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 useful to understand some of the other tools that are used for reverse engineering binaries. Many of these tools predate Ghidra and continue to be useful for quick glimpses into files as well as for double-checking the work that Ghidra does. As we will see, Ghidra rolls many of the capabilities of these tools 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 filename 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 Subsystems for Linux[1] (WSL). This command is also available to Windows users by installing either Cygwin[2] or MinGW.[3] 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). 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 (often referred to as *magic numbers*[4]) known to be unique to specific file types. 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 capability to identify a large number of 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

1. See *https://docs.microsoft.com/en-us/windows/wsl/about*.

2. See *http://www.cygwin.com/*.

3. See *http://www.mingw.org/*.

4. A *magic number* is a special tag value required by some file format specifications whose presence indicates conformance to such specifications. In some cases, humorous reasons surround the selection of magic numbers. The `MZ` tag 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 it is an easily remembered sequence of hex digits.

THE WSL ENVIRONMENT

The Windows Subsystem for Linux provides a GNU/Linux command-line environment directly within Windows without the need to create a virtual machine. During WSL installation, users choose a Linux distribution and can then run it on the WSL. This provides access to common command-line free software (`grep`, `awk`), compilers (`gcc`, `g++`), interpreters (Perl, Python, Ruby), networking utilities (`nc`, `ssh`), and many others. Once WSL has been installed, many programs written for use with Linux can be compiled and executed on Windows systems.

binary was linked (statically or dynamically) and whether the binary was stripped or not.

```
ghidrabook# 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 12.0,
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 12.0,
statically linked, FreeBSD-style, not stripped
ch2_ex_freebsd_static_strip: ELF 32-bit LSB executable, Intel 80386,
version 1 (FreeBSD), for FreeBSD 12.0,
statically linked, FreeBSD-style, stripped
ch2_ex_linux: ELF 32-bit LSB executable, Intel 80386,
version 1 (SYSV), for GNU/Linux 3.2.0,
dynamically linked (uses shared libs),
not stripped
ch2_ex_linux_static: ELF 32-bit LSB executable, Intel 80386,
version 1 (SYSV), for GNU/Linux 3.2.0,
statically linked, not stripped
ch2_ex_linux_static_strip: ELF 32-bit LSB executable, Intel 80386,
version 1 (SYSV), for GNU/Linux 3.2.0,
statically linked, stripped
ch2_ex_linux_stripped: ELF 32-bit LSB executable, Intel 80386,
version 1 (SYSV), for GNU/Linux 3.2.0,
dynamically linked (uses shared libs), stripped
```

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 some 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 containing only

the two characters `MZ` will be identified as an *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. While a stripped binary will be smaller than its unstripped counterpart, the behavior of the stripped binary will remain unchanged.

PE Tools

PE Tools[5] is a collection of tools useful for analyzing both running processes and executable files on Windows systems. Figure 2-1 shows the primary interface offered by PE Tools, which displays a list of active processes and provides access to all of the PE Tools utilities.

From the process list, users can dump a process's memory image to a file or utilize the PE Sniffer utility to determine what compiler was used to build the executable or whether the executable was processed by any known obfuscation utilities. The Tools menu offers similar options for analysis of disk files. Users can view a file's PE header fields by using the embedded PE Editor utility, which also allows for easy modification of any header values. Modification of PE headers is often required when attempting to reconstruct a valid PE from an obfuscated version of that file.

