5

BASIC BINARY ANALYSIS IN LINUX

Even in the most complex binary analysis, you can accomplish surprisingly advanced feats by combining a set of basic tools in the right way. This can save you hours of work implementing equivalent functionality on your own. In this chapter, you'll learn the fundamental tools you'll need to perform binary analysis on Linux.

Instead of simply showing you a list of tools and explaining what they do, I'll use a *Capture the Flag (CTF)* challenge to illustrate how they work. In computer security and hacking, CTF challenges are often played as contests, where the goal is typically to analyze or exploit a given binary (or a running process or server) until you manage to capture a flag hidden in the binary. The flag is usually a hexadecimal string, which you can use to prove that you completed the challenge as well as unlock new challenges.

In this CTF, you start with a mysterious file called *payload*, which you can find on the VM in the directory for this chapter. The goal is to figure out how to extract the hidden flag from *payload*. In the process of analyzing *payload* and looking for the flag, you'll learn to use a wide range of basic binary analysis tools that are available on virtually any Linux-based system (most of them as part of GNU `coreutils` or `binutils`). I encourage you to follow along.

Most of the tools you'll see have a number of useful options, but there are far too many to cover exhaustively in this chapter. Thus, it's a good idea to check out the man page for every tool using the command `man tool` on the VM. At the end of the chapter, you'll use the recovered flag to unlock a new challenge, which you can complete on your own!

5.1 Resolving Identity Crises Using file

Because you received absolutely no hints about the contents of *payload*, you have no idea what to do with this file. When this happens (for instance, in reverse engineering or forensics scenarios), a good first step is to figure out what you can about the file type and its contents. The `file` utility was designed for this purpose; it takes a number of files as input and then tells you what type each file is. You may remember it from Chapter 2, where I used `file` to find out the type of an ELF file.

The nice thing about `file` is that it isn't fooled by extensions. Instead, it searches for other telltale patterns in the file, such as magic bytes like the `0x7f ELF` sequence at the start of ELF files, to find out the file type. This is perfect here because the *payload* file doesn't have an extension. Here's what `file` tells you about *payload*:

```
$ file payload
payload: ASCII text
```

As you can see, *payload* contains ASCII text. To examine the text in detail, you can use the `head` utility, which dumps the first few lines (10 by default) of a text file to `stdout`. There's also an analogous utility called `tail`, which shows you the last few lines of a file. Here's what the `head` utility's output shows:

```
$ head payload
H4sIAKiT61gAA+xaD3RTVZq/Sf9TSKL8aflnn56ioNJJSiktDpqUlL5oOUpbYEVIOzRtI2naSV5K
YVOHTig21jqojH9mnRV35syZPWd35ZzZOOXHxWBHYJydXf4ckRldZRUxBRzxz2CFQvb77ru3ee81
AZdZZ92z+XrS733fu993v/v/vnt/bqmVfNNkBlqOcCFyy6KFZiUHKi1buMhMLAvMiOoXWSzlZYtA
v2hRWRkRzN94ZEChoOQKCAJp8fdcNt2V3v8fpe9X1y7T63Rjsp7cTlCKGq1UtjL9yPUJGyupIHnw
/zoym2SDnKVIZyVWFR9hrjnPZeky4JcJvwq9LFforSo+i6XjXKfgWaoSWFX8mclExQkRxuww1uOz
Ze3x2UOqfpDFcUyvttMzuxFmN8LScO54er26fJns18DODaxcnNtZOrsiPVLdh1ILPudey/xda1Xx
MpauTGN3L9hlk69PJsZXsPxS1YvA4uect8N3fN7m8rLv+Frm+7z+UM/8nory+eVlJcHOklIak4ml
rbm7kabn9SiwmKcQuQ/g+3n/OJj/byfuqjvO9uKVj8889O6TvxXM+G4qSbRbX1TQCZnWPNQVwG86
/F7+4IkHl1a/eebY91bPemngU8OpI58YNjrWD16u3P3wuzaJ3kh4i6vpuhT6g7rkfs6kODtS6P8l
hf6NFPocfXL9yRTpSOny+NtJ8vR3pOhfl8J/bgr9VynOb6bQkxTl+ixF+p+mON+qx743k+wWmlT6
```

That definitely doesn't look human-readable. Taking a closer look at the alphabet used in the file, you can see that it consists of only alphanumeric characters and the characters + and /, organized in neat rows. When you see a file that looks like this, it's usually safe to assume that it's a *Base64* file.

Base64 is a widely used method of encoding binary data as ASCII text. Among other things, it's commonly used in email and on the web to ensure that binary data transmitted over a network isn't accidentally malformed by services that can handle only text. Conveniently, Linux systems come with a tool called `base64` (typically as part of GNU `coreutils`) that can encode and decode Base64. By default, `base64` will encode any files or `stdin` input given to it. But you can use the `-d` flag to tell `base64` to decode instead. Let's decode *payload* to see what you get!

```
$ base64 -d payload > decoded_payload
```

This command decodes *payload* and then stores the decoded contents in a new file called `decoded_payload`. Now that you've decoded *payload*, let's use `file` again to check the type of the decoded file.

```
$ file decoded_payload
decoded_payload: gzip compressed data, last modified: Tue Oct 22 15:46:43 2019, from Unix
```

Now you're getting somewhere! It turns out that behind the layer of Base64 encoding, the mysterious file is actually just a compressed archive that uses `gzip` as the outer compression layer. This is an opportunity to introduce another handy feature of `file`: the ability to peek inside zipped files. You can pass the `-z` option to `file` to see what's inside the archive without extracting it. Here's what you should see:

```
$ file -z decoded_payload
decoded_payload: POSIX tar archive (GNU) (gzip compressed data, last modified:
                 Tue Oct 22 19:08:12 2019, from Unix)
```

You can see that you're dealing with multiple layers that you need to extract, because the outer layer is a `gzip` compression layer and inside that is a `tar` archive, which typically contains a bundle of files. To reveal the files stored inside, you use `tar` to unzip and extract `decoded_payload`, like this:

```
$ tar xvzf decoded_payload
ctf
67b8601
```

As shown in the `tar` log, there are two files extracted from the archive: *ctf* and *67b8601*. Let's use `file` again to see what kinds of files you're dealing with.

