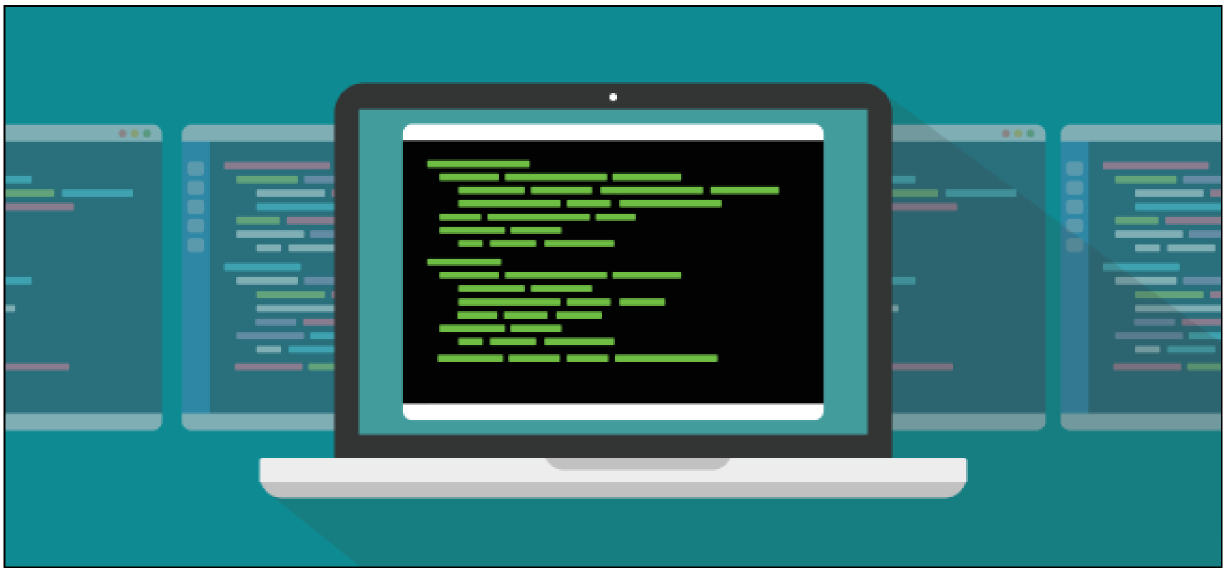
How to Peek Inside Binary Files From the Linux Command Line


fatmawati achmad zaenuri/Shutterstock

Have a mystery file? The Linux `file` command will quickly tell you what type of file it is. If it's a binary file, though, you can find out even more about it. `file` has a whole raft of stablemates that will help you analyze it. We'll show you how to use some of these tools.

Identifying File Types

Files usually have characteristics that allow software packages to identify which type of file it is, as well as what the data within it represents. It wouldn't make sense to try to open a PNG file in an MP3 music player, so it's both useful and pragmatic that a file carries with it some form of ID.

This might be a few signature bytes at the very beginning of the file. This allows a file to be explicit about its format and content. Sometimes, the file type is inferred from a distinctive aspect of the internal organization of the data itself, known as the file architecture.

Some operating systems, like Windows, are completely guided by a file's extension. You can call it gullible or trusting, but Windows assumes any file with the DOCX extension really is a DOCX word processing file. Linux isn't like that, as you'll soon see. It wants proof and looks inside the file to find it.

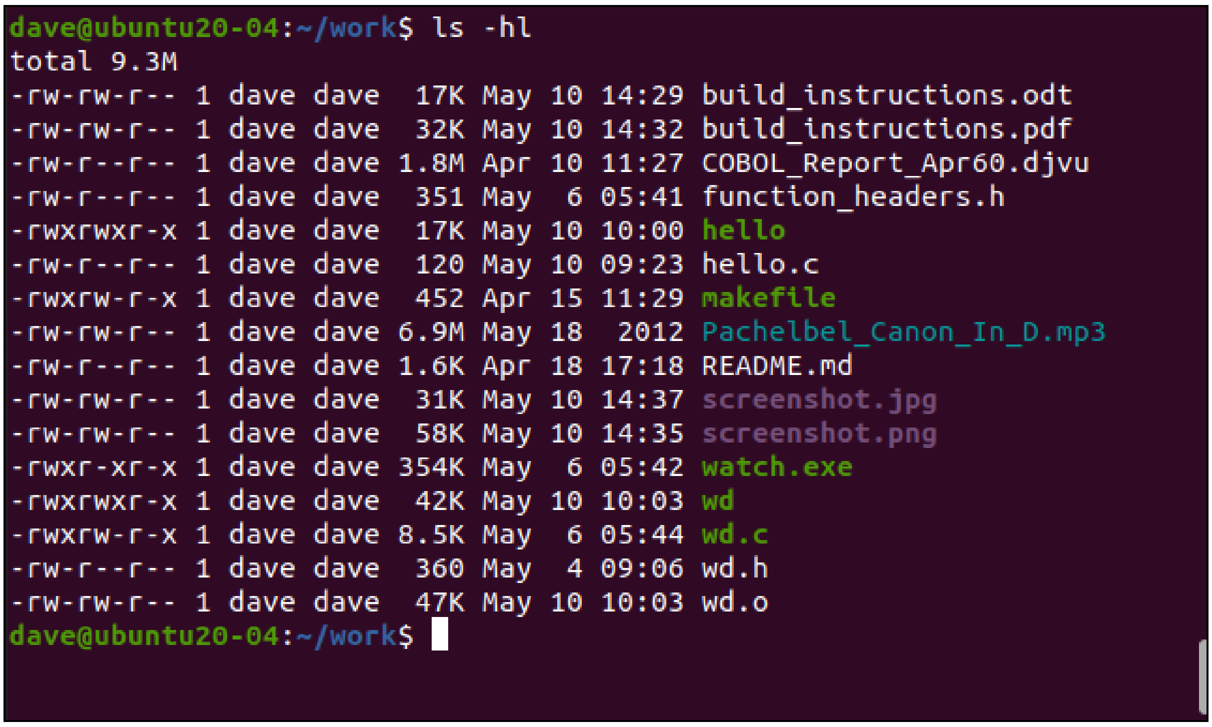
The tools described here were already installed on the Manjaro 20, Fedora 21, and Ubuntu 20.04 distributions we used to research this article. Let's start our investigation by using the `file` command.

