Research into dynamically linked ELF executables/libraries

Armijn Hemel
Tjaldur Software Governance Solutions

April 5, 2013

Scope of talk

I researched dynamically linked ELF files:

- find errors in code and build scripts
- see if by accident derivative works are created

Focus today is on legal issues. The presented research and methods are experimental.

Dynamic linking: legal debate

The GPL license talks about “derivative works”.

It is unclear if dynamic linking creates a derivative work and if the GPL “spreads by linking”. Opinions are divided about this.

Linking Schminking

My goal today

- leave you with a better understanding of how dynamic linking works (and sometimes not works)
- hopefully kickstart a debate where we can get more clarity

And now the fun technical stuff!

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.

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

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
- easier to do replacements with alternatives with the same binary interface
- 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*.

It works in most situations, but it is a bit fragile (and out of scope for this talk).

Discovering *declared* dependencies

```
$ 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: many developers think that this list indicates whether or not something is a derivative work.

Undefined symbols

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

- functions
- variables

These are recorded as *undefined* symbols in the binary at *build time*

Discovering undefined function symbols

```
$ 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
...
```

Discovering defined function symbols

```
$ 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
...
```

Discovering defined and undefined variable symbols

Like function names, except replace `FUNC` with `OBJECT`.

Resolving manually

Resolving these symbols manually is actually quite easy!

1. get list of *declared* dependencies for a binary, such as `busybox`
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 leftover symbols, or superfluous dependencies

Of course, actual bookkeeping is a little bit trickier than this.

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]
```

License of libraries

Some of the declared dependencies have no corresponding source code available, so it has to be assumed that these are proprietary closed source libraries:

- `libutility.so`
- `libnvram.so`
- `libapcfg.so`
- `libaplog.so`

This would mean that there is likely a license violation!

Or is there?

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 proprietary libraries (`libutility.so` and `libapcfg.so`) *are* used by BusyBox, so there is still a license violation (in my opinion).

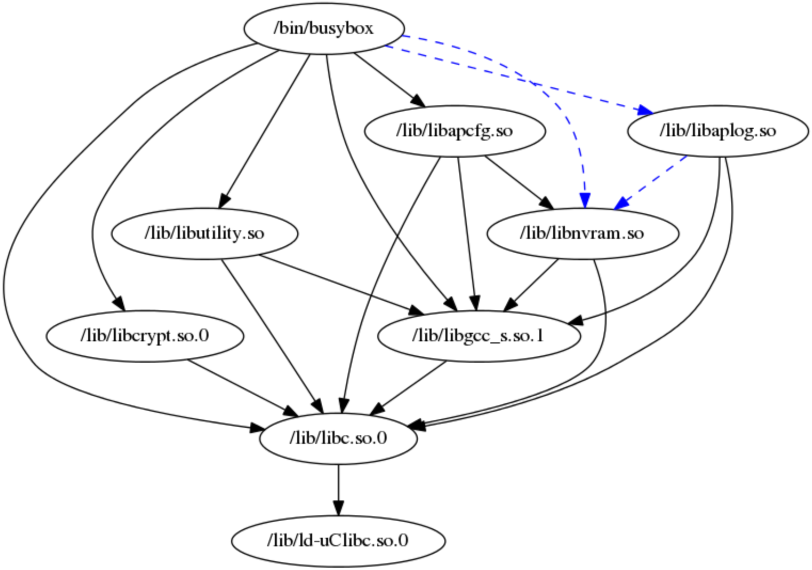
Recursively resolving dependencies

`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`, so this library now also is likely covered by the GPL.

But they are not used by BusyBox *directly*! A different implementation of `libapcfg.so` might not use `libnvram.so` at all, but still have the same interface towards BusyBox, so these dependencies should not be in BusyBox!

Picture even Shane can understand



Analysing the source

There are two places in the BusyBox source code where the dependencies on the unused libraries are defined:

1. `Rules.mak`: `LIBRARIES+= -lutility -lnvram -lapcfg -laplog`
2. `shell/sercommCli.c` includes `../../include/apcfg.h` which in turn includes `nvram.h` and `utility.h` which are the header files for the libraries.

`Rules.mak` causes the dependencies to be recorded in the `busybox` binary. The second one is discarded by the compiler (since nothing from the libraries is used) and irrelevant.

The role of the compiler

Most compilers are smart and do not generate bytecode for unused code or record function names or variable names of unused code.

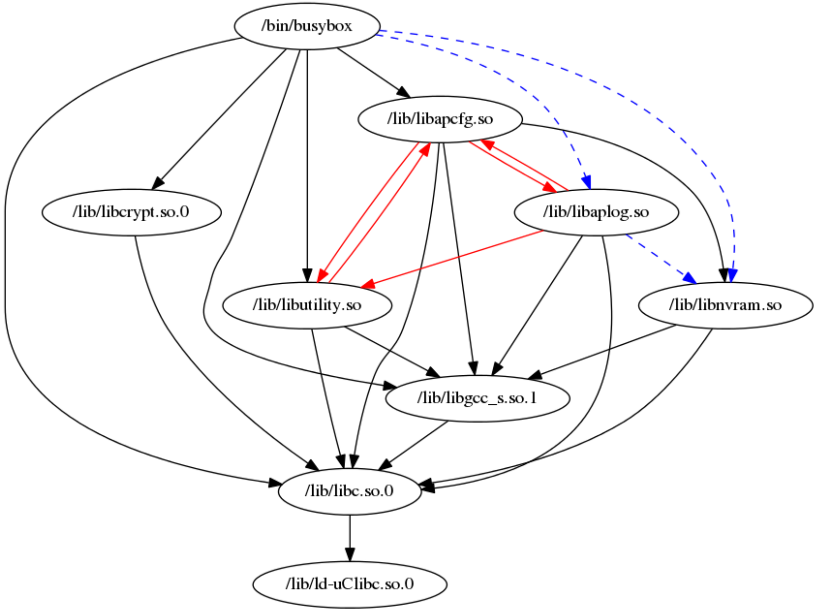
If binary code that is linked with is only known at *run time* the compiler cannot do these checks.

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



Going even further

What else could happen?

- two libraries that offer the same symbols, and both are declared dependencies (I have seen examples in the wild)
- libraries that do not implement all symbols that are needed (leading to undefined behaviour at run time)

But this is future work.

Isolated incident or common problem?

I analysed more firmwares and found the same issue (or unspecified dependencies) in nearly all firmwares in one or more binaries.

So it is actually a widespread problem!

Derivative work: yes or no

Does dynamic linking *always* make a program a derivative work? Is the mere fact something is dynamically linked strong enough?

After all, undeclared dependencies recorded in the binary are needed at run time otherwise the program *will not start*.

Solutions

There are several solutions to these bogus dependencies:

- fix the code and rebuild the software, verify until it is clean
- use *patchelf* to remove/change/add dependencies and verify the programs still run. This is the only solution if you don't have the source code to rebuild.

Future research

Detection and picture generation is already in the Binary Analysis Tool, but needs work to deal with edge cases. Future work includes:

- detecting unfulfilled dependencies
- detecting duplicate dependencies (different libraries, same symbols, all dependencies)
- analysing more firmwares
- expanding to other languages (for example Java)
- decorating the graph with more information (licensing, etc.)

Contact

Any questions