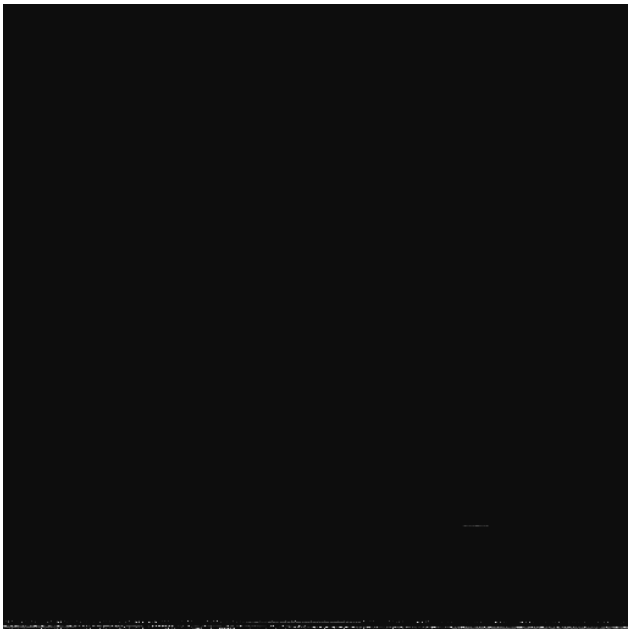
```
$ file ctf
ctf: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32,
BuildID[sha1]=29aeb60bcee44b50d1db3a56911bd1de93cd2030, stripped
```

The first file, *ctf*, is a dynamically linked 64-bit stripped ELF executable. The second file, called *67b8601*, is a bitmap (BMP) file of 512 × 512 pixels. Again, you can see this using `file` as follows:

```
$ file 67b8601
67b8601: PC bitmap, Windows 3.x format, 512 x 512 x 24
```

This BMP file depicts a black square, as you can see in Figure 5-1a. If you look carefully, you should see some irregularly colored pixels at the bottom of the figure. Figure 5-1b shows an enlarged snippet of these pixels.

Before exploring what this all means, let's first take a closer look at *ctf*, the ELF file you just extracted.



(a) The complete figure

(b) Enlarged view of some of the colored pixels at the bottom

Figure 5-1: The extracted BMP file, 67b8601

5.2 Using ldd to Explore Dependencies

Although it's not wise to run unknown binaries, since you're working in a VM, let's try running the extracted *ctf* binary. When you try to run the file, you don't get far.

```
$ ./ctf
./ctf: error while loading shared libraries: lib5ae9b7f.so:
       cannot open shared object file: No such file or directory
```

Before any of the application code is even executed, the dynamic linker complains about a missing library called *lib5ae9b7f.so*. That doesn't sound like a library you normally find on any system. Before searching for this library, it makes sense to check whether *ctf* has any more unresolved dependencies.

Linux systems come with a program called `ldd`, which you can use to find out on which shared objects a binary depends and where (if anywhere) these dependencies are on your system. You can even use `ldd` along with the `-v` flag to find out which library versions the binary expects, which can be useful for debugging. As mentioned in the `ldd man` page, `ldd` may run the binary to figure out the dependencies, so it's not safe to use on untrusted binaries unless you're running it in a VM or another isolated environment. Here's the `ldd` output for the *ctf* binary:

```
$ ldd ctf
        linux-vdso.so.1 =>  (0x00007fff6edd4000)
        lib5ae9b7f.so => not found
        libstdc++.so.6 => /usr/lib/x86_64-linux-gnu/libstdc++.so.6 (0x00007f67c2cbe000)
        libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007f67c2aa7000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f67c26de000)
        libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007f67c23d5000)
        /lib64/ld-linux-x86-64.so.2 (0x0000561e62fe5000)
```

Luckily, there are no unresolved dependencies besides the missing library identified earlier, *lib5ae9b7f.so*. Now you can focus on figuring out what this mysterious library is and how you can obtain it in order to capture the flag!

Because it's obvious from the library name that you won't find it in any standard repository, it must reside somewhere in the files you've been given so far. Recall from Chapter 2 that all ELF binaries and libraries begin with the magic sequence `0x7f ELF`. This is a handy string to look for in search of your missing library; as long as the library is not encrypted, you should be able to find the ELF header this way. Let's try a simple `grep` for the string `'ELF'`.

```
$ grep 'ELF' *
Binary file 67b8601 matches
Binary file ctf matches
```

As expected, the string `'ELF'` appears in *ctf*, which is not surprising because you already know it's an ELF binary. But you can see that this string is also in *67b8601*, which at first glance appeared to be an innocent bitmap file. Could there be a shared library hidden within the bitmap's pixel data? It would certainly explain those strangely colored pixels you saw in Figure 5-1b! Let's examine the contents of *67b8601* in more detail to find out.

Quickly Looking Up ASCII Codes

When interpreting raw bytes as ASCII, you'll often need a table that maps byte values in various representations to ASCII symbols. You can use a special man page called `man ascii` for quick access to such a table. Here's an excerpt of the table from `man ascii`:

```
Oct   Dec   Hex   Char                        Oct   Dec   Hex   Char
──────────────────────────────────────────────────────────────────────
000   0     00    NUL '\0' (null character)   100   64    40    @
001   1     01    SOH (start of heading)      101   65    41    A
002   2     02    STX (start of text)         102   66    42    B
003   3     03    ETX (end of text)           103   67    43    C
004   4     04    EOT (end of transmission)   104   68    44    D
005   5     05    ENQ (enquiry)               105   69    45    E
006   6     06    ACK (acknowledge)           106   70    46    F
007   7     07    BEL '\a' (bell)             107   71    47    G
...
```

As you can see, this is an easy way to look up the mappings from octal, decimal, and hexadecimal encodings to ASCII characters. This is much faster than googling for an ASCII table!

5.3 Viewing File Contents with xxd

To discover exactly what's in a file without being able to rely on any standard assumptions about the file contents, you'll have to analyze it at the byte level. To do this, you can use any numeric system to display bits and bytes on the screen. For instance, you could use the binary system, displaying all the ones and zeros individually. But because that makes for some tedious analysis, it's better to use the *hexadecimal system*. In the hexadecimal system (also known as *base 16*, or *hex* for short), digits range from 0 to 9 (with the usual meaning) and then from *a* to *f* (where *a* represents the value 10 and *f* represents 15). In addition, because a byte has $256 = 16 \times 16$ possible values, it fits exactly in two hexadecimal digits, making this a convenient encoding for compactly displaying bytes.

To display the bytes of a file in hexadecimal representation, you use a *hex-dumping* program. A *hex editor* is a program that can also edit the bytes

in the file. I'll get back to hex editing in Chapter 7, but for now let's use a simple hex-dumping program called `xxd`, which is installed on most Linux systems by default.

Here are the first 15 lines of output from `xxd` for the bitmap file you're analyzing:

```
$ xxd 67b8601 | head -n 15
00000000: 424d 3800 0c00 0000 0000 3600 0000 2800  BM8.......6...(.
00000010: 0000 0002 0000 0002 0000 0100 1800 0000  ................
00000020: 0000 0200 0c00 c01e 0000 c01e 0000 0000  ................
00000030: 0000 0000 ❶7f45 4c46 0201 0100 0000 0000  .....ELF........
00000040: 0000 0000 0300 3e00 0100 0000 7009 0000  ......>.....p...
00000050: 0000 0000 4000 0000 0000 0000 7821 0000  ....@.......x!..
00000060: 0000 0000 0000 0000 4000 3800 0700 4000  ........@.8...@.
00000070: 1b00 1a00 0100 0000 0500 0000 0000 0000  ................
00000080: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000090: 0000 0000 f40e 0000 0000 0000 f40e 0000  ................
000000a0: 0000 0000 0000 2000 0000 0000 0100 0000  ......  ........
000000b0: 0600 0000 f01d 0000 0000 0000 f01d 2000  ..............  .
000000c0: 0000 0000 f01d 2000 0000 0000 6802 0000  ......  .....h...
000000d0: 0000 0000 7002 0000 0000 0000 0000 2000  ....p.........  .
000000e0: 0000 0000 0200 0000 0600 0000 081e 0000  ................
```

As you can see, the first output column shows the offset into the file in hexadecimal format. The next eight columns show hexadecimal representations of the bytes in the file, and on the rightmost side of the output, you can see an ASCII representation of the same bytes.

