My ld.so

Version 1
5 December 2016

Epita systems/security laboratory 2018 <`recrutement@lse.epita.fr`>

Copyright

Contents

Obligations

- ▷ Read the *entire* subject
- ▷ Follow the rules
- ▷ Respect the submission

Submission

Project managers:	Alpha ABDOULAYE < `alpha@lse.epita.fr`> Pierre MARSAIS < `pierre.marsais@lse.epita.fr`> Francis VISOIU MISTRIH < `francis@lse.epita.fr`>
Project markup:	**`[LDSO]`**
Developers:	1
Submission method:	Git ACU
Submission deadline:	18 December 2016 at 11h42
Project duration:	2 weeks
Newsgroup:	labos.lse
Architecture/OS :	x86_64 Linux
Language(s):	C++
Compiler:	**`g++ 6.2.1 - clang++ 3.9.0`**
Compiler options:	**`-std=c++14 -Wall -Wextra -Werror`**
Allowed includes:	All

Instructions

The following instructions are fundamental:

Respect them, otherwise your mark may be multiplied by 0.

They are clear, non-ambiguous and each of them has a precise objective.

They are non-negotiable and you must follow them carefully.

Do not dither to ask if you do not understand any of the rules.

- ▷ **Instruction 0:** A file **`AUTHORS`** must be present in the root directory of your submission. This file must contain the login of each member of your group, leader first, in the following format : a star **`*`**, a space , then your login (ex: login_x) followed by a newline.

  Example:

  ```
  sh$ cat -e AUTHORS
  * login_x$
  sh$
  ```

  If this file is not present you will get a non-negotiable mark of 0.

- ▷ **Instruction 1:** Respect carefully the output samples' format.

  Examples are indented: **`$`** symbolizes the prompt, use it as a landmark.

- ▷ **Instruction 2:** You have to **automagically** produce a source code using the coding style when it exists for the language you are using. In any case, your code must be clean, readable, never exceeding 80 columns.

- ▷ **Instruction 3:** File and directory permissions are mandatory and stand for the whole project: main directory, main directory's files, subdirectories, subdirectories' files, etc.

- ▷ **Instruction 4:** When an executable is requested, you must only provide its source code. It will be compiled by us.

- ▷ **Instruction 5:** If a file named **`configure`** is present in the root directory of your submission, it will be executed before processing the compilation. This file must have the execution permissions.

- ▷ **Instruction 6:** Your submission must respect the following directory tree, where **`login_x`** has to be replaced by the login of the leader of your group :

  ```
  login_x-my_ldso/AUTHORS *
  login_x-my_ldso/autogen.sh *
  login_x-my_ldso/configure.ac *
  login_x-my_ldso/Makefile.am *
  login_x-my_ldso/README *
  login_x-my_ldso/TODO *
  ```

The following files are mandatory:

`AUTHORS`	contains the authors of the project. Must be EPITA style compliant.
`autogen.sh`	prepares all the build system for compilation.
`configure.ac`	in charge of generating a configure script with autoconf.
`Makefile.am`	describes how the Makefile will be generated by automake.
`README`	describes the project and the used algorithms in a proper english. Also explains how to use your project.
`TODO`	describes the tasks to complete. Need to be updated regularly.

Your test-suite will be executed during your oral examination (if any) by an assistant.

▷ **Instruction 7**

The root directory of your submission must contain a file named **`Makefile`** with the following **mandatory** rules:

`all`	compiles your project with the correct compiling options.
`clean`	removes all temporary and compiler-generated files.
`distclean`	follows the behavior of **clean** and also removes binaries and libraries.

Your project will be tested by executing , then launching the resulting executable file.

▷ **Instruction 8** A *non-clean* tarball is a tarball containing forbidden files: **`*~`**, **`*.o`**, **`*.a`**, **`*.so`**, **`*#*`**, **`*core`**, **`*.log`**, binaries, etc.

