A. Dynamic Libraries

[...] every major operating system implements them but that does not mean they are a good idea.

Rob Pike, *9fans mailing list*, 2008

Static libraries are just a collection of object files. In Linux, a "`.a`" file 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.

Because of its 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.

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 multiple 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. Now the libraries can now be updated on disk without updating the executables, and all new processes will automatically pick the updated version.

R As long as the new library has a different filename (e.g., a different version number), running processes (e.g., server) 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.

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, see Section A.4), slightly slower libraries (because of the way they are compiled, see Section A.3.1) and, above all, a lot of management complexity due to possible incompatibilities among library versions (see Section A.6). 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 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.

A.1 Basic workflow

Shared libraries are implemented as ELF DSOs (Dynamic Shared Objects), with file names typically ending in ".so" (for Shared Object). 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 enter at least twice in the lifetime of an executable:

1. at build time or, more precisely, at link time;
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 dynamic linking. Note that it 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* that we mentioned briefly in Section 6.3.1), 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.

Without going into further details, we can already make some observations.

- The DSOs found at load time (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

f2.c

```
#include <stdio.h>
extern long myvar;
void f2() {
  myvar++;
  printf("libx:f2()\n");
}
```

f1.c

```
extern void f2(void);
long myvar;
long f1() {
  f2();
  return myvar;
}
```

Figure A.1 – Example files for `libx.so`

(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, see Section A.2.1 below), and code all instructions to either use relative addresses, or fetch absolute addresses from the GOT (see Section A.3.1). 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 dynamic 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 (see Section A.5), by the main program, which may want to programmatically load dynamic objects at runtime ("`man 3 dlopen`"), and, finally, by attackers.

A.2 Building a dynamically linked executable

Let's see how to create a dynamically linked executable. We will use the files in Figure A.1 as an example. You can download the example code from

https://lettieri.iet.unipi.it/hacking/dynlib-1.0.zip

Unzip the file and enter the `dynlib-1.0` directory. The `pre-built/libx.so` file contains a dynamic library built from the files in Figure A.1 (we will see how to do build `libx.so` in Section A.3). We can create a program that uses any of the symbols exported by the library (in this example the symbols are `f1`, `f2` and `myvar`), e.g.:

```
#include <stdio.h>
extern long f1(void);
int main()
{
  printf("%ld\n", f1());
}
```

Put the above in a `main.c` file and compile it to `main.o`. To link it to the library, we can simply add the `pre-built/libx.so` file to the list of files to link:

```
$ gcc -o main main.o pre-built/libx.so
```

R Your Linux distribution may have enabled some compilation options that may be a source of confusion at this point and, for the purpose of the current discussion, are better disabled. In particular, you should pass the additional flags `-no-pie`, `-fcf-protection=none`, `-fno-PIC`, and possibly others. The `Makefile` in the `dynlib-1.0` directory contains the exact commands.

Alternatively, we can have the linker *search* for the library with this (probably more familiar) command:

```
$ gcc -o main main.o -lx
```

Where `x` is what is left after stripping the `lib` prefix and the `.so` suffix. Note that for this second option to work, the library file must have a name that follows the `lib`*name*`.so` or lib*name*`.a` convention, and must be placed in one of the directories that the linker automatically looks for. You can add new directories to the list using the `-L`*directory-path* option. The linker will be able to find it with the following command:

```
$ gcc -o main main.o -Lpre-built -lx
```

The linker will recognize the difference between static and dynamic libraries (not based on the names, but on their actual contents) and act accordingly. If both static and dynamic versions of the same library are found during the search, 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 removed by `strip` and go into a loadable read-only ELF segment, along with other sections related to symbol lookup, relocation, and versioning. Their contents will be available in memory when the dynamic linker starts (see Section A.4). We can see the symbols in the `.dynsym` section with

```
$ nm -D main
```

You can check that the dynamic symbols are still there, even if you run “`strip main`”, while the “normal” symbols have been removed (“`nm main`” without the `-D` option will complain that there are no symbols). The dynamic symbols cannot be removed, because they are needed by the dynamic linker.

The linker also adds the `.dynamic` and `.interp` sections. The `.dynamic` section contains information needed by the dynamic linker and 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 `NEEDED`, and the corresponding value is the name (actually, its `SONAME`, see Section A.6) of a DSO containing the definitions of symbols needed by the object. We can read the `.dynamic` section of our executable with

```
$ readelf -d main
```

When you run the above command, you should see two `NEEDED` entries:

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

(The first column is the numeric value of the tag.) We can see that our executable needs `libx.so` and also the C library, which was added by default by `gcc`.

