Peeking Inside Executables And Libraries To Make Debugging Easier

Originalartikel

Backup

<html> <p>At first glance, both the executables that a compiler produces, and the libraries that are used during the building process seem like they&#8217;re not very accessible. They are these black boxes that make an application go, or make the linker happy when you hand it the &#8216;right&#8217; library file. There is also a lot to be said for not digging too deeply into either, as normally things will Just Work&#8482; without having to bother with such additional details.</p> <p>The thing is that both executables and libraries contain a lot of information that normally is just used by the OS, toolchain, debuggers and similar tools. Whether these files are in Windows PE format, old-school Linux

```
a.out
```

or modern-day

```
.elf
```

, when things go south during development, sometimes one has to break out the right tools to inspect them in order to make sense of what is happening.</p> <p>This article will focus primarily on the Linux platform, though most of it also applies to BSD and MacOS, and to some extent Windows.</p> <h2>Opening the Black Box</h2> <p><img data-attachment-id=„414852“ data-permalink=„https://hackaday.com/2020/05/28/peeking-inside-executables-and-libraries-to-make-debugging-easier/elf-layout-basic-themed/“ data-orig-file=„https://hackaday.com/wp-content/uploads/2020/05/elf-layout-basic-themed.png“ data-orig-size=„789,840“ data-comments-opened=„1“ data-image-meta=„{&quot;aperture&quot;:&quot;0&quot;,&quot;credit&quot;:&quot;&quot;,&quot;camera&quot;:&quot;&quot;,&quot;caption&quot;:&quot;&quot;,&quot;created_timestamp&quot;:&quot;0&quot;,&quot;copyright&quot;:&quot;&quot;,&quot;focal_length&quot;:&quot;0&quot;,&quot;iso&quot;:&quot;0&quot;,&quot;shutter_speed&quot;:&quot;0&quot;,&quot;title&quot;:&quot;&quot;,&quot;orientation&quot;:&quot;0&quot;}“ data-image-title=„elf-layout-basic-themed“ data-image-description=„“ data-medium-file=„https://hackaday.com/wp-content/uploads/2020/05/elf-layout-basic-themed.png?w=376“ data-large-file=„https://hackaday.com/wp-content/uploads/2020/05/elf-layout-basic-themed.png?w=587“ class=„alignright wp-image-414852 size-medium“ src=„https://hackaday.com/wp-content/uploads/2020/05/elf-layout-basic-themed.png?w=376“ alt=„“ width=„376“ height=„400“ srcset=„https://hackaday.com/wp-content/uploads/2020/05/elf-layout-basic-themed.png 789w, https://hackaday.com/wp-content/uploads/2020/05/elf-layout-basic-themed.png?resize=235,250 235w, https://hackaday.com/wp-content/uploads/2020/05/elf-layout-basic-themed.png?resize=376,400 376w, https://hackaday.com/wp-content/uploads/2020/05/elf-layout-basic-themed.png?resize=587,625