Using the file Command

We've got a collection of different file types in our current directory. They're a mixture of document, source code, executable, and text files.

The `ls` command will show us what's in the directory, and the `-hl` (human-readable sizes, long listing) option will show us the size of each file:

```
ls -hl
```



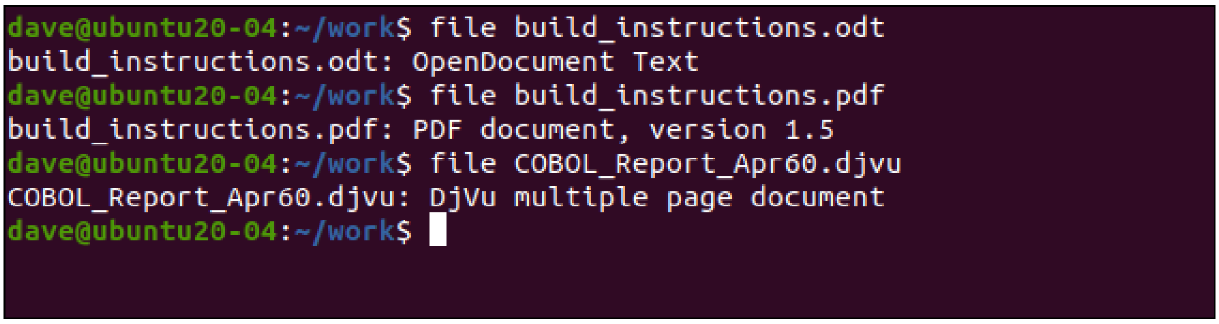
```
dave@ubuntu20-04:~/work$ ls -hl
total 9.3M
-rw-rw-r-- 1 dave dave  17K May 10 14:29 build_instructions.odt
-rw-rw-r-- 1 dave dave  32K May 10 14:32 build_instructions.pdf
-rw-r--r-- 1 dave dave 1.8M Apr 10 11:27 COBOL_Report_Apr60.djvu
-rw-r--r-- 1 dave dave  351 May  6 05:41 function_headers.h
-rwxrwxr-x 1 dave dave  17K May 10 10:00 hello
-rw-r--r-- 1 dave dave  120 May 10 09:23 hello.c
-rwxrw-r-x 1 dave dave  452 Apr 15 11:29 makefile
-rw-rw-r-- 1 dave dave 6.9M May 18  2012 Pachelbel_Canon_In_D.mp3
-rw-r--r-- 1 dave dave 1.6K Apr 18 17:18 README.md
-rw-rw-r-- 1 dave dave  31K May 10 14:37 screenshot.jpg
-rw-rw-r-- 1 dave dave  58K May 10 14:35 screenshot.png
-rwxr-xr-x 1 dave dave 354K May  6 05:42 watch.exe
-rwxrwxr-x 1 dave dave  42K May 10 10:03 wd
-rwxrw-r-x 1 dave dave 8.5K May  6 05:44 wd.c
-rw-r--r-- 1 dave dave  360 May  4 09:06 wd.h
-rw-rw-r-- 1 dave dave  47K May 10 10:03 wd.o
dave@ubuntu20-04:~/work$
```

Let’s try `file` on a few of these and see what we get:

```
file build_instructions.odt
```

```
file build_instructions.pdf
```

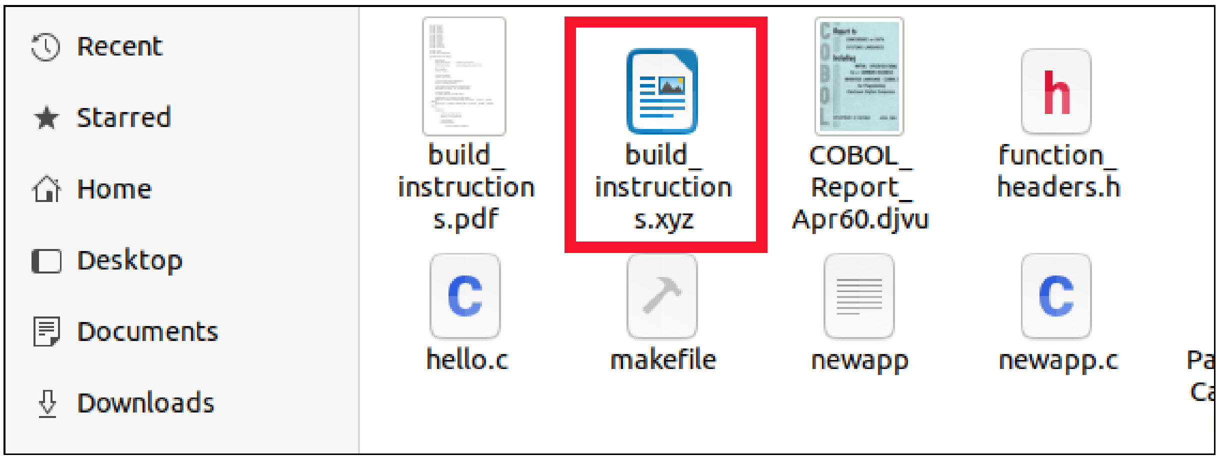
```
file COBOL_Report_Apr60.djvu
```



```
dave@ubuntu20-04:~/work$ file build_instructions.odt
build_instructions.odt: OpenDocument Text
dave@ubuntu20-04:~/work$ file build_instructions.pdf
build_instructions.pdf: PDF document, version 1.5
dave@ubuntu20-04:~/work$ file COBOL_Report_Apr60.djvu
COBOL_Report_Apr60.djvu: DjVu multiple page document
dave@ubuntu20-04:~/work$
```

