同时它们也可以用来与 IDA 对照。就如我们所见到的，IDA 已把这些工具的许多功能都集合到了它的用户界面中，这为逆向工程提供了一个简一的，集成的环境。然而，尽管 IDA 拥有一个集成的调试器，但我们并不打算讨论它，因为单就这个主题就可以出本书了。

分类工具（Classification Tools）

当我们一开始遇到一个不熟悉的文件时，问一些比较简单的问题，如“这是什么文件”，通常会很有用的。回答这问题的首要原则就是千万不要依靠这文件的扩展名来决定它是什么类型的文件。这甚至也是第二、三、四条原则。一旦你已经明白了文件扩展名对确定文件的类型没有任何意义之后，你也就会考虑学习使用如下的几个工具了。

file

file 是一个标准的工具，存在于*NIX 类操作系统和 Windows 下的 Cygwin[1]程序中。**file** 试图通过检查文件里的某些特定域来确认文件的类型。在某些情况下，**file** 能检测出常见的字符串，如“*#!/bin/sh*”（shell 脚本文件），或“*<html>*”(HTML 文件)。然而检测某些非 ASCII 文件将会变得更加困难，在这种情况下，**file** 首先判断该文件是否是符合某定已知的文件格式。在大多情况下，它是通过搜索一个唯一的已知特征值（通常被称为幻数（magic number）[2]）来决定文件的类型的。下面的十六进制表列出了一些常用文件类型的幻数。

```
Windows PE executable file
00000000   4D 5A 90 00   03 00 00 00   04 00 00 00   FF FF 00 00   MZ.............
00000010   B8 00 00 00   00 00 00 00   40 00 00 00   00 00 00 00   ........@.......

Jpeg image file
00000000   FF D8 FF E0   00 10 4A 46   49 46 00 01   01 01 00 60   ......JFIF.....`
00000010   00 60 00 00   FF DB 00 43   00 0A 07 07   08 07 06 0A   .`.....C........

Java .class file
00000000   CA FE BA BE   00 00 00 32   00 98 0A 00   2E 00 3E 08   .......2......>.
00000010   00 3F 09 00   40 00 41 08   00 42 0A 00   43 00 44 0A   .?..@.A..B..C.D.
```

file 可以识别大量的文件格式，这其中包括许多 ASCII 文本格式，许多可执行文件格式和一些数据文件格式。**file** 使用 magic file 来管理幻数检测。不同的操作系统拥有不同的 magic flie，但是一般情况下位于*/usr/share/file/magic*，*/usr/share/misc/magic* 或*/edt/magic* 文件里。请参阅 **file** 的文档查询更多关于 magic file 的信息。

---

1. 更多资料请访问 http://www.cygwin.com/.

2. 幻数(magic number)被很多文件格式规范用来表明该文件符合该文件格式。有时候幻数的选择还伴随着有趣的原由呢，如，MS-DOS 可执行文件文件头的 MZ 特征值是由一位最初的 MS-DOS 结构师 Mark Zbikowski 的名字的每个单词的第一个字母组成，又如，十六进制数 0xcafebabe 是众所周知的 Java .class 文件的幻数，选择它仅公是因为它是一个很容易记的十六进制串。

CYGWIN 环境

Cygwin 是为 Windows 操作系统提供一个类 Linux 的命令行和其相关的程序的一个工具集。安装的时候，有很多的安装包供您选择，包括编绎器（如 gcc， g++），解释器（如 Perl, Python, Ruby），网络工具集（如 nc，ssh），当然也包括很多其它工具。cygwin 安装完成后，你就可以在 Windows 操作系统中编绎和执行许多 Linux 下的程序了。