587w“ referrerpolicy=„no-referrer“ /></p> <p>Regardless of which platform you&#8217;re on, executable and library formats all have a number of common sections. There is of course the section with the actual instructions, as well as the section with all of the text strings and constant values that we put in the code before we compiled it. If we instructed the compiler to generate debug symbols and told the linker to leave those in place, we also have the debug symbols included in its own section. We will look at those later in this article.</p> <p>In the <a href=„https://en.wikipedia.org/wiki/Executable_and_Linkable_Format“ target=„_blank“>ELF</a> (Executable and Linkable Format) that is commonly used on Linux and many other operating systems, the rough layout follows this diagram. Not all of these sections are required, and their inclusion depends on what options were selected when the executable file was created.</p> <p>A quick overview of an executable file&#8217;s properties can be obtained with the <a href=„https://linux.die.net/man/1/file“ target=„_blank“>file</a> utility:</p> <pre class=„code“>ELF 32-bit LSB shared object, Intel 80386, version 1 (GNU/Linux), dynamically linked, interpreter /lib/ld-linux.so.2, for GNU/Linux 3.2.0, BuildID[sha1]=0558c7ef0f6845826d012b4ccc14948a2ffe8277, stripped</pre> <p>This output tells us that we&#8217;re dealing with a 32-bit binary, compiled for the x86 architecture, which uses a number of shared libraries, and which has had its debug symbols stripped.</p> <p>If debug symbols are still present, we get:</p> <pre class=„code“>ELF 32-bit LSB shared object, Intel 80386, version 1 (GNU/Linux), dynamically linked, interpreter /lib/ld-linux.so.2, for GNU/Linux 3.2.0, BuildID[sha1]=0558c7ef0f6845826d012b4ccc14948a2ffe8277, with debug_info, not stripped </pre> <p>In this particular case, we are dealing with a binary that was compiled on Raspbian Buster for x86, which is a 32-bit version of Linux, so that all matches.</p> <p>For a Windows executable file we get the following, less expansive output:</p> <pre class=„code“>PE32+ executable (GUI) x86-64, for MS Windows</pre> <p>This tells us that we are dealing with a PE (Windows) executable, compiled for the 64-bit x86-64 architecture.</p> <p>As one may have guessed at this point, libraries, both dynamic and shared, use the <a href=„https://en.wikipedia.org/wiki/Comparison_of_executable_file_formats“ target=„_blank“>same format</a> as the executables, so for example examining an

```
.so
```

shared library file on Linux would generate almost the same output when we use the <em>file</em> command.</p> <h2>Sharing Responsibly</h2> <p>Unique to (desktop) operating systems is the ability to load dynamic (shared) libraries when the application is started. Here the assumption is made that the required libraries are present on the host system, and in the search path for the library loader (an OS component). Libraries can also be versioned to indicate different revisions. This usually happens via the filename, with the generic name (e.g.

```
libfoo.so
```

) <a href=„https://linux.die.net/man/1/ln“ target=„_blank“>symlinked</a> to the actual file (

```
libfoo.so.0.1
```

). If there&#8217;s a mismatch with the version, this can result in a symbol error, which we&#8217;ll look at in the next section.</p> <p>When an executable uses shared library files, it is easy to check which direct dependencies (encoded in the executable file) it uses, by checking the executable with the <a href=„https://linux.die.net/man/1/ldd“ target=„_blank“>ldd</a> utility, which has a gotcha that it does not work well with the older

```
a.out
```

format. This isn&#8217;t really an issue with modern day development on Windows, Linux/BSD, and MacOS, which use the PE (PE32+), ELF and Mach-O formats, respectively. For embedded development (e.g. ARM Cortex-M) the ELF format is also used as an intermediary format before generating the binary image.</p> <h2>Listing Dependencies</h2> <p>The basic output from

```
ldd
```

shows where direct dependencies are found on the filesystem, and which dependencies are not found. For example, this is the (heavily) abbreviated output from

```
ldd
```

for

```
ffplay.exe
```

under MSYS2 on Windows:</p> <pre>$ ldd /mingw64/bin/ffplay.exe

```
      ntdll.dll =&gt; /c/Windows/SYSTEM32/ntdll.dll (0x77780000)
      kernel32.dll =&gt; /c/Windows/system32/kernel32.dll (0x77660000)
      KERNELBASE.dll =&gt; /c/Windows/system32/KERNELBASE.dll
(0x7fefd730000)
      msvcrt.dll =&gt; /c/Windows/system32/msvcrt.dll (0x7fefed80000)
      SHELL32.dll =&gt; /c/Windows/system32/SHELL32.dll (0x7fefdab0000)
      SHLWAPI.dll =&gt; /c/Windows/system32/SHLWAPI.dll (0x7fefda10000)
      GDI32.dll =&gt; /c/Windows/system32/GDI32.dll (0x7feff0e0000)
      USER32.dll =&gt; /c/Windows/system32/USER32.dll (0x77560000)
      LPK.dll =&gt; /c/Windows/system32/LPK.dll (0x7fefeb30000)
      USP10.dll =&gt; /c/Windows/system32/USP10.dll (0x7feff6e0000)
      SDL2.dll =&gt; /mingw64/bin/SDL2.dll (0x644c0000)
      [...]
```

