Shared Libraries: Understanding Dynamic Loading

September 17, 2016

In this post, I will attempt to explain the inner workings of how dynamic loading of shared libraries works in Linux systems. This post is long - for a TL;DR, please read the debugging cheat sheet.

This post is not a how-to guide, although it does show how to compile and debug shared libraries and executables. It's optimized for understanding of the inner workings of how dynamic loading works. It was written to eliminate my knowledge debt on the subject, in order to become a better programmer. I hope that it will help you become better, too.

What Are Shared Libraries?

A library is a file that contains compiled code and data. Libraries in general are useful because they allow for fast compilation times (you don't have to compile all sources of your dependencies when compiling your application) and modular development process. Static Libraries are linked into a compiled executable (or another library). After the compilation, the new artifact contains the static library's content. Shared Libraries are loaded by the executable (or other shared library) at runtime. That makes them a little more complicated in that there's a whole new field of possible hurdles which we will discuss in this post.

Example Setup

To explore the world of shared libraries, we'll use one example throughout this post. We'll start with three source files:

`main.cpp` will be the main file for our executable. It won't do much - just call a function from a `random` library which we'll compile:

```
#include "random.h"

int main() {
    return get_random_number();
}
```

The `random` library will define a single function in its header file, `random.h` :

```
int get_random_number();
```

It will provide a simple implementation in its source file, `random.cpp` :

```
#include "random.h"
```

```
int get_random_number(void) {
    return 4;
}
```

Note: I'm running all of my examples on Ubuntu 14.04.

Compiling a Shared Library

Before compiling the actual library, we'll create an object file from `random.cpp` :

```
$ clang++ -o random.o -c random.cpp
```

In general, build tools don't print to the standard output when everything is okay. Here are all the parameters explained:

- `-o random.o` : Define the output file name to be `random.o` .
- `-c` : Don't attempt any linking (only compile).
- `random.cpp` : Select the input file.

Next, we'll compile the object file into a shared library:

```
$ clang++ -shared -o librandom.so random.o
```

The new flag is `-shared` which specifies that a shared library should be built. Notice that we called the shared library `librandom.so` . This is not arbitrary - shared libraries should be called `lib<name>.so` for them to link properly later on (as we'll see in the linking section below).

Compiling and Linking a Dynamic Executable

First, we'll create a shared object for `main.cc` :

```
$ clang++ -o main.o -c main.cpp
```

This is exactly the same as before with `random.o` . Now, we'll try to create an executable:

```
$ clang++ -o main main.o
main.o: In function `main':
main.cpp:(.text+0x10): undefined reference to `get_random_number()'
clang: error: linker command failed with exit code 1 (use -v to see invocatio
```

Okay, so we need to tell `clang` that we want to use `librandom.so` . Let's do that [1] :

```
$ clang++ -o main main.o -lrandom
/usr/bin/ld: cannot find -lrandom
clang: error: linker command failed with exit code 1 (use -v to see invocatio
```

Hmmmmph. We told our compiler we want to use a `librandom` file. Since it's loaded dynamically, why do we need it in compile time? Well, the reason is that we need to make sure that the libraries we depend on contain all the symbols needed for our executable. Also note that we specified `random` as the name of the library, and not `librandom.so` . Remember there's a convention regarding library file naming? This is where it's used.

So, we need to let `clang` know where to search for shared libraries. We do this with the `-L` flag. Note that paths specified by `-L` only affect the search path when linking - not during runtime. We'll specify the current directory:

```
$ clang++ -o main main.o -lrandom -L.
```

Great. Now let's run it!

```
$ ./main
./main: error while loading shared libraries: librandom.so: cannot open share
```

This is the error we get when a dependency can't be located. It will happen before our application even runs one line of code, since shared libraries are loaded before symbols in our executable.

This raises several questions:

- How does `main` know it depends on `librandom.so` ?
- Where does `main` look for `librandom.so` ?
- How can we tell `main` to look for `librandom.so` in this directory?

