Dynamic Libraries

G. Lettieri

25 October 2023

1 Introduction

Static libraries are just a collection of object files. In Linux, an `.a` is just an *archive* of `.o` files created using the `ar`(1) command, an ancient archive-management tool that survives today only for this purpose[1] During linking, the link editor extracts the object files from the archive as needed and adds them to the list of files to link. The resulting executable keeps no record of the fact that some objects originally came from a library (except perhaps in debugging info, if any).

Dynamic libraries attempt to address some of the (perceived) shortcomings of static libraries:

- Objects used in mutiple executables (e.g., those extracted from the C library) are copied multiple times wasting disk and central memory space;
- if a library needs to be updated to fix a bug, all executables built with the old library must be identified and rebuilt using the new one.

Dynamic libraries solve these problems by having "incomplete" executables that are linked with the libraries at load time. The libraries can now be updated without updating the executables[2]. In addition, the libraries are built and linked in such a way that multiple processes can share almost all of their contents. The price of all this is slower executable startup times (because of the dynamic linkage), slightly slower libraries (because of the way they are compiled) and, above all, a lot of management complexity due to possible incompatibilities among library versions. Some people (like the `go` developers) think that the price to pay is too high while the benefits are either marginal or non-existent (space is not a problem nowadays, and library version incompatibilities often cause executable updates anyway) and so they only use static libraries. From a

[1] Because of this specialized use, modern GNU `ar` can also add a symbol index to the archive all by itself. In the past a separate tool, `ranlib`, was needed.

[2] What happens to long-running executables, such as servers and daemons, that where loaded and linked with the old library? As long as the new library has a different filename (e.g., a different version number), running processes will not be affected, not even if the old library is deleted (Unix will keep it around as long as there are running processes that reference it). However, if we want the servers and daemons to use the new library, we need to restart them.

security standpoint, dynamic libraries are both a problem and an opportunity: on the one hand they introduce a number of exploitable data structures and code (GOT/PLT, writable constructor and destructor lists, ...), but on the other hand, they enable a more effective randomization of address spaces.

2 Basic workflow

Shared libraries are implemented as ELF DSOs (Dynamic Shared Objects). This is a type of ELF file that contains both *program* and *section* headers and, therefore, contains both loadable code and data, as well as relocation information. The address space of a process can contain the image of an EXE ELF file plus any number of DSOs (including zero).

DSOs are used at least twice during the lifetime of an executable:

1. at build time, when the executable is linked;
2. at load time.

(They can also be used at runtime, by explicitly loading DSOs with the `dlopen()` function.) At build time the linker only resolves undefined symbols in the executable and remembers which DSOs define them. It adds the names of these DSOs to a special `.dynamic` section in the executable ELF file. The executable is specially marked so that the kernel can know that, when the program will be loaded, it will need dyamic linking. Note that is not the kernel that will do the linking: a userspace dynamic linker will be called to do the job. For this reason, the executable must contain the filesystem path of the required dynamic linker in a special `.interp` section (by writing different paths in this section, each executable can have its own dynamic linker).

At load time (*each time the executable is loaded*), the kernel loads the ELF file as usual, but then it also loads the specified dynamic linker in the same address space, pushes some additional information on the stack (the *auxiliary vector*), and finally transfers control to the dynamic linker. The main purpose of the auxiliary vector is to tell the dynamic linker where the kernel has loaded the main executable.

The dynamic linker uses the information contained in the `.dynamic` section and in the auxiliary vector to:

1. find all needed DSOs and resolve all undefined symbols (recursively);
2. patch the EXE and all the DSOs to link them together;
3. run all the initialization code ("constructors").

Finally, control goes to the executable entry point.

We can already make some observations.

- The DSOs found in step 1 above are not necessarily the same as those identified at build time. They are not even managed by the same people,

since the latter ones belong to the developers, while the former ones belong to the administrator of the system on which the executable is running. Great care must be taken to ensure that the two sets of DSOs are at least compatible.

