# ELF Loaders, Libraries and Executables on Linux

REF:

https://dtrugman.medium.com/elf-loaders-libraries-and-executables-on-linux-e5cfce318f94

The target of the following stream of ASCII characters is to shed some light on an area that is probably not common knowledge even for experienced system developers: loaders, libraries and executables in the ELF Linux ecosystem.

First, we’ll try to understand what happens when we run a basic program on our Linux machine. Then we’ll discuss libraries and what they bring to the table. Cover the differences between static and dynamic libraries and executables, and finally dive into the inner-works of the dynamic loader, learning how to control, configure and manipulate it.

Hopefully, we can cover enough concepts, tools and approaches to ultimately provide you with an efficient tool-set for tackling related issues when managing, deploying or developing applications for Linux systems.

Here are the subjects we’ll attempt to recount:

* Behind the scenes of “Hello World”
* Executable and Linkable Format (ELF)
* Who the hell needs libraries?!
* Libraries needed by an executable
* Static enumeration of needed libraries
* Dynamic enumeration of needed libraries
* Loader who?
* Use your own loader
* What does the loader load?
* R[UN]PATH
* NODEFLIB

# Behind the scenes of “Hello World”

I want to believe that everybody have seen the following piece of C code at least once in their lives:

#include <stdio.h>void main() {

printf("Hello World!\n");

}

And then we've probably compiled and executed it:

$ gcc main.c -o myapp # <--- Compile

$ ./myapp # <--- Execute

Hello World!

But, did we really understand what happened there during these two trivial steps we went through? What is this myapp file that got created? How could our program use the printf method without implementing it? And what happened when we executed it?

We’ll try to address each of these questions in the same order in the following paragraphs.

## Executable and Linkable Format (ELF)

People don’t really realize that, but the ELF file format is a cornerstone for all their magical applications! ELF is a standard file format for executables, libraries, and more.

Just like in our “Hello World” application; myapp is an ELF file (without even knowing it) and used other modules, such as the dynamic loader (which we’ll talk about later) to execute it.

By design, the ELF format is flexible, extensible, and cross-platform. It supports different endianness and address sizes so it can be used on any CPU or instruction set architecture. This has promoted its adoption by many different operating systems, and made it the de facto standard for Linux and other Unix-like systems.

Since the ELF format is a topic on its own, we won’t dwell into that, but if you want some more information, take a look at this [comprehensive ELF standard](https://refspecs.linuxfoundation.org/elf/elf.pdf).

## Who the hell needs libraries?

One of the most important concepts in Software Engineering is code reuse. There are numerous reasons for not writing the same code more than once, or using different instances of the same code snippet.

This is just like in our “Hello World” application, where we used the printf method without implementing it. Because printing to the screen is required by so many applications out there, it was implemented once, and is now readily available for anyone writing C or C++ applications.

In order to implement this, libraries were invented. Libraries allow programmers to pack code into reusable modules. Then, every program that requires the same functionality can simply use the already-existing library. There are actually two kinds of libraries:

Static libraries (.a files): Libraries that become part of the executable (are actually embedded into it) during linking. Which means that once the executable is ready, it doesn't require any additional files.

Dynamic libraries (.so files): Libraries that are shipped separately from the executable, and are required to be present at run-time. We can use these libraries in two different ways, and the difference is mainly whether or not the libraries are present during compile-time:

1. The executable links with the library at compile time. The executable can call methods defined in the library as if both shared their code-base. There is no need to explicitly load the library, and the loader handles it auto-magically during runtime. A classic example is any C program that links and uses glibc.
2. The executable does NOT link with the library at compile time. Instead, it loads and (possibly) unloads the library during execution-time only. The executable “unravels” and uses the library’s API during run-time using functionality provided by the dynamic loader. This is accomplished using dlopen (See [man](https://linux.die.net/man/3/dlopen)(3) page for more information). A classic examples is an application that loads plug-ins during runtime.

