RECRUTEMENT LSE — Subject

version **#deploy-2021-v1.0**

LSE <recrutement@lse.epita.fr>

Contents

*. https://intra.assistants.epita.fr

1 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 shared libraries needed by a program, and perform the necessary runtime relocations.

A concise `README` file is welcomed to explain what you have done, all the features you implemented, and what you tried to do. Of course, you code needs to be clear, it will be part of your grade.

Since this project will not be evaluated automatically, you are free to format any output like you see fit. They should be readable and usable without reading the code though.

If you have any question, do not hesitate to contact us on the newsgroup `labos.lse`, or by email (for a personal issue) at `recrutement@lse.epita.fr`, and use the following tags: `[RECRUT]` `[LDSO]`. Don't ask us if you need to handle some kind of edge case, if you think about it, you should implement it and tell us about it in your `README`.

The loader that you will produce needs to work at least on linux x86_64.

Since you are implementing **`ld.so`** you can't really use any external libraries. Headers should be the ones provided by the kernel, compiler, with the exception of `elf.h` and `link.h`.

Have fun and impress us!

2 Useful Headers and Documentation

2.1 Recommended reading

- ld.so(8)
- binutils documentation: https://sourceware.org/binutils/docs/
- Linkers and Loaders: https://linker.iecc.com/ (http://www.becbapatla.ac.in/cse/naveenv/docs/LL1.pdf)
- Ulrich Drepper Website: https://www.akkadia.org/drepper/
  - Using ELF in glibc 2.1: https://www.akkadia.org/drepper/elftut1.ps
  - How To Write Shared Libraries: slides (https://www.akkadia.org/drepper/ukuug2002slides.pdf), paper (https://www.akkadia.org/drepper/dsohowto.pdf)

2.1.1 Implementations

- glibc and musl source code: 2 libc implementation with their loaders.
- openbsd ld.so implementation: https://cvsweb.openbsd.org/src/libexec/ld.so/ and https://cvsweb.openbsd.org/src/lib/csu/

2.1.2 ELF related information

- `elf.h`
- `link.h`
- `elf(5)` has less information than `elf.h`, but some general explanations are there.
- Tool Interface Standards (TIS) Executable and Linking File Format (ELF) Specification: http://refspecs.linuxbase.org/elf/elf.pdf
- Itanium C++ ABI: https://itanium-cxx-abi.github.io/cxx-abi/

2.1.3 Auxiliary Vector

- getauxval(3)
- elf.h (look for ElfW(auxv_t))
- "getauxval() and the auxiliary vector" (Michael Kerrisk): https://lwn.net/Articles/519085/
- About ELF Auxiliary Vectors: http://articles.manugarg.com/aboutelfauxiliaryvectors
- linux kernel elf loader code: https://github.com/torvalds/linux/blob/v4.19/fs/binfmt_elf.c#L230

2.1.4 About symbol lookups

- https://flapenguin.me/2017/05/10/elf-lookup-dt-gnu-hash/
- https://blogs.oracle.com/solaris/gnu-hash-elf-sections-v2
- https://docs.oracle.com/cd/E23824_01/html/819-0690/chapter6-48031.html

2.2 Headers

Since ld.so is responsible for loading dynamic libraries, you can't really depend on any. You can use any header you want, but, only some of them really make sense:

- `elf.h` and `link.h`: duh!

- linux kernel headers contains most of the constants you need, they are located at:
  - `/usr/include/linux`
  - `/usr/include/asm`
  - `/usr/include/asm-generic`
- Your toolchain headers, like `stddef.h`, `stdarg.h`. For gcc, it'll be located somewhere like `/usr/lib/gcc/x86_64-pc-linux-gnu/$VERSION/include`. If you need to find them programmatically, you can call:

```
$ gcc --print-file-name=include
/usr/lib/gcc/x86_64-pc-linux-gnu/8.2.1/include
$ clang --print-file-name=include
/usr/lib/clang/7.0.0/include
```

3 Prelude: Readelf

The principal objective of this part is to familiarize yourself with the ELF file format. You should take time to read the structures and understand their role.

