2

REVERSING AND DISASSEMBLY TOOLS

With some disassembly background under our belts, and before we begin our dive into the specifics of Ghidra, it will be helpful to understand some of the other tools used to reverse engineer binaries. Many of these tools predate Ghidra and continue to be useful for taking quick glimpses into files, as well as for double-checking the work that Ghidra does. As we will see, Ghidra rolls many of these tools' capabilities into its user interface to provide a single, integrated environment for reverse engineering.

Classification Tools

When first confronted with an unknown file, it is often useful to answer simple questions such as "What is this thing?" The first rule of thumb when attempting to answer that question is to *never* rely on a file extension to determine what a file actually is. That is also the second, third, and fourth rules of thumb. Once you have become an adherent of the "file extensions

are meaningless" line of thinking, you may wish to familiarize yourself with one or more of the following utilities.

file

The `file` command is a standard utility, included with most *nix-style operating systems, as well as the Windows Subsystem for Linux (WSL). The `file` command attempts to identify a file's type by examining specific fields within the file. In some cases, `file` recognizes common strings such as `#!/bin/sh` (a shell script) and `<html>` (an HTML document).

> **THE WSL ENVIRONMENT**
>
> The Windows Subsystem for Linux (WSL) enables users to run a full GNU/Linux environment directly on Windows without needing a virtual machine or dual-boot setup. WSL supports running Linux distributions with access to a wide range of command line tools (`grep`, `awk`), compilers (`gcc`, `g++`), interpreters (Perl, Python, Ruby), networking utilities (`curl`, `ssh`), and many others. With WSL 2, users gain enhanced performance and full system call compatibility, enabling many Linux-native applications to be compiled and executed seamlessly on Windows systems with the bonus of a choice between a command line or GUI.

Files containing non-ASCII content present somewhat more of a challenge. In such cases, `file` attempts to determine whether the content appears to be structured according to a known file format. In many cases, it searches for specific tag values known to be unique to specific file types, often referred to as magic numbers.

A *magic number* is a unique sequence of bytes in a file that can be used to identify its format. This allows classification tools to easily identify a file's type regardless of the file's extension. In some cases, magic numbers are selected for humorous reasons. The `MZ` byte sequence in MS-DOS executable file headers represents the initials of Mark Zbikowski, one of the original architects of MS-DOS, while the hex value `0xcafebabe`, the well-known magic number associated with Java *.class* files, was chosen because the sequence of hex digits spells an easily remembered word.

The following hex listings show several examples of magic numbers used to identify some common file types:

```
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.
```

The `file` command has the ability to identify many file formats, including several types of ASCII text files and various executable and data file formats. The magic number checks performed by `file` are governed by rules contained in a *magic file*. The default magic file varies by operating system, but common locations include */usr/share/file/magic*, */usr/share/misc/magic*, and */etc/magic*. Please refer to the documentation for `file` for more information concerning magic files.

In some cases, `file` can distinguish variations within a given file type. The following listing demonstrates `file`'s ability to identify not only several variations of ELF binaries but also information pertaining to how the binary was linked (statically or dynamically) and whether the binary was stripped.

```
ghidrabook# file ch2_ex_*
  ch2_ex_x64:         ELF 64-bit LSB shared object, x86-64, version 1 (SYSV),
                      dynamically linked, interpreter /lib64/l, for GNU/Linux
                      3.2.0, not stripped
  ch2_ex_x64_dbg:     ELF 64-bit LSB shared object, x86-64, version 1 (SYSV),
                      dynamically linked, interpreter /lib64/l, for GNU/Linux
                      3.2.0, with debug_info, not stripped
  ch2_ex_x64_static: ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux),
                      statically linked, for GNU/Linux 3.2.0, not stripped
  ch2_ex_x64_strip:   ELF 64-bit LSB shared object, x86-64, version 1 (SYSV),
                      dynamically linked, interpreter /lib64/l, for GNU/Linux
                      3.2.0, stripped
  ch2_ex_x86:         ELF 32-bit LSB shared object, Intel 80386, version 1
                      (SYSV), dynamically linked, interpreter /lib/ld-, for
                      GNU/Linux 3.2.0, not stripped
  ch2_ex_x86_dbg:     ELF 32-bit LSB shared object, Intel 80386, version 1
                      (SYSV), dynamically linked, interpreter /lib/ld-, for
                      GNU/Linux 3.2.0, with debug_info, not stripped
  ch2_ex_x86_static: ELF 32-bit LSB executable, Intel 80386, version 1
                      (GNU/Linux), statically linked, for GNU/Linux 3.2.0,
                      not stripped
  ch2_ex_x86_strip:   ELF 32-bit LSB shared object, Intel 80386, version 1
                      (SYSV), dynamically linked, interpreter /lib/ld-, for
                      GNU/Linux 3.2.0, stripped
  ch2_ex_Win32:       PE32 executable (console) Intel 80386, for MS Windows
  ch2_ex_x64:         PE32+ executable (console) x86-64, for MS Windows
```