A *non-clean* tarball is also a tarball containing files with inappropriate permissions.

A *non-clean* tarball will automatically remove two points from your final mark.

▷ **Instruction 9** Your work has to be submitted on time. Any late submission, even one second late, will result in a non-negotiable mark of 0 in the best case scenario.

▷ **Instruction 10:** Functions or commands not explicitly allowed are forbidden. Any abuse will result in a non-negotiable mark of -21.

▷ **Instruction 11:** Cheating, exchanges of source code, tests, test frameworks or coding style checking tools are severely punished with a non-negotiable mark of -42.

▷ **Advice:** Do not wait until the last minute to start the project.

Introduction

This project is the opportunity to show us that you are able to work on LSE's projects. You will approach some notions linked to the manipulation of binaries produced by your compiler. You will have to write a dynamic linker, the program in charge of loading and linking the shared libraries needed by a program, and performing all the relocations needed at run-time.

A concise README file is welcomed to explain what you have done or what you tried to do. Of course, your code will be clear and properly commented.

This project is divided into 4 steps. The last exercises are not totally independent. If you have any question, do not hesitate to contact us either on the newsgroup **`labos.lse`**, on the IRC channel **`#lse@irc.rezosup.fr`** or by email at **`recrutement@lse.epita.fr`**, and use the following tags: **`[RECRUT][LDSO]`**.

Have fun and impress us!

1 Level 1: my_ldd

Synopsis

```
      ./my_ldd /path/to/executed/binary
```

1.1 Objective

In this threshold, you will have to find the list of all the shared libraries needed by your executable to be run. In order to do that, you will write a program that behaves like `ldd(1)`.

Prerequisites: Before doing that, you will have to compile your programs so that your own written dynamic linker (which must be a shared library) will be called, instead of the one already present on your system. Once compiled correctly, the `PT_INTERP` section of your binary that holds the path to the dynamic linker will refer to your dynamic linker. The program interpreter refered there will be first called before the program, allowing it to perform all the needed work before going to the pogram's entry point.

```
sh$ readelf -l /path/to/executed/binary
(...)
PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
               0x00000000000001f8 0x00000000000001f8  R E    8
INTERP         0x0000000000000238 0x0000000000400238 0x0000000000400238
               0x0000000000000009 0x0000000000000009  R      1
    [Requesting program interpreter: my_ld.so]
LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
               0x00000000000005b8 0x00000000000005b8  R E    200000
(...)
```

Furthermore, as you are writing the program in charge of the handling of run-time relocations, you will most likely prefer not to have such relocations in your ld.so. In order to avoid that, you will need to change the visibility of your symbols, and more specifically the visibility of the functions' symbols. Remember to disable the implicit linking of standard libraries when compiling the dynamic linker.

Finally, do not forget to define and indicate the entry point of your shared library to know what will be called first. Generally, your system will look for the `_start` symbol to act as the entry point. This must be a minimal assembly function to call the actual main function of your dynamic linker.

See:
- `ld.so(8)` manpage
- GCC wiki page concerning the visibility attribute [1]

[1] https://gcc.gnu.org/wiki/Visibility

1.2 Your work: printing the ELF's needed libraries

Find the program's needed libraries Step by step:

- Retrieve the traced process ELF program header from the auxiliary vector.
- Find the dynamic segment.
- Get the name of the library in the entry with the `DT_NEEDED` tag.

The value found is an offset into the string table, which can also be found in the dynamic segment.

Calling `ldd` must be equivalent to invoking your dynamic linker with the `LD_TRACE_LOADED_OBJECTS` environment variable set to 1.

```
sh$ my_ldd /path/to/executed/binary
      libfoo.so.1 => /usr/lib/libfoo.so.1
      libc.so.6 => /usr/lib/libfc.so.6
      libbar.so.4  => /usr/lib/libbar.so.4

sh$ LD_TRACE_LOADED_OBJECTS=1 /path/to/executed/binary
      libfoo.so.1 => /usr/lib/libfoo.so.1
      libc.so.6 => /usr/lib/libfc.so.6
      libbar.so.4  => /usr/lib/libbar.so.4
```

See:
- `ld.so(1)` manpage for information about the search order of the libraries paths.