You can change the number of bytes displayed per line using the `xxd` program's `-c` option. For instance, `xxd -c 32` will display 32 bytes per line. You can also use `-b` to display binary instead of hexadecimal, and you can use `-i` to output a C-style array containing the bytes, which you can directly include in your C or C++ source. To output only some of the bytes, you can use the `-s` (seek) option to specify a file offset at which to start, and you can use the `-l` (length) option to specify the number of bytes to dump.

In the `xxd` output for the bitmap file, the ELF magic bytes appear at offset `0x34` ❶, which corresponds to 52 in the decimal system. This tells you where in the file the suspected ELF library begins. Unfortunately, finding out where it ends is not so trivial because there are no magic bytes delimiting the end of an ELF file. Thus, before you try to extract the complete ELF file, begin by extracting only the ELF header instead. This is easier since you know that 64-bit ELF headers contain exactly 64 bytes. You can then examine the ELF header to figure out how large the complete file is.

To extract the header, you use `dd` to copy 64 bytes from the bitmap file, starting at offset 52, into a new output file called *elf_header*.

```
$ dd skip=52 count=64 if=67b8601 of=elf_header bs=1
64+0 records in
```

```
64+0 records out
64 bytes copied, 0.000404841 s, 158 kB/s
```

Using `dd` is incidental here, so I won't explain it in detail. However, `dd` is an extremely versatile[1] tool, so it's worth reading its man page if you aren't already familiar with it.

Let's use `xxd` again to see whether it worked.

```
$ xxd elf_header
00000000: ❶7f45 4c46 0201 0100 0000 0000 0000 0000  .ELF............
00000010: 0300 3e00 0100 0000 7009 0000 0000 0000  ..>.....p.......
00000020: 4000 0000 0000 0000 7821 0000 0000 0000  @.......x!......
00000030: 0000 0000 4000 3800 0700 4000 1b00 1a00  ....@.8...@.....
```

That looks like an ELF header! You can clearly see the magic bytes at the start ❶, and you can also see that the `e_ident` array and other fields look reasonable (refer to Chapter 2 for a description of these fields).

5.4 Parsing the Extracted ELF with readelf

To view the details of the ELF header you just extracted, it would be great if you could use `readelf`, like you did in Chapter 2. But will `readelf` work on a broken ELF file that contains nothing but a header? Let's find out in Listing 5-1!

Listing 5-1: The `readelf` output for the extracted ELF header

```
❶ $ readelf -h elf_header
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x970
  Start of program headers:          64 (bytes into file)
❷ Start of section headers:          8568 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         7
```

1. And dangerous! It's so easy to accidentally overwrite crucial files with `dd` that the letters *dd* have often been said to stand for *destroy disk*. Needless to say, use this command with caution.

```
❸  Size of section headers:           64 (bytes)
❹  Number of section headers:         27
   Section header string table index: 26
readelf: Error: Reading 0x6c0 bytes extends past end of file for section headers
readelf: Error: Reading 0x188 bytes extends past end of file for program headers
```

The `-h` option ❶ tells `readelf` to print only the executable header. It still complains that the offsets to the section header table and program header table point outside the file, but that's okay. What matters is that you now have a convenient representation of the extracted ELF header.

Now, how can you figure out the size of the complete ELF using nothing but the executable header? In Figure 2-1 of Chapter 2, you learned that the last part of an ELF file is typically the section header table and that the offset to the section header table is given in the executable header ❷. The executable header also tells you the size of each section header ❸ and the number of section headers in the table ❹. This means you can calculate the size of the complete ELF library hidden in your bitmap file as follows:

$$\begin{aligned} size &= e\_shoff + (e\_shnum \times e\_shentsize) \\ &= 8,568 + (27 \times 64) \\ &= 10,296 \end{aligned}$$

In this equation, *size* is the size of the complete library, *e_shoff* is the offset to the section header table, *e_shnum* is the number of section headers in the table, and *e_shentsize* is the size of each section header.

Now that you know that the size of the library should be 10,296 bytes, you can use `dd` to extract it completely, as follows:

```
$ dd skip=52 count=10296 if=67b8601 ❶of=lib5ae9b7f.so bs=1
10296+0 records in
10296+0 records out
10296 bytes (10 kB, 10 KiB) copied, 0.0287996 s, 358 kB/s
```

The `dd` command calls the extracted file *lib5ae9b7f.so* ❶ because that's the name of the missing library the *ctf* binary expects. After running this command, you should now have a fully functioning ELF shared object. Let's use `readelf` to see whether all went well, as shown in Listing 5-2. To keep the output brief, let's only print the executable header (`-h`) and symbol tables (`-s`). The latter should give you an idea of the functionality that the library provides.

Listing 5-2: The `readelf` output for the extracted library, lib5ae9b7f.so

```
$ readelf -hs lib5ae9b7f.so
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
```

```
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x970
  Start of program headers:          64 (bytes into file)
  Start of section headers:          8568 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         7
  Size of section headers:           64 (bytes)
  Number of section headers:         27
  Section header string table index: 26

Symbol table '.dynsym' contains 22 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx Name
     0: 0000000000000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000000008c0     0 SECTION LOCAL  DEFAULT    9
     2: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND __gmon_start__
     3: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _Jv_RegisterClasses
     4: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND _ZNSt7__cxx1112basic_stri@GL(2)
     5: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND malloc@GLIBC_2.2.5 (3)
     6: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_deregisterTMCloneTab
     7: 0000000000000000     0 NOTYPE  WEAK   DEFAULT  UND _ITM_registerTMCloneTable
     8: 0000000000000000     0 FUNC    WEAK   DEFAULT  UND __cxa_finalize@GLIBC_2.2.5 (3)
     9: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND __stack_chk_fail@GLIBC_2.4 (4)
    10: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND _ZSt19__throw_logic_error@ (5)
    11: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND memcpy@GLIBC_2.14 (6)
❶   12: 0000000000000bc0   149 FUNC    GLOBAL DEFAULT   12 _Z11rc4_encryptP11rc4_sta
❷   13: 0000000000000cb0   112 FUNC    GLOBAL DEFAULT   12 _Z8rc4_initP11rc4_state_t
    14: 0000000000202060     0 NOTYPE  GLOBAL DEFAULT   24 _end
    15: 0000000000202058     0 NOTYPE  GLOBAL DEFAULT   23 _edata
❸   16: 0000000000000b40   119 FUNC    GLOBAL DEFAULT   12 _Z11rc4_encryptP11rc4_sta
❹   17: 0000000000000c60     5 FUNC    GLOBAL DEFAULT   12 _Z11rc4_decryptP11rc4_sta
    18: 0000000000202058     0 NOTYPE  GLOBAL DEFAULT   24 __bss_start
    19: 00000000000008c0     0 FUNC    GLOBAL DEFAULT    9 _init
❺   20: 0000000000000c70    59 FUNC    GLOBAL DEFAULT   12 _Z11rc4_decryptP11rc4_sta
    21: 0000000000000d20     0 FUNC    GLOBAL DEFAULT   13 _fini
```

As hoped, the complete library seems to have been extracted correctly. Although it's stripped, the dynamic symbol table does reveal some interesting exported functions (❶ through ❺). However, there seems to be some gibberish around the names, making them difficult to read. Let's see if that can be fixed.