The `file` utility and similar utilities are not foolproof. It is quite possible for a file to be misidentified simply because it happens to bear the identifying marks of a particular file format. You can see this for yourself by using a hex editor to modify the first 4 bytes of any file to the Java magic number sequence: `CA FE BA BE`. The `file` utility will incorrectly identify the newly modified file as *compiled Java class data*. Similarly, a text file that contains only

the two characters `MZ` will be identified as *MS-DOS executable*. A good approach to take in any reverse engineering effort is to never fully trust the output of any tool until you have correlated that output with several tools and manual analysis.

STRIPPING BINARY EXECUTABLE FILES

Stripping a binary is the process of removing symbols from the binary file. Binary object files contain symbols as a result of the compilation process. Some of these symbols are utilized during the linking process to resolve references between files when creating the final executable file or library. In other cases, symbols may be present to provide additional information for use with debuggers. Following the linking process, many of the symbols are no longer required. Options passed to the linker can cause the linker to remove the unnecessary symbols at build time. Alternatively, a utility named `strip` may be used to remove symbols from existing binary files. Although a stripped binary will be smaller than its unstripped counterpart, the behavior of the stripped binary will remain unchanged.

PE-bear

PE-bear, a tool available for Windows and Linux, is useful for quickly looking at 32-bit and 64-bit Windows executable files. You can find it at *https://github.com/hasherezade/pe-bear*. It provides a graphical display of a PE file's structure, including listing and graphical views of each file section, hex dumps, and disassembly listings of section content.

By using its included signatures, PE-bear can also identify whether an executable was processed by any known obfuscation utilities. Finally, PE-bear includes a PE editor, allowing you to easily modify header values and section content. You'll often need to modify PE headers when attempting to reconstruct a valid PE from an obfuscated version of that file.

Detect-It-Easy

Detect-It-Easy (DiE) is a tool designed primarily for feature extraction and basic identification of executable files, and it's available at *https://github.com/horsicq/Detect-It-Easy*. DiE uses signatures to identify common executable and container file formats, such as PE, ELF, ZIP, and APK. Further, DiE attempts to identify the compiler used to build the executable and any obfuscation utilities that may have been used on the file.

Features extracted by DiE include lists of strings, hashes of individual sections, and entropy measures across all portions of the binary. Many of the tool's additional capabilities overlap with those of PE-bear, including the ability to summarize file headers and perform basic disassembly.

BINARY FILE OBFUSCATION

Obfuscation is any attempt to obscure the true meaning of something. When applied to executable files, obfuscation is any attempt to hide the true behavior of a program. Programmers may employ obfuscation for a number of reasons. Commonly cited examples include protecting proprietary algorithms and obscuring malicious intent. Almost all forms of malware utilize obfuscation in an effort to hinder analysis. Tools to help program authors generate obfuscated programs are widely available. We further discuss obfuscation tools and techniques and their associated impacts on the reverse engineering process in Chapter 21.

Summary Tools

Once we've performed the initial classification of a file, we will need more sophisticated tools to extract detailed information. These highly specialized tools are typically designed to work with specific file formats and often won't function correctly outside of that narrow scope. Each tool understands the structure of a particular file type and is capable of parsing it to extract targeted, format-specific information.

nm

When source files are compiled to object files, compilers must embed information regarding the location of any global (external) symbols into the object file so that the linker can resolve references to those symbols when it combines object files to create an executable. Unless instructed to strip symbols from the final executable, the linker generally carries symbols from the object files over into the resulting executable. According to the man page, the `nm` utility lists symbols from object files.