To answer these question, we'll have to go a little deeper into the structure of these files.

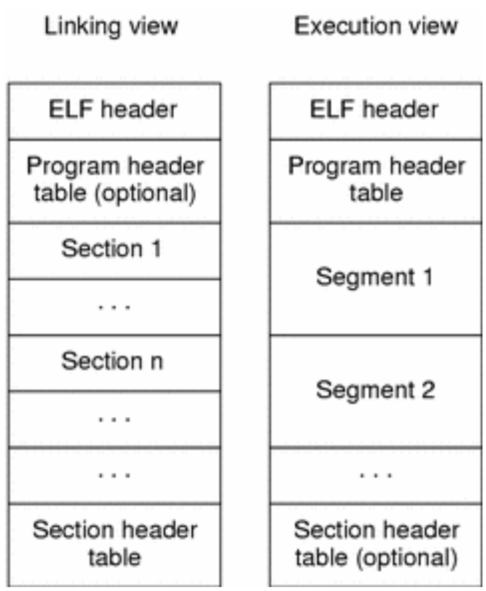
ELF - Executable and Linkable Format

The shared library and executable file format is called ELF (Executable and Linkable Format). If you check out the Wikipedia article you'll see that it's a hot mess, so we won't go over all of it. In summary, an ELF file contains:

- ELF Header
- File Data, which may contain:
  - Program header table (a list of *segment headers*)
  - Section header table (a list of *section headers*)
  - Data pointed to by the above two headers

The ELF header specifies the size and number of segments in the program header table and the size and number of sections in the section header table. Each such table consists of fixed size entries (I use *entry* to describe either a *segment header* or a *section header* in the appropriate table). Entries are headers and they contain a “pointer” (an offset in the file) to the location of the actual body of the segment or section. That body exists in the data part of the file. To make matters more complicated - each *section* is a part of a *segment*, and a *segment* can contain many *sections*.

In effect, the same data is referenced as either part of a *segment* or a *section* depending on the current context. *sections* are used when linking and *segments* are used when executing.



We’ll use `readelf` to... well, *read* the *ELF*. Let’s start by looking at the ELF header of `main` :

```
$ readelf -h main
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:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
```

```
  Version:                           0x1
  Entry point address:               0x4005e0
  Start of program headers:          64 (bytes into file)
  Start of section headers:          4584 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         9
  Size of section headers:           64 (bytes)
  Number of section headers:         30
  Section header string table index: 27
```

We can see that this is an ELF file (64-bit) on Unix. Its type is `EXEC`, which is an executable file - as expected. It has 9 program headers (meaning it has 9 *segments*) and 30 section headers (i.e., *sections*).

Next up - program headers:

```
$ readelf -l main

Elf file type is EXEC (Executable file)
Entry point 0x4005e0
There are 9 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                 0x00000000000001f8 0x00000000000001f8  R E    8
  INTERP         0x0000000000000238 0x0000000000400238 0x0000000000400238
                 0x000000000000001c 0x000000000000001c  R      1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x000000000000089c 0x000000000000089c  R E    200000
  LOAD           0x0000000000000dd0 0x0000000000600dd0 0x0000000000600dd0
                 0x0000000000000270 0x0000000000000278  RW     200000
  DYNAMIC        0x0000000000000de8 0x0000000000600de8 0x0000000000600de8
                 0x0000000000000210 0x0000000000000210  RW     8
  NOTE           0x0000000000000254 0x0000000000400254 0x0000000000400254
                 0x0000000000000044 0x0000000000000044  R      4
```

```
  GNU_EH_FRAME   0x0000000000000774 0x0000000000400774 0x0000000000400774
                 0x0000000000000034 0x0000000000000034  R      4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     10
  GNU_RELRO      0x0000000000000dd0 0x0000000000600dd0 0x0000000000600dd0
                 0x0000000000000230 0x0000000000000230  R      1

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr
   03     .init_array .fini_array .jcr .dynamic .got .got.plt .data .bss
   04     .dynamic
   05     .note.ABI-tag .note.gnu.build-id
   06     .eh_frame_hdr
   07
   08     .init_array .fini_array .jcr .dynamic .got
```

Again, we see that we have 9 program headers. Their types are `LOAD` (two of those), `DYNAMIC`, `NOTE`, etc. We can also see the section ownership of each segment.