## Libraries needed by an executable

Now that we know what libraries are, we’ll define two new terms:

* Static executable: An executable that doesn't depend on any libraries.
* Dynamic executable: An executable that depends on other libraries.

So, given an executable, how can we tell if it’s dynamic or static? Does it require any additional libraries or not? We can use a simple command called file .

For example, when we use it on our “Hello World” application, we see that it is dynamically linked (emphasis added), thus requiring dynamic libraries:

$ file myapp

myapp: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0, BuildID[sha1]=f385b7c6c03b6b5c8c416e3c4a2267030ca095aa, not stripped

> Note: Static executables are ‘statically linked’

But, how is this possible? We haven’t told anyone anything about additional libraries! Why is it dynamically linked?

Well, it turns out that by default, when we compile a C/C++ application, the compiler is programmed to automatically use the standard libraries: glibc.so for C and additionally libstdc++.so for C++.

Apart from that, in real life, when we compile an application, we usually tell the linker to add additional libraries using the -l flag. For example, if we’re using a publicly available library like boost, we can tell the compiler to link against that library (i.e. require it during runtime) by passing the -lboost parameter during compilation/linkage.

> Note: Required libraries are specified only by their basename, whereas the lookup directories are defined using additional parameters such as -L or system default paths, which we’ll discuss later.

Now that we know our application is a dynamic executable, how can we tell which libraries it requires?

## Static enumeration of needed libraries

There are multiple ways to probe for libraries required by a specific executable. We will use an awesome tool called patchelf. Though its mostly intended to do what its name suggests, patching ELFs, it has a useful option called --print-needed that prints a list of libraries (basenames) required by this executable?

For example, if we use it on our “Hello World” application, we get the following output:

$ patchelf --print-needed myapp

libc.so.6

We can accomplish the same task by directly examining the ELF dynamic sections. Each needed library is represented by an entry in the .dynamic section, so we can simply print the entire .dynamic section using the readelf -d <path-to-elf> command, and look for the keyword NEEDED:

$ readelf -d myapp | grep NEEDED

0x0000000000000001 (NEEDED) Shared library: [libc.so.6]

## Dynamic enumeration of needed libraries

But, how can we check what library the loader is actually going to load?libc.so.6 can exist at multiple locations, which one are we actually going to use?

In order to collect additional information, we can use a great script called ldd (List Dynamic Dependencies). This script enumerates all the dynamic libraries required by a specific executable. And, unlike the static enumeration methods, this tool describes the full path(!) of the library we are going to use. For example, the output for our “Hello World” application is:

$ ldd myapp

linux-vdso.so.1 (0x00007fffed5b3000)

libc.so.6 => /lib/x86\_64-linux-gnu/libc.so.6 (0x00007fa412e60000)

/lib64/ld-linux-x86-64.so.2 (0x00007fa413600000)

As you can see, ldd has a much richer output. Right now, we’ll ignore the first and last line, and focus on the line in bold, that follows the format:

<lib-basename> => <lib-full-path> (lib-load-addr)

This time, for each of the needed libraries, we can see the actual instance to be loaded and the load address. For instance, libc.so.6 will be loaded from /lib/x86\_64-linux-gnu/libc.so.6 .

How does ldd collect this information? Well, it basically loads the application (without executing it) and traces the loader’s actions. If we run it using Bash’s trace mode enabled (bash -x $(which ldd) myapp), we’ll see that the entire script evaluates to a single line (I removed all the empty export statements around it to improve readability):

LD\_TRACE\_LOADED\_OBJECTS=1 /lib64/ld-linux-x86-64.so.2 myapp

At this point, one might ask himself, WTF?! Well, yeah, this is definitely peculiar! What are we trying to do here? What is this weird ld-linux-x86-64.so.2 file? Isn't it a library (according to its .so.2 extension)?

Well, I've been using the ‘loader’ term intermittently in the last few paragraphs, but this is the first time we have actually seen it in action. This is probably a good time to entertain ourselves with some more interesting facts about it.