When used to examine an object file (a compiled file that typically has a *.o* extension and is not yet linked into a final executable), `nm`'s default output yields the names of any functions and global variables declared in the file. Here is sample output of the `nm` utility:

```
ghidrabook# gcc -c ch2_nm_example.c
ghidrabook# nm ch2_nm_example.o
                 U exit
                 U fwrite
  000000000000002e t get_max
                 U _GLOBAL_OFFSET_TABLE_
                 U __isoc99_scanf
  00000000000000a6 T main
  0000000000000000 D my_initialized_global
  0000000000000004 C my_uninitialized_global
                 U printf
                 U puts
```

```
                 U rand
                 U srand
                 U __stack_chk_fail
                 U stderr
                 U time
  0000000000000000 T usage
ghidrabook#
```

We see that `nm` lists each symbol, along with information about the symbol. The letter codes indicate the type of symbol being listed. In this example, we see the following letter codes:

U An undefined symbol (usually an external symbol reference).

T A symbol defined in the text section (usually a function name).

t A local symbol defined in the text section. In a C program, this usually equates to a static function.

D An initialized data value.

C An uninitialized data value.

NOTE *Uppercase letter codes are used for global symbols, whereas lowercase letter codes are used for local symbols. More information, including a full explanation of the letter codes, can be found in the man page for `nm`.*

The utility displays somewhat more information when used to list symbols from an executable file. During linking, symbols are resolved to virtual addresses when possible, which makes more information available when `nm` is run. Here is truncated sample output from `nm` used on an executable:

```
ghidrabook# gcc -o ch2_nm_example ch2_nm_example.c
ghidrabook# nm ch2_nm_example
  ...
                 U fwrite@@GLIBC_2.2.5
  0000000000000938 t get_max
  0000000000201f78 d _GLOBAL_OFFSET_TABLE_
                 w __gmon_start__
  000000000000c5c r __GNU_EH_FRAME_HDR
  0000000000000730 T _init
  0000000000201d80 t __init_array_end
  0000000000201d78 t __init_array_start
  000000000000b60 R _IO_stdin_used
                 U __isoc99_scanf@@GLIBC_2.7
                 w _ITM_deregisterTMCloneTable
                 w _ITM_registerTMCloneTable
  000000000000b50 T __libc_csu_fini
  000000000000ae0 T __libc_csu_init
                 U __libc_start_main@@GLIBC_2.2.5
  00000000000009b0 T main
  0000000000202010 D my_initialized_global
  000000000020202c B my_uninitialized_global
```

```
                 U printf@@GLIBC_2.2.5
                 U puts@@GLIBC_2.2.5
                 U rand@@GLIBC_2.2.5
  0000000000000870 t register_tm_clones
                 U srand@@GLIBC_2.2.5
                 U __stack_chk_fail@@GLIBC_2.4
  0000000000000800 T _start
  0000000000202020 B stderr@@GLIBC_2.2.5
                 U time@@GLIBC_2.2.5
  0000000000202018 D __TMC_END__
  000000000000090a T usage
ghidrabook#
```

At this point, some of the symbols (`main`, for example) have been assigned virtual addresses, new ones (such as `__libc_csu_init`) have been introduced as a result of the linking process, some (such as `my_unitialized_global`) have had their symbol type changed, and others remain undefined as they continue to reference external symbols. In this case, the binary we are examining is dynamically linked, and the undefined symbols are defined in the shared C library.

ldd

When creating an executable, the linker must resolve the location of any library functions that executable references. The linker has two methods for resolving calls to library functions: *static linking* and *dynamic linking*. Command line arguments provided to the linker determine which of the two methods is used. An executable may be statically linked, dynamically linked, or both.

When static linking is requested, the linker combines an application's object files with a copy of the required library to create an executable file. At runtime, there is no need to locate the library code because it is already contained within the executable. Static linking has a few advantages: It results in slightly faster function calls and makes it easier to distribute binaries because no assumptions need to be made regarding the availability of library code on users' systems.