5.5 Parsing Symbols with nm

C++ allows functions to be *overloaded*, which means there may be multiple functions with the same name, as long as they have different signatures. Unfortunately for the linker, it doesn't know anything about C++. For example, if there are multiple functions with the name `foo`, the linker has no idea how to resolve references to `foo`; it simply doesn't know which version of `foo` to use. To eliminate duplicate names, C++ compilers emit *mangled* function names. A mangled name is essentially a combination of the original function name and an encoding of the function parameters. This way, each version of the function gets a unique name, and the linker has no problems disambiguating the overloaded functions.

For binary analysts, mangled function names are a mixed blessing. On the one hand, mangled names are more difficult to read, as you saw in the `readelf` output for *lib5ae9b7f.so* (Listing 5-2), which is programmed in C++. On the other hand, mangled function names essentially provide free type information by revealing the expected parameters of the function, and this information can be useful when reverse engineering a binary.

Fortunately, the benefits of mangled names outweigh the downsides because mangled names are relatively easy to *demangle*. There are several standard tools you can use to demangle mangled names. One of the best known is `nm`, which lists symbols in a given binary, object file, or shared object. When given a binary, `nm` by default attempts to parse the static symbol table.

```
$ nm lib5ae9b7f.so
nm: lib5ae9b7f.so: no symbols
```

Unfortunately, as this example shows, you can't use `nm`'s default configuration on *lib5ae9b7f.so* because it has been stripped. You have to explicitly ask `nm` to parse the dynamic symbol table instead, using the `-D` switch, as shown in Listing 5-3. In this listing, "…" indicates that I've truncated a line and continued it on the next line (mangled names can be quite long).

Listing 5-3: The `nm` output for lib5ae9b7f.so

```
$ nm -D lib5ae9b7f.so
                 w _ITM_deregisterTMCloneTable
                 w _ITM_registerTMCloneTable
                 w _Jv_RegisterClasses
0000000000000c60 T _Z11rc4_decryptP11rc4_state_tPhi
0000000000000c70 T _Z11rc4_decryptP11rc4_state_tRNSt7__cxx1112basic_...
                   ...stringIcSt11char_traitsIcESaIcEEE
0000000000000b40 T _Z11rc4_encryptP11rc4_state_tPhi
0000000000000bc0 T _Z11rc4_encryptP11rc4_state_tRNSt7__cxx1112basic_...
                   ...stringIcSt11char_traitsIcESaIcEEE
0000000000000cb0 T _Z8rc4_initP11rc4_state_tPhi
                 U _ZNSt7__cxx1112basic_stringIcSt11char_traitsIcESaIcEE9_...
```

```
                    ...M_createERmm
                  U _ZSt19__throw_logic_errorPKc
0000000000202058 B __bss_start
                  w __cxa_finalize
                  w __gmon_start__
                  U __stack_chk_fail
0000000000202058 D _edata
0000000000202060 B _end
0000000000000d20 T _fini
00000000000008c0 T _init
                  U malloc
                  U memcpy
```

This looks better; this time you see some symbols. But the symbol names are still mangled. To demangle them, you have to pass the `--demangle` switch to `nm`, as shown in Listing 5-4.

Listing 5-4: Demangled `nm` output for lib5ae9b7f.so

```
$ nm -D --demangle lib5ae9b7f.so
                  w _ITM_deregisterTMCloneTable
                  w _ITM_registerTMCloneTable
                  w _Jv_RegisterClasses
0000000000000c60 T ❶rc4_decrypt(rc4_state_t*, unsigned char*, int)
0000000000000c70 T ❷rc4_decrypt(rc4_state_t*,
                                std::__cxx11::basic_string<char, std::char_traits<char>,
                                std::allocator<char> >&)
0000000000000b40 T ❸rc4_encrypt(rc4_state_t*, unsigned char*, int)
0000000000000bc0 T ❹rc4_encrypt(rc4_state_t*,
                                std::__cxx11::basic_string<char, std::char_traits<char>,
                                std::allocator<char> >&)
0000000000000cb0 T ❺rc4_init(rc4_state_t*, unsigned char*, int)
                  U std::__cxx11::basic_string<char, std::char_traits<char>,
                       std::allocator<char> >::_M_create(unsigned long&, unsigned long)
                  U std::__throw_logic_error(char const*)
0000000000202058 B __bss_start
                  w __cxa_finalize
                  w __gmon_start__
                  U __stack_chk_fail
0000000000202058 D _edata
0000000000202060 B _end
0000000000000d20 T _fini
00000000000008c0 T _init
                  U malloc
                  U memcpy
```

Finally, the function names appear human-readable. You can see five interesting functions, which appear to be cryptographic functions

implementing the well-known RC4 encryption algorithm.[2] There's a function called `rc4_init`, which takes as input a data structure of type `rc4_state_t`, as well as an unsigned character string and an integer ❺. The first parameter is presumably a data structure where the cryptographic state is kept, while the next two are probably a string representing a key and an integer specifying the length of the key, respectively. You can also see several encryption and decryption functions, each of which takes a pointer to the cryptographic state, as well as parameters specifying strings (both C and C++ strings) to encrypt or decrypt (❶ through ❹).

As an alternative way of demangling function names, you can use a specialized utility called `c++filt`, which takes a mangled name as the input and outputs the demangled equivalent. The advantage of `c++filt` is that it supports several mangling formats and automatically detects the correct mangling format for the given input. Here's an example using `c++filt` to demangle the function name `_Z8rc4_initP11rc4_state_tPhi`:

```
$ c++filt _Z8rc4_initP11rc4_state_tPhi
rc4_init(rc4_state_t*, unsigned char*, int)
```

At this point, let's briefly recap the progress so far. You extracted the mysterious payload and found a binary called *ctf* that depends on a file called *lib5ae9b7f.so*. You found *lib5ae9b7f.so* hidden in a bitmap file and successfully extracted it. You also have a rough idea of what it does: it's a cryptographic library. Now let's try running *ctf* again, this time with no missing dependencies.

When you run a binary, the linker resolves the binary's dependencies by searching a number of standard directories for shared libraries, such as */lib*. Because you extracted *lib5ae9b7f.so* to a nonstandard directory, you need to tell the linker to search that directory too by setting an environment variable called `LD_LIBRARY_PATH`. Let's set this variable to contain the current working directory and then try launching *ctf* again.

```
$ export LD_LIBRARY_PATH=`pwd`
$ ./ctf
$ echo $?
1
```

Success! The *ctf* binary still doesn't appear to do anything useful, but it runs without complaining about any missing libraries. The exit status of *ctf* contained in the `$?` variable is 1, indicating an error. Now that you have all the required dependencies, you can continue your investigation and see whether you can coax *ctf* into getting past the error so that you can reach the flag you're trying to capture.