## Loader who?

As you could guess by now, the dynamic loader is a single static executable ELF file that goes by a name usually reserved by libraries.

The weird thing about it, is that it can be run either indirectly, by running some dynamically linked program or shared object, or directly, just like we’ve seen in the ldd example.

> Note: The default loader (on 64-bit machines) can be found at /lib64/ld-linux.so.2 and is actually a symbolic link to the real loader file /lib/x86\_64-linux-gnu/ld-<version>.so.

## Use your own loader

Well, must we use the system’s loader? Can you BYOL? On the one hand, this is something every systems programmer wishes to avoid. The different package managers take care of many compatibility issues, and once you step off that train and into the wild, a lot of things can go wrong at every turn you take.

On the other hand, there is a method of distributing software as an encapsulated bundle that depends on it. When following this convention, applications are compiled against and shipped with a very specific set of libraries (I consider the loader to be one of them), thus creating a package that is (almost) independent of the hosting system. To accomplish this goal, we have to “patch” our executable and tell it to use our own loader.

To find out which loader an ELF “expects”, we can use the file app (emphasis added):

$ file myapp

myapp: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0, BuildID[sha1]=f385b7c6c03b6b5c8c416e3c4a2267030ca095aa, not stripped

Whereas to change the loader for a specific ELF, we can use patchelf and specify the interpreter and the elf to patch:

$ patchelf --set-interpreter <path-to-interpreter> <path-to-elf># For example:

$ patchelf --set-interpreter "/opt/pkg/lib/ld.so.2" /opt/pkg/app

But, when shipping such a specially crafted package, one also has to make sure that loader loads the libraries he shipped with it, and not the ones pre-installed on the target system. To accomplish that, we have to understand how the loader works, and how it decided which libraries to load.

## What does the loader load?

When we load a static executable, there is no need for the dynamic loader, because all the required libraries were integrated into the executable during link-time.

However, when we load a dynamic executable, the system depends on the loader to do the heavy lifting.

The loader always starts by loading:

* All the libraries specified by the environment variable LD\_PRELOAD
* All the libraries listed in /etc/ld.so.preload

Then, it tries to satisfy each direct dependency (library) string specified as NEEDED in the ELF’s .dynamic section.

If one of these libraries requires an additional library, not required by the executable, this library is referred to as an ‘indirect dependency’. And once we finished loading all the direct dependencies, the loader loads the indirect ones.

For each (direct/indirect) dependency, the loader follows this decision making process:

First, if the dependency string contains a slash, e.g. /mylib (can occur if one was specified during link time), it is interpreted as a (relative or absolute) pathname and loaded from there and no further lookup is required.

Otherwise, the loader follows these steps (Each step contains a condition. If that condition is not satisfied, the loader will skip it):

Step 1: Using the directories in DT\_RPATH

* Condition: DT\_RUNPATH attribute does not exist
* Note: DT\_RPATH is deprecated

Step 2: Using the environment variable LD\_LIBRARY\_PATH

* Condition: The executable is not being run in secure-execution mode (Formally, when AT\_SECURE has a nonzero value. Informally, this happens when we set the file’s SUID bit, for example)
* Example: When the SETUID bit is set, and the process’s real and effective UIDs differ, the loader will ignore it
* Note: LD\_LIBRARY\_PATH can be overridden by executing the dynamic linker directly with the option --library-path , e.g: /lib64/ld-linux-x86–64.so.2 --library-path "/my/libs:/my/other/libs" myapp

Step 3: Using the directories in DT\_RUNPATH

* Condition: The loader is loading a direct dependency.
* Example: If our binary myapp needs a single library a.so , and a.so requires b.so , when the loader will be looking for b.so (which is not a direct dependency of myapp ), it will skip this step!
* Note: This is different from DT\_RPATH, which is always applied.

Step 4: From the cache file /etc/ld.so.cache

* Condition: The ELF doesn't contain the NODEFLIB flag.