Disadvantages of static linking include larger resulting executables and greater difficulty upgrading programs when library components change. Programs are more difficult to update because they must be relinked every time a library is changed. From a reverse engineering perspective, static linking also complicates matters somewhat. If we are faced with the task of analyzing a statically linked binary, there is no easy way to answer the questions "Which libraries are linked into this binary?" and "Which of these functions is a library function?" Chapter 13 discusses the challenges encountered while reverse engineering statically linked code.

Dynamic linking differs from static linking in that the linker does not need to make a copy of any required libraries. Instead, the linker simply inserts references to any required libraries (often *.so* or *.dll* files) into the final

executable, usually resulting in much smaller executable files. Upgrading library code is much easier in these cases; the vendor maintains a single copy of a library, and many binaries reference that copy, replacing the outdated library with a new version.

One of the disadvantages of using dynamic linking is that it requires a more complicated loading process. All of the necessary libraries must be located and loaded into memory, as opposed to loading one statically linked file that contains all of the library code. Another disadvantage of dynamic linking is that vendors must distribute not only their own executable file but also all library files upon which that executable depends. Attempting to execute a program on a system that does not contain all the required library files will result in an error.

The following output demonstrates the creation of dynamically and statically linked versions of a program, the size of the resulting binaries, and the manner in which `file` identifies those binaries:

```
ghidrabook# gcc -o ch2_example_dynamic ch2_example.c
ghidrabook# gcc -o ch2_example_static ch2_example.c -static
ghidrabook# ls -l ch2_example_*
  -rwxrwxr-x 1 ghidrabook ghidrabook  12944 Nov  7 10:07 ch2_example_dynamic
  -rwxrwxr-x 1 ghidrabook ghidrabook 963504 Nov  7 10:07 ch2_example_static
ghidrabook# file ch2_example_*
  ch2_example_dynamic: ELF 64-bit LSB executable, x86-64, version 1 (SYSV),
  dynamically linked, interpreter /lib64/l, for GNU/Linux 3.2.0,
  BuildID[sha1]=e56ed40012accb3734bde7f8bca3cc2c368455c3, not stripped
  ch2_example_static:  ELF 64-bit LSB executable, x86-64, version 1 (GNU/Linux),
  statically linked, for GNU/Linux 3.2.0,
  BuildID[sha1]=430996c6db103e4fe76aea7d578e636712b2b4b0, not stripped
ghidrabook#
```

For dynamic linking to function properly, dynamically linked binaries must indicate which libraries they depend on, along with the specific resources required from each of those libraries. As a result, unlike statically linked binaries, it is quite simple to determine the libraries on which a dynamically linked binary depends. The `ldd` (*list dynamic dependencies*) utility is a tool used to list the dynamic libraries an executable requires. In the following example, we run `ldd` to determine the libraries on which the Apache web server depends:

```
ghidrabook# ldd /usr/sbin/apache2
  linux-vdso.so.1 =>  (0x00007fffc1c8d000)
  libpcre.so.3 => /lib/x86_64-linux-gnu/libpcre.so.3 (0x00007fbeb7410000)
  libaprutil-1.so.0 => /usr/lib/x86_64-linux-gnu/libaprutil-1.so.0 (0x00007fbeb71e0000)
  libapr-1.so.0 => /usr/lib/x86_64-linux-gnu/libapr-1.so.0 (0x00007fbeb6fa0000)
  libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007fbeb6d70000)
  libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fbeb69a0000)
  libcrypt.so.1 => /lib/x86_64-linux-gnu/libcrypt.so.1 (0x00007fbeb6760000)
  libexpat.so.1 => /lib/x86_64-linux-gnu/libexpat.so.1 (0x00007fbeb6520000)
  libuuid.so.1 => /lib/x86_64-linux-gnu/libuuid.so.1 (0x00007fbeb6310000)
```

```
  libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007fbeb6100000)
  /lib64/ld-linux-x86-64.so.2 (0x00007fbeb7a00000)
ghidrabook#
```

The `ldd` utility is available on Linux and BSD systems. On macOS systems, similar functionality is available using the `otool` utility with the `-L` option: `otool -L` *`filename`*. On Windows systems, the `dumpbin` utility, part of the Visual Studio tool suite, can be used to list dependent libraries: `dumpbin /dependents` *`filename`*.