在某些情况下，**file** 还可以识别同一类文件的微小差异。下面的列表示范了 **file** 不仅可以识别几种不同的 ELF 文件，而且还可以给出这些文件是怎么连接的（静态或动态）和有没被去除符号信息（**stripped**）等一些相关信息。

```
idabook# file ch2_ex_*
ch2_ex.exe:                  MS-DOS executable PE  for MS Windows (console)
                             Intel 80386 32-bit
ch2_ex_upx.exe:              MS-DOS executable PE  for MS Windows (console)
                             Intel 80386 32-bit, UPX compressed
ch2_ex_freebsd:              ELF 32-bit LSB executable, Intel 80386,
                             version 1 (FreeBSD), for FreeBSD 5.4,
                             dynamically linked (uses shared libs),
                             FreeBSD-style, not stripped
ch2_ex_freebsd_static:       ELF 32-bit LSB executable, Intel 80386,
                             version 1 (FreeBSD), for FreeBSD 5.4,
                             statically linked, FreeBSD-style, not stripped
ch2_ex_freebsd_static_strip: ELF 32-bit LSB executable, Intel 80386,
                             version 1 (FreeBSD), for FreeBSD 5.4,
                             statically linked, FreeBSD-style, stripped
ch2_ex_linux:                ELF 32-bit LSB executable, Intel 80386,
                             version 1 (SYSV), for GNU/Linux 2.6.9,
                             dynamically linked (uses shared libs),
                             not stripped
ch2_ex_linux_static:         ELF 32-bit LSB executable, Intel 80386,
                             version 1 (SYSV), for GNU/Linux 2.6.9,
                             statically linked, not stripped
ch2_ex_linux_static_strip:   ELF 32-bit LSB executable, Intel 80386,
                             version 1 (SYSV), for GNU/Linux 2.6.9,
                             statically linked, stripped
ch2_ex_linux_stripped:       ELF 32-bit LSB executable, Intel 80386,
                             version 1 (SYSV), for GNU/Linux 2.6.9,
                             dynamically linked (uses shared libs), stripped
```

Strip 二进制执行文件

Strip 一个二进制文件是从二进制文件中把符号去除的一个过程。编绎时，编绎器会为目标文件创建符号信息。这里有些符号用于解决链接过程各文件之间的引用问题，从而产生最终的可执行文件或库文件。有时候,符号也被用来为调试器提供额外的信息。整个链接过程中，有许多的符号已不在需要了。创建(build)文件时,我们可以在编绎程序的过程中，通过给链接器传递特定选项来去除一些无用的符号。此外，一个叫作 strip 的工具可以用于去除二进制文件的符号信息。虽然一个被去除符号的文件会比原来的要小，但它的功能将保持不变。

然而 **file** 和其类似的工具一样也有出错的时候。一个文件被误认也是很正常，原因很简单，有些识别标签碰巧出现在了某类文件里。你可以打开任意十六进制文件编辑器来实验一下，用它打开任何一个文件并把前四个字符改成 Java 的幻数序列：CA　FE　BA BE。此时 **file** 就会将这个修改的文件误认为是 compiled java class data 文件。同样的，一个仅含 MZ 两个字符的文件将被误认为是一个 MS-DOS 的可执行文件。在逆向工程中一个应该采取的很好的方法就是千万不要完全相信任何工具的输出结果，除非使用多种工具的确认或是你亲自动手分析。

PE Tools

PE Tools[3] 是用于分析 windows 平台上进程和可执行文件的一个工具集。图 2-1 是 **PE Tools** 的主界面，这里罗列出了所有活动进程，通过该介面你能够访问该工具集的所有功能。