5. See *https://github.com/petoolse/petools*.

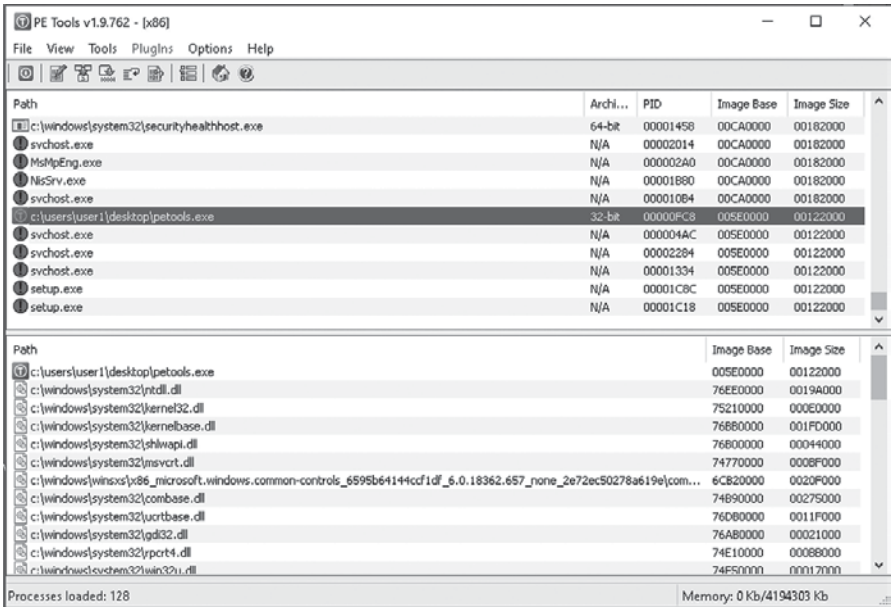

Figure 2-1: The PE Tools utility

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. Nearly all forms of malware utilize obfuscation in an effort to hinder analysis. Tools are widely available to assist program authors in generating obfuscated programs. Obfuscation tools and techniques and their associated impact on the reverse engineering process will be discussed further in Chapter 21.

PEiD

PEiD[6] is another Windows tool whose primary purposes are to identify the compiler used to build a particular Windows PE binary and to identify any tools used to obfuscate a Windows PE binary. Figure 2-2 shows the use of PEiD to identify the tool (ASPack in this case) used to obfuscate a variant of the Gaobot[7] worm.

6. See *https://github.com/wolfram77web/app-peid*.

7. See *https://www.trendmicro.com/vinfo/us/threat-encyclopedia/malware/GAOBOT/*.

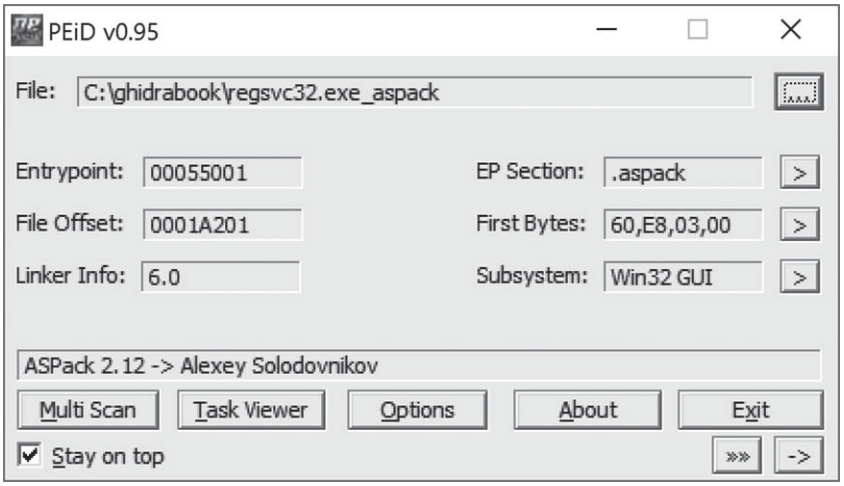

Figure 2-2: The PEiD utility

Many additional capabilities of PEiD overlap those of PE Tools, including the ability to summarize PE file headers, collect information on running processes, and perform basic disassembly.