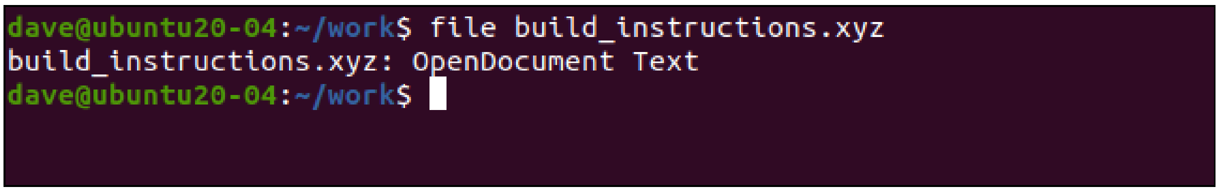
The three file formats are correctly identified. Where possible, `file` gives us a bit more information. The PDF file is reported to be in the version 1.5 format.

Even if we rename the ODT file to have an extension with the arbitrary value of XYZ, the file is still correctly identified, both within the `Files` file browser and on the command line using `file`.



Within the `Files` file browser, it's given the correct icon. On the command line, `file` ignores the extension and looks inside the file to determine its type:

```
file build_instructions.xyz
```



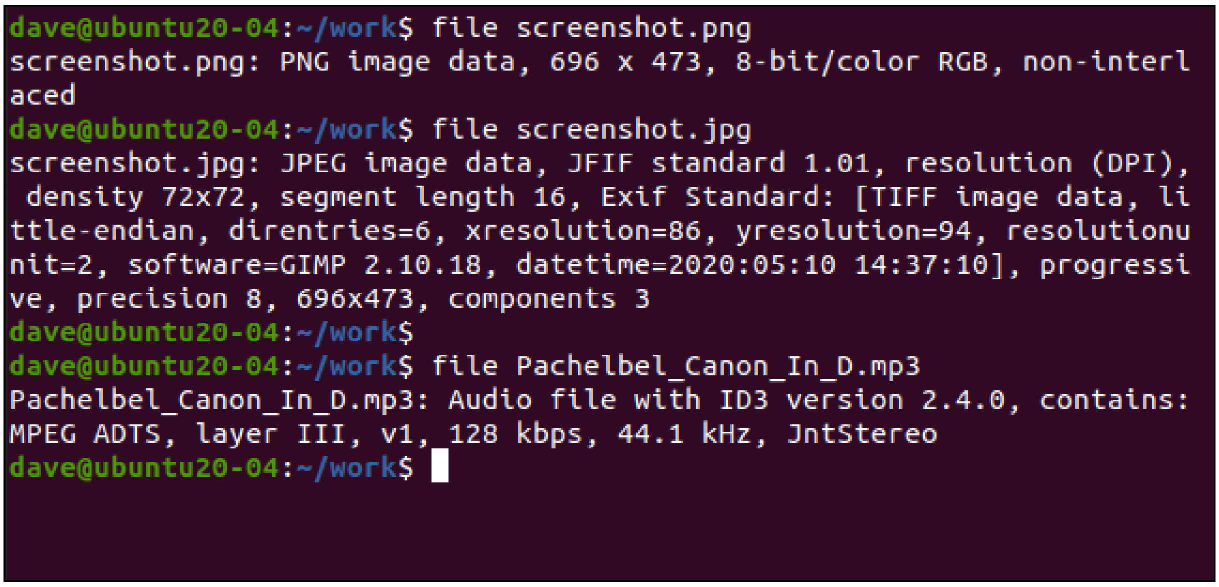
```
dave@ubuntu20-04:~/work$ file build_instructions.xyz
build_instructions.xyz: OpenDocument Text
dave@ubuntu20-04:~/work$
```

Using `file` on media, such as image and music files, usually yields information regarding their format, encoding, resolution, and so on:

```
file screenshot.png
```

```
file screenshot.jpg
```

```
file Pachelbel_Canon_In_D.mp3
```