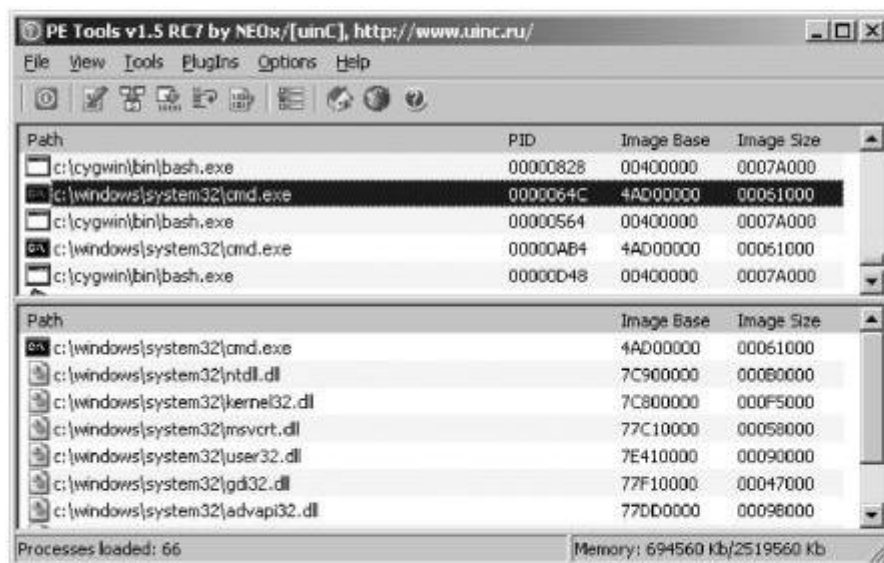

图 2-1：　**PE Tools** 工具

你可以从进程列表中抓取（DUMP）一个进程的内存印象然后把它转存到文件中，也可以使用 PE 嗅探工具检测这个程序由什么编译器编译的以及判断其是否被已知的混淆工具(obfuscation Tool)处理过。同时 **PE tools** 也为本地文件提供了类似的菜单。你可以使用自带的 PE Editor 工具来查看 PE 文件头的各个域，你也可以用它对这些域的值做简单的修改。从已经被混淆了的文件中重建一个有效的文件，往往需要手工修改 PE 文件头。

二进制文件的混淆技术

混淆（Obfuscation）即掩盖事件的真相。对二进制文件来说，混淆即掩盖程序的真实行为。程序员有很多理由使用它。混淆技术常用来保护专有算法和以及隐藏程序的恶意行为。几乎所有的恶意程序都使用混淆来阻碍对它的分析。程序员可以使用的混淆工具到处可见。混淆工具与技术以及其对逆向工程所产生的影响将会在第二十一章做进一步的探讨。

PEiD

PEiD[4]是 Windows 下另一款主要用于识别程序所使用的编译器和使用的混淆技术的 PE 文

3. 更多资料请查询 http://petools.org.ru/petools.shtml

4. 更多资料请查询 http://peid.has.it/.

件工具。图 2-2 演示了使用 **PEiD** 识别一个 Gaobot[5]蠕虫的变种所使用的混淆工具（此处为 ASPack）。

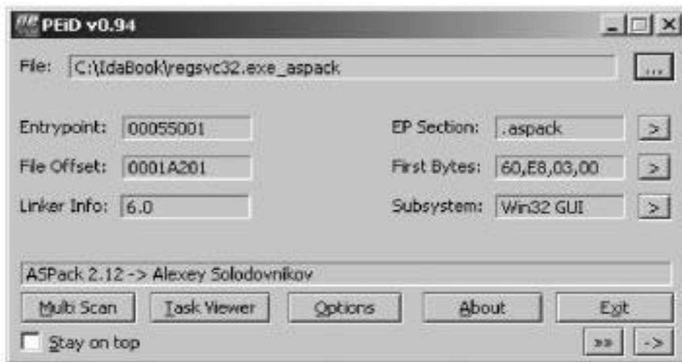

图 2-2 ：PEiD 工具

PEiD 有与 **PE Tools** 很多相似功能，包括概要 PE 文件头，收集运行中程序的信息，而且者能做些基本的反汇编。

总结性的工具（Summary Tools）

由于我们的目的是逆向二进制文件，在我们大概的把一个程序归类后，我们需要更专业的工具来获得更多的信息。这一节我们将讨论那些比之前只能识别文件格式工具更加高级的工具。在大多情况下，这些工具都对某一特定的文件特别了解，它们被用于获取要处理的文件的详细信息。