- The DSOs mentioned in the executable may themselves be dynamically linked, i.e., each of them may mention further DSOs that need to be loaded (and so on). This can result in the same DSO being mentioned several times. Usually only one copy of each DSO is loaded. However, it is possible for multiple versions of the same DSO to coexist, as long as they are named differently (e.g., by using the version number as part of the name).

- Linking requires patching, i.e., modifying the executable and the DSOs to fix relocatable addresses: how can the images be shared among several processes, then? The solution is to isolate all relocatable addresses in a data structure, the Global Offset Table (GOT for short), and code all instructions to either use relative addresses, or fetch absolute addresses from the GOT. In this way, all the code and all read-only data sections can be shared by all processes that have loaded the same DSO. Only the writable data sections and the GOT must be private for each process. Note that this applies to *all* loaded DSOs besides the main executable: each one of them will have its own GOT.

- *Entire* DSOs are loaded, not just the needed object files as in a static library. This seems to go against the desire to save space. However, on-demand paging comes to the rescue, as it will bring into memory only the DSO parts that are actually used (another reason to keep all the relocations in the same place, otherwise the dynamic linker would fault-in many more pages during the patching phase).

Note that the linker code and data structures remain there while the executable is running and can still be used by the runtime support for the *lazy-binding* mechanism (explained below), by the main program, which may want to programmatically load dynamic objects at runtime and, finally, by attackers.

2.1 Building a shared library

Building a shared library is more like building an executable than a static library: the set of object files is assembled into a single ELF file, rather than into an archive. The easiest way to do this is to pass the `-shared` option to `gcc`.

Note that the same object file (an `.o` file which is the result of compilation and/or assembly) can be used several times for different purposes: it can be included in an executable ELF, a static library, or into a dynamic library (ELF DSO). To enable sharing, objects destined for dynamic libraries should be

compiled with `-fpic` or `-fPIC`, which ask for Position Independent Code[3]. In fact, a shared object may end up running at different addresses in each process that shares it, so it cannot make any assumptions about the addresses of its own parts. A non-shared object does not have this limitation, since it can be patched as needed once the addresses are known.

However, even if we don't want sharing, `-fpic` is still needed, as it is the only way to enable *interposing*, which is a requirement for any library. "Interposing" means that any program linking to a library must be able to redefine any of the symbols exported by the library. Suppose that a function `f1()` defined in the library internally calls a function `f2()` exported by the library. Suppose further that the main program redefines `f2()`. In the final process image, the function `f1()` must call the redefined `f2()` and not the one originally defined in the library. For static libraries this either means that each exported symbol lives in its own object file (so that already defined symbols are never extracted from the library), or that exported symbols are defined "weak" (the linker only picks a weak symbol if it is otherwise undefined). In either case, the call site in `f1()` will contain a relocation which will be filled with the correct address during the normal linking process. Dynamic libraries are different, as patching the text section should be avoided to allow sharing. This means that `f1()` must not assume to know the address of `f2()`; instead, it should calculate it based on information that is only available at runtime. This is almost the same problem already solved by Position Independent Code and is therefore treated in a similar way.

Exercises

2.1. When a developer creates a program that uses non-standard dynamic libraries, shipping the program to users becomes cumbersome: the users must obtain and install these libraries in order to run the program. To simplify the installation, the developer can package the libraries and ship them with the program. However, dynamic libraries must be placed in specific directories that depend on the (possibly unknown) configuration of the users' systems, otherwise the dynamic loader won't find them. Another possibility is to put the search path of the libraries in the program itself: this way, the developer can choose where to put the libraries, independent of the user's configuration. The `gcc` compiler and the GNU libc dynamic loader support this feature with the `RUNPATH` dynamic variable, which can be written in the program by passing the `-Wl,-rpath=<dirs>` option to the compiler, where `<dirs>` is a colon-separated list of directories. However, if not used carefully, this feature can be exploited by attackers. Connect with

```
ssh -p 4422 dll1@lettieri.iet.unipi.it
```

[3]The only difference between the two options is that the all-caps one tries to overcome some size limitations in some architectures, possibly at the expense of efficiency. On x86 the two options are equivalent.

with password `dll1` and try to steal the `flag.txt` file by abusing the `dll1` program.