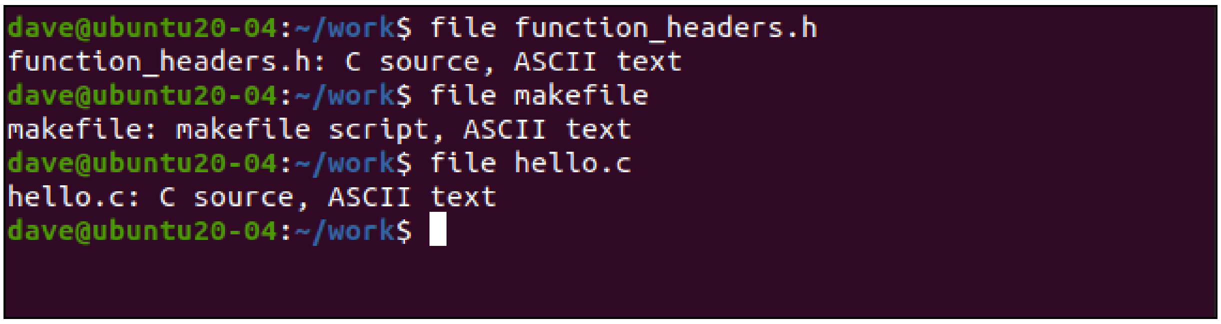
```
dave@ubuntu20-04:~/work$ file screenshot.png
screenshot.png: PNG image data, 696 x 473, 8-bit/color RGB, non-interl
aced
dave@ubuntu20-04:~/work$ file screenshot.jpg
screenshot.jpg: JPEG image data, JFIF standard 1.01, resolution (DPI),
 density 72x72, segment length 16, Exif Standard: [TIFF image data, li
ttle-endian, direntries=6, xresolution=86, yresolution=94, resolutionu
nit=2, software=GIMP 2.10.18, datetime=2020:05:10 14:37:10], progressi
ve, precision 8, 696x473, components 3
dave@ubuntu20-04:~/work$
dave@ubuntu20-04:~/work$ file Pachelbel_Canon_In_D.mp3
Pachelbel_Canon_In_D.mp3: Audio file with ID3 version 2.4.0, contains:
MPEG ADTS, layer III, v1, 128 kbps, 44.1 kHz, JntStereo
dave@ubuntu20-04:~/work$
```

Interestingly, even with plain-text files, `file` doesn't judge the file by its extension. For example, if you have a file with the ".c" extension, containing standard plain text but not source code, `file` doesn't mistake it for a genuine C source code file:

```
file function+headers.h
```

```
file makefile
```

```
file hello.c
```



```
dave@ubuntu20-04:~/work$ file function_headers.h
function_headers.h: C source, ASCII text
dave@ubuntu20-04:~/work$ file makefile
makefile: makefile script, ASCII text
dave@ubuntu20-04:~/work$ file hello.c
hello.c: C source, ASCII text
dave@ubuntu20-04:~/work$
```

`file` correctly identifies the header file (".h") as part of a C source code collection of files, and it knows the makefile is a script.

Using file with Binary Files

Binary files are more of a "black box" than others. Image files can be viewed, sound files can be played, and document files can be opened by the appropriate software package. Binary files, though, are more of a challenge.

For example, the files "hello" and "wd" are binary executables. They are programs. The file called "wd.o" is an object file. When source code is compiled by a compiler, one or more object files are created. These contain the machine code the computer will eventually execute when the finished program runs, together with information for the linker. The linker checks each object file for function calls to libraries. It links them to any libraries the program uses. The result of this process is an executable file.

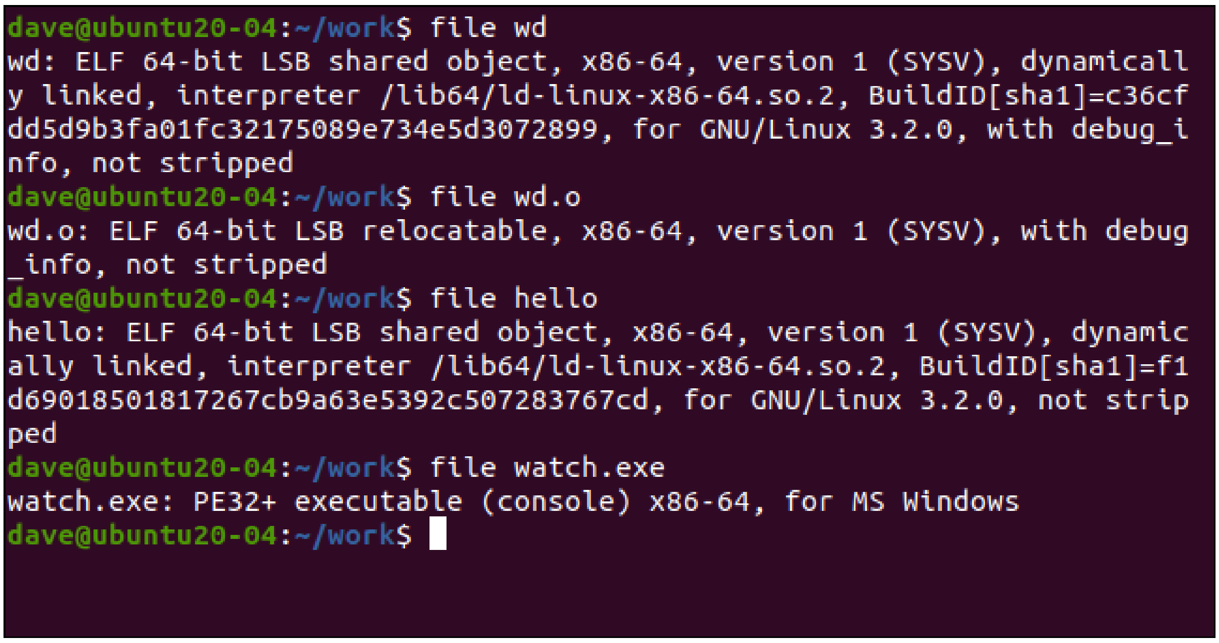
The file "watch.exe" is a binary executable that has been cross-compiled to run on Windows:

```
file wd
```

```
file wd.o
```

```
file hello
```

```
file watch.exe
```



```
dave@ubuntu20-04:~/work$ file wd
wd: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamicall
y linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=c36cf
dd5d9b3fa01fc32175089e734e5d3072899, for GNU/Linux 3.2.0, with debug_i
nfo, not stripped
dave@ubuntu20-04:~/work$ file wd.o
wd.o: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), with debug
_info, not stripped
dave@ubuntu20-04:~/work$ file hello
hello: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamic
ally linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=f1
d69018501817267cb9a63e5392c507283767cd, for GNU/Linux 3.2.0, not strip
ped
dave@ubuntu20-04:~/work$ file watch.exe
watch.exe: PE32+ executable (console) x86-64, for MS Windows
dave@ubuntu20-04:~/work$
```