In practice, `ldd` prints the libraries needed by the program, and also those libraries' dependencies. At the moment, you cannot get all the libraries' dependencies, as it requires to perform the following steps. As the project will go on, you should be able to display all of them properly.

1.3 Information

1.3.1 Executable and Linking Format (ELF)

ELF is an executable format used on UNIX platforms. It contains a set of sections, used at link-time, and a set of segments, used at run-time. In this project, we will mostly be interested by:

PT_LOAD segments: Will be loaded into memory by the program loader.

PT_DYNAMIC segment: Holds information for dynamic linking.

See:
- `elf(5)` manpage
- `/usr/include/elf.h` header
- `readelf` program and its associated manpage `readelf(1)`

1.3.2 Auxiliary Vector (auxv)

Auxiliary vectors are a mechanism to transfer some kernel level information to user space processes. There are multiple ways to retrieve data contained in auxiliary vectors, for example, the `getauxval(3)` function or the file `auxv` in the procfs.

However, as you don't have access to the functions of any standard library (remember, you are writing the dynamic linker), you will have to access it by yourself. The auxiliary vector is located above the argument list and the list of environment variables.

Program headers can be found in the auxv entry with the `AT_PHDR` tag.

See:

- `getauxval(3)` manpage
- `/usr/include/linux/auxvec.h` header
- Article about the auxiliary vector available here

2 Level 2: link map

Synopsis

```
    /path/to/executed/binary [args]
```

2.1 Objective

Construct a link map, representing all the dependencies of a program, and the informations needed in order to proceed to the relocations of symbols.

2.2 Your work: build your link map

2.2.1 Elf loading

Now that you have your program's dependencies, you will have to open and map them in memory to be able to access all the necessary information. This run-time information is located in the ELF's segments.

The parts of an ELF that need to be loaded and mapped are the `PT_LOAD` segments. Generally, one of them is the code segment, and the other one is the data segment. They can be mapped anywhere in memory, but need to keep their relative offsets intact. Be careful, the size of the data in the file might differ from the size in memory once loaded.

Do no forget that these are shared libraries, so the informations contained in the ELF are always relative to the location where they have been mapped. Every virtual address is actually an offset from the first loaded segment, which is generally mapped starting from the start of the file.

2.2.2 Parse dynamic segment

Once the libraries are correctly loaded, you will need to parse their dynamic segment, to retrieve informations about the relocations to perform. You need to find the location of the relocation table, and also the relocations associated with the procedure linkage table (PLT), which correspond to functions locations. You will later need other entries in the dynamic segment in order to correctly go on with the dynamic linking.

See:

– `elf(5)` manpage
– `/usr/include/elf.h` header
– `readelf` program and its associated manpage `readelf(1)`
– System V Application Binary Interface chapter 5 here

3 Level 3: relocations

Synopsis

```
        /path/to/executed/binary [args]
```

3.1 Objective

Process the different relocations and resolve the associated symbols. This means that you will have to find all the symbols and place them at their definitive location. These relocation must be performed by the dynamic linker because their final address is only known at run-time.

3.2 Your work

3.2.1 Find symbols

You will need to go through the relocation table and retrieve the name of the associated symbols. As usual, the value found in the entry is an index into the string table. The symbol can then be located with the help of two tables: the hash table and the symbol table.

Relocate Step by step:

- Detemine the symbol name's hash value. This will depend on the ELF's hash type.
- Find the associated symbol in the symbol table.
- Relocate the symbol to its final location using the symbol type and relocation info.

See:
– Ulrich Drepper's article about shared libraries handling available here

3.2.2 Perform relocations

When the symbol is found, it can be relocated to its final run-time location with the help of the associated relocation entry and the symbol information.