The `.interp` section is filled with the argument of the `-loader` option, that `gcc` passes to the linker by default when we build a dynamic executable. We can read its contents with

```
$ readelf -p.interp main
```

The default interpreter for AMD64 Linux binaries should be `/lib64/ld-linux-x86-64.so.2`.

A.2.1 The Global Offset Table

Although it is an executable, our `main` program still contains undefined symbols and relocation instructions. We can read the relocations with the following command:

```
$ readelf -r main
```

In the output of the above command, we should see a few lines that mention `printf()` and `f1()`, the two functions that `main.c` uses without giving a definition for them:

```
000000404000 [...] printf@GLIBC_2.2.5 + 0
000000404008 [...] f1 + 0
```

Only the dynamic linker will know the addresses of these functions, and therefore it will have to patch these addresses back into the loaded executable image. The first column is the address of the first byte of the (loaded) binary that should be patched, once the value of the expression in the last column becomes known. The exact addresses that you read in the first column may differ from those above; however, if you check the load address of the sections of our executable with

```
$ readelf -WS main
```

you should be able to see that the relocation addresses are *not* in the `.text` section and are not part of any `call` instruction, as we might have expected. Instead, they fall into a section called `.got` (`.got.plt` in recent binaries). This is a section that contains the Global Offset Table, or GOT for short. The GOT is a table of pointers created *by the static linker* (not the compiler), that contains an entry for each undefined symbol in the final executable. These symbols are expected to be defined in other DSOs, and the code generated by the compiler should not try to access them directly, but rather access them indirectly via the corresponding pointer in the GOT.

To generate this code, the compiler needs several pieces of information: where is the GOT? what GOT entry do I need? This information is only available to the static linker, so the compiler prepares the instructions assuming a dummy value (typically zero), and then creates the proper relocation instructions for the static linker. We don't see these relocations in the final executable, since the static linker has already resolved them, but we can see them if we run "`readelf -r`" on the `main.o` file: this time we will see relocations in the `.text` section, i.e., in the code generated by the compiler.

With this technique, the code of the final executable will not need any further patching when it is loaded, just like the code of traditional statically linked executables, and will be shared among all processes that run the same program. All relocations are restricted to the GOT, which is part of the *private* data memory of each process.

R Note that the GOT may be created also for reasons other than dynamic linking, such as to overcome the addressing limitations of some architectures. For example, depending on the memory model used, 64b Intel/AMD architectures may require a GOT to address data that can be located anywhere in the 64b address space. For another reason to have a GOT, see Section A.3.2 below.

The downside of all this is that accessing data and functions that reside in other DSOs is slightly more expensive than accessing data or functions that reside in the executable, since the access requires at least one additional indirection through the GOT. Note that when we use static libraries, the imported library objects become part of the executable, so accessing their data and functions is as fast as accessing data and functions defined in the executable itself.

A.3 Building a shared library

Now let's go back to Figure A.1 and let's see how we can obtain the `libx.so` shared library from those files. To build a shared library we start with a set of object files, just like for a static library. In the example, these are `f1.o` and `f2.o` obtained from the compilation of `f1.c` and `f2.c`, respectively. However, there is a difference in how the object files should be compiled. We explain this difference in Sections A.3.1 and A.3.2 below.

Once the set of object files is ready, there is another difference in how the final library file is obtained. In fact, building a shared library is more like building an executable: the set of object files is assembled into a single ELF file, rather than into an archive. We talk about the process in Section A.3.3 below.

A.3.1 Position Independent Code

Since libraries are meant to be used by any program that may need them—programs that are unknown at the time when the library is built—, they must be ready to be loaded at any address. However, in general, binary code and data cannot be loaded at random addresses without modification. Consider the "`return myvar`" statement in function `f1()` in Figure A.1, and assume that the compiler uses the following AMD64 assembly instruction to copy `myvar` in the `rax` register:

```
        mov rax, QWORD PTR [myvar]
```

where `myvar` is an undefined symbol.

R `gcc` will select this instruction if run with `-mcmodel=large` and `-O`, otherwise it will prefer the rip-relative instructions described below.