2.1.1 The Global Offset Table

Position independence in shared objects is achieved through indirection: the code does not hard-code addresses in the instructions (not even relative addresses), but fetches them from a table of pointers, the Global Offset Table, or GOT for short, which is created by the linker and filled by the dynamic loader. The same GOT is also used to store the addresses of symbols used but not defined in the DSO, i.e., external symbols imported from other DSOs. The dynamic loader will fill these entries as well. To let the dynamic loader fill the GOT, the linker creates relocation entries not different from the entries created by the assembler. The GOT is just a data structure in the data part of the DSO which is patched at load time. Note that this means that the GOT is not shared among all the loaders of the same DSO, but this is also true for all data sections, as we have seen.

Note that the GOT may be generated also for reasons other than PIC, e.g., to overcome the addressing limitations of some architectures. For example, depending on the memory model used, 64b Intel/AMD architectures may need a GOT to address data which can reside anywhere in the 64b address space.

The code that accesses an entry in the GOT needs several pieces of information: where is the GOT? which GOT entry do I need? The code usually accesses the GOT using RIP-relative addresses (since the GOT is mostly used by PIC code). The offset of the GOT from RIP is filled in by the linker (via a relocation) and then the code only needs to know its RIP. This is easy for AMD64, since rip-relative addressing is available in the instruction set. Intel 32b code, instead, uses a method called "thunking": there is a call to a thunk function which copies the rip saved on the top of its stack into a register and then returns. The address of the GOT thus computed is usually held in a register, but the ABI gives no guarantee on the contents of this register at function entry, so each function must usually recompute it.

The other part of the information needed to access the GOT, i.e., the index of the entry, is only known to the linker, since it is the linker that builds the GOT after collecting all the needed entries. There is a special relocation that the assembler generates and that causes the linker to fill in this index where needed.

One final note: it should be clear from the above that there is one GOT for each DSO and dynamically linked executable. In a process address space where many DSOs have been loaded there will be several GOTs: the one from the executable and one for each loaded DSO.

2.1.2 Lazy binding and the Procedure Linkage Table

Filling a GOT entry requires a non-trivial amount of work from the dynamic loader, which must look up the corresponding symbol into all loaded DSOs until

it finds a match (ELF files come which an hash table of dynamic symbols for just this purpose). However, in each process run it is possible that many GOT entries will not be actually be used. This is especially true for entries found in the dynamic libraries' GOTs: remember, for example, that the entire libc is loaded, even if you only call `printf`. All the libc GOT entries not directly or indirectly used by `printf` will not be needed by your program. For a large program using many dynamic libraries (which may in turn load other dynamic libraries), the number of these unused entries may add up to a considerable number, so it is tempting to avoid filling them. The usual technique used to solve this kind of problems is "laziness": delay the work until the latest time before it is actually needed. If it is never needed, the work will not be performed. Even if most entries are actually needed, laziness can improve startup time by distributing the work more evenly during runtime.

Entries corresponding to functions are first needed when the function is actually called. Laziness, here, is implemented by letting the code call a stub function instead of the actual function. The stub function, when called for the first time, calls the dynamic loader (which, remember, is still there in the address space) to resolve the symbol, then calls the real function. The stub also makes sure that the next time the code will call directly into the real function, skipping the stub. This is achieved using the GOT: the GOT entries that should point to the imported library functions actual point to the stub. The stub then replaces the entry with a pointer to the real function.

The stubs are also created by the linker, which then must also be able to generate code. This arrangement is needed since only the linker has the complete list of all the external functions needed by the executable or DSO. The stubs are put one after the other in a Procedure Linkage Table, or PLT for short. If the code calls the external function `foo`, the call is redirected to `foo@plt`, which is the label of the corresponding stub. The first stub instruction then jumps indirectly through the `foo` entry in the GOT. The dynamic loader, however, initially fills this entry with a pointer to the stub itself, to a set of instructions that eventually call the dynamic loader, passing it the index of the `foo` GOT entry (since there is a stub for each entry, this is a constant in the stub itself, computed at link time). The dynamic loader will then resolve the symbol, update the the GOT entry and call the function. Note that the stub must know the address of the dynamic loader, which is unknown until load time: an entry of the GOT is reserved for this purpose and is filled by the dynamic loader, during initialization, with the address of its symbol-lookup routine. All stubs jump to a common code (at the start of the PLT) which calls through this entry of the GOT. The dynamic loader must also know which of the several loaded DSOs is calling the look up routine. For this purpose, another entry of the GOT is also reserved, and contains a pointer to a per-DSO data structure allocated by the dynamic loader. The common stub code passes this pointer as parameter to the lookup routine.