BEWARE YOUR TOOLS!

While `ldd` may appear to be a simple tool, the `ldd` man page states that "you should never employ `ldd` on an untrusted executable, since this may result in the execution of arbitrary code." While this is unlikely in most cases, it provides a reminder that running even apparently simple software reverse engineering (SRE) tools may have unintended consequences when examining untrusted input files. While we hope it is obvious that executing untrusted binaries is unlikely to be safe, it is wise to take precautions even when statically analyzing untrusted binaries and to assume that the computer on which you perform SRE tasks, along with any data on it or other hosts connected to it, may be compromised as a result of SRE activities.

objdump

Whereas `ldd` is fairly specialized, `objdump` is extremely versatile. Its purpose is to display information from object files. This is a fairly broad goal, and to accomplish it, `objdump` responds to more than 50 command line options tailored to extract various pieces of information. You can use this tool to display the following data (and much more) related to object files:

Section headers Summary information for each of the sections in the program file.

Private headers Program memory layout information and other information required by the runtime loader, including a list of required libraries, such as that produced by `ldd`.

Debugging information Any debugging information embedded in the program file.

Symbol information Symbol table information, dumped in a manner similar to the `nm` utility.

Disassembly listing The `objdump` tool performs a linear sweep disassembly of sections of the file marked as code. When disassembling x86 code, `objdump` can generate either AT&T or Intel syntax, and the disassembly can be captured as a text file. This type of text file is called a disassembly *dead listing*, and while these files can certainly be used for reverse engineering, they are difficult to navigate effectively and even more difficult to modify in a consistent and error-free manner.

The `objdump` tool is available as part of the GNU binutils tool suite and can be found on Linux, FreeBSD, and Windows (via WSL). Note that `objdump` relies on the *Binary File Descriptor library (libbfd)*, a component of binutils, to access object files and thus is only capable of parsing file formats supported by libbfd (including ELF and PE, among others). For ELF-specific parsing, you could also use a utility named `readelf`, which offers most of the same capabilities as `objdump` but does not rely upon libbfd.

otool

The `otool` utility is most easily described as an `objdump`-like option for macOS, and it is useful for parsing information about macOS Mach-O binaries. The following listing demonstrates how `otool` displays the dynamic library dependencies for a Mach-O binary, thus performing a function similar to `ldd`:

```
ghidrabook# file ch2_ex_macOS_arm
  ch2_ex_macOS_arm: Mach-O 64-bit executable arm64
ghidrabook# otool -L ch2_ex_macOS_arm
  ch2_ex_macOS_arm:
    /usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version 1351.0.0)
```

The `otool` utility can be used to display information related to a file's headers and symbol tables and to perform disassembly of the file's code section. You can find more information about the tool's capabilities on the associated man page.

dumpbin

Like `otool` and `objdump`, the `dumpbin` utility is capable of displaying a wide range of information related to Windows PE files. It's included in Microsoft's Visual Studio suite of tools. The following listing shows how `dumpbin` displays the dynamic dependencies of the Windows Notepad program in a manner similar to `ldd`:

```
$ dumpbin /dependents C:\Windows\System32\notepad.exe
Microsoft (R) COFF/PE Dumper
Copyright (C) Microsoft Corporation.  All rights reserved.

Dump of file c:\Windows\System32\notepad.exe

File Type: EXECUTABLE IMAGE

  Image has the following dependencies:

    GDI32.dll
    USER32.dll
    api-ms-win-crt-string-l1-1-0.dll
    api-ms-win-crt-runtime-l1-1-0.dll
    ...
```

```
  Image has the following delay load dependencies:

    ADVAPI32.dll
    COMDLG32.dll
    PROPSYS.dll
    SHELL32.dll
    WINSPOOL.DRV
    urlmon.dll

  Summary

        3000 .data
        1000 .didat
        ...
```

Additional `dumpbin` options offer the ability to extract information from various sections of a PE binary, including symbols, imported function names, exported function names, and disassembled code. You can find more information related to the use of `dumpbin` on the Microsoft website.