When the source file is assembled and linked, the `myvar` symbol will be mapped to a memory address, say `0x406020`, and the above instruction becomes

```
        movabs rax, QWORD PTR [0x406020]
```

which, in the final binary code, corresponds to the bytes

```
        48 a1 20 60 40 00 00 00 00 00
```

We can see that the address `0x406020` is written in the instruction itself (starting from the 3rd byte, in little endian). This means that `myvar` cannot be loaded at a different address, unless the instruction is patched. This is normal for a static library, since each program that needs the library will have its own private, suitably patched copy of the code. In a *shared* library, instead, patching the code is undesirable, because it limits or completely prevents one of the selling points of its technology: the fact that the library can be shared among several processes.

Therefore, object files meant for shared libraries are typically compiled as Position Independent Code (PIC for short). PIC code does not assume to know the addresses of variables and functions statically, but computes them at runtime using information provided by the loader. There are basically two models that are used to implement PIC, often in combination: relative addresses and indirect addresses. We will explore them in turn.

A.3.1.1 Relative addresses

Continuing with our example, the static linker can allocate myvar at a statically known *offset* from a *base address* chosen by the loader. The loader then communicates the base address by storing it in an agreed-upon register, call it the *base register*. The compiler generates instructions that add the offset to the base address stored in the base register. If the referenced symbol is in another compilation unit, the offset may be unknown to the compiler, but the the static linker will be able to patch it into the code once it has assembled the final image. The complete address will be obtained only at runtime, by adding the constant offset and the base register each time. The code can be shared among several processes, since each one of them will have a private copy of the base register.

If we use this solution, the above instruction should be changed to something like

```
        mov rax, QWORD PTR [rbx + myvar_offset]
```

where we have assumed that the base register is rbx.

The price to pay, beside a probably negligible slowdown due to the new additions, is that at least one register, such as rbx, must be reserved for this purpose and cannot be used for general computation, or it must be saved and restored very often. An important special case is when myvar is at a known offset from the *instruction pointer*, since this register is already unavailable for general computation. In AMD64 the instruction becomes

```
        mov rax, QWORD PTR myvar[rip]
```

R Note that we are using the label myvar, that should stand for the absolute address of myvar, but this doesn't mean that the instruction will contain this address. The special syntax myvar[rip] will cause the assembler and/or static linker to compute the *offset* of myvar from the rip of the next instruction. Only this offset will be stored in the instruction. Some assembler uses the more confusing "[rip + myvar]" syntax here, but the meaning is the same.

This form of rip-relative addressing is not available on 32 bit x86 systems. The same effect can only be achieved by first loading eip into a general register—a process called "thunking":

```
        call thunk
        ...
thunk:  mov ebx, DWORD PTR [esp]
        ret
```

We call a function that immediately stores the contents of the top of the stack (i.e., the eip saved by the call) into a register and then returns.

A.3.1.2 Indirect addresses

Another way for the code to access variables and functions whose addresses are unknown at compile and (static) link time, is to access them indirectly through pointers. In this solution the static linker creates a table of pointers that is filled by the dynamic linker at load time (or later, see the Lazy Binding in Section A.5). In our example, one entry of the table will contain the address of myvar, and the code must load this pointer first, whenever it wants to access myvar.

This solution is more flexible than the one in Section A.3.1.1, since we can load myvar at any address. However, it is also more expensive. The following instructions can be used to access myvar:

```
        mov rcx, QWORD PTR myvar_entry[rip]
        mov rax, QWORD PTR [rcx]
```

First we need to load `myvar`'s pointer from the table into a temporary register (`rcx` in this case) and then load `myvar` indirectly through this register. We have assumed that `myvar`'s entry in the table can be accessed using (`rip`-based) relative addressing, which is typical. We can see how an access to `myvar` requires an additional access to the pointer table (of course the compiler can optimize a sequence of such accesses by storing `myvar`'s address in a register, but this also increases the pressure on register allocation).

This more expensive solution is the one that is typically implemented in shared libraries, because of the additional requirement that we examine next.

A.3.2 Interposing

Libraries should support *interposing*. This means that any program linking to a library must be able to redefine any of the symbols exported by the library. Suppose, for example, that we link `libx` to the following main program:

```
#include <stdio.h>
extern long f1(void);
void f2()
{
  printf("main:f2()\n");
}
int main()
{
  printf("%ld\n", f1());
}
```

With respect to the main program of Section A.2, we have provided a definition for `f2()`. Notice in Figure A.1 that `f2()` is called internally by `f1()`. In the final executable, `f1()` must call the `f2()` we redefined, not the one available in the library: the final program should print "`main:f2()`" instead of "`libx:f2()`", and `myvar` should not be incremented.