Laziness is typically not implemented for GOT entries pointing to global data, probably because it is not trivial to remove the stub from the code path after the first access and it is also not worth the trouble, since there should be

very few global data objects anyway.

2.1.3 Constructors and destructors

DSOs can have initialization code that is automatically run before the executable `main()` function, and cleanup up code that is run during normal execution (when `main()` returns or the process calls `exit()`). GCC allows the programmer to specially mark any function as a "constructor" using the `__attribute__((constructor))` syntax. A pointer to such a function will be added to the `.init_array` section, so that a table of function pointers will be automatically built when the linker will put together all the `.init_array` sections contained in all the object files. This table will be located and walked over by the dynamic linker when the DSO will be loaded into a process. Something similar is done for destructors, which will be called when the DSO is unloaded (usually when the program terminates).

Exercises

2.1. The GOT, PLT, constructors and destructors tables all provide useful writeable function pointers that can be overwritten by attackers. Depending on the type of bug that the attacker is trying to exploit, these pointers may be much more convenient than the RIP addresses stored on the stack. Try to solve the `canary1` exercise again, this time overwriting one of these new data structures instead of the saved RIP.

2.1.4 Versioning

Since dynamic executables and the libraries they depend on are distributed separately, we face the problem of *versioning*, i.e., knowing which version of each library and executable can be safely dynamically linked.

Libraries may change between releases in at least two ways: by introducing new functionalities while continuing to support the old ones, or introducing incompatible changes. Version numbers usually try to reflect this by having a part that changes when new, backward compatible changes are introduced, and another one that signals incompatible ("breaking") changes. There is unfortunately no universally agreed upon standard, but "Semantic versioning" is the most widely adopted scheme. In this scheme, the version number is decomposed as "major.minor.patch", where "major" is bumped when breaking changes are introduced, "minor" for non-breaking changes, and "patch" only for very small changes such as bug fixes. When an executable (or a library depending on other libraries) is built, we want to remember the major.minor version of the libraries used, call it $M.m$. At load time we can accept any version $M.m'.p$ with exactly the same major version M, a minor version $m' \geq m$, and any patch level p. This requirements reflect the fact that our executable may depend on functions not found in older releases, but it should not be affected by the non breaking changes introduced in more recent minor releases.

In linux systems these requirements are partially satisfied using some symbolic links and the DSOs "SONAME" (Shared Object NAME) tag in the `.dynamic` ELF section. For example, when a library `lib`x with version $M.m.p$ is built, it is tagged with a SONAME `lib`x`.so.`$M.m$. The file system were the program is built will contain the file `lib`x`.so.`$M.m.p$ and a symbolic link `lib`x`.so` pointing to it. At link time, the linker option `-l$x$` will cause the linker to look up the file `lib`x`.so` and find the symbolic link. The linker will then read the SONAME tag and copy it into a DT_NEEDED tag of the executable. When the executable is shipped, we can assume that the target system will contain a `lib`x`.so.`$M.m.p'$ library installed, and a `lib`x.so.$M.m$ symbolic link pointing to it. At load time the dynamic linker will read the DT_NEEDED tag in the executable `.dynamic` section and look for a `lib`x.so.$M.m$ file, thus finding the symbolic link. In this way, an executable built using version $M.m.p$ will be run using version $M.m.p'$ of `lib`x, thus ignoring the patch level. The solution is not able to automatically accept a minor $m' \geq m$, but this limitation can be sometimes overcome by adding more symbolic links.