Summary Tools

Since our goal is to reverse engineer binary program files, we are going to need more sophisticated tools to extract detailed information following initial classification of a file. The tools discussed in this section, by necessity, are far more aware of the formats of the files that they process. In most cases, these tools understand a very specific file format, and the tools are utilized to parse input files to extract very specific information.

nm

When source files are compiled to object files, compilers must embed information regarding the location of any global (external) symbols so that the linker will be able to 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 `nm` is used to examine an intermediate object file (a *.o* file rather than an executable), the default output yields the names of any functions and global variables declared in the file. Sample output of the `nm` utility is shown next:

```
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#
```

Here we see that `nm` lists each symbol, along with some information about the symbol. The letter codes are used to 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.

Somewhat more information is displayed when `nm` is used to display symbols from an executable file. During linking, symbols are resolved to virtual addresses (when possible), which results in more information being available when `nm` is run. Truncated sample output from `nm` used on an executable is shown here:

```
ghidrabook# gcc -o ch2_nm_example ch2_nm_example.c
ghidrabook# nm ch2_nm_example
*--- SOME CONTENT OMITTED FOR BREVITY ---*
                 U fwrite@@GLIBC_2.2.5
0000000000000938 t get_max
0000000000201f78 d _GLOBAL_OFFSET_TABLE_
                 w __gmon_start__
0000000000000c5c r __GNU_EH_FRAME_HDR
0000000000000730 T _init
0000000000201d80 t __init_array_end
0000000000201d78 t __init_array_start
0000000000000b60 R _IO_stdin_used
                 U __isoc99_scanf@@GLIBC_2.7
                 w _ITM_deregisterTMCloneTable
```

```
                 w _ITM_registerTMCloneTable
0000000000000b50 T __libc_csu_fini
0000000000000ae0 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 (`__libc_csu_init`) have been introduced as a result of the linking process, some (`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 an executable is created, the location of any library functions referenced by that executable must be resolved. 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.[8]

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. Advantages of static linking are that (1) it results in slightly faster function calls and (2) distribution of binaries is easier because no assumptions need be made regarding the availability of library code on users' systems. Disadvantages of static linking include (1) larger resulting executables and (2) 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 complicates matters somewhat. If we are faced with the task of analyzing a statically linked binary, there is no easy way

8. For more information on linking, consult John R. Levine's *Linkers and Loaders* (San Francisco: Morgan Kaufmann, 1999)

to answer the questions "Which libraries are linked into this binary?" and "Which of these functions is a library function?" Chapter 13 will discuss the challenges encountered while reverse engineering statically linked code.

Dynamic linking differs from static linking in that the linker has no 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 when dynamic linking is utilized. Since a single copy of a library is maintained and that copy is referenced by many binaries, replacing the single outdated library with a new version results in any new process based on a binary that dynamically links to that library using the updated 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 happens to contain 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#
```

In order 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 required by any executable. In the following example, `ldd` is used 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 it is hopefully 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 may be compromised as a result of SRE activities, along with any data on it or other hosts connected to it.

objdump

Whereas `ldd` is fairly specialized, `objdump` is extremely versatile. The purpose of `objdump` is to "display information from object files[9]." This is a fairly broad goal, and in order to accomplish it, `objdump` responds to a large number (30+) of command line options tailored to extract various pieces of information from object files. `objdump` can be used 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`.

9. See *http://www.sourceware.org/binutils/docs/binutils/objdump.html#objdump/*.

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. Such a 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[10] tool suite and can be found on Linux, FreeBSD, and Windows (via WSL or Cygwin). Note that `objdump` relies on the Binary File Descriptor library (libbfd), a component of binutils, to access object files and thus is capable of parsing file formats supported by libbfd (ELF and PE among others). For ELF-specific parsing, a utility named `readelf` is also available. The `readelf` utility offers most of the same capabilities as `objdump`, and the primary difference between the two is that `readelf` 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 osx_example
osx_example: Mach-O 64-bit executable x86_64
ghidrabook# 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 1281.0.0)
```

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. For more information regarding the capabilities of `otool`, please refer to the associated man page.

dumpbin

The `dumpbin` command line utility is included with Microsoft's Visual Studio suite of tools. Like `otool` and `objdump`, `dumpbin` is capable of displaying a wide range of information related to Windows PE files. The following listing

10. See *http://www.gnu.org/software/binutils/*.

shows how dumpbin displays the dynamic dependencies of the Windows calculator program in a manner similar to ldd:

```
$ dumpbin /dependents C:\Windows\System32\notepad.exe
Microsoft (R) COFF/PE Dumper Version 12.00.40629.0
Copyright (C) Microsoft Corporation.  All rights reserved.