To achieve this effect with static libraries, we can put the definition of each exported symbol in its own object file, since the linker only extracts the objects that resolve an undefined symbol from static libraries. Alternatively, we can mark all exported symbols as "weak" (the linker will only pick a weak symbol if it is otherwise undefined).

R In `gcc` the latter can be obtained with declarations like "`void __attribute__((weak)) f2(void)`". In the symbol table, the `f2` symbol will have type `WEAK` instead of `GLOBAL` or `LOCAL`.

In either case, the call site in `f1()` will be the target of a relocation and will be filled with the correct address during the normal linking process. There is no performance penalty at runtime.

Shared libraries are different, as this patching of `f1()` should now happen whenever `libx.so` is loaded by a new process, and this would result in each process having a different copy of the library code. To avoid this, `f1()` must not assume to know the address of `f2()`; instead, it should calculate it based on private per-process information that is only available at load time. Now, the same pointer table described in Section A.3.1.2 can also be used to support interposing, since it introduces indirection between a call site and its target. The dynamic library's pointer table will have an entry for `f2()`, and `f1()` will call `f2()` indirectly through this entry. At load time, the dynamic linker will fill this entry, either with the address of `f2()` found in the library, or with the address of any `f2()` found in the executable.

Of course, the GOT from Section A.2.1 can also play the role of the pointer table we are discussing. To enable interposing in `gcc`, we need to compile object files with the `-fPIC` option. Note that the "PIC" in the option name is a misnomer: by enabling interposing we *also* get PIC, but if PIC was our only goal, we could have achieved it by less expensive means such as those described in Section A.3.1.1. See Section 8.6 for why this is important.

To return to our example, we can compile our `f1.c` and `f2.c` files as follows:

```
$ gcc -fPIC -c f1.c
$ gcc -fPIC -c f2.c
```

R We have used two separate commands to emphasize the fact that these are two separate, independent compilations. You can also write "`gcc -fPIC -c f1.c f2.c`", but the fact doesn't change: `gcc` will call the underlying compiler two times.

A.3.3 Static link of the dynamic library

The easiest way to create a shared library is to pass the `-shared` option to `gcc` when we link together all the objects of the library. In our example, we issue the following command:

```
$ gcc -shared -o libx.so f1.o f2.o
```

The `libx.so` file that we get is an ELF file of type DYN. We can `strip` it if we want, but the information needed by the dynamic linker will not be removed, just like with dynamically linked executables. Therefore, the exported symbols will always be visible with "`nm -D libx.so`".

Note that our shared library will have its own GOT, for the reasons explained in Section A.3.2, but also to allow dynamic linking to other shared objects. In particular, `f2()` uses `printf()`, which is defined in a different DSO (the C library), and therefore needs a GOT entry to access it. When our executable is fully loaded, there will be three GOTs in memory: the one coming from the executable, the one in `libx.so`, and another one coming from the C library.

A.3.3.1 Constructors and destructors

DSOs (and also EXEs) 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()`). The `gcc` compiler 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 static 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).

A.4 Loading the executable

Loading a dynamically linked executable is an activity that starts in the kernel, when the `execve()` primitive is called, and is completed in userspace by the dynamic linker.

A.4.1 The kernel

To load a dynamically linked executable, the kernel first performs the same actions as for any executable, interpreting the ELF file and loading the loadable ELF 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 (see also Section 6.3.1). Then it also loads (in a different part of the address space) the executable mentioned in the `.interp` section and pushes the auxiliary vector onto the stack. Finally, it passes control to the entry point of the *interpreter*, instead of the executable.

A.4.2 The dynamic linker

The interpreter is a statically linked, position-independent executable. For AMD64 systems it is the `/lib64/ld-linux-x86-64.so.2` dynamic linker mentioned above. This is the case we will consider.

The dynamic linker uses the auxiliary vector on the stack to locate the executable in the address space, and in particular its `.dynamic` section (see Section A.2). It then starts looking for all the libraries mentioned in the `NEEDED` tags of the `.dynamic` section, loading each 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 libraries will end up somewhere between the data section and the stack.