- readelf: header, segments, symboles, sections

For this, you will first create a simple binary named `dummy-readelf` that will read some of the elf structures of a file taken as a parameter.

```
$ dummy-readelf $binary
```

This should display the same kind of information as:

```
$ readelf -hlsSd $binary
```

Don't take too much time on the output, the goal is to explore the structures, not give us a perfect readelf clone.

You should display:

- Elf header
- Program headers
- Section headers (with their names)
- all symtabs (with symbol names)
- dynamic section

Advice:

- For all structures, you should display all members.
- `mmap(2)` will probably be the easiest way to read the file.
- `ElfW()` is your friend
- try to write reusable code for lookup/dumping your structures, it'll be useful later when trying to debug `ld.so`

3.1 2 different formats

In order to cope with 32/64 bit architecture, there are tow classes of ELF structures, Elf32 and Elf64.

It can be a pain to write code that is usable with the two formats. In order to avoid code duplication, there is a useful macro called `ElfW` in `link.h`. This macro is used to build the typename which will have the correct wordsize.

Here is an example:

```
/* From elf.h */
/* Type of addresses.  */
typedef uint32_t Elf32_Addr;
typedef uint64_t Elf64_Addr;

/* from link.h */
/* We use this macro to refer to ELF types independent of the native wordsize.
   `ElfW(TYPE)' is used in place of `Elf32_TYPE' or `Elf64_TYPE'.  */
#define ElfW(type)   _ElfW (Elf, __ELF_NATIVE_CLASS, type)
#define _ElfW(e,w,t) _ElfW_1 (e, w, _##t)
```

(continues on next page)

(continued from previous page)

```
#define _ElfW_1(e,w,t)       e##w##t

/* now we can declare an addr in the native wordsize */
ElfW(addr) addr;
```

This allows building a 32bit and 64bit version without too much code duplication.

4 Given Code base

In order to give you some hints and not take too much time with your architecture, we are giving you some code to start. This contains:

- an interpreter that only chainloads into the executable
- a sample libc that contains
  - printf
  - malloc
  - syscall wrappers
- some libraries and test programs (with or without dependencies)
  - `test-standalone`: depends on only one lib, but don't use it.
  - `test-onelib`: same code, depends on only one lib
  - `test-libs`: same code, one lib, but lib depends on multiple ones.

You should take time to read and understand all the code base, especially the build system. In order to debug your code, you have to understand what is done here.

This is a start, you will need to write code inside nearly all the files. If you don't like the architecture, feel free to do something else, this is just here to help you.

```
.
|-- Makefile
|-- include
|   |-- compiler.h
|   |-- ctype.h
|   |-- malloc-internal.h
|   |-- printf
|   |   `-- boot.h
|   |-- stdio.h
|   |-- stdlib.h
|   |-- string.h
|   |-- syscall.h
|   |-- types.h
|   `-- unistd.h
|-- ldso
|   |-- exported-symbols.map
|   |-- ldso.c
|   `-- ldso_start.S
|-- libc
|   |-- crt0.S
|   |-- libc_start_main.c
|   |-- malloc.c
|   |-- printf.c
|   |-- stdio.c
|   |-- string.c
|   |-- unistd.c
|   `-- useless.c
`-- tests
    `-- test-standalone.c
```