nm

当源代码文件被编译成目标文件时，编译器必需嵌入一些全局（或外部）符号地址的信息，这样的话当链接器把目标文件链在一起生成可执行文件时才能解决对这些符号的引用问题。除非被指明要从最终的可执行文件中去除符号信息，一般来说链接器都要在最终的可执行文件中载入很多的符号信息。根据它的手册，我们可以知道 **nm** 的主要功能在于列出目标文件的符号信息。

当 **nm** 用于检测一个中间级的目标文件时（扩展名为 .O 的文件而不是可执行文件）时，默认的输出是在这文件中声明的函数名和全局变量名。下面列出了 **nm** 的样例输出：

```
idabook# gcc -c ch2_example.c
idabook# nm ch2_example.o
         U __stderrp
         U exit
         U fprintf
00000038 T get_max
00000000 t hidden
00000088 T main
00000000 D my_initialized_global
00000004 C my_unitialized_global
         U printf
         U rand
         U scanf
         U srand
         U time
00000010 T usage
idabook#
```

5.更多信息请查询：http:// securityresponse.symantec.com/security_response/writeup.jsp?docid=2005-030723-2605-99

这里我们可以看到 **nm** 列出了每个符号与其相关的信息。这字母码用于表明所列出来的符号的类型。在上面的例子中，我们看到了下列的这些字母码，现在给出其解释:

U 未定义的符号，通常是一个外部变量的引用

T 定义在代码节的符号，通常是一个函数名。

t 定义在代码节的局部符号。在 C 程序中，这通常等同于静态函数（static function）.

D 已初始化的数据

C 未初始化的数据

注意： *大写字母用于表示全局符号，小写字母用于表示局部符号表。更详尽的解释可查询* **nm** *的手册页。*

大体上来讲，用 **nm** 列出二进制文件的符号有助于我们获得更多的信息。链接时，符号指向于虚拟地址（如果支持虚拟存取的话），这使 **nm** 能够获得更多的信息。下面是用 **nm** 打开可执行文件的部分输出：

```
idabook# gcc -o ch2_example ch2_example.c
idabook# nm ch2_example
         <. . .>
         U exit
         U fprintf
080485c0 t frame_dummy
08048644 T get_max
0804860c t hidden
08048694 T main
0804997c D my_initialized_global
08049a9c B my_unitialized_global
08049a80 b object.2
08049978 d p.0
         U printf
         U rand
         U scanf
         U srand
         U time
0804861c T usage
idabook#
```

此时，有些符号（如 main）被赋上了虚拟地址，链接过程也会引入一些新的符号（如 frame_dummy），也有些（如 my_unitialized_global）符号的类型改变了，而另外一些由于仍旧引用外部符号因而仍为未定义。我们检测的这个例子是动态链接的，因此这些未定义符号存在于 C 语言共享库文件当中。更多关于 **nm** 的信息可以在它的手册中找到。

ldd

生成一个可执行文件时，必须解决库函数引用这一问题。链接器有两种方法来解决库函数的引用问题：静态链接和动态链接。链接器通过传入的选项以决定使用哪一种方法。一个可执行文件既可有静态链接，动态链接，也可两者都有[6]。

当使用静态链接时，链接器把程序的目标文件和要使用的库文件组合起来生成一个可执行文件。因为库函数代码已经包含于可执行文件了，就不必再进行定位了。静态链接的优点有：（1）更快的函数调用；（2）程序发布将变得更加简单，因为这不用对用户系统的库函

6.关于链接的更多信息请参见 John Levine 一文 *Linkers and Loaders* (San Francisco:Morgan Kaufmann，2000).