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

The dynamic linker then performs all the relocations found in the ELF files, in particular filling the GOT tables of the EXE and all loaded DSOs (but see Section A.5 for exceptions). Then it arranges for the destructors to be called at normal program exit (using `atexit()`), runs all the constructors, and finally jumps to the entry point of the executable.

Let's return to our example. When we try to run our `main` executable, the dynamic linker will try to find `libx.so` and `libc.so.6`. The latter is not a problem, since it is installed in one of the default directories that the linker already knows. However, if `libx.so` is still in the current directory, the linker will not be able to find it. We can work around this by setting the environment library `LD_LIBRARY_PATH`. This variable uses the same syntax as the `PATH` variable (including its quirks, such as empty paths that are equivalent to the current directory) and can be used to add directories to the dynamic linker search path. Therefore, we can run our program with

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

Exercise A.1 The dynamic linker ignores the `LD_LIBRARY_PATH` variable when running in "secure-execution mode". This mode is entered when the kernel has placed an `AT_SECURE` tag in the auxiliary vector, which it does, for example, when the process has changed its effective uid or gid. Why do you think we need these precautions? ■

Another possibility is to write the search path of the library in the program itself. This is useful for developers who want to ship their programs with the required dynamic libraries, and don't want to make assumptions about the dynamic linker configuration of their users. The `gcc` compiler and the Linux dynamic linker support this feature with the `RUNPATH` dynamic variable, which can be written in the program by passing the `-Wl,-rpath=`*directories* option to the compiler, where *directories* is a colon-separated list of directories.

Exercise A.2 — dll1. If not used carefully, this feature can be exploited by attackers. Try to abuse the RUNPATH of the set-uid `dll1` program in the *dll1* challenge. ■

A.5 Lazy binding and the Procedure Linkage Table

Filling a GOT entry requires a non-trivial amount of work from the dynamic linker, which must look up the corresponding symbol into the EXE and all loaded DSOs until it finds a match (ELF files come with 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 and, if it is never needed, the work will not be performed. Even if most entries are actually needed, laziness can still be useful since it improves startup time by distributing the work more evenly during runtime.

GOT entries corresponding to functions are first needed when the function is actually called the first time. 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 linker (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. This is achieved using the GOT: the GOT entries that should point to the imported library functions initially point back to the stub itself. Once the symbol is resolved, the dynamic linker replaces the entry with a pointer to the real function.

■ **Example A.1** Let's see this in action in our example. Our main executable calls two external functions: `f1()`, defined in `libx.so`, and `printf()`, defined in the C library. The linker will prepare a `.got.plt` section containing the GOT entries for `f1()` and `printf()`. It will also prepare a `.plt` section containing the *Procedure Linkage Table* (PLT for short). The PLT contains the stubs we mentioned above, one for `f1()` and another for `printf()`. The stubs are small functions generated by the linker itself, with entry points `f1@plt` and `printf@plt`. If we disassemble the `main` executable and search for the call to `f1`, we see:

```
    40113a: call   401040 <f1@plt>
```

The static linker has patched the `call` instruction and redirected it to `f1@plt`. We can disassemble the `.plt` section of the `main` executable with

```
  $ objdump -d -j.plt -Mintel main
```

In the output of the command we can recognize the two stubs (plus some other code above the first stub, unhelpfully labelled `printf@plt-0x10`). The code for `f1@plt` is:

```
  401040: jmp QWORD PTR [rip+0x2fc2]    # 404008
  401046: push 1
  40103b: jmp 401020
```

The first instruction is an indirect jump through a pointer stored at address "`rip` (of the next instruction) plus `0x2fc2`". The disassembler has done the sum for us, and the result is shown in the comment on the right: the pointer is stored at address `0x404008`. Remember the two GOT relocations that we saw at the beginning of Section A.2.1: this is the address of the entry of the GOT that should contain the address of `f1`. This is true in general: each stub has a corresponding entry in the GOT or, more