Taking the last one first, `file` tells us the "watch.exe" file is a PE32+ executable, console program, for the x86 family of processors on Microsoft Windows. PE stands for portable executable format, which has 32- and 64-bit versions. The PE32 is the 32-bit version, and the PE32+ is the 64-bit version.

The other three files are all identified as Executable and Linkable Format (ELF) files. This is a standard for executable files and shared object files, such as libraries. We'll take a look at the ELF header format shortly.

What might catch your eye is that the two executables ("wd" and "hello") are identified as Linux Standard Base (LSB) shared objects, and the object file "wd.o" is identified as an LSB relocatable. The word executable is obvious in its absence.

Object files are relocatable, meaning the code inside them can be loaded into memory at any location. The executables are listed as shared objects because they've been created by the linker from the object files in such a way that they inherit this capability.

This allows the Address Space Layout Randomization (ASMR) system to load the executables into memory at addresses of its choosing. Standard executables have a loading address coded into their headers, which dictate where they're loaded into memory.

ASMR is a security technique. Loading executables into memory at predictable addresses makes them susceptible to attack. This is because their entry points, and the locations of their functions, will always be known to attackers. Position Independent Executables (PIE) positioned at a random address overcome this susceptibility.

If we compile our program with the `gcc` compiler and provide the `-no-pie` option, we'll generate a conventional executable.

The `-o` (output file) option lets us provide a name for our executable:

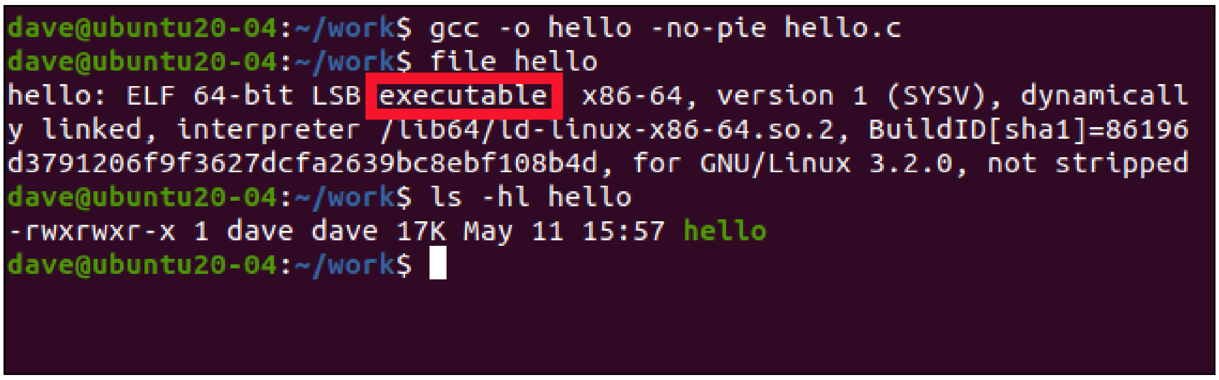
```
gcc -o hello -no-pie hello.c
```

We'll use `file` on the new executable and see what has changed:

```
file hello
```

The size of the executable is the same as before (17 KB):

```
ls -hl hello
```



```
dave@ubuntu20-04:~/work$ gcc -o hello -no-pie hello.c
dave@ubuntu20-04:~/work$ file hello
hello: ELF 64-bit LSB executable  x86-64, version 1 (SYSV), dynamicall
y linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=86196
d3791206f9f3627dcfa2639bc8ebf108b4d, for GNU/Linux 3.2.0, not stripped
dave@ubuntu20-04:~/work$ ls -hl hello
-rwxrwxr-x 1 dave dave 17K May 11 15:57 hello
dave@ubuntu20-04:~/work$
```

The binary is now identified as a standard executable. We're doing this for demonstration purposes only. If you compile applications this way, you'll lose all advantages of the ASMR.