数的有效性做出任何假设。缺点有：（1）可执行文件将会变得很臃肿；（2）另外如若库函数发生改变，程序的升级将变得非常困难。因为每一个库文件改变后，程序都得重新链接一遍。在逆向工程师看来，静态链接或多或少增加了些难度。当遇到一个静态链接的程序时，就没有特别简单的判断该程序引用了何种库文件的方法了。第十二章将挑战逆向分析静态链接程序。

动态链接和静态链接不同，链接器不需要复制库函数。取而代之的是简单的把需要引用的库文件（通常是.so 或.dl 文件）插入到最终的可执行文件里，这就使生成的可执行文件体积更小。使用动态链接时，更新库文件将变得十分简单。因为如果有一个库文件要修改，而这个库文件又被很多的程序所引用，那么我们仅需要把新的库文件替换掉原来的库文件就行了，这时引用它的所有程序都更新好了。一个动态链接的缺点是其带来更加复杂的加载过程。必须得找到所有库文件并将其载入内存当中。使用动态链接的另一缺点是销售商在发布可执行程序时也要发布这个程序所使需的所有的库文件。试图在系统里运行一个没有包含该程序所需的所有的库文件时将会出错。

下面的输出示范了一个程序的动态和静态版本的生成过程，程序的大小以及 **file** 工具是如何识别这两个程序的：

```
idabook# gcc -o ch2_example_dynamic ch2_example.c
idabook# gcc -o ch2_example_static ch2_example.c --static
idabook# ls -l ch2_example_*
-rwxr-xr-x  1 root  wheel    6017 Sep 26 11:24 ch2_example_dynamic
-rwxr-xr-x  1 root  wheel  167987 Sep 26 11:23 ch2_example_static

idabook# file ch2_example_*
ch2_example_dynamic: ELF 32-bit LSB executable, Intel 80386, version 1
        (FreeBSD), dynamically linked (uses shared libs), not stripped
ch2_example_static:  ELF 32-bit LSB executable, Intel 80386, version 1
        (FreeBSD), statically linked, not stripped
idabook#
```

为了确保在链接函数的过程中不出现错误，动态链接的程序必须得指定其使用了的库文件。因此，和静态链接不同，很容易就能找到动态链接程序所使用的库文件。**ldd**(list dynamic dependencies)是一个用来列出程序所需动态库文件的工具。在下面这个例子，**ldd** 检测出了 Apache web 服务器程序所依赖的库：

```
idabook# ldd /usr/local/sbin/httpd
/usr/local/sbin/httpd:
        libm.so.4 => /lib/libm.so.4 (0x280c5000)
        libaprutil-1.so.2 => /usr/local/lib/libaprutil-1.so.2 (0x280db000)
        libexpat.so.6 => /usr/local/lib/libexpat.so.6 (0x280ef000)
        libiconv.so.3 => /usr/local/lib/libiconv.so.3 (0x2810d000)
        libapr-1.so.2 => /usr/local/lib/libapr-1.so.2 (0x281fa000)
        libcrypt.so.3 => /lib/libcrypt.so.3 (0x2821a000)
        libpthread.so.2 => /lib/libpthread.so.2 (0x28232000)
        libc.so.6 => /lib/libc.so.6 (0x28257000)
idabook#
```

Linux 和 BSD 操作系统提供了 **ldd** 这个工具。在 OS X 系统上，使用 **otool** 工具，并带上 –L 选项：**otool –L** ***filename*** 就可以实现类似的功能。在 Windows 系统，Visual Studio 工具集自带了一个工具，dumpbin，使用命令：**dumpbin /dependents** ***filename*** 可以列出该文件所依赖的库文件。

objdump

不同于 **ldd** 的专业，**objdump** 的功用相当广泛。**objdump** 的目标是“显示目标文件的信息”[7]. 这可是一个相当宏大的目标，为了实现这一目标，**objdump** 使用大量选项（大于 30 个）试图将目标文件里所有信息都显示出来。**objdump** 可以显示的目标文件的信息如下（甚至更多）：