2. RC4 is a widely used stream cipher, noted for its simplicity and speed. If you're interested, you can find more details about it at *https://en.wikipedia.org/wiki/RC4*. Note that RC4 is now considered broken and should not be used in any new real-world projects!

5.6 Looking for Hints with strings

To figure out what a binary does and what kinds of inputs it expects, you can check whether the binary contains any helpful strings that can reveal its purpose. For instance, if you see strings containing parts of HTTP requests or URLs, you can safely guess that the binary is doing something web related. When you're dealing with malware such as a bot, you might be able to find strings containing the commands that the bot accepts, if they're not obfuscated. You might even find strings left over from debugging that the programmer forgot to remove, which has been known to happen in real-world malware!

You can use a utility called `strings` to check for strings in a binary (or any other file) on Linux. The `strings` utility takes one or more files as input and then prints any printable character strings found in those files. Note that `strings` doesn't check whether the found strings were really intended to be human readable, so when used on binary files, the `strings` output may include some bogus strings as a result of binary sequences that just happen to be printable.

You can tweak the behavior of `strings` using options. For example, you can use the `-d` switch with `strings` to print only strings found in data sections in a binary instead of printing all sections. By default, `strings` prints only strings of four characters or more, but you can specify another minimum string length using the `-n` option. For our purposes, the default options will suffice; let's see what you can find in the *ctf* binary using `strings`, as shown in Listing 5-5.

Listing 5-5: Character strings found in the ctf *binary*

```
$ strings ctf
❶ /lib64/ld-linux-x86-64.so.2
lib5ae9b7f.so
❷ __gmon_start__
_Jv_RegisterClasses
_ITM_deregisterTMCloneTable
_ITM_registerTMCloneTable
_Z8rc4_initP11rc4_state_tPhi
...
❸ DEBUG: argv[1] = %s
❹ checking '%s'
❺ show_me_the_flag
>CMb
-v@P?
flag = %s
guess again!
❻ It's kinda like Louisiana. Or Dagobah. Dagobah - Where Yoda lives!
;*3$"
zPLR
```

```
GCC: (Ubuntu 5.4.0-6ubuntu1~16.04.4) 5.4.0 20160609
❼ .shstrtab
.interp
.note.ABI-tag
.note.gnu.build-id
.gnu.hash
.dynsym
.dynstr
.gnu.version
.gnu.version_r
.rela.dyn
.rela.plt
.init
.plt.got
.text
.fini
.rodata
.eh_frame_hdr
.eh_frame
.gcc_except_table
.init_array
.fini_array
.jcr
.dynamic
.got.plt
.data
.bss
.comment
```

Here, you can see some strings that you'll encounter in most ELF files. For example, there's the name of the program interpreter ❶, as found in the `.interp` section, and some symbolic names found in `.dynstr` ❷. At the end of the `strings` output, you can see all the section names as found in the `.shstrtab` section ❼. But none of these strings is very interesting for the purposes here.

Fortunately, there are also some more useful strings. For example, there is what appears to be a debug message, which suggests that the program expects a command line option ❸. There are also checks of some sort, presumably performed on an input string ❹. You don't yet know what the value of the command line option should be, but you could try some of the other interesting-looking strings, such as `show_me_the_flag` ❺, that might work. There's also a mysterious string ❻ that contains a message whose purpose is unclear. You don't know what the message means at this point, but you do know from your investigation of *lib5ae9b7f.so* that the binary uses RC4 encryption. Perhaps the message is used as an encryption key?

Now that you know that the binary expects a command line option, let's see whether adding an arbitrary option gets you any closer to revealing the flag. For lack of a better guess, let's simply use the string `foobar`, like this:

```
$ ./ctf foobar
checking 'foobar'
$ echo $?
1
```

The binary now does something new. It tells you that it's checking the input string you gave it. But the check doesn't succeed because the binary still exits with an error code after the check. Let's take a gamble and try one of the other interesting-looking strings that you found, such as the string `show_me_the_flag`, which looks promising.

```
$ ./ctf show_me_the_flag
checking 'show_me_the_flag'
ok
$ echo $?
1
```

That did it! The check now appears to succeed. Unfortunately, the exit status is still 1, so there must be something else missing. To make things worse, the `strings` results don't provide any more hints. Let's take a more detailed look at *ctf*'s behavior to determine what to do next, starting with the system and library calls *ctf* makes.

5.7 Tracing System Calls and Library Calls with strace and ltrace

To make forward progress, let's investigate the reason that *ctf* exits with an error code by looking at *ctf*'s behavior just before it exits. There are many ways that you could do this, but one way is to use two tools called `strace` and `ltrace`. These tools show the system calls and library calls, respectively, executed by a binary. Knowing the system and library calls that a binary makes can often give you a good high-level idea of what the program is doing.

Let's start by using `strace` to investigate *ctf*'s system call behavior. In some cases, you may want to attach `strace` to a running process. To do this, you need to use the `-p` *`pid`* option, where *`pid`* is the process ID of the process you want to attach to. However, in this case, it suffices to run *ctf* with `strace` from the start. Listing 5-6 shows the `strace` output for the *ctf* binary (some parts are truncated with "`...`").