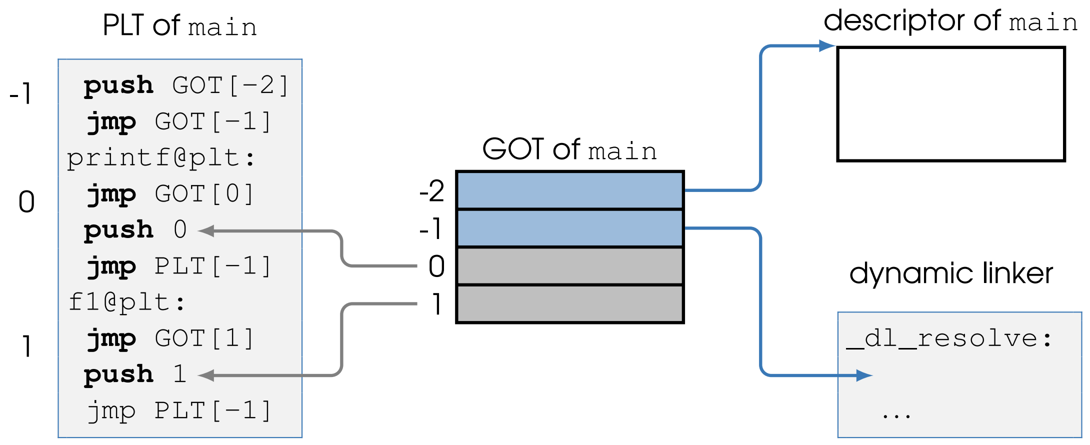

Figure A.2 – Initial arrangement of PLT and GOT in the example executable

precisely, in that part of GOT that is stored in the `.got.plt` section. We can see the contents of the `.got.plt` section, initialized by the static linker, with:

```
$ objdump -s -j.got.plt main
```

the output (edited) is as follows:

```
403fe8 f83d4000 00000000 00000000 00000000
403ff8 00000000 00000000 36104000 00000000
404008 46104000 00000000
```

The first column is an address, followed by 16 bytes stored at that address. Each entry of the GOT is a QWORD (8 bytes), so each line of the output shows two entries. The bytes along the line are grouped by 4 just for readability, but they are not interpreted as integers. This means that we see the QWORD values reversed: the address stored in the entry at `0x404008` is therefore `0x0000000000401046`. This is the address of the second line of the `f1@plt` stub above. We can repeat the exploration for `printf@plt` and we come to the arrangement shown in Figure A.2. In the Figure we have used some pseudo-C for the references to the GOT/PLT in the assembly instructions, to make them more readable. We have also added some negative indexes, both in the GOT and in the PLT: these entries are explained below, in Section A.5.1. The color of the arrows indicate which linker has written the pointer in the GOT: gray for the static linker and blue for the dynamic one. The blue pointers are still `NULL` in the `.got.plt` prepared by the static linker: they will be set at load time.

With this arrangement, the first instruction of each stub is essentially a NOP. With Lazy Binding enabled, this arrangement is not changed by the dynamic linker at load time: the dynamic linker will *ignore* (for the moment) the relocations in the `.got.plt` section, leaving the GOT entries in the state prepared by the static linker.

Let's now follow the control flow during the first call of `f1`, using Figure A.3 for reference. At run-time, when and if the `f1@plt` stub is called (step 1 in the Figure), the stub will make the useless jump to its second instruction (step 2) and continue from there. The purpose of the instructions below the first jump is to call the `_dl_resolve` function of dynamic linker and ask it to fill entry 1 of the GOT of `main`, i.e. the entry of `f1` (steps 3 and 4, see Section A.5.1 below for the details). At this point, the dynamic linker pops the information pushed on the stack by the PLT stub; this information directs it to the `f1` relocation it had ignored at load time; it resolves the `f1` symbol, fills entry 1 of the

