ELF deep dive

Armijn Hemel, MSc
Tjaldur Software Governance Solutions

April 28, 2017

About Armijn

- using Open Source software since 1994
- MSc Computer Science from Utrecht University (The Netherlands), exploring reproducible builds (NixOS)
- core team `gpl-violations.org` from 2005 - May 2012
- owner Tjaldur Software Governance Solutions since May 2011
- NixOS Foundation board member

I don't want to be a lawyer. Nothing I say is legal advice.

About Tjaldur Software Governance Solutions

- (custom) tooling
- research (provenance, build tracing, audit techniques, supply chain issues)
- GPL compliance engineering (many cases involving Patrick McHardy)
- audits (for OSADL and others)

Today’s talk

It is time to revisit ELF as a lot of people get confused by the whole “linking” concept.

Some say: dynamic linking and static linking are the same (license wise). Others say they are the same. But really: “it depends”.

Let’s see what is really going on!

Note: this talk refines my talk from LLW2013.

Executable formats

There are many executable formats:

- a.out – old Unix-format
- ELF – new Unix-format, default on Linux since 1995
- bFLT – format for embedded Linux systems without a memory management unit ("MMU-less")
- Java class files
- Dex/OAT – Android file formats
- Python bytecode
- PE (Windows)
- ...

Today's focus is on ELF.

What is ELF?

ELF is the “Executable and Linkable Format”. It is (for now) the most widely used form for binaries on Linux and other Unix(-like) systems.

For C/C++/assembler this is the default format.

Before we can dive into the meaty ELF bits let’s walk through the process of compiling source code into a binary.

Note: I will provide an *extremely simplified* overview.

Step 1: source code pre-processing

First source code is pre-processed. Actions taken:

- macros expanded
- comments stripped
- superfluous whitespace stripped
- conditional statements (ifdef, etc.) processed
- ...

The result is still source code, but without any of the “cruft” like license texts (which usually are in comments).

Step 2: parsing source code

In the second step the source code is tokenized (using a lexical analyzer). A parser then processes the tokens and stores the result in a so called “abstract syntax tree” (AST), which is used in subsequent steps.

An optional step here could include program optimization/transformation (example: dead code elimination).

Step 3: code generation

In the code generation step the AST is traversed and machine code (ELF format, hardware is being generated, although there likely will be intermediate steps (intermediate language, with more optimization passes, and so on).

For C/C++/assembler files so called “object files” (.o files) are generated (ELF relocatable files).

Step 4: linking

Finally the object files are combined with information from the C library, header files and other libraries to create a library or executable.

Static vs dynamic linking

Static linking: the object code from the program, plus the C library, are combined into one single executable.

ELF also allows "dynamic linking": an executable is combined at *run time* with libraries to create a runnable program.

Dynamic linking *partially* delays linking until *run time*, meaning that for analysis you have to take the run time environment (or possibly the execution of a program!) into account.

Why dynamic linking?

Dynamic linking has advantages:

- sharing code between programs at run time to reduce memory usage
- easier to do upgrades for all programs that use a particular library
- use less disk/flash space

How dynamic linking works (simplified)

1. during *build time* symbols (function names, variables) that a program needs from a library at *run time* are recorded in the binary as “undefined”.
2. a list of libraries needed at run time is recorded into the binary.
3. at run time the symbols that are “undefined” are resolved by a program called the “dynamic linker”, which uses the list of declared dependencies to find libraries that could possibly have the needed symbols.

Important: build time and run time environments can be *different*.

Tools for inspecting ELF binaries

Important information related to dynamic linking can be extracted from ELF files. For this we need tools, for example:

- readelf (standard program on almost every Linux, from GNU binutils)
- radare2 (very steep learning curve, non-standard, extremely powerful)

Today I am using readelf, but if you want to automate ELF analysis then it is strongly recommended to use radare2.

Dynamic linker configuration (Linux)

The location of the dynamic linker is configured in the ELF file:

```
$ readelf --program-headers /bin/vim | grep interpreter
      [Requesting program interpreter:
            /lib64/ld-linux-x86-64.so.2]
```

The dynamic linker itself can be instructed to look into certain directories for finding dynamically linked libraries, typically `/etc/ld.so.conf` and files inside `/etc/ld.so.conf.d/`. Some programs add their own directories with libraries when they are installed (example: Qt).

The order in which (directories with) libraries are searched depends on the configuration of the dynamic linker.

Discovering *declared* dependencies

The names of libraries that should be searched are also recorded in the binary:

```
$ readelf -a busybox | grep NEEDED
 0x00000001 (NEEDED)   Shared library: [libutility.so]
 0x00000001 (NEEDED)   Shared library: [libnvram.so]
 0x00000001 (NEEDED)   Shared library: [libapcfg.so]
 0x00000001 (NEEDED)   Shared library: [libaplog.so]
 0x00000001 (NEEDED)   Shared library: [libcrypt.so.0]
 0x00000001 (NEEDED)   Shared library: [libgcc_s.so.1]
 0x00000001 (NEEDED)   Shared library: [libc.so.0]
```

NB: this list is *only* used to instruct the dynamic linker where to look for symbols. It does not mean that content from the library is *actually* used.