Step 5: In the default path /lib[64] , and then /usr/lib[64]

* Condition: The ELF doesn't contain the NODEFLIB flag.

Do you feel like five different preconditioned steps is enough? Well, lucky for us, that’s it for now. If the loader goes through all five steps and still it cannot find a library, the load process fails, and a similar error message is printed:

/bin/myapp: error while loading shared libraries: libncursesw.so.5: cannot open shared object file

Now that we know what guides the loader while trying to do its job, let’s take it to the next step, and see how we can advise. To do that, we’ll recap and elaborate on some of the UPPERCASE keywords we've just seen.

## R[UN]PATH

Both RPATH and RUNPATH are optional entries in the .dynamic section of ELF executables or shared libraries.

As we already understand, their goal is to empower the developer and allow him to alter the behavior of the dynamic loader.

In case you didn't read the last chapter, or just to sharpen your grasp of these values, let’s recap:

* If RUNPATH isn't set, the loader looks for every library in the directories specified by RPATH .
* If RUNPATH is set, the loader looks for direct dependency libraries in the directories specified by RUNPATH.
* RPATH is deprecated, and the use of RUNPATH is encouraged instead

Having a good understanding of the consequences, we can now start talking about the next phase, how can we manipulate it?

First, before we start breaking things, we should be able to know what we’re dealing with. Given an ELF file, we can examine R[UN]PATH values using:

readelf -d <path-to-elf> | egrep "RPATH|RUNPATH"

When RPATH and RUNPATH are not set, there won’t be any output, otherwise, we’ll see the dynamic section entry:

$ readelf -d ./example | egrep "RPATH|RUNPATH"

0x000000000000001d (RUNPATH) Library runpath: [/my/patched/libs]

Now, for the fun part. We can manipulate an ELF in two very different ways:

1. Set at compile time. The GNU linker ld supports the -rpath option. All -rpath options are concatenated and added to the final executable (See [ld(1)](https://linux.die.net/man/1/ld) for more information).
2. Manipulate an existing ELF file using the beloved patchelf :

# Clearing RPATH & RUNPATH

patchelf --remove-rpath <path-to-elf># Setting RPATH

patchelf --force-rpath --set-rpath <desired-rpath> <path-to-elf># Setting RUNPATH

patchelf --set-rpath <desired-rpath> <path-to-elf>

## NODEFLIB

This next one is a flag that we can find in an optional .dynamic section called FLAGS\_1. In a nutshell, it tells the loader to avoid loading libraries from:

* The loader cache file
* Default system locations

> Note: This request holds, even if it means that the load operation will fail.

How can we tell if this flag is set? Well, once more, we print out the dynamic sections of the ELF and look for the NODEFLIB expression:

readelf -d <path-to-elf> | grep NODEFLIB

And how can we manipulate it? Similarly to R[UN]PATH:

* Option 1: During compile time, by passing the -z nodefaultlib flag to the GNU linker.
* Option 2: Manipulate an existing ELF using our revered tool of course: patchelf --no-default-lib <path-to-elf>

This is the end, and I hope you have enjoyed scrolling down to the bottom of this article :)

Now seriously, I hope that after reading this, you feel better equipped for fighting those sorts of issues on your system. If you think that this article could have used an additional chapter, please let me know!

If you feel that you’ve gained some insights from reading this, I would really appreciate it if you clap!

## Resources:

* [Executable and Linkable Format](https://en.wikipedia.org/wiki/Executable_and_Linkable_Format) on Wikipedia
* [Dynamic loader manual](https://linux.die.net/man/8/ld.so): Know it all, just like the gods wanted
* [GNU Linker manual](https://linux.die.net/man/1/ld): Useful for linker flags like nodefaultlib or rpath
* [Python pwnypack package documentation](https://pwnypack.readthedocs.io/en/latest/pwnypack/elf.html): Great reference for ELF types and flags values. For example, supported flags in the FLAGS and FLAGS\_1 sections