节头（Section headers）

文件节的信息汇总。

私有头（Private headers）

程序的内存分布及其运行时需要加载的，可以使用 ldd 显示的库文件的信息。

调试信息（Debugging information）

提取出程序里的调试信息。

符号信息（Symbol information）

像 ***nm*** 那样，导出符号表信息。

反汇编列表（Disassembly listing）

objdump 对标为代码的节采用一次性扫描反汇编。当反汇编 x86 代码时，**objdump** 可以输出 AT&T 或是 Intel 格式的代码，并且反汇编代码还可以另存为一个文本文件，这样的文本文件叫做反汇编列表。尽管这样的文件可以用于逆向工程，但是浏览起来并不方便，而且编辑时非常容易出错。

objdump 是 GNU 二进制工具集（GNU binutils[8]）的一部分，Linux，FreeBSD 甚至 Windows（通过 Cygwin）都提供了这工具。**objdump** 依赖于二进制文件描述库（Binary File Descriptor library， 简称 libbfd，为 binutils 的组件）对目标文件进行访问，因此它只可以解析出 libbfd 支持的文件格式（如 ELF， PE 等)。一个叫做 **readelf** 的工具也可用于分析 ELF 文件。**readelf** 几乎提供了 **objdump** 所有的功能，但是 **readelf** 并不依赖于 libbfd。

otool

otool 可以简单认为是一个 OS X 平台下的 **objdump** 的实现，可用于分析 OS X 平台的 Mach-O 二进制文件。下表演示了 **otool** 显示 mach-o 文件依赖的其他文件的列表。该功能和 **ldd** 完全一样。

```
idabook# file osx_example
osx_example: Mach-O executable ppc
idabook# otool -L osx_example
osx_example:
        /usr/lib/libstdc++.6.dylib (compatibility version 7.0.0, current version 7.4.0)
        /usr/lib/libgcc_s.1.dylib (compatibility version 1.0.0, current version 1.0.0)
        /usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version 88.1.5)
```

Otool 也可以输出文件头和符号表的信息，并且可以对代码节进行反汇编。更多关于 **otool** 功能的信息请参考它的手册。

7.更多信息请查询：http:// www.gnu.org/software/binutils/manual/html_chapter/binutils_4.html

8.更多信息请查询：http://www.gnu.org/software/binutils/.

dumpbin

dumpbin 是集中于微软 Visual Studio 开发包中的一个命令行工具。和 ***otool*** 和 **objdump** 一样，**dumpbin** 可以输出大量的 PE 文件信息。下面的例子演示了如何用 **dumpbin** 列出 windows 自带计算器程序所使用的库文件。这和 ldd 非常想像。

```
$ dumpbin /dependents calc.exe
Microsoft (R) COFF/PE Dumper Version 8.00.50727.762
Copyright (C) Microsoft Corporation.  All rights reserved.

Dump of file calc.exe

File Type: EXECUTABLE IMAGE

  Image has the following dependencies:

    SHELL32.dll
    msvcrt.dll
    ADVAPI32.dll
    KERNEL32.dll
    GDI32.dll
    USER32.dll
```

dumpbin 的其它选项可以提取出 PE 文件各个节的相关信息，包括符号，输入函数名，输出函数名和反汇编代码。更多的信息可以访问 MSDN(Mircrosoft Developer Network[9]).

c++filt

支持重载的语言都存在一个区分使用同一名称不同重载函数的方法。下面这个例子展示了一个名为 demo 的重载函数的不同函数原型：

```
void demo(void);
void demo(int x);
void demo(double x);
void demo(int x, double y);
void demo(double x, int y);
void demo(char* str);
```