Finally - section headers:

```
$ readelf -S main
There are 30 section headers, starting at offset 0x11e8:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS         0000000000400238  00000238
       000000000000001c  0000000000000000   A       0     0     1
  [ 2] .note.ABI-tag     NOTE             0000000000400254  00000254
       0000000000000020  0000000000000000   A       0     0     4

  [..]

  [21] .dynamic          DYNAMIC          0000000000600de8  00000de8
       0000000000000210  0000000000000010  WA       6     0     8
```

```
  [..]

  [28] .symtab           SYMTAB           0000000000000000  00001968
       0000000000000618  0000000000000018          29    45     8
  [29] .strtab           STRTAB           0000000000000000  00001f80
       000000000000023d  0000000000000000           0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), l (large)
  I (info), L (link order), G (group), T (TLS), E (exclude), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

I trimmed this one for the sake of brevity. We see our 30 sections listed with various names (e.g., `.note.ABI-tag`) and types (e.g., `SYMTAB`).

You might be confused by now. Don't worry - it won't be on the test. I'm explaining this because we're interested in a specific part of this file: In their *Program Header Table*, ELF files can have (and shared libraries in particular must have) a *segment header* that describes a *segment* of type `PT_DYNAMIC`. This segment owns a section called `.dynamic` which contains useful information to understand dynamic dependencies.

Direct Dependencies

We can use the `readelf` utility to further explore the `.dynamic` section of our executable [2]. In particular, this section contains all of the dynamic dependencies of our ELF file. We only specified `librandom.so` as a dependency, so we would expect there to be exactly one dependency listed:

```
$ readelf -d main | grep NEEDED
 0x0000000000000001 (NEEDED)             Shared library: [librandom.so]
 0x0000000000000001 (NEEDED)             Shared library: [libstdc++.so.6]
 0x0000000000000001 (NEEDED)             Shared library: [libm.so.6]
 0x0000000000000001 (NEEDED)             Shared library: [libgcc_s.so.1]
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
```

We can see `librandom.so` , which we specified, but we also get four extra dependencies we didn't expect. These dependencies seem to appear in all compiled shared libraries. What are they?

- `libstdc++` : The standard C++ library.
- `libm` : A library that contains basic math functions.
- `libgcc_s` : The GCC (GNU Compiler Collection) runtime library.
- `libc` : The C library: the library which defines the 'system calls' and other basic facilities such as `open` , `malloc` , `printf` , `exit` , etc.

Okay - so we know that `main` knows it depends on `librandom.so` . So why can't `main` find `librandom.so` in runtime?

Runtime Search Path

`ldd` is a tool that allows us to see *recursive* shared library dependencies. That means we can see the complete list of all shared libraries an artifact needs at runtime. It also allows us to see *where* these dependencies are located. Let's run it on `main` and see what happens:

```
$ ldd main
	linux-vdso.so.1 =>  (0x00007fff889bd000)
	librandom.so => not found
	libstdc++.so.6 => /usr/lib/x86_64-linux-gnu/libstdc++.so.6 (0x00007f0
	libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007f07c52bf000)
	libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007f07c50a9
	libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f07c4ce4000)
	/lib64/ld-linux-x86-64.so.2 (0x00007f07c58c9000)
```

Right off the bat, we see that `librandom.so` is listed - but not found. We can also see that we have two additional libraries (`vdso` and `ld-linux-x86-64`). They are indirect dependencies. More importantly, we see that `ldd` reports the *location* of the libraries. Consider `libstdc++` . `ldd` reports its location to be `/usr/lib/x86_64-linux-gnu/libstdc++.so.6` . How does it know?

Each shared library in our dependencies is searched in the following locations [3], in order:

1. Directories listed in the executable's `rpath`.
2. Directories in the `LD_LIBRARY_PATH` environment variable, which contains colon-separated list of directories (e.g., `/path/to/libdir:/another/path`)
3. Directories listed in the executable's `runpath`.
4. The list of directories in the file `/etc/ld.so.conf`. This file can include other files, but it is basically a list of directories - one per line.
5. Default system libraries - usually `/lib` and `/usr/lib` (skipped if compiled with `-z nodefaultlib`).

Fixing our Executable

Alright. We validated that `librandom.so` *is* a listed dependency, but it can't be found. We know where dependencies are searched for. We'll make sure that our directory is not actually on the search path by using `ldd` again:

```
$ LD_DEBUG=libs ldd main
      [..]

      3650:     find library=librandom.so [0]; searching
      3650:      search cache=/etc/ld.so.cache
      3650:      search path=/lib/x86_64-linux-gnu/tls/x86_64:/lib/x86_64-lin
      3650:       trying file=/lib/x86_64-linux-gnu/tls/x86_64/librandom.so
      3650:       trying file=/lib/x86_64-linux-gnu/tls/librandom.so
      3650:       trying file=/lib/x86_64-linux-gnu/x86_64/librandom.so
      3650:       trying file=/lib/x86_64-linux-gnu/librandom.so
      3650:       trying file=/usr/lib/x86_64-linux-gnu/tls/x86_64/librandom.
      3650:       trying file=/usr/lib/x86_64-linux-gnu/tls/librandom.so
      3650:       trying file=/usr/lib/x86_64-linux-gnu/x86_64/librandom.so
      3650:       trying file=/usr/lib/x86_64-linux-gnu/librandom.so
      3650:       trying file=/lib/tls/x86_64/librandom.so
      3650:       trying file=/lib/tls/librandom.so
      3650:       trying file=/lib/x86_64/librandom.so
      3650:       trying file=/lib/librandom.so
      3650:       trying file=/usr/lib/tls/x86_64/librandom.so
      3650:       trying file=/usr/lib/tls/librandom.so
      3650:       trying file=/usr/lib/x86_64/librandom.so
      3650:       trying file=/usr/lib/librandom.so
```

```
        [..]
```

I trimmed the output since it's very… chatty. It's no wonder our shared library is not found - the directory where `librandom.so` is located is not in the search path! The most ad-hoc way to solve this is to use `LD_LIBRARY_PATH`:

```
$ LD_LIBRARY_PATH=. ./main
```

It works, but it's not very portable. We don't want to specify our lib directory every time we run our program. A better way is to put our dependencies *inside the file*.

Enter `rpath` and `runpath`.

rpath and runpath

`rpath` and `runpath` are the most complex items in our runtime search path "checklist". The `rpath` and `runpath` of an executable or shared library are optional entries in the `.dynamic` section we reviewed earlier [4]. They are both a list of directories to search for.

The only difference between `rpath` and `runpath` is the order they are searched in. Specifically, their relation to `LD_LIBRARY_PATH` - `rpath` is searched in *before* `LD_LIBRARY_PATH` while `runpath` is searched in *after*. The meaning of this is that `rpath` cannot be changed dynamically with environment variables while `runpath` can.

Let's bake `rpath` into our executable and see if we can get it to work:

```
$ clang++ -o main main.o -lrandom -L. -Wl,-rpath,.
```

The `-Wl` flag passes the following, comma-separated, flags to the linker. In this case, we pass `-rpath .` . To set `runpath` instead, we would also have to pass `--enable-new-dtags` [5] . Let's examine the result:

```
$ readelf main -d | grep path
 0x000000000000000f (RPATH)              Library rpath: [.]