c++filt

Languages that allow function overloading must have a mechanism for distinguishing between the many overloaded versions of a function since each version has the same name. The following C++ example shows the prototypes for several overloaded versions of a function named `demo`:

```
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);
```

As a general rule, two functions cannot have the same name in an object file. So, compilers derive unique names for overloaded functions by incorporating information that describes the type sequence of the function arguments. The process of deriving unique names for functions with identical names is called *name mangling*. If we use `nm` to dump the symbols from the compiled version of the preceding C++ code, we might see something like the following (filtered to focus on versions of `demo`):

```
ghidrabook# g++ -o ch2_cpp_example ch2_cpp_example.cc
ghidrabook# nm ch2_cpp_example | grep demo
  000000000000060b T _Z4demod
  0000000000000626 T _Z4demodi
  0000000000000601 T _Z4demoi
  0000000000000617 T _Z4demoid
```

```
0000000000000635 T _Z4demoPc
00000000000005fa T _Z4demov
```

The C++ standard does not define a standard name mangling scheme, leaving compiler designers to develop their own. To decipher the mangled variants of `demo` shown here, we need a tool that understands the name mangling scheme of our compiler (`g++`, in this case).

This is precisely the purpose of `c++filt`. This utility treats each input word as if it were a mangled name and then attempts to determine the compiler used to generate that name. If the name appears to be a valid mangled name, it outputs the demangled version of the name. When `c++filt` does not recognize a word as a mangled name, it simply outputs the word with no changes.

If we pass the results of `nm` from the preceding example through `c++filt`, the tool recovers the demangled function names, as shown here:

```
ghidrabook# nm ch2_cpp_example | grep demo | c++filt
  000000000000060b T demo(double)
  0000000000000626 T demo(double, int)
  0000000000000601 T demo(int)
  0000000000000617 T demo(int, double)
  0000000000000635 T demo(char*)
  00000000000005fa T demo()
```

It is important to note that the mangled names contain additional information about functions that `nm` does not normally provide. This information can be extremely helpful in reverse engineering situations. In more complex cases, it may include data regarding class names or function-calling conventions.

Deep Inspection Tools

So far, we have discussed tools that perform a cursory analysis of files based on minimal knowledge of those files' internal structures. We have also seen tools capable of extracting specific pieces of data from files based on very detailed knowledge of a file's structure. In this section, we discuss tools designed to extract specific types of information independently of the type of file being analyzed.

strings

It is occasionally useful to ask more generic questions regarding file content that do not necessarily require any specific knowledge of a file's structure. One such question is "Does this file contain any embedded strings?" Of course, we must first answer the question "What exactly constitutes a string?"

Let's loosely define a *string* as a consecutive sequence of printable characters. This definition is often augmented to specify a minimum length and a specific character set. Thus, we could specify a search for all sequences of at least four consecutive ASCII printable characters and print the results to the console. Searches for such strings generally do not depend on the file's structure. You can search for strings in an ELF binary just as easily as you can search for strings in a Microsoft Word document.

The `strings` utility is designed specifically to extract string content from files, often without regard for the format of those files. Using `strings` with its default settings (which look for 7-bit ASCII sequences of at least four characters) might yield something like the following:

```
ghidrabook# strings ch2_example
  /lib64/ld-linux-x86-64.so.2
  libc.so.6
  exit
  srand
  __isoc99_scanf
  puts
  time
  __stack_chk_fail
  printf
  stderr
  fwrite
  __libc_start_main
  GLIBC_2.7
  GLIBC_2.4
  GLIBC_2.2.5
  _ITM_deregisterTMCloneTable
  __gmon_start__
  _ITM_registerTMCloneTable
  usage: ch4_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
  GCC: (Ubuntu 7.4.0-1ubuntu1~18.04.1) 7.4.0
  ...
```

Unfortunately, while some of these strings look like they might be program outputs, other strings appear to be function names and library names. We should be careful not to jump to any conclusions; analysts often fall into the trap of attempting to deduce the behavior of a program based on the output of `strings`, but the presence of a string within a binary in no way indicates that the binary ever uses the string in any manner.