Be careful, before resolving the symbols of any shared object, all the symbols in its dependencies need to be resolved first. The order here is really important, and the completion of this task will depend on the way your link map is built.

Each relocation entry has a specific type, which correspond to a unique kind of computation to perform. Depending on this relocation type, the symbol's address will be resolved in different ways.

See:
– System V Application Binary Interface Chapter 4.4 here

3.3 Infomation

3.3.1 Hash section

To find the symbol, the string's hash value must be computed with the help of the hash table. There are two main styles of hash tables: the SYSV ELF hash, and the more recent GNU hash. The hash table style will be different for every ELF, so you need to handle the two hash types.

See:
- SYSV ELF hash described here
- GNU ELF hash described here

3.3.2 Symtab

The symbol table can be located (as usual) in the dynamic section, with the entry identified with the `DT_SYMTAB` tag. You can then access each entry's information with the structures defined by the ELF format.

See:
- `elf(5)` manpage
- `/usr/include/elf.h` header

3.4 Lazy binding

In the pevious step, the relocations associated to the procedure linkage table were retrieved from the dynamic segment. These relocations correspond to the shared object's functions. A function's symbol can be resolved with all the other relocations when loading the program. However, it can also be deferred until the first call to the function in the program, by using the GOT and the PLT: this is known as lazy binding.

The dynamic linker can alse chose to defer or not those symbols' relocations depending on the envionment variables set (`LD_BIND_NOW` or `LD_BIND_LAZY`). If both values are specified, `LD_BIND_NOW` take precedence.

```
sh$ LD_BIND_NOW=1 /path/to/executed/binary
sh$ LD_BIND_LAZY=1 /path/to/executed/binary
```

When a function is called, the program jumps to an entry inside the PLT. In the PLT, the program then jumps to an associated global offset table entry. If it is the first call to that function, the program will be back at the PLT, will jump in the first PLT entry which will be in charge of resolving the function with the information about the relocation gathered after all the previous jumps. In the subsequent calls to the function, the symbol will already be resolved and the appropriate function will be called.

The resolver consist of some assembly code in charge of saving the registers and preparing the arguments for the call to the actual function in charge of resolving the symbol. After the call to this function, the registers must be of course restored.

Here again, the location of the GOT (which on x86_64 is the only information you need in order to reach your goal) can be found in the dynamic section.

See:
- System V Application Binary Interface Chapter 5.2 available here

4 Level 4: system libraries

Synopsis

```
    ./path/to/executed/binary [args]
```

4.1 Objective

Finally, you will be able to correctly test your dynamic linker. In order to do that, you will need to write your own C standard library (libc). Furthermore, you will also write your own dynamic linking library (libdl).

```
sh$ ls /path/to/build/dir
libmydl.so libmylibc.so
```

4.2 Your work

4.2.1 Libc

The dynamic linker and the C library are tightly coupled and cooperate in order to run programs with (or without) shared libraries. The main concern here will be to call the main function of the program you want to execute.

Like for your dynamic linker, you need to define an assembly function responsible for setting up the stack, and preparing arguments before calling the actual C function in charge of the call of the program's main function. This assembly function will be called when the dynamic linker will jump to the entry point of the program after performing all the relocations. The entry point of the program can be found in the ELF header.

Your executed programs must be compiled with your brand new libc, and of course do not forget to deactivate the standard libraries linking.