- **include:** contains all headers used for the libc. There should not be any ldso only headers files in here.
  - `compiler.h`: sample macros that should be builtin
  - `malloc-internal.h` and `printf/boot.h`: files to adapt `malloc` and `printf` to the build/libc.
  - `syscall.h`: implementation of the syscall wrappers.
- **ldso:** contains the base implementation of ldso.
  - `exported-symbols.map`: this file is used to export symbols from ldso to other dso (with the `--version-script` ld option). It'll be useful in order to implement libdl for example. see LD manual [1]. for the exact syntax of the file.
  - `ldso_start.S`: contains the startup function
  - `ldso.c`: base code for `ld.so`
- **libc:** contains a base implementation for a libc. Most of the files are quite explicit.
  - `crt0.S`: contains `_start`, this will be linked inside binaries and calls `__libc_start_main`.
  - `libc_start_main.c`: contains the startup code of the libc, and calls `main()`.
  - `useless.c`: placeholder function used to build a minimal library.
  - `malloc.c`: implementation of dlmalloc [2].
  - `printf.c`: printf implementation (from linux boot code)
  - `string.c`: we are using gcc builtins instead of implementing them, this does the trick, and is quite efficient.
- **tests:** code base for tests, only one for the moment. Needless to say, there are not enough tests here, this will test only the simple cases (almost no relocs, simple dependency graph, ...).

[1] https://www.gnu.org/software/gnulib/manual/html_node/LD-Version-Scripts.html
[2] http://g.oswego.edu/dl/html/malloc.html

5 Building the loader

5.1 First part: looking at the loaded binary

For starters, let's just display some information about our binary. This will be useful later to debug your code.

5.1.1 Display Auxiliary vector information

For this first part, we will start looking into the environment and display all the information stored inside the Auxiliary Vector.

As `ld.so` can't really take arguments, all its configuration is passed through environment variables.

If `LD_SHOW_AUXV` is set (not necessarly to 1, any value is valid), your interpreter should display all the Auxiliary Vector content on stderr.

Here is an example with the glibc loader:

```
$ LD_SHOW_AUXV=1 /usr/bin/echo example
AT_SYSINFO_EHDR: 0x7ffc3cdc6000
AT_HWCAP:        bfebfbff
AT_PAGESZ:       4096
AT_CLKTCK:       100
AT_PHDR:         0x564b71565040
AT_PHENT:        56
AT_PHNUM:        11
AT_BASE:         0x7f32f1ba2000
AT_FLAGS:        0x0
AT_ENTRY:        0x564b715675f0
AT_UID:          1000
AT_EUID:         1000
AT_GID:          1000
AT_EGID:         1000
AT_SECURE:       0
AT_RANDOM:       0x7ffc3cd67de9
AT_HWCAP2:       0x0
AT_EXECFN:       /usr/bin/echo
AT_PLATFORM:     x86_64
example
```

5.1.2 Display libraries: `ldd`

`ldd` is a shell script that uses `ld.so` to display all the necessary libraries for an executable or library. There are multiple options for it, but basically, it just set `LD_TRACE_LOADED_OBJECTS` env variable and executes the binary. When seeing this variable, the interpreter will load the libraries, and display them on stdout.

You will have to reproduce this behavior. This step is quite important, and you will have to rewrite it several times in this project. For the moment, we will just display the dependencies of the binary (not the libraries dependencies).

You'll have to:

- look into the dynamic segment of the loaded binary
- find the `DT_NEEDED` entries
- find the libraries (for the moment, you can consider that your library path is the current directory only, but you'll add `LD_LIBRARY_PATH` handling afterwards)
- display all this.

Here is an example, with the glibc loader:

```
$ LD_TRACE_LOADED_OBJECTS=1 /usr/bin/echo
   linux-vdso.so.1 (0x00007ffec889c000)
   libc.so.6 => /usr/lib/libc.so.6 (0x00007f14be682000)
   /lib64/ld-linux-x86-64.so.2 (0x00007f14be88d000)
```

Later, you'll have to add ld.so entry, vdso, and loaded addresses.

The final part of this part will be only to load all your libraries, but instead of jumping inside the binary, just display the content of your link map.

5.2 Loading libraries and Building the link_map

Now that you can see a little bit more what is your binary, we need to start loading libraries, and starting to build our link map.

The link map definition is in `<link.h>`. It describe a dso. You should start to build that list. At first it should contain:

- the binary
- ld.so
- vdso (you can do this later)
- loaded libraries

Since we're not trying to locate the symbols yet, there is still no relocation work, this will come in a second part.