</pre> <p>Dependencies shown for the average executable can be pretty massive (the full list is about eight times this length), but it&#8217;s useful as a quick sanity check to see not only whether a dependency has been fulfilled, but also whether the application loader has picked the right library. It can happen for example that a system has two different versions of a library (e.g. in <em>/usr/shared/bin</em> and <em>/usr/bin</em>), which can lead to the hilarious situation where you spend half a day debugging different libraries and application versions, rolling back &#8216;known working&#8217; code versions and losing your sanity.</p> <p>Another thing which a tool like

```
ldd
```

shows is at which address the library has been loaded, but that&#8217;s useful only for truly advanced levels of debugging and optimization.</p> <h2>When Symbols Go AWOL</h2> <p>Things get fun when we talk about symbols in the context of executable and library formats. This is not about debug symbols, which are a completely different topic, but the symbols that are integral to making it possible for sections of code to be found, whether while executing, or while linking object files and static libraries together. Missing symbols lead to fun run-time errors as well, where an &#8216;entry point&#8217; is not found in some shared library.</p> <p>A quick way to fix such issues is usually to ensure that you have the matching versions of the libraries for the code or executable file. Sometimes this all checks out, and the application loader or linker tool is still giving you lip about missing symbols, so what gives?</p> <p><img data-attachment-id=„218893“ data-permalink=„https://hackaday.com/2020/05/28/peeking-inside-executables-and-libraries-to-make-debugging-easier/turingtapes/“ data-orig-file=„https://hackaday.com/wp-content/uploads/2016/08/turingtapes.jpg“ data-orig-size=„3000,1815“ data-comments-opened=„1“ data-image-meta=„{&quot;aperture&quot;:&quot;0&quot;,&quot;credit&quot;:&quot;&quot;,&quot;camera&quot;:&quot;&quot;,&quot;caption&quot;:&quot;&quot;,&quot;created_timestamp&quot;:&quot;0&quot;,&quot;copyright&quot;:&quot;&quot;,&quot;focal_length&quot;:&quot;0&quot;,&quot;iso&quot;:&quot;0&quot;,&quot;shutter_speed&quot;:&quot;0&quot;,&quot;title&quot;:&quot;&quot;,&quot;orientation&quot;:&quot;1&quot;}“ data-image-title=„TuringTapes“ data-image-description=„“ data-medium-file=„https://hackaday.com/wp-content/uploads/2016/08/turingtapes.jpg?w=400“ data-large-file=„https://hackaday.com/wp-content/uploads/2016/08/turingtapes.jpg?w=800“ class=„alignright size-medium wp-image-218893“ src=„https://hackaday.com/wp-content/uploads/2016/08/turingtapes.jpg?w=400“ alt=„“ width=„400“ height=„242“ srcset=„https://hackaday.com/wp-content/uploads/2016/08/turingtapes.jpg 3000w, https://hackaday.com/wp-content/uploads/2016/08/turingtapes.jpg?resize=250,151 250w, https://hackaday.com/wp-content/uploads/2016/08/turingtapes.jpg?resize=400,242 400w, https://hackaday.com/wp-content/uploads/2016/08/turingtapes.jpg?resize=800,484 800w, https://hackaday.com/wp-content/uploads/2016/08/turingtapes.jpg?resize=1536,929 1536w, https://hackaday.com/wp-content/uploads/2016/08/turingtapes.jpg?resize=2048,1239 2048w“ referrerpolicy=„no-referrer“ />In the case of linking code, it can be as simple as the wrong linking order, as toolchains for most languages use an opportunistic linking style that remembers missing symbols, but does not remember symbols it has already seen. While in languages like Ada this is not an issue, in C-style languages, determining the linking order in the commands given to the linker tool is essential.</p> <p>Another issue is where a language (like C++) supports overloading functions to support different arguments and return types, and name mangling is used (to get a unique symbol). If a header file was compiled in C++ mode, when it&#8217;s supposed to be linked against a library that was compiled as C code, without name mangling, this would make the linker tool give the &#8216;missing symbol&#8217; error for those functions.</p> <p>In order to figure out whether a missing symbol is truly missing, improperly mangled, left unmangled or in another library or object file, one can use a utility like <a href=„https://linux.die.net/man/1/readelf“ target=„_blank“>readelf</a> to check which symbols are actually in the file. Note that (obviously) readelf only supports ELF-style files. A more generic utility that focuses on just symbols in a variety of formats is <a href=„https://linux.die.net/man/1/nm“ target=„_blank“>nm</a>. For example, this output from the <a href=„https://en.wikipedia.org/wiki/Nm_(Unix)“ target=„_blank“>Wikipedia entry</a> on nm:</p> <pre># nm test.o 0000000a T _Z15global_functioni 00000025 T _Z16global_function2v 00000004 b _ZL10static_var 00000000 t _ZL15static_functionv 00000004 d _ZL15static_var_init 00000008 b _ZZ15global_functioniE16local_static_var 00000008 d _ZZ15global_functioniE21local_static_var_init

```
       U __gxx_personality_v0
```