Here are some final notes on the use of `strings`:

- By default, `strings` gives no indication of where, within a file, a string is located. You can use the `-t x` command line argument to have `strings` print file offset information for each string found.
- In addition, `strings` searches for ASCII printable sequences. The `strings` man page describes the options available for searching for strings made up of other character encodings such as Unicode.

WHY DID STRINGS CHANGE?

Historically, when `strings` was used on executable files, it would, by default, search only for character sequences in the loadable, initialized data sections of the binary file. This required `strings` to parse the binary file to find those sections, using libraries such as libbfd. When it was used for parsing untrusted binaries, vulnerabilities in libraries could potentially result in arbitrary code execution. As a result, the default behavior for `strings` was changed to examine the entire binary file without parsing for loadable initialized data sections (synonymous with the use of the `-a` flag). The historical behavior can be invoked using the `-d` flag.

Disassemblers

Earlier, we mentioned that dedicated tools can generate dead listing–style disassemblies of binary object files. You can disassemble PE, ELF, and Mach-O binaries using `dumpbin`, `objdump`, and `otool`, respectively. However, none of those tools can deal with arbitrary blocks of binary data. You will occasionally be confronted with a binary file that does not conform to a widely used file format, in which case you will need tools capable of beginning the disassembly process at user-specified offsets.

Two examples of such *stream disassemblers* for the x86 instruction set are `ndisasm` and `diStorm3`. The utility `ndisasm` is included with Netwide Assembler (NASM). The following example illustrates the use of `ndisasm` to disassemble a piece of shellcode generated using the Metasploit framework:

```
ghidrabook#  msfvenom -p linux/x64/shell_find_port -f raw > findport
ghidrabook#  ndisasm -b 64 findport
  00000000  4831FF            xor rdi,rdi
  00000003  4831DB            xor rbx,rbx
  00000006  B314              mov bl,0x14
  00000008  4829DC            sub rsp,rbx
  0000000B  488D1424          lea rdx,[rsp]
  0000000F  488D742404        lea rsi,[rsp+0x4]
  00000014  6A34              push byte +0x34
  00000016  58                pop rax
  00000017  0F05              syscall
  00000019  48FFC7            inc rdi
```

```
  0000001C  66817E024A67      cmp word [rsi+0x2],0x674a
  00000022  75F0              jnz 0x14
  00000024  48FFCF            dec rdi
  00000027  6A02              push byte +0x2
  00000029  5E                pop rsi
  0000002A  6A21              push byte +0x21
  0000002C  58                pop rax
  0000002D  0F05              syscall
  0000002F  48FFCE            dec rsi
  00000032  79F6              jns 0x2a
  00000034  4889F3            mov rbx,rsi
  00000037  BB412F7368        mov ebx,0x68732f41
  0000003C  B82F62696E        mov eax,0x6e69622f
  00000041  48C1EB08          shr rbx,byte 0x8
  00000045  48C1E320          shl rbx,byte 0x20
  00000049  4809D8            or rax,rbx
  0000004C  50                push rax
  0000004D  4889E7            mov rdi,rsp
  00000050  4831F6            xor rsi,rsi
  00000053  4889F2            mov rdx,rsi
  00000056  6A3B              push byte +0x3b
  00000058  58                pop rax
  00000059  0F05              syscall
ghidrabook#
```

The flexibility of stream disassembly is useful in many situations. One scenario involves the analysis of computer network attacks in which network packets may contain shellcode. Stream disassemblers can disassemble the portions of the packet that contain shellcode to analyze the payload's behavior. Another situation involves the analysis of ROM images for which no layout reference can be located. Portions of the ROM will contain data, while other portions will contain code. Stream disassemblers can be used to disassemble just those portions of the image thought to be code.

Summary

The tools discussed in this chapter are not necessarily the best of their kind. However, they do represent tools commonly available for anyone who wishes to reverse engineer binary files. More importantly, they represent the types of tools that motivated much of the development of Ghidra. In future chapters, we occasionally highlight stand-alone tools that provide functionality similar to that integrated into Ghidra. An awareness of these tools will greatly enhance your understanding of the Ghidra user interface and the many informational displays that Ghidra offers.