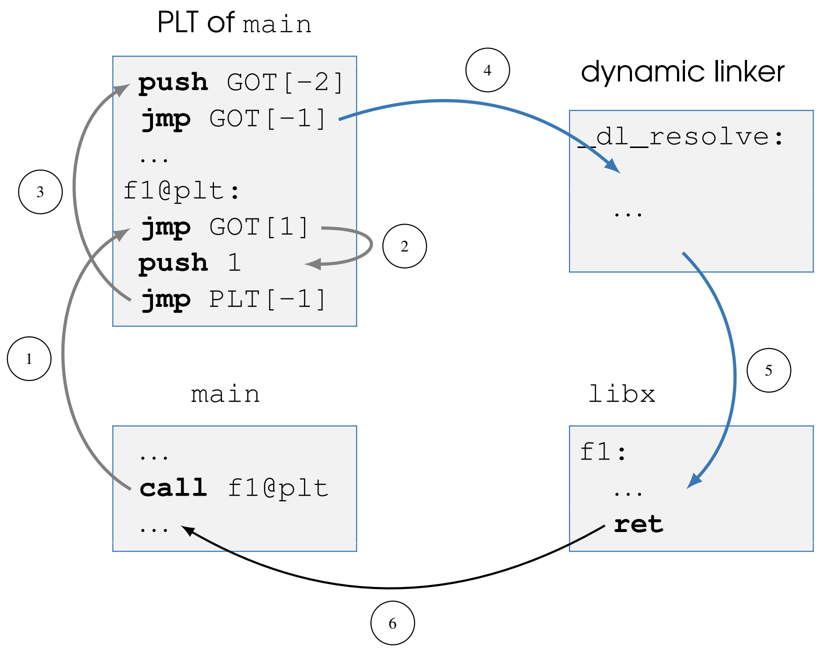

Figure A.3 – Control flow during the first call to `f1`

GOT and finally jumps to `f1()` (step 5). When `f1` returns, the top of the stack still contains the return address pushed by the original "`call f1@plt`", so control will return to the main program (step 6).

Since the GOT has been updated, the next time `f1@plt` is called, the jump instruction will go directly to `f1()` and the control flow will be as in Figure A.4.

R Note that even after the first call, calling `f1()` still requires a "`call f1@plt`" (step 1 of Figure A.4) followed by the indirect jump through the GOT (step 2). If `f1()` were in a static library, only a "`call f1`" would be required.

■

Lazy binding is usually 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.

A.5.1 Calling the dynamic linker

All stubs of our main program end with a jump to address `0x401020`, which we have represented as a jump to the PLT entry at index -1 in the Figures. If we look up address `0x401020` in the disassembler output we find:

```
401020:  push QWORD PTR [rip+0x2fca]   # 403ff0
401026:  jmp  QWORD PTR [rip+0x2fcc]   # 403ff8
```

This code (generated by the static linker) uses two entries of the GOT stored above the one with index 0 (entries -1 and -2 in Figure A.2). These entries are filled by the dynamic linker at load time: the second entry (the one used in the `jmp`) points to the function that implements the symbol lookup (`_dl_resolve` in our dynamic linker), and the first entry (the one used in the `push`) contains a pointer to a data structure, allocated by the dynamic linker, that describes the EXE or DSO that owns this GOT. Together with the index passed on the stack by the PLT stub, this information allows the lookup function to identify the entry that needs to be filled.

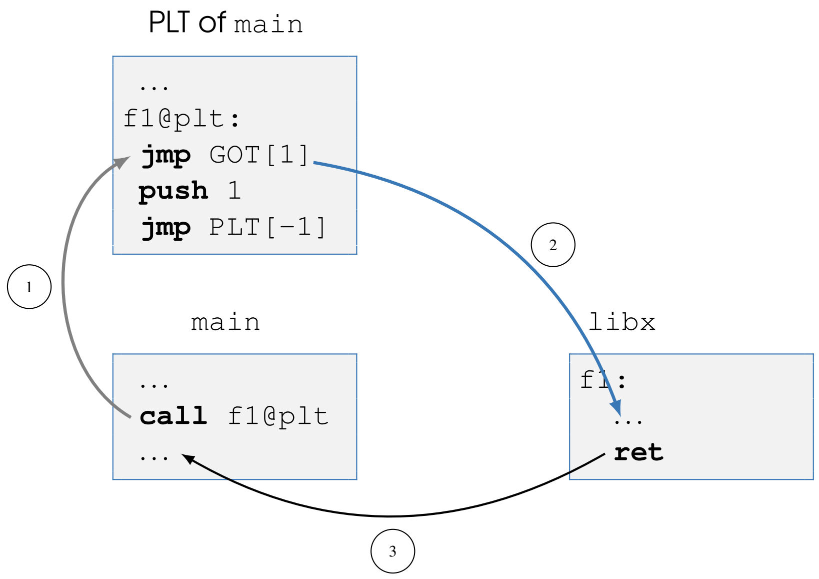

Figure A.4 – Control flow during the second and later calls of `f1`

Note that only the dynamic linker knows the address of its own lookup routine and the address of the data structure describing the EXE or DSO, so the static linker cannot fill these GOT entries. However, the static linker doesn't even create *relocations* for these entries: it just allocates them and adds the code in PLT[-1] that uses them, tacitly assuming that the dynamic linker will fill them with the correct addresses.