00000000 B global_var 00000000 D global_var_init 0000003b T main 00000036 T non_mangled_function </pre> <p>This shows what the output from nm looks like when a C++ compiler is used. Nm can be instructed to demangle symbols to make it easier to read if that&#8217;s necessary. Regardless, its output tells us whether a symbol exists in the file or is undefined (&#8216;U&#8217;). It will also detail where the symbol is defined (which section) and what type of symbol it is (if relevant). In the above example we see one undefined symbol (&#8216;U&#8217;), a couple of text (code) section symbols (&#8216;T&#8217; &amp; &#8216;t&#8217;), one symbol in the uninitialized data section (BSS, &#8216;B&#8217; &amp; &#8216;b&#8217;) and two in the initialized data section (&#8216;D&#8217; &amp; &#8216;d&#8217;).</p> <p>Of these, we&#8217;d just need to hand the linker a library or object file that contains the one undefined symbol to make this code link and produce an executable.</p> <h2>Last Resort: Tracing Application Startup</h2> <p>Annoyingly, sometimes everything seems in order, yet the application fails to start, or quits half-way through with a mysterious message. This is where a utility like <a href=„https://linux.die.net/man/1/strace“ target=„_blank“>strace</a> can be extremely useful, as it traces all system calls involving the application from the moment that the application starts. Often, the issue with an application not loading is due to an indirect dependency that cannot be loaded, an environmental setting that is inappropriate, or a file that was accidentally set to read-only.</p> <p>Simply firing up strace with the application as argument will output a list of the system calls as made by the application, including errors, such as a missing file:</p> <pre>open(„/foo/bar“, O_RDONLY) = -1 ENOENT (No such file or directory) </pre> <p>Or a missing library dependency:</p> <pre>open(„/usr/lib/libfoo.so“, O_RDONLY|O_CLOEXEC) = -1 ENOENT (No such file or directory)</pre> <h2>Wrapping Up</h2> <p>Obviously none of this is the end-all, be-all of debugging the linking and running of executables, binaries, and an assortment of related issues. As with so many things in life, in the end it&#8217;s mostly experience that counts. Over time one will develop an intuition for where the problem likely lies, as well as how to find out the culprit as quickly as possible.</p> <p>Having spent many years in commercial software development and having survived a range of (overly) ambitious hobby projects, I can definitely say that there is a lot of knowledge that I wish I had had sooner. On the other hand, the act of discovering why some things were not working and correcting this injustice against the order of the world was usually rewarding in itself.</p> <p>That said, one has to pick their battles wisely. Sometimes learning things from scratch isn&#8217;t worth it, and leaning on the knowledge of others is nothing to be ashamed of. Especially when it&#8217;s Friday afternoon and the client expects delivery of the new version on Monday. Hopefully this article has been helpful in that regard.</p> </html>