一般情况下，不可能出现两个名称相同的函数。因此为了使用重载函数，编绎器会为重载函数加上参数类型信息而为这些不同的重载函数生成惟一的函数名。使用相同的函数名，却能访问不同的函数的过程，被称为命名改编（name mangling [10]）。当我们使用 **nm** 来查看使用之前 C++代码编绎的文件的符号时，我们就可以看到如下信息(为了突显 demo 重载函数，过滤其他的信息):

9.更多信息请查询：http://msdn2.microsoft.com/en-us/library/clh23y6c(VS.71).aspx

10.更多信息请查询：http://en.wikipedia.org/wiki/Name_mangling.

```
idabook# g++ -o cpp_test cpp_test.cpp
idabook# nm cpp_test | grep demo
0804843c T _Z4demoPc
08048400 T _Z4demod
08048428 T _Z4demodi
080483fa T _Z4demoi
08048414 T _Z4demoid
080483f4 T _Z4demov
```

C++标准没有给出命名改编的标准方案，而是让编译器设计者开发自己的方案。为了解析出重载函数 demo 的不同版本，我们就必须得使用能哆识别编绎该文件的编绎器（这里是 g++）所使用的命名改编方法的工具。而这正是 **c++filt** 的主要目的。起初，**c++filt** 将每个输入的名称认为是命名改编后的名称，然后才会去分析生成该名称的编绎器类型。如果这恬巧是一个合法的命名改编名称，那么就输出该命名改编后的原始名称。当 **c++filt** 不能识别出该命名改编名称时，它就原样输出。

如果转送上例 nm 的结果给 **c++filt**，就能找到原始的函数名称了。如下表所示:

```
idabook# nm cpp_test | grep demo | c++filt
0804843c T demo(char*)
08048400 T demo(double)
08048428 T demo(double, int)
080483fa T demo(int)
08048414 T demo(int, double)
080483f4 T demo()
```

ee

值得注意的是，改编了的名字可能包含了该函数的一些额外信息，但是一般情况下 **nm** 并不能显示这些信息。而有时候这些信息在逆向工程时非常有用。在有些复杂的情况下，这些信息还可能包含该函数的类名称及其调用方式。

更深入性的工具（Deep Inspection Tools）

到目前为止，我们已经介绍了一些使用少量文件内部结构信息来对文件进行大略分析的工具。同样，我们也介绍了许多基于文件内部详细信息来进行文件分析的工具。这一节我们主要讨论那些与文件类型无关的文件分析工具。

strings

有时候，思考那些与特定文件格式无关的普遍性问题非常有用。例如您可能会问，“这文件里包含字符串么？”。那么首先我们就得回答“什么是字符串”这个问题。起初，我们可以假设一个字符串即一串可打印的字符，而且字符串定义往往要指定一个最小长度和一个特定字符集才有意义。因此，我们可以指定搜索至少 4 个字节以上的可打印的 ASCII 字符串然后将其显示在输出终端。一般情况下，文件格式不会影响这种字符串的搜索。你可以在 ELF 二进制文件里搜索字符串，而这就像在微软的 Word 文档里搜索字符串一样简单。

strings 工具可以从不同格式文件当中提取字符串。使用默认选项（至少 4 个字符的 7 位 ASCII 序列）时，**strings** 可能输出与下面类似的结果：

```
idabook# strings ch2_example
/lib/ld-linux.so.2
__gmon_start__
libc.so.6
_IO_stdin_used
exit
srand
puts
time
printf
stderr
fwrite
scanf
__libc_start_main
GLIBC_2.0
PTRh
[^_]
usage: ch2_example [max]
A simple guessing game!
Please guess a number between 1 and %d.
Invalid input, quitting!
Congratulations, you got it in %d attempt(s)!
Sorry too low, please try again
Sorry too high, please try again
```