$ ./main
```

The executable runs, but this added `.` to the `rpath`, which is the current working directory. This means it won't work from a different directory:

```
$ cd /tmp
$ ~/code/shared_lib_demo/main
/home/nurdok/code/shared_lib_demo/main: error while loading shared libraries:
```

We have several ways to solve this. The easiest way is to copy `librandom` to a directory that is in our search path (such as `/lib`). A more complicated way, which, *obviously*, is what we're going to do - is to specify `rpath` relative to the executable.

$ORIGIN

Paths in `rpath` and `runpath` can be absolute (e.g., `/path/to/my/libs/`), relative to the current working directory (e.g., `.`), but they can also be *relative to the executable*. This is achieved by using the `$ORIGIN` variable [6] in the `rpath` definition:

```
$ clang++ -o main main.o -lrandom -L. -Wl,-rpath,"\$ORIGIN"
```

Notice that we need to escape the dollar sign (or use single quotes), so that our shell won't try to expand it. The result is that `main` works from every directory and finds `librandom.so` correctly:

```
$ ./main
$ cd /tmp
$ ~/code/shared_lib_demo/main
```

Let's use our toolkit to make sure:

```
$ readelf main -d | grep path
 0x000000000000000f (RPATH)              Library rpath: [$ORIGIN]

$ ldd main
        linux-vdso.so.1 =>  (0x00007ffe13dfe000)
        librandom.so => /home/nurdok/code/shared_lib_demo/./librandom.so (0x0
        [..]
```

Runtime Search Path: Security

If you ever changed your Linux user password from the command line, you may have used the `passwd` utility:

```
$ passwd
Changing password for nurdok.
(current) UNIX password:
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
```

The password hash is stored in `/etc/shadow`, which is root protected. How then, you might ask, your non-root user can change that file?

The answer is that the `passwd` program has the `setuid` bit set, which you can see with `ls` :

```
$ ls -l `which passwd`
-rwsr-xr-x 1 root root 39104 2009-12-06 05:35 /usr/bin/passwd
#  ^--- This means that the "setuid" bit is set for user execution.
```

It's the `s` (the fourth character of the line). All programs that have this permission bit set run as the owner of that program. In this example, the user is root (third word of the line).

"What does that have to do with shared libraries?", you ask. We'll see with an example.

We'll now have `librandom` in a `libs` directory next to `main` and we'll bake `$ORIGIN/libs` [7] in our `main` 's `rpath` :

```
$ ls
libs  main
$ ls libs
librandom.so
$ readelf -d main | grep path
 0x000000000000000f (RPATH)              Library rpath: [$ORIGIN/libs]
```

If we run `main` , it works as expected. Let's turn on the `setuid` bit for our `main` executable and make it run as `root` :

```
$ sudo chown root main
$ sudo chmod a+s main
$ ./main
./main: error while loading shared libraries: librandom.so: cannot open share
```

Alright, `rpath` doesn't work. Let's try setting `LD_LIBRARY_PATH` instead:

```
$ LD_LIBRARY_PATH=./libs ./main
./main: error while loading shared libraries: librandom.so: cannot open share
```

What's going on here?

For security reasons, when running an executable with elevated privileges (such as `setuid`, `setgid`, special capabilities, etc.), the search path list is different than normal: `LD_LIBRARY_PATH` is ignored, as well as any path in `rpath` or `runpath` that contains `$ORIGIN`.

The reason is that using these search path allows to exploit the elevated privileges executable to run as `root`. Details about this exploit can be found here. Basically, it allows you to make the elevated privileges executable load your own library, which will run as root (or a different user). Running your own code as root pretty much gives you absolute control over the machine you're using.

If your executable needs to have elevated privileges, you'll need to specify your dependencies in absolute paths, or place them in the default locations (e.g., `/lib`).

An important behavior to note here is that, for these kind of applications, `ldd` lies to our face:

```
% ldd main
        linux-vdso.so.1 =>  (0x00007ffc2afd2000)
        librandom.so => /home/nurdok/code/shared_lib_demo/libs/librandom.so (
        libstdc++.so.6 => /usr/lib/x86_64-linux-gnu/libstdc++.so.6 (0x00007f1
        libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007f1f660c0000)
        libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007f1f65eaa
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f1f65ae5000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f1f668cc000)
```

`ldd` doesn't care about `setuid` and it expands `$ORIGIN` when it is searching for our dependencies. This can be quite a pitfall when debugging dependencies on `setuid` applications.

Debugging Cheat Sheet

If you ever get this error when running an executable:

```
$ ./main
./main: error while loading shared libraries: librandom.so: cannot open share
```

You can try doing the following:

- Find out what dependencies are missing with `ldd <executable>`.
- If you don't identify them, you can check if they are direct dependencies by running `readelf -d <executable> | grep NEEDED`.
- Make sure the dependencies actually exist. Maybe you forgot to compile them or move them to a `libs` directory?
- Find out where dependencies are searched by using `LD_DEBUG=libs ldd <executable>`.
- If you need to add a directory to the search:
    - Ad-hoc: add the directory to the `LD_LIBRARY_PATH` environment variable.
    - Baked in the file: add the directory to the executable or shared library's `rpath` or `runpath` by passing `-Wl,-rpath,<dir>` (for `rpath`) or `-Wl,--enable-new-dtags,-rpath,<dir>` (for `runpath`). Use `$ORIGIN` for paths relative to the executable.
- If `ldd` shows that no dependencies are missing, see if your application has elevated privileges. If so, `ldd` might lie. See security concerns above.

If you still can't figure it out - you'll need to read the whole thing again :)

Sources

- "ELF (Execuable and Linkable Format)"/ Wikipedia
- "Linker and Libraries Guide" / Oracle
- The GNU C Library (glibc)
- "Shared Libraries" / The Linux Documentation Project
- "Where do executables look for shared objects at runtime" / Unix & Linux SE
- "Application Binary Interface"
- "The ELF format - how programs look from the inside" / Christian Aichinger
- "Rpath" / Wikipedia
- "GNU Dynamic Loader Search Directories" / TechBlog
- "ld.so: Dynamic Link library support for the Linux OS"
- "How does the 'passwd' command gain root user permissions?" / Unix & Linux SE
- "ELF Object File Format" / nairobi-embedded

Discuss this post at Hacker News, /r/Programming, or the comment section below.
Follow me on Twitter, Facebook or Google+
Thanks to Hannan Aharonov, Yonatan Nakar and Shachar Ohana for reading drafts of this.

Similar Posts

- My Contribution to LEDE (OpenWrt): A Hacktoberfest Adventure
- Multiple Custom Commands in Shell Startup (for Terminator)
- Making History with Bash

<< See All Posts

6 replies

Neeraja_Neeru May '18

Hi Amir, Very well and precisely explained. Thanks for sharing.

Joe_Higgins Jul '18

Thanks! Very useful & concise article.

JHArp Nov '18

The best tutorial I have read for a long time.
Perfect.
Thanks for the help and your time

---

lurban58 17 Jan

Registered just to say thank you. Your article presumably saved me at least half a day of research.

ichernev 23 Jan

Thank you very much for the great article!

It helped me troubleshoot a very peculiar issue. I just want to point out, that `ldd -v` which lists recursive dependencies, does not list the ones that are not found.

So the only true way to find out where a (missing) dependency is coming from is to use `readelf -d <exe or lib> | grep 'NEEDED\|RPATH\|RUNPATH'` and trace every dependency by hand.

Also keep in mind that for a given shared lib/executable its dependencies are searched according to the RPATH/RUNPATH specified by that dependency. So even if a path is in your executable RUNPATH, a library inside that runpath can still be missing, because its an indirect dependency of another library, which does not have RUNPATH (i.e it assumes that libs are put inside a globally accessible /etc/ld.so.conf location).

AKPKUMAR 21 Feb

Hi, I am facing a issue which I will best try and describe as below.
I have 2 libraries in my environment which both exposes the same api.
libraryone : is a pc library,