Extracting symbols

Simplified version: two types of (undefined) symbols are resolved at *run time*:

- functions
- variables

These are recorded as *undefined* symbols in the binary at *build time* and resolved at *run time*.

Discovering undefined symbols

ELF files declare the names of symbols that are needed to the outside world:

For functions (for variables: use “OBJECT” instead of “FUNC”):

```
$ readelf -W --dyn-syms busybox  | grep FUNC | grep UND
...
17: 0044e620 148 FUNC GLOBAL DEFAULT UND sigprocmask
21: 0044e610 564 FUNC GLOBAL DEFAULT UND free
22: 0044e600  72 FUNC GLOBAL DEFAULT UND raise
23: 00000000  24 FUNC GLOBAL DEFAULT UND xdr_int
24: 0044e5f0  64 FUNC GLOBAL DEFAULT UND tcsetpgrp
25: 0044e5e0  80 FUNC GLOBAL DEFAULT UND strpbrk
...
```

The names are in the last column.

Discovering defined symbols

ELF files also export the names of symbols that are exposed to the outside world (and that can be resolved):

For functions (for variables: use “OBJECT” instead of “FUNC”):

```
$ readelf -W --dyn-syms busybox  | grep FUNC | grep -v UND
...
11: 0044b2f8 228 FUNC GLOBAL DEFAULT 10 getgrent_r
18: 0044695c 144 FUNC GLOBAL DEFAULT 10 print_login_prompt
20: 0044a830 164 FUNC GLOBAL DEFAULT 10 bb_do_delay
27: 004477a0 780 FUNC GLOBAL DEFAULT 10 bb_parse_mode
...
```

The names are in the last column.

Resolving manually

Resolving these symbols manually is actually quite easy!

1. get list of *declared* dependencies for a binary
2. get list of *undefined* symbols from said binary
3. for each of the declared dependencies (libraries, etc.) get a list of *defined* symbols
4. match undefined symbols from the binary with defined symbols of the dependencies
5. stop when all dependencies are resolved, or when all dependencies have been scanned
6. report unresolved symbols, or superfluous dependencies

Do this recursively for all dependencies and you get a linking graph.

Superfluous dependencies

Superfluous dependencies happen *all the time*.

- sloppy build scripts (declaring too much)
- linker at *build time* does not thoroughly verify if the symbols are present in the dependency because at *run time* it might be a different library

Dependencies that are listed (“`NEEDED`” in output of readelf) are not necessarily actual dependencies.

Superfluous dependencies are a configuration error. If you ever encounter someone says something to the contrary, then PatchELF is your friend: `https://nixos.org/patchelf.html`

Example: Trendnet TEW-636APB

The Trendnet TEW-636APB is a wireless access point. Originally the device was made by Sercomm from Taiwan.

The firmware is a complete update: the whole flash is overwritten during the update, so we have the complete operating system, libraries, and so on, in the firmware update.

It has a GPL source code release, so we can verify our findings in the build scripts and source code.

The example program used is BusyBox.

Declared dependencies of BusyBox in the TEW-636APB

```
$ readelf -a busybox | grep NEEDED
 0x00000001 (NEEDED)   Shared library: [libutility.so]
 0x00000001 (NEEDED)   Shared library: [libnvram.so]
 0x00000001 (NEEDED)   Shared library: [libapcfg.so]
 0x00000001 (NEEDED)   Shared library: [libaplog.so]
 0x00000001 (NEEDED)   Shared library: [libcrypt.so.0]
 0x00000001 (NEEDED)   Shared library: [libgcc_s.so.1]
 0x00000001 (NEEDED)   Shared library: [libc.so.0]
```

Analysis using resolving symbols

The method described above reports that `libnvram.so` and `libaplog.so` are *not used* by BusyBox (but they *are* needed on the system to satisfy the dynamic linker).

Manual verification of the BusyBox binary and the libraries *confirms* this.