Pay attention that in order to load the libraries, you need to look into the Program Headers of each library and map them at the correct address. One simple method is to make one big mapping, with all Program headers, call `mprotect` with the correct permission for each segment, and `munmap` all the holes.

Also there are some segments that are not only contained inside the binary. The data segment will contains the `.bss` section, so you'll see a segment that have a bigger size in memory than what is described inside the file.

You can start handling dependencies here, since you'll be able to see into the dynamic segment of the libraries.

5.3 Relocations!

This is the big part.

There are two tables for relocations inside a dso. As usual, their location can be found inside the dynamic segment.

- `DT_RELA`, `DT_REL`: contains static relocations that should be done at load.
- `DT_JMPREL`: contains the relocation for the GOT.

5.3.1 Symbol Resolution

In order to be able to resolve the relocations, you need to look into symbols. You have multiple ways to do that. A full featured loader will use:

- `DT_HASH` or `DT_GNU_HASH`: an hash map of all symbols
- there is also a bloom filter[1] structure inside the `GNU_HASH` to easily reject some symbols. This bloom filter is useful in real case, to avoid too much lookup, but is really not necessary in the case of this project, since you will only load small libraries.
- These two method will look up inside the `DT_SYMTAB` and `DT_STRTAB` to find the symbols definitions.

We could start simply by doing a search inside the `DT_SYMTAB`, but there is a big issue here: The size of the symtab is not inside the binary! For a first quick hack, you can see that by construction, the static linker will always put the strtab just after the symtab, and thus allowing you to calculate the size.

[1] https://en.wikipedia.org/wiki/Bloom_filter

5.4 What to do now?

Let's pull that all together:

For each dso, starting with the loaded executable, we need to:

- make its relocations
- load the needed dsos
- rinse and repeat
- jump into the entry point

Keep in mind that `ld.so` is also a dso. This means that it is possible that it have internal relocations that needs to be resolved first. Most of the time this will not be the case, since the code is position independant.

For symbol resolution, you need to look into each element in the link map, and return the first found symbol that matches the name provided.

5.4.1 `LD_BIND_NOW`

A first step is to simply resolve all the relocations, aka `LD_BIND_NOW` behavior.

This should give you a working binary that can call library functions.

5.4.2 `LD_BIND_LAZY`

The second step, here you'll need to create a function that will be relocated inside the GOT, and resolve the called symbols, only when called the first time. This will look a lot like the precedent version, except for the wrapper code around symbol resolution.

6 Going further

Now that we have a working loader, we can start adding more features into it. Here is a list of features that you can implement:

libdl: let's support libdl also, and allow programs to load dso's at runtime. Pay attention that you should not write a complete library, just a thin wrapper that will call your loader code.

`LD_LIBRARY_PATH`: look up into multiple directories for libraries

`LD_PRELOAD`: this one is quite simple, just some libraries to load at the beginning of the link map.

`RELRO`: make your loader more secure, keep your relocation tables read only! There is two type of RELRO, now, and partial. "now" should be easy enough, "partial" is a little more complicated, since a signal can be received at any moment, and leave the mapping writable a little too long. Without thread support, this should be simple enough though.

`RPATH`: another way to find libraries, look into `ld.so(8)` for more explanations.

`DT_DEBUG`: let's make gdb work with your loader!

PIE executable: there is multiple types of executable, handle all of them! This will be simple if your code well separated.

Other architectures: i386 (should nearly work out of the box), arm/arm64 (more difficult to test but simple enough), most of the work is to have a good architecture, and handle more relocations types.

Constructor and Destructor support: `DT_INIT`, `DT_FINI` and co needs to be handled too. For this you will need to modify the libc a little bit more, in order to call them.

Bloom Filter: You can now rework your symbol resolution to accelerate it a little more.

Impress us: anything else that you can think of (tls variable, glibc support, rtld-audit...)

Ça va bien se passer.