Dump of file notepad.exe

File Type: EXECUTABLE IMAGE

  Image has the following delay load dependencies:

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

  Image has the following dependencies:

    GDI32.dll
    USER32.dll
    msvcrt.dll
    …
```

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. Additional information related to the use of dumpbin is available via the Microsoft website.[11]

c++filt

Languages that allow function overloading must have a mechanism for distinguishing among 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, it is not possible to have two functions with the same name in an object file. In order to allow overloading, compilers derive unique names for overloaded functions by incorporating information describing the type sequence of the function arguments. The process of deriving unique

11. See *https://docs.microsoft.com/en-us/cpp/build/reference/dumpbin-command-line.*

names for functions with identical names is called *name mangling*.[12] 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. In order to decipher the mangled variants of `demo` shown here, we need a tool that understands our compiler's (g++ in this case) name-mangling scheme. This is precisely the purpose of the `c++filt` utility. `c++filt` treats each input word as if it were a mangled name and then attempts to determine the compiler that was 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`, it is possible to recover the demangled function names, as seen 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 mangled names contain additional information about functions that `nm` does not normally provide. This information can be extremely helpful in reversing engineering situations, and in more complex cases, this extra information 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 structure. 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

12. For an overview of name mangling, refer to *http://en.wikipedia.org/wiki/Name_mangling*.

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—questions that don't 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 are generally not limited in any way by the structure of a file. 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 (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
…
```

WHY DID STRINGS CHANGE?

Historically, when `strings` was used on executable files it would, by default, only search for character sequences in the loadable, initialized data sections of the binary file. This required that `strings` parse the binary file to find those sections, using libraries such as libbfd. When it was used for parsing untrusted binaries, vulnerabilities in libraries[13] 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.

13. See CVE-2014-8485 and *http://lcamtuf.blogspot.com/2014/10/psa-dont-run-strings-on-untrusted-files.html*.

Unfortunately, while we see some strings that look like they might be output by the program, other strings appear to be function names and library names. We should be careful not to jump to any conclusions regarding the behavior of the program. Analysts often fall into the trap of attempting to deduce the behavior of a program based on the output of `strings`. Remember, the presence of a string within a binary in no way indicates that the string is ever used in any manner by that binary.

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. Use the `-t` command-line argument to have `strings` print file offset information for each string found.
- Many files utilize alternate character sets. Utilize the `-e` command-line argument to cause `strings` to search for wide characters such as 16-bit Unicode.

Disassemblers

As mentioned earlier, a number of tools are available to generate dead listing–style disassemblies of binary object files. PE, ELF, and Mach-O binaries can be disassembled using `dumpbin`, `objdump`, and `otool`, respectively. None of those, however, 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 `diStorm`.[14] `ndisasm` is a utility included with the Netwide Assembler (NASM).[15] The following example illustrates the use of `ndis-`

14. See *https://github.com/gdabah/distorm*.

15. See *http://www.nasm.us*.

asm to disassemble a piece of shellcode generated using the Metasploit framework.[16]

```
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 be used to disassemble the portions of the packet that contain shellcode in order to analyze the behavior of the malicious payload. 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.

16. See *https://metasploit.com/*.

Summary

The tools discussed in this chapter are not necessarily the best of their breed. They do, however, represent tools commonly available for anyone who wishes to reverse engineer binary files. More important, they represent the types of tools that motivated much of the development of Ghidra. In future chapters, we will 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.