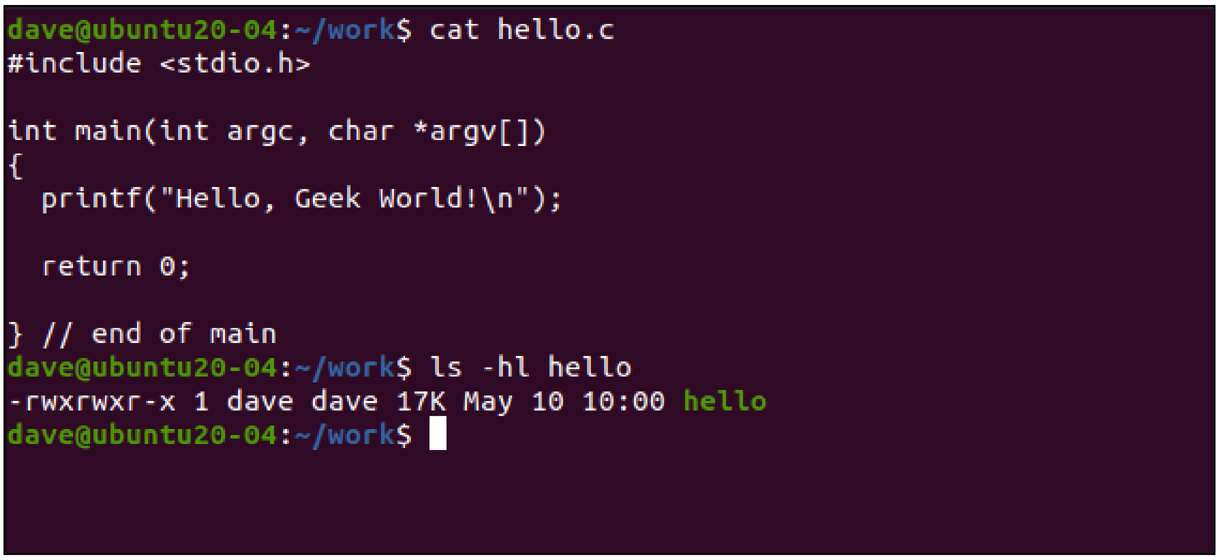
Why Is an Executable So Big?

Our example `hello` program is 17 KB, so it could hardly be called big, but then, everything's relative. The source code is 120 bytes:

```
cat hello.c
```

What's bulking out the binary if all it does is print one string to the terminal window? We know there's an ELF header, but that's only 64-bytes long for a 64-bit binary. Plainly, it must be something else:

```
ls -hl hello
```



```
dave@ubuntu20-04:~/work$ cat hello.c
#include <stdio.h>

int main(int argc, char *argv[])
{
  printf("Hello, Geek World!\n");

  return 0;

} // end of main
dave@ubuntu20-04:~/work$ ls -hl hello
-rwxrwxr-x 1 dave dave 17K May 10 10:00 hello
dave@ubuntu20-04:~/work$ 
```

Let's scan the binary with the `strings` command as a simple first step to discover what's inside it. We'll pipe it into `less`:

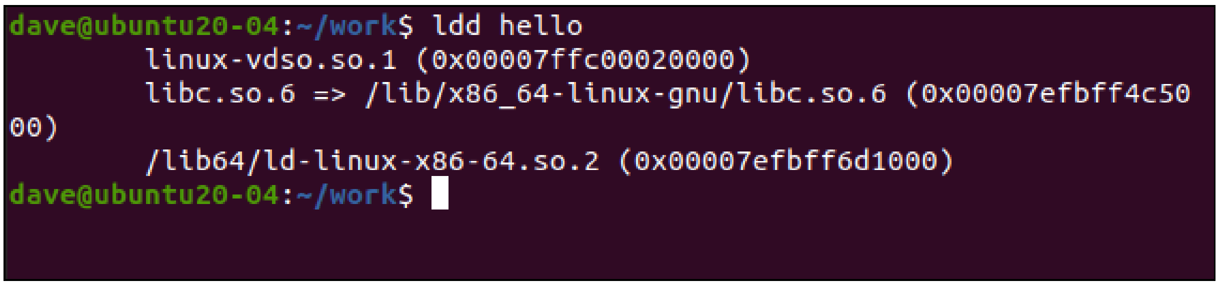
```
strings hello | less
```

```
dave@ubuntu20-04:~/work$ strings hello | less 
```

There are many strings inside the binary, besides the "Hello, Geek world!" from our source code. Most of them are labels for regions within the binary, and the names and linking information of shared objects. These include the libraries, and functions within those libraries, on which the binary depends.

The `ldd` command shows us the shared object dependencies of a binary:

```
ldd hello
```



```
dave@ubuntu20-04:~/work$ ldd hello
        linux-vdso.so.1 (0x00007ffc00020000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007efbff4c50
00)
        /lib64/ld-linux-x86-64.so.2 (0x00007efbff6d1000)
dave@ubuntu20-04:~/work$ 
```

There are three entries in the output, and two of them include a directory path (the first does not):

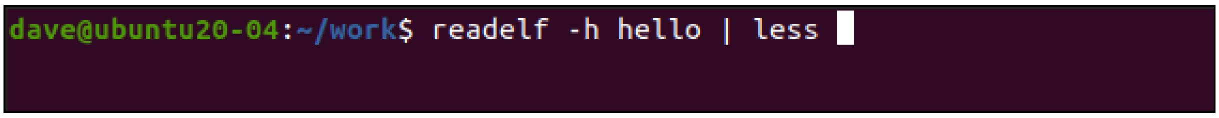
- **linux-vdso.so:** Virtual Dynamic Shared Object (VDSO) is a kernel mechanism that allows a set of kernel-space routines to be accessed by a user-space binary. This avoids the overhead of a context switch from user kernel mode. VDSO shared objects adhere to the Executable and Linkable Format (ELF) format, allowing them to be dynamically linked to the binary at runtime. The VDSO is dynamically allocated and takes advantage of ASMR. The VDSO capability is provided by the standard GNU C Library if the kernel supports the ASMR scheme.
- **libc.so.6:** The GNU C Library shared object.
- **/lib64/ld-linux-x86-64.so.2:** This is the dynamic linker the binary wants to use. The dynamic linker interrogates the binary to discover what dependencies it has. It launches those shared objects into memory. It prepares the binary to run and be

able to find and access the dependencies in memory. Then, it launches the program.