不过， 有此些字符串看起来像程序的输出的字符串，另外一些则看起来像函数名和库文件名。我们不应该根据这些字符串信息就冒然断定程序的行为。逆向工程时，我们往往容易错误地依赖 ***strings*** 工具输出的信息来推断程序的行为。有一点得记住：程序里有这些字符串并不表示程序会以任何的方式使用它。

使用 **strings** 还得注意如下几个问题：

- 当 **strings** 用于可执行文件时，默认情况下，**strings** 只会扫描可加载的和初始化了的节，一定要牢记这一点。使用-a 可以迫使 **strings** 扫描整个文件。
- **strings** 并不会指定字符串在文件中的位置。但使用-t 选项选项可以输出每一个字符串的文件偏移值。
- 许多文件工具都可以更改字符集。使用-e 选项，**strings** 就可以搜索宽字符了，如 16 位的 Unicode.

反汇编器（Disassembliers）

如前所述，有很多工具都可以生成目标文件的反汇编代码。你可以使用 **dumpbin**，**objdump** 以及 **otool** 来反汇编相应的 PE，ELF 及 MACH-O 类型文件。这些工具都不能够处理其他格式的文件数据。因此当你遇到与一个不使用常规文件格式的文件时，你就需要一个能从指定位置开始反汇编的工具了。

有两个 x86 指令集的流式反汇编程序（stream disassemblers）：**ndisasm** 和 **diStorm** [11]。**ndisasm** 是 NASM（Netwide Assembler [12]）下的一个工具。下面是一个使用 **ndisasm** 反汇编由 Metasploit framework[13]生成的 shellcode 的例子

11.更多信息请访问：http://www.ragestorm.net/distorm/.

12.更多信息请访问：http://nasm.souceforge.net/.

13.更多信息请访问：http://www.metasploit.com/

```
idabook# ./msfpayload linux_ia32_findsock CPORT=4444 R > fs
idabook# ls -l fs
-rw-r--r-- 1 ida ida 62 Oct 30 15:49 fs
idabook# ndisasm -u fs
00000000  31D2              xor edx,edx
00000002  52                push edx
00000003  89E5              mov ebp,esp
00000005  6A07              push byte +0x7
00000007  5B                pop ebx
00000008  6A10              push byte +0x10

0000000A  54                push esp
0000000B  55                push ebp
0000000C  52                push edx
0000000D  89E1              mov ecx,esp
0000000F  FF01              inc dword [ecx]
00000011  6A66              push byte +0x66
00000013  58                pop eax
00000014  CD80              int 0x80
00000016  66817D02115C      cmp word [ebp+0x2],0x5c11
0000001C  75F1              jnz 0xf
0000001E  5B                pop ebx
0000001F  6A02              push byte +0x2
00000021  59                pop ecx
00000022  B03F              mov al,0x3f
00000024  CD80              int 0x80
00000026  49                dec ecx
00000027  79F9              jns 0x22
00000029  52                push edx
0000002A  682F2F7368        push dword 0x68732f2f
0000002F  682F62696E        push dword 0x6e69622f
00000034  89E3              mov ebx,esp
00000036  52                push edx
00000037  53                push ebx
00000038  89E1              mov ecx,esp
0000003A  B00B              mov al,0xb
0000003C  CD80              int 0x80
```

很多时候，流式反汇编灵活性非常有用处。例如在分析可能包含 shellcode 的网络包的攻击时，流式反汇编能够从网络包的 shellcode 位置开始反汇编以分析其恶意行为。另外一种情况是分析那些格式未知的 ROM 时，有些地方是数据，而有些地方是代码。此时，可以只使用流式反汇编来反汇编代码部分数据。

总结(Summary)

本章所讨论的工具可能不是该类里边最好的，但它们是所有从事逆向工程的人都能使用的经典工具。更为重要的是，它们代表了推动 IDA 开发的工具类型。在接下来的章节，我们还会讨论这些工具。深入理解这些工具能够大大加深你对 IDA 用户界面及其输出信息的理解。