See:
- Musl libc wiki page about ldso/libc relation available here
- `elf(5)` manpage

4.2.2 Libdl

The dynamic linking library allows you to load and unload shared libraries, and access their symbols during the program execution. You must produce a library exporting the API described below.

```
void* my_dlopen(const char* filename, int flags);
void* my_dlsym(void* handle, const char* symbol);
char* my_dlerror(void);
int my_dlclose(void* handle);
```

Any pogram linked with your libc and your libld must be able to call those functions, load libraries, and call functions from these libraries inside the program, even if the libraries are not linked initially with the program.

```
(...)
int (*fcn)(const char*);
void* handle = my_dlopen("libfoo.so", RTLD_NOW);
if (!handle)
    return -1;
fcn = (int(*)(const char*))my_dlsym(handle, "my_bar");
if (!fcn) {
    char* err = my_dlerror();
    /* Handle error */
}
int res = fcn("LSE");
if (my_dlclose(handle) != 0) {
    char* err = my_dlerror();
    /* Handle error */
}
(...)
```

See:

- `dlopen(3)` manpage
- `dlsym(3)` manpage
- `dlerror(3)` manpage

5 Level 5: bonus

Synopsis

```
      /path/to/executed/binary
```

5.1 Objective

You just finished writing a minimal dynamic linker. Now you will have to improve it and add other interesting funcionnalities. The features described here are grouped by their corresponding threshold. You are free to implement them in any order you want, depending on which seems the most interesting to you.

5.2 Your work

5.2.1 my_ldd

LD_LIBRARY_PATH Handle the `LD_LIBRARY_PATH` environment variable, to specify a list of directories in which to search for ELF libraries at run-time.

```
sh$ LD_PRELOAD="/path/to/libfoo:$ORIGIN/$LIB" /path/to/binary
```

Symbolic links Follow symbolic link and print the actual path of the library on your system, and the hexadecimal address at which it was loaded.

```
sh$ my_ldd /path/to/executed/binary
      libfoo.so.1 => /usr/lib/libfoo-3.46.so (0x00007fbbed1cf000)
      libc.so.6 => /usr/lib/libc-2.24.so (0x00007fbbecc2d000)
      libbar.so.4 => /usr/lib/libbar-5.39.so (0x00007fa8a4f7d000)

sh$ LD_TRACE_LOADED_OBJECTS=1 /path/to/executed/binary
      libfoo.so.1 => /usr/lib/libfoo-3.46.so (0x00007fbbed1cf000)
      libc.so.6 => /usr/lib/libc-2.24.so (0x00007fbbecc2d000)
      libbar.so.4 => /usr/lib/libbar-5.39.so (0x00007fa8a4f7d000)
```

Options Handle all `ldd` options

5.2.2 link map

LD_PRELOAD When a program is called with the `LD_PRELOAD` environment variable set to a particular value, the library located at the specified path must be loaded first before all the other ones.

```
sh$ LD_PRELOAD="/path/to/libfoo /path/to/libbar" /path/to/binary
```

5.2.3 relocations

Initialization and Termination functions Handle functions that must be called when the files are loaded (initialization) and when they are unloaded or the process terminated (finalization). Once again, every useful piece information needed will be present in the dynamic segment.

Thread local storage Handle relocations associated to thread local symbols. This means that everything related to thread local storage must be set up beforehand by the dynamic linker before performing the relocations, by using the `PT_TLS` segment among other things. You will also need to add support for thead-local storage in your libc.

See:

– Ulrich Drepper's article concerning ELF Handling For Thead-Local Storage [2]

5.2.4 system libraries

vDSO Handle the virtual dynamic shared object (vDSO) in your libc and ldso.

See:

– `vdso(7)` manpage

Thread library Write your own pthread library, compatible with your libc and dynamic linker.

Libdl extension Add the `my_dladdr` function implemention to your standard library.

See:

– `dladdr(3)` manpage

ldconfig Write `my_ldconfig`, compatible with your dynamic linker.

See:

– `ldconfig(8)` manpage

ld.so environment Handle all the environment variables used by `ld.so`.

Multi Arch Support different architectures.

GNU libc Be Glibc compliant!

Free-porn Impress us!

5.2.5 CTF

Do not forget the CTF [3] ;)

[2] `https://www.akkadia.org/drepper/tls.pdf`
[3] `https://ctf.lse.epita.fr/`