The ELF Header

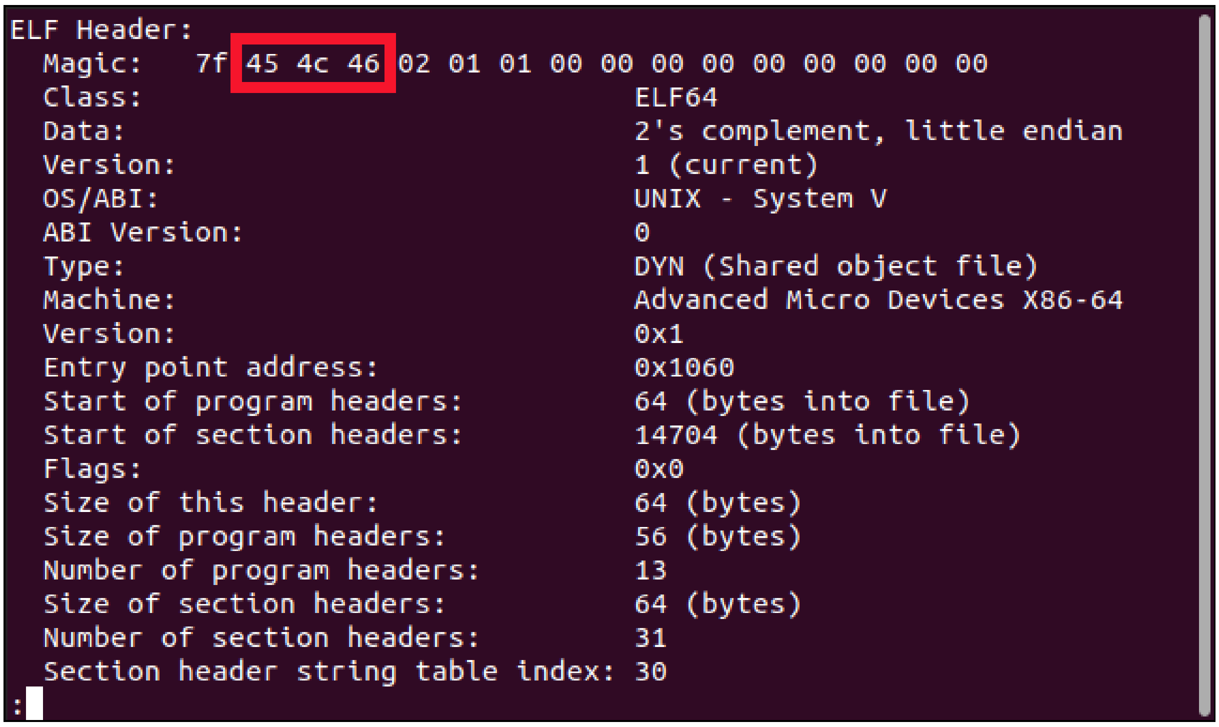
We can examine and decode the ELF header using the `readelf` utility and the `-h` (file header) option:

```
readelf -h hello
```



```
dave@ubuntu20-04:~/work$ readelf -h hello | less
```

The header is interpreted for us.



```
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:               0x1060
  Start of program headers:          64 (bytes into file)
  Start of section headers:          14704 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         13
  Size of section headers:           64 (bytes)
  Number of section headers:         31
  Section header string table index: 30
:
```

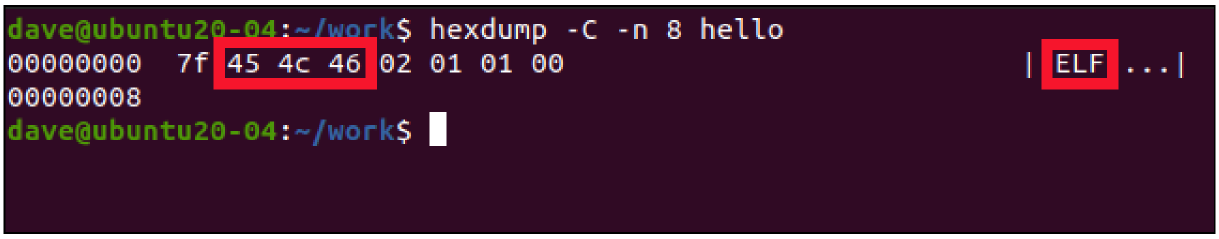
The first byte of all ELF binaries is set to hexadecimal value 0x7F. The next three bytes are set to 0x45, 0x4C, and 0x46. The first byte is a flag that identifies the file as an ELF binary. To make this crystal clear, the next three bytes spell out “ELF” in ASCII:

- **Class:** Indicates whether the binary is a 32- or 64-bit executable (1=32, 2=64).
- **Data:** Indicates the endianness in use. Endian encoding defines the way in which multibyte numbers are stored. In big-endian encoding, a number is stored with its most significant bits first. In little-endian encoding, the number is stored with its least significant bits first.
- **Version:** The version of ELF (currently, it’s 1).
- **OS/ABI:** Represents the type of application binary interface in use. This defines the interface between two binary modules, such as a program and a shared library.
- **ABI Version:** The version of the ABI.
- **Type:** The type of ELF binary. The common values are `ET_REL` for a relocatable resource (such as an object file), `ET_EXEC` for an executable compiled with the `-no-pie` flag, and `ET_DYN` for an ASMR-aware executable.
- **Machine:** The instruction set architecture. This indicates the target platform for which the binary was created.
- **Version:** Always set to 1, for this version of ELF.
- **Entry Point Address:** The memory address within the binary at which execution commences.