The hardest problem, however, comes when a system contains executables (or libraries) that depend on different, incompatible versions of the same library. Even worse, the same executable may depend on two such versions, e.g., one version directly and another version indirectly, through the dependencies of some linked library. The former problem can be solved in the previous scheme by having both library versions installed, with `lib`x`.so` pointing to the newest one, so that new executables are built against the latest version, while older executables will find the older library version using the SONAME. The latter problem is probably unsolvable, since loading two different versions of the same library is unlikely to work (think of some global state that should be maintained by the library, or of some data structure created by one version and passed to the other one).

Linux implements a finer grained scheme that can be used to solve all of the above problems, with some cooperation from the library authors. Rather than having only per-library versions, linux libraries can assign a different version to each symbol and they can export several versions of the same symbol, thus resolving internally any compatibility problem. Versions are assigned to symbols using a "version script" when the shared library is built (`--version-script` option of the linker). They show up in the symbol tables with the "symbol@version" syntax. Each symbol also has a default version, which uses two at-signs instead of one. The default version is picked by the linker when linking the library to an executable at build time. The executable thus remembers the version of each symbol it needs. At load time, the dynamic linker searches the library for the exact version of each symbol.

2.2 Building a dynamically linked executable

The linker will tell the difference between static and dynamic libraries and act accordingly. If both static and dynamic versions of the same library are available (`.a` vs `.so` files), the linker will choose the dynamic one unless `-static` is

passed.

Symbols resolved by dynamic linking go into a `.dynsym` section, while the relative strings go into their own `dynstr` section. These sections are not stripped by `strip` and will go into a loadable read-only ELF segment, together with other sections related to symbol look-up, relocation and versioning.

The linker also adds the `.dynamic` and `.interp` sections. The `.dynamic` section is organized as a table of "tag" and "value" entries. The tags are agreed-upon codes that determine what the value means. One possible tag is `DT_NEEDED`, and the corresponding value is the `SONAME` of a DSO that contains the definitions of symbols needed by the object. The DSOs are looked up in the file system in a set of standard places, plus all the directories mentioned in `-L` flags. The `.interp` section is filled with the argument of the `-loader` option.

The linker will also create a GOT and a PLT as required. This is actually triggered by the presence, in the linked object files, of relocations related to the GOT and the PLT.

The linker also selects the default version of each symbol, as found in the linked DSO, and records it in the dynamic symbol table. The set of versions (from all imported symbols) is also recorded in the executable, so that a quick check on the available versions can be performed at load time.

2.3 Loading the executable

Loading an executable is an activity that starts in the kernel, when one of the `exec()` primitives is executed, and is completed in user space by the system dynamic loader.

2.3.1 The kernel

To load a dynamically linked executable, the kernel initially performs the same actions as for any executable, interpreting the ELF file and loading the text and data sections into memory (actually, creating the page tables that link virtual addresses to the location of the corresponding pages in the file), then creating and filling the stack and so on. Then, it also loads (in a different part of the address space) the executable mentioned in the `.interp` section and pushes on the stack the *auxiliary vector*. Finally, it yields control to the entry point of the *interpreter*, instead of the executable.

2.3.2 The dynamic linker

The interpreter is a statically linked, position-independent executable, usually living in the `/lib` directory with a name like `ld-linux.so` or something like that.

The interpreter uses the auxiliary vector on the stack to locate the executable and, in particular, its `.dynamic` section. It then starts to look up all the libraries mentioned in the `DT_NEEDED` tags of the `.dynamic` section, loading

each one of them and their dependencies (recursively). To load the libraries it actually uses the `mmap()` system call to map the needed segments of the library ELF files into the address space of the current process.

The loader looks for each library in many directories in turn. The set of directories depends on its own configuration, on the configuration of the system, on options built into the executable itself, and also on the preferences of the user running the executable (`man ld-linux` for the details).

The loader then performs all the relocations found in the in the ELF files, in particular filling the GOT tables of all the loaded DSOs. Then it arranges for the destructors to be called at normal program exit (using `atexit()`), it runs all the constructors, and finally it jumps to the entry point of the executable.