Exercise A.3 — canary1b. The GOT and the constructors/destructors tables (Section A.3.3.1) all provide useful writable 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. The *canary1b* challenge is very similar to *canary1* (Ex. 7.6), but overwriting the saved `rip` is much more difficult. Try to overwrite one of these new data structures instead. ■

A.6 Versioning

Since dynamic executables and the libraries they depend on may be 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`*name* with version $M.m.p$ is built, it is tagged with a `SONAME` `lib`*name*`.so.`$M.m$. The file system were the program is built will contain the file `lib`*name*`.so.`$M.m.p$ and a symbolic link `lib`*name*`.so` pointing to it. At link time, the linker option `-l`*name* will cause the linker to look up the file `lib`*name*`.so` and find the symbolic link. The linker will then read the `SONAME` tag and copy it into the `NEEDED` tag of the executable. When the executable is shipped, we can assume that the target system will contain a `lib`*name*`.so.`$M.m.p'$ library installed, and a `lib`*name*.so.$M.m$ symbolic link pointing to it. At load time the dynamic linker will read the `NEEDED` tag in the executable `.dynamic` section and look for a `lib`*name*.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`*name*, 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.

■ **Example A.2** We can add a `SONAME` to our `libx.so` library by using the `-Wl,-soname=` option when we link the library, e.g.:

```
$ gcc -shared -o libx.so.1.0.0 f1.o f2.o -Wl,-soname=libx.so.1.0
```

We have also followed the conventions described above, and named our library with the full $M.m.p$ version number.

The `SONAME` is stored as another entry in the `.dynamic` section of the library: if we run "`readelf -d libx.so.1.0.0`" we should see (among other lines):

```
0x000000000000000e (SONAME)      Library soname: [libx.so.1.0]
```

Now we link the main executable exactly as in Section A.2:

```
$ gcc -o main main.o libx.so.1.0.0
```

R Here we have used the full filename for the library, but typical installations will create a symbolic link from `libx.so` to `libx.so.1.0.0`, so that developers don't need to know the exact version of the installed library. The symlink is also required to make `-lx` work.

If we run "`readelf -d main`" we can confirm that the static linker used the library `SONAME` instead of the filename:

```
0x0000000000000001 (NEEDED)      Shared library: [libx.so.1.0]
```

This also means that, when we try to run the executable, the dynamic linker will look for a file named `libx.so.1.0`, which doesn't exist. We can fix this by creating another symlink to `libx.so.1.0.0`, this time from `libx.so.1.0`. ■

A.6.1 The diamond problem

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 (this is sometimes called *the Diamond Problem*, since the dependency graph looks like a diamond). The former problem can be solved in the previous scheme by having both library versions installed, with `lib`*name*`.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 old `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).

■ **Example A.3** In our running example, both the `main` executable and the `libx.so` library use `printf()` and therefore need the C library. This didn't cause any problems, since we linked both of them with the same version of the C library (the one installed on our system), and therefore the dynamic linker was able to load the library only once. If, instead, `main` and `libx.so` had been linked against two incompatible C library versions (e.g., with different `SONAME`s), both of them would have been needed, and most certainly they would have clashed at runtime. ■

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.

■ **Example A.4** We have already seen this per-symbol versions in Section A.2.1, since the relocation for the GOT entry of `printf` actually mentioned the `printf@GLIBC_2.2.5` symbol. We can add per-symbol versions to our library too, by creating a version script, e.g.:

```
LIBX_1.0 {
  global:
    f1; f2;
  local:
    *;
};
```

Note that the script has other purposes, besides assigning versions to symbols: it can be used to say which symbols are part of the public interface of the library (global) and which symbols are instead internal (local). The above script is saying that `f1` and `f2` are public and all the others are internal.

Note how this allows us to say that `myvar` is *not* exported: we wouldn't be able to say this with standard C keywords (we cannot declare it `static`, since it is declared in `f1.c` and used in `f2.c`). This features where added to reduce the startup time of very large programs such as OpenOffice, which used tens of dynamic libraries with thousands of unnecessary global symbols.

In addition, we are assigning version `LIBX_1.0` to all the symbols. The version is an arbitrary string that will be appended to each symbol. To use the script we put it in a `libx.map` file and pass it when we link the library:

```
$ gcc -shared -o libx.so f1.o f2.o -Wl,-version-script=libx.map
```

If we run "`nm -D libx.so`" we can see that all our symbols have a `@LIBX_1.0` suffix. If we link the main exectuable against our new library, the `f1` relocation will also show this suffix. ■