Listing 5-6: System calls executed by the `ctf` *binary*

```
$ strace ./ctf show_me_the_flag
❶ execve("./ctf", ["./ctf", "show_me_the_flag"], [/* 73 vars */]) = 0
brk(NULL)                                = 0x1053000
```

```
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f703477e000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
❷ open("/ch3/tls/x86_64/lib5ae9b7f.so", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or ...)
stat("/ch3/tls/x86_64", 0x7ffcc6987ab0) = -1 ENOENT (No such file or directory)
open("/ch3/tls/lib5ae9b7f.so", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
stat("/ch3/tls", 0x7ffcc6987ab0) = -1 ENOENT (No such file or directory)
open("/ch3/x86_64/lib5ae9b7f.so", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
stat("/ch3/x86_64", 0x7ffcc6987ab0) = -1 ENOENT (No such file or directory)
open("/ch3/lib5ae9b7f.so", O_RDONLY|O_CLOEXEC) = 3
❸ read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0p\t\0\0\0\0\0\0"..., 832) = 832
fstat(3, st_mode=S_IFREG|0775, st_size=10296, ...) = 0
mmap(NULL, 2105440, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f7034358000
mprotect(0x7f7034359000, 2097152, PROT_NONE) = 0
mmap(0x7f7034559000, 8192, PROT_READ|PROT_WRITE, ..., 3, 0x1000) = 0x7f7034559000
close(3)                                = 0
open("/ch3/libstdc++.so.6", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, st_mode=S_IFREG|0644, st_size=150611, ...) = 0
mmap(NULL, 150611, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f7034759000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
❹ open("/usr/lib/x86_64-linux-gnu/libstdc++.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0 \235\10\0\0\0\0\0"..., 832) = 832
fstat(3, st_mode=S_IFREG|0644, st_size=1566440, ...) = 0
mmap(NULL, 3675136, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f7033fd6000
mprotect(0x7f7034148000, 2097152, PROT_NONE) = 0
mmap(0x7f7034348000, 49152, PROT_READ|PROT_WRITE, ..., 3, 0x172000) = 0x7f7034348000
mmap(0x7f7034354000, 13312, PROT_READ|PROT_WRITE, ..., -1, 0) = 0x7f7034354000
close(3)                                = 0
open("/ch3/libgcc_s.so.1", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/x86_64-linux-gnu/libgcc_s.so.1", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0p*\0\0\0\0\0\0"..., 832) = 832
fstat(3, st_mode=S_IFREG|0644, st_size=89696, ...) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f7034758000
mmap(NULL, 2185488, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f7033dc0000
mprotect(0x7f7033dd6000, 2093056, PROT_NONE) = 0
mmap(0x7f7033fd5000, 4096, PROT_READ|PROT_WRITE, ..., 3, 0x15000) = 0x7f7033fd5000
close(3)                                = 0
open("/ch3/libc.so.6", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P\t\2\0\0\0\0\0"..., 832) = 832
fstat(3, st_mode=S_IFREG|0755, st_size=1864888, ...) = 0
mmap(NULL, 3967392, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f70339f7000
mprotect(0x7f7033bb6000, 2097152, PROT_NONE) = 0
```

```
mmap(0x7f7033db6000, 24576, PROT_READ|PROT_WRITE, ..., 3, 0x1bf000) = 0x7f7033db6000
mmap(0x7f7033dbc000, 14752, PROT_READ|PROT_WRITE, ..., -1, 0) = 0x7f7033dbc000
close(3)                                = 0
open("/ch3/libm.so.6", O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib/x86_64-linux-gnu/libm.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\0V\0\0\0\0\0\0"..., 832) = 832
fstat(3, st_mode=S_IFREG|0644, st_size=1088952, ...) = 0
mmap(NULL, 3178744, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7f70336ee000
mprotect(0x7f70337f6000, 2093056, PROT_NONE) = 0
mmap(0x7f70339f5000, 8192, PROT_READ|PROT_WRITE, ..., 3, 0x107000) = 0x7f70339f5000
close(3)                                = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f7034757000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f7034756000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f7034754000
arch_prctl(ARCH_SET_FS, 0x7f7034754740) = 0
mprotect(0x7f7033db6000, 16384, PROT_READ) = 0
mprotect(0x7f70339f5000, 4096, PROT_READ) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f7034753000
mprotect(0x7f7034348000, 40960, PROT_READ) = 0
mprotect(0x7f7034559000, 4096, PROT_READ) = 0
mprotect(0x601000, 4096, PROT_READ)     = 0
mprotect(0x7f7034780000, 4096, PROT_READ) = 0
munmap(0x7f7034759000, 150611)          = 0
brk(NULL)                               = 0x1053000
brk(0x1085000)                          = 0x1085000
fstat(1, st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...) = 0
❺ write(1, "checking 'show_me_the_flag'\n", 28checking 'show_me_the_flag'
) = 28
❻ write(1, "ok\n", 3ok
) = 3
❼ exit_group(1) = ?
+++ exited with 1 +++
```

When tracing a program from the start, `strace` includes all the system calls used by the program interpreter to set up the process, making the output quite verbose. The first system call in the output is `execve`, which is called by your shell to launch the program ❶. After that, the program interpreter takes over and starts setting up the execution environment. This involves setting up memory regions and setting the correct memory access permissions using `mprotect`. Additionally, you can see the system calls used to look up and load the required dynamic libraries.

Recall that in Section 5.5, you set the `LD_LIBRARY_PATH` environment variable to tell the dynamic linker to add your current working directory to its search path. This is why you can see the dynamic linker searching for the *lib5ae9b7f.so* library in a number of standard subfolders in your current working directory, until it finally finds the library in the root of your working

directory ❷. When the library is found, the dynamic linker reads it and maps it into memory ❸. The setup process is repeated for other required libraries, such as *libstdc++.so.6* ❹, and it accounts for the vast majority of the `strace` output.

It isn't until the last three system calls that you finally see application-specific behavior. The first system call used by *ctf* itself is `write`, which is used to print `checking 'show_me_the_flag'` to the screen ❺. You see another `write` call to print the string `ok` ❻, and finally, there's a call to `exit_group`, which leads to the exit with status code 1 ❼.

That's all interesting, but how does it help you figure out how to extract the flag from *ctf*? The answer is that it doesn't! In this case, `strace` didn't reveal anything helpful, but I still wanted to show you how it works because it can be useful for understanding a program's behavior. For instance, observing the system calls executed by a program is useful not only for binary analysis but also for debugging.

Looking at *ctf*'s system call behavior didn't help much, so let's try library calls. To view the library calls executed by *ctf*, you use `ltrace`. Because `ltrace` is a close relative of `strace`, it takes many of the same command line options, including `-p` to attach to an existing process. Here, let's use the `-i` option to print the instruction pointer at every library call (this will be useful later). We'll use `-C` to automatically demangle C++ function names. Let's run *ctf* with `ltrace` from the start, as shown in Listing 5-7.

Listing 5-7: Library calls made by the ctf *binary*

```
$ ltrace -i -C ./ctf show_me_the_flag
❶ [0x400fe9] __libc_start_main (0x400bc0, 2, 0x7ffc22f441e8, 0x4010c0 <unfinished ...>
❷ [0x400c44] __printf_chk (1, 0x401158, 0x7ffc22f4447f, 160checking 'show_me_the_flag') = 28
❸ [0x400c51] strcmp ("show_me_the_flag", "show_me_the_flag") = 0
❹ [0x400cf0] puts ("ok"ok) = 3
❺ [0x400d07] rc4_init (rc4_state_t*, unsigned char*, int)
               (0x7ffc22f43fb0, 0x4011c0, 66, 0x7fe979b0d6e0) = 0
❻ [0x400d14] std::__cxx11::basic_string<char, std::char_traits<char>,
               std::allocator<char> >:: assign (char const*)
               (0x7ffc22f43ef0, 0x40117b, 58, 3) = 0x7ffc22f43ef0
❼ [0x400d29] rc4_decrypt (rc4_state_t*, std::__cxx11::basic_string<char,
               std::char_traits<char>, std::allocator<char> >&)
               (0x7ffc22f43f50, 0x7ffc22f43fb0, 0x7ffc22f43ef0, 0x7e889f91) = 0x7ffc22f43f50
❽ [0x400d36] std::__cxx11::basic_string<char, std::char_traits<char>,
               std::allocator<char> >:: _M_assign (std::__cxx11::basic_string<char,
               std::char_traits<char>, std::allocator<char> > const&)
               (0x7ffc22f43ef0, 0x7ffc22f43f50, 0x7ffc22f43f60, 0) = 0
❾ [0x400d53] getenv ("GUESSME") = nil
[0xffffffffffffffff] +++ exited (status 1) +++
```

As you can see, this output from `ltrace` is a lot more readable than the `strace` output because it isn't polluted by all the process setup code.