The other entries are sizes and numbers of regions and sections within the binary so their locations can be calculated.

A quick peek at the first eight bytes of the binary with `hexdump` will show the signature byte and "ELF" string in the first four bytes of the file. The `-C` (canonical) option gives us the ASCII representation of the bytes alongside their hexadecimal values, and the `-n` (number) option lets us specify how many bytes we want to see:

```
hexdump -C -n 8 hello
```

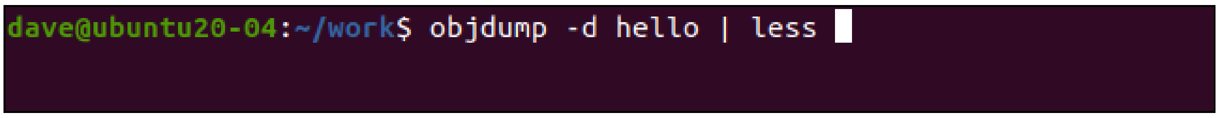


```
dave@ubuntu20-04:~/work$ hexdump -C -n 8 hello
00000000  7f 45 4c 46 02 01 01 00                           |.ELF....|
00000008
dave@ubuntu20-04:~/work$
```

objdump and the Granular View

If you want to see the nitty-gritty detail, you can use the `objdump`command with the `-d` (disassemble) option:

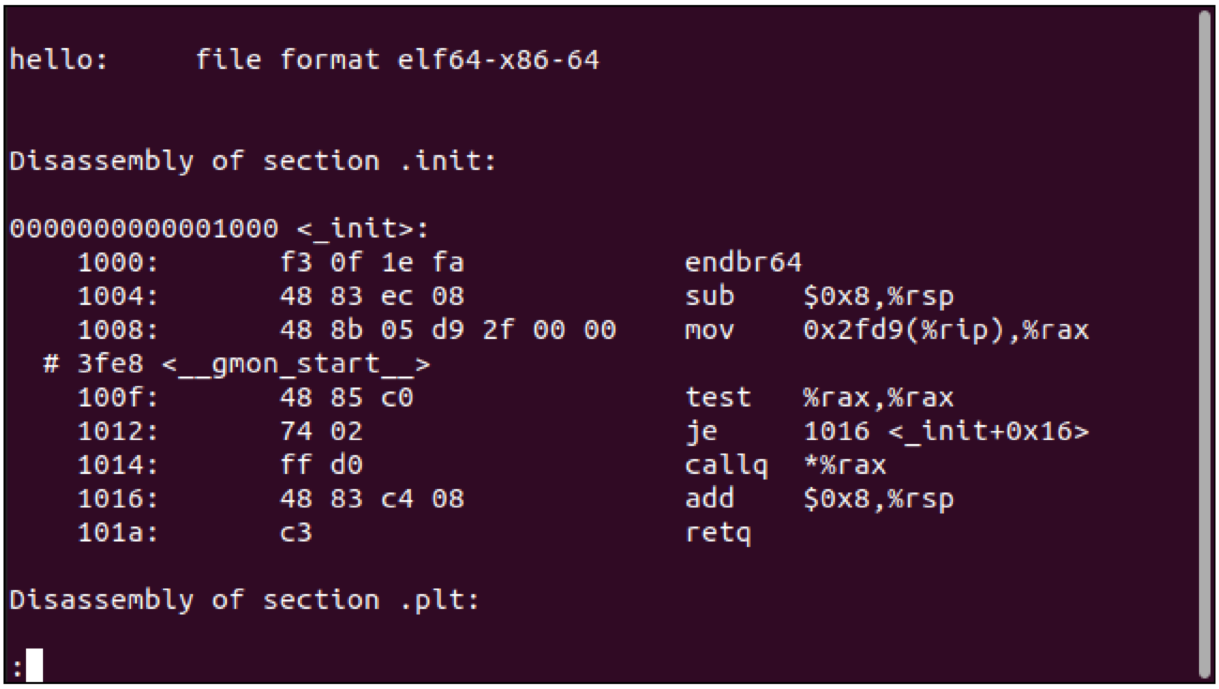
```
objdump -d hello | less
```



```
dave@ubuntu20-04:~/work$ objdump -d hello | less
```

This disassembles the executable machine code and displays it in hexadecimal bytes alongside the assembly language equivalent. The address location of the first bye in each line is shown on the far left.

This is only useful if you can read assembly language, or you're curious what goes on behind the curtain. There's a lot of output, so we piped it into `less`.



```

hello:     file format elf64-x86-64


Disassembly of section .init:

0000000000001000 <_init>:
    1000:       f3 0f 1e fa             endbr64
    1004:       48 83 ec 08             sub    $0x8,%rsp
    1008:       48 8b 05 d9 2f 00 00    mov    0x2fd9(%rip),%rax
  # 3fe8 <__gmon_start__>
    100f:       48 85 c0                test   %rax,%rax
    1012:       74 02                   je     1016 <_init+0x16>
    1014:       ff d0                   callq  *%rax
    1016:       48 83 c4 08             add    $0x8,%rsp
    101a:       c3                      retq

Disassembly of section .plt:

:
```

Compiling and Linking

There are many ways to compile a binary. For example, the developer chooses whether to include debugging information. The way the binary is linked also plays a role in its contents and size. If the binary references share objects as external dependencies, it will be smaller than one to which the dependencies statically link.

Most developers already know the commands we’ve covered here. For others, though, they offer some easy ways to rummage around and see what lies inside the binary black box.