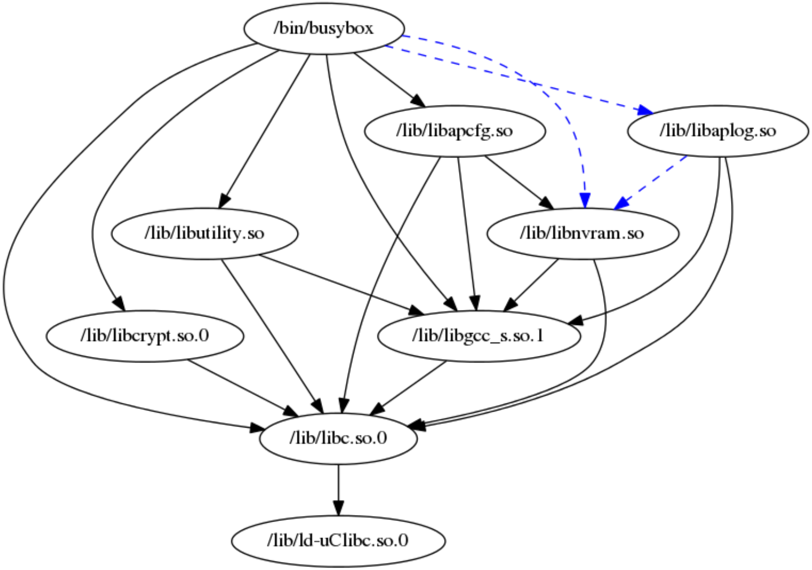
The other libraries (`libutility.so` and `libapcfg.so`) *are* used by BusyBox.

Recursively resolving dependencies

`libnvram.so` and `libaplog.so` could still be used by the other libraries!

Further inspection reveals that `libnvram.so` actually *is* used by `libapcfg.so` (which in turn is used by BusyBox).

But these two libraries are not used by BusyBox *directly*!


/bin/busybox
/lib/libapcfg.so
/lib/libaplog.so
/lib/libutility.so
/lib/libnvram.so
/lib/libcrypt.so.0
/lib/libgcc_s.so.1
/lib/libc.so.0
/lib/ld-uClibc.so.0

Analysing the source

In `Rules.mak` it is clear to see why these dependencies are recorded in the binary:

```
LIBRARIES+= -lutility -lnvram -lapcfg -laplog
```

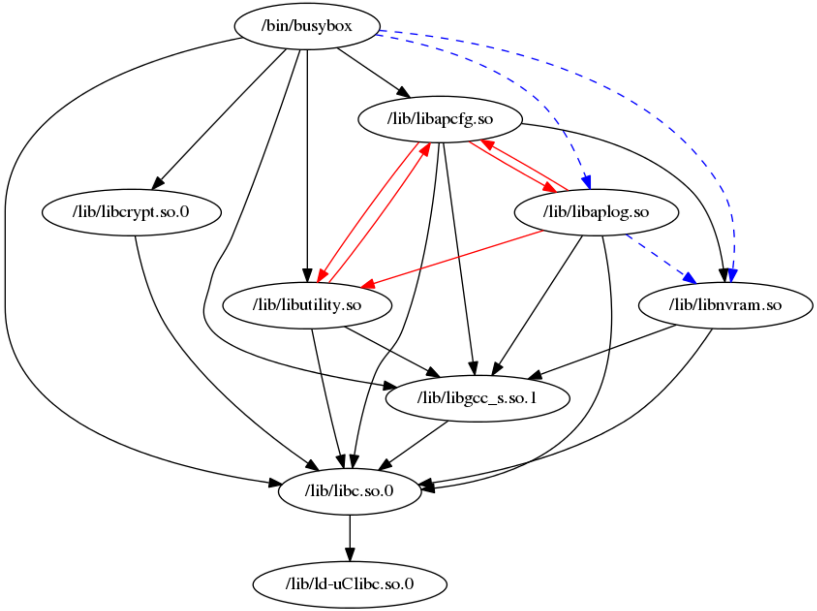
It doesn't say anything about symbols being used.

The bigger picture

Question: Are there any more pitfalls, for example dependencies that are *undeclared* and hidden?

Answer: yes.

The bigger picture (literally): hidden dependencies



LOLWUT?

How could this happen?

Looking at the sources: the dependencies are not there as source code, but only as binary code. The libraries were compiled in a different environment with a different configuration and then transplanted into another build environment.

This is the reality in the consumer electronics world.

Going even further

What else could happen?

- two libraries that offer the same symbols, and both are declared dependencies (I vaguely remember seeing examples in the wild). It then depends on the configuration of the dynamic linker which library is picked up first.
- libraries that do not implement all symbols that are needed (leading to undefined behaviour at run time)

But detecting this is future work.

Stepping back

What does this all mean?

1. in dynamic linking symbols are only resolved at *run time*. This means that a full description of the run time environment (including changes over time) is needed to draw conclusions.
2. linking information recorded in a binary is not necessarily correct, but is just there to pass information to the dynamic linker.

It would be possible to make a non-deterministic dynamic linker, for example by reconfiguring the dynamic linker every few minutes.

In practice

So is this really a problem in practice?

I think not:

- on most desktop and server machines build time and run time are actually the same
- on embedded systems it is usually possible to analyze packages in the whole environment
- symbol clashes are (I think) very rare
- no one has created a non-deterministic dynamic linker and no one should!

Wrapping up

Dynamic linking and static linking really *are* different from a licensing point of view.

However, in practice problems are very unlikely to happen and dynamic and static linking can, in most cases, be treated as equivalent (from a licensing point of view).

Contact

Any questions? Feel free to contact me!

- `armijn@tjaldur.nl`
- `http://www.tjaldur.nl/`