The first library call is `__libc_start_main` ❶, which is called from the `_start` function to transfer control to the program's `main` function. Once `main` is started, its first library call prints the now familiar `checking ...` string to the screen ❷. The actual check turns out to be a string comparison, which is implemented using `strcmp`, and verifies that the argument given to *ctf* is equal to `show_me_the_flag` ❸. If this is the case, `ok` is printed to the screen ❹.

So far, this is mostly behavior you've seen before. But now you see something new: the RC4 cryptography is initialized through a call to `rc4_init`, which is located in the library you extracted earlier ❺. After that, you see an `assign` to a C++ string, presumably initializing it with an encrypted message ❻. This message is then decrypted with a call to `rc4_decrypt` ❼, and the decrypted message is assigned to a new C++ string ❽.

Finally, there's a call to `getenv`, which is a standard library function used to look up environment variables ❾. You can see that *ctf* expects an environment variable called `GUESSME`! The name of this variable may well be the string that was decrypted earlier. Let's see whether *ctf*'s behavior changes when you set a dummy value for the `GUESSME` environment variable as follows:

```
$ GUESSME='foobar' ./ctf show_me_the_flag
checking 'show_me_the_flag'
ok
guess again!
```

Setting `GUESSME` results in an additional line of output that says `guess again!`. It seems that *ctf* expects `GUESSME` to be set to another specific value. Perhaps another `ltrace` run, as shown in Listing 5-8, will reveal what the expected value is.

Listing 5-8: Library calls made by the ctf *binary after setting the* `GUESSME` *environment variable*

```
$ GUESSME='foobar' ltrace -i -C ./ctf show_me_the_flag
...
[0x400d53] getenv("GUESSME") = "foobar"
❶ [0x400d6e] std::__cxx11::basic_string<char, std::char_traits<char>,
              std::allocator<char> >::assign(char const*)
              (0x7fffc7af2b00, 0x401183, 5, 3) = 0x7fffc7af2b00
❷ [0x400d88] rc4_decrypt(rc4_state_t*, std::__cxx11::basic_string<char,
              std::char_traits<char>, std::allocator<char> >&)
              (0x7fffc7af2b60, 0x7fffc7af2ba0, 0x7fffc7af2b00, 0x401183) = 0x7fffc7af2b60
[0x400d9a] std::__cxx11::basic_string<char, std::char_traits<char>,
              std::allocator<char> >::_M_assign(std::__cxx11::basic_string<char,
              std::char_traits<char>, std::allocator<char> > const&)
              (0x7fffc7af2b00, 0x7fffc7af2b60, 0x7700a0, 0) = 0
[0x400db4] operator delete(void*)(0x7700a0, 0x7700a0, 21, 0) = 0
❸ [0x400dd7] puts("guess again!"guess again!) = 13
[0x400c8d] operator delete(void*)(0x770050, 0x76fc20, 0x7f70f99b3780, 0x7f70f96e46e0) = 0
[0xffffffffffffffff] +++ exited (status 1) +++
```

After the call to `getenv`, *ctf* goes on to assign ❶ and decrypt ❷ another C++ string. Unfortunately, between the decryption and the moment that `guess again` is printed to the screen ❸, you don't see any hints regarding the expected value of `GUESSME`. This tells you that the comparison of `GUESSME` to its expected value is implemented without the use of any library functions. You'll need to take another approach.

5.8 Examining Instruction-Level Behavior Using objdump

Because you know that the value of the `GUESSME` environment variable is checked without using any well-known library functions, a logical next step is to use `objdump` to examine *ctf* at the instruction level to find out what's going on.[3]

From the `ltrace` output in Listing 5-8, you know that the `guess again` string is printed to the screen by a call to `puts` at address `0x400dd7`. Let's focus the `objdump` investigation around this address. It will also help to know the address of the string to find the first instruction that loads it. To find this address, you can look at the `.rodata` section of the *ctf* binary using `objdump -s` to print the full section contents, as shown in Listing 5-9.

Listing 5-9: The contents of ctf*'s* `.rodata` *section as shown by* `objdump`

```
$ objdump -s --section .rodata ctf

ctf:     file format elf64-x86-64

Contents of section .rodata:
 401140 01000200 44454255 473a2061 7267765b  ....DEBUG: argv[
 401150 315d203d 20257300 63686563 6b696e67  1] = %s.checking
 401160 20272573 270a0073 686f775f 6d655f74   '%s'..show_me_t
 401170 68655f66 6c616700 6f6b004f 89df919f  he_flag.ok.O....
 401180 887e009a 5b38babe 27ac0e3e 434d6285  .~..[8..'..>CMb.
 401190 55868954 3848a34d 00192d76 40505e3a  U..T8H.M..-v@P^:
 4011a0 00726200 666c6167 203d2025 730a00❶67  .rb.flag = %s..g
 4011b0 75657373 20616761 696e2100 00000000  uess again!.....
 4011c0 49742773 206b696e 6461206c 696b6520  It's kinda like
 4011d0 4c6f7569 7369616e 612e204f 72204461  Louisiana. Or Da
 4011e0 676f6261 682e2044 61676f62 6168202d  gobah. Dagobah -
 4011f0 20576865 72652059 6f646120 6c697665   Where Yoda live
 401200 73210000 00000000                    s!......
```

Using `objdump` to examine *ctf*'s `.rodata` section, you can see the `guess again` string at address `0x4011af` ❶. Now let's take a look at Listing 5-10, which

3. Remember from Chapter 1 that `objdump` is a simple disassembler that comes with most Linux distributions.

shows the instructions around the `puts` call, to find out what input *ctf* expects for the `GUESSME` environment variable.

Listing 5-10: Instructions checking the value of `GUESSME`

```
$ objdump -d ctf
...
❶  400dc0: 0f b6 14 03     movzx  edx,BYTE PTR [rbx+rax*1]
   400dc4: 84 d2           test   dl,dl
❷  400dc6: 74 05           je     400dcd <_Unwind_Resume@plt+0x22d>
❸  400dc8: 3a 14 01        cmp    dl,BYTE PTR [rcx+rax*1]
   400dcb: 74 13           je     400de0 <_Unwind_Resume@plt+0x240>
❹  400dcd: bf af 11 40 00  mov    edi,0x4011af
❺  400dd2: e8 d9 fc ff ff  call   400ab0 <puts@plt>
   400dd7: e9 84 fe ff ff  jmp    400c60 <_Unwind_Resume@plt+0xc0>
   400ddc: 0f 1f 40 00     nop    DWORD PTR [rax+0x0]
❻  400de0: 48 83 c0 01     add    rax,0x1
❼  400de4: 48 83 f8 15     cmp    rax,0x15
❽  400de8: 75 d6           jne    400dc0 <_Unwind_Resume@plt+0x220>
...
```

The `guess again` string is loaded by the instruction at `0x400dcd` ❹ and is then printed using `puts` ❺. This is the failure case; let's work our way backward from here.

The failure case is reached from a loop that starts at address `0x400dc0`. In each iteration of the loop, it loads a byte from an array (probably a string) into `edx` ❶. The `rbx` register points to the base of this array, while `rax` indexes it. If the loaded byte turns out to be `NULL`, then the `je` instruction at `0x400dc6` jumps to the failure case ❷. This comparison to `NULL` is a check for the end of the string. If the end of the string is reached here, then it's too short to be a match. If the byte is not `NULL`, the `je` falls through to the next instruction, at address `0x400dc8`, which compares the byte in `edx` against a byte in another string, based at `rcx` and indexed by `rax` ❸.

If the two compared bytes match up, then the program jumps to address `0x400de0`, where it increases the string index ❻, and checks whether the string index is equal to `0x15`, the length of the string ❼. If it is, then the string comparison is complete; if not, the program jumps into another iteration of the loop ❽.

From this analysis, you now know that the string based at the `rcx` register is used as a ground truth. The program compares the environment string taken from the `GUESSME` variable against this ground truth. This means that if you can dump the ground truth string, you can find the expected value for `GUESSME`! Because the string is decrypted at runtime and isn't available statically, you'll need to use dynamic analysis to recover it instead of using `objdump`.

5.9 Dumping a Dynamic String Buffer Using gdb

Probably the most used dynamic analysis tool on GNU/Linux is `gdb`, or the GNU Debugger. As the name suggests, `gdb` is mainly for debugging, but it can be used for a variety of dynamic analysis purposes. In fact, it's an extremely versatile tool, and there's no way to cover all of its functionality in this chapter. However, I'll go over some of the most-used features of `gdb` you can use to recover the expected value of `GUESSME`. The best place to look up information on `gdb` is not the man page but *http://www.gnu.org/software/gdb/documentation/*, where you'll find an extensive manual covering all the supported `gdb` commands.

Like `strace` and `ltrace`, `gdb` has the ability to attach to a running process. However, because *ctf* is not a long-running process, you can simply run it with `gdb` from the start. Because `gdb` is an interactive tool, when you start a binary under `gdb`, it's not immediately executed. After printing a startup message with some usage instructions, `gdb` pauses and waits for a command. You can tell that `gdb` is waiting for a command by the command prompt: `(gdb)`.

Listing 5-11 shows the sequence of `gdb` commands needed to find the expected value of the `GUESSME` environment variable. I'll explain each of these commands as I discuss the listing.

Listing 5-11: Finding the expected value of `GUESSME` using `gdb`

```
$ gdb ./ctf
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./ctf...(no debugging symbols found)...done.
❶ (gdb) b *0x400dc8
Breakpoint 1 at 0x400dc8
❷ (gdb) set env GUESSME=foobar
❸ (gdb) run show_me_the_flag
Starting program: /home/binary/code/chapter3/ctf show_me_the_flag
checking 'show_me_the_flag'
ok
```

```
❹ Breakpoint 1, 0x0000000000400dc8 in ?? ()
❺ (gdb) display/i $pc
  1: x/i $pc
  => 0x400dc8:    cmp    (%rcx,%rax,1),%dl
❻ (gdb) info registers rcx
  rcx            0x615050 6377552
❼ (gdb) info registers rax
  rax            0x0      0
❽ (gdb) x/s 0x615050
  0x615050:      "Crackers Don't Matter"
❾ (gdb) quit
```

One of the most basic functions of any debugger is setting a *breakpoint*, which is simply an address or a function name at which the debugger will "break" execution. Whenever the debugger reaches a breakpoint, it pauses execution and returns control to the user, waiting for a command. To dump the "magic" string against which the `GUESSME` environment variable is compared, you set a breakpoint at address `0x400dc8` ❶ where the comparison happens. In `gdb`, the command for setting a breakpoint at an address is `b *address` (`b` is a short version of the command `break`). If symbols are available (they aren't in this case), you can set a breakpoint at the entry point of a function using the function's name. For instance, to set a breakpoint at the start of `main`, you would use the command `b main`.

After setting the breakpoint, you need to do one more thing before you can start the execution of *ctf*. You still need to set a value for the `GUESSME` environment variable to prevent *ctf* from exiting prematurely. In `gdb`, you can set the `GUESSME` environment variable using the command `set env GUESSME=foobar` ❷. Now, you can begin the execution of *ctf* by issuing the command `run show_me_the_flag` ❸. As you can see, you can pass arguments to the `run` command, which it then automatically passes on to the binary you're analyzing (in this case, *ctf*). Now, *ctf* begins executing normally, and it should continue doing so until it hits your breakpoint.

When *ctf* hits the breakpoint, `gdb` halts the execution of *ctf* and returns control to you, informing you that a breakpoint was hit ❹. At this point, you can use the `display/i $pc` command to display the instruction at the current program counter (`$pc`), just to make sure you're at the expected instruction ❺. As expected, `gdb` informs you that the next instruction to be executed is `cmp (%rcx,%rax,1),%dl`, which is indeed the comparison instruction you're interested in (in AT&T format).

Now that you've reached the point in *ctf*'s execution where `GUESSME` is compared against the expected string, you need to find out the base address of the string so that you can dump it. To view the base address contained in the `rcx` register, use the command `info registers rcx` ❻. You can also view the contents of `rax`, just to ensure that the loop counter is zero, as expected ❼. It's also possible to use the command `info registers` without specifying any register name. In that case, `gdb` will show the contents of all general-purpose registers.

You now know the base address of the string you want to dump; it starts at address `0x615050`. The only thing left to do is to dump the string at that address. The command to dump memory in `gdb` is `x`, which is capable of dumping memory in many granularities and encodings. For instance, `x/d` dumps a single byte in decimal representation, `x/x` dumps a byte in hexadecimal representation, and `x/4xw` dumps four hexadecimal words (which are 4-byte integers). In this case, the most useful version of the command is `x/s`, which dumps a C-style string, continuing until it encounters a `NULL` byte. When you issue the command `x/s 0x615050` to dump the string you're interested in ❽, you can see that the expected value of `GUESSME` is `Crackers Don't Matter`. Let's exit `gdb` using the `quit` command ❾ to try it!

```
$ GUESSME="Crackers Don't Matter" ./ctf show_me_the_flag
checking 'show_me_the_flag'
ok
flag = 84b34c124b2ba5ca224af8e33b077e9e
```

As this listing shows, you've finally completed all the necessary steps to coax *ctf* into giving you the secret flag! On the VM in the directory for this chapter, you'll find a program called *oracle*. Go ahead and feed the flag to *oracle*, like this: `./oracle 84b34c124b2ba5ca224af8e33b077e9e`. You've now unlocked the next challenge, which you can complete on your own using your new skills.

5.10 Summary

In this chapter, I introduced you to all the essential Linux binary analysis tools you need to be an effective binary analyst. While most of these tools are simple enough, you can combine them to implement powerful binary analyses in no time! In the next chapter, you'll explore some of the major disassembly tools and other, more advanced analysis techniques.

Exercise

1. A New CTF Challenge

Complete the new CTF challenge unlocked by the *oracle* program! You can complete the entire challenge using only the tools discussed in this chapter and what you learned in Chapter 2. After completing the challenge, don't forget to give the flag you found to the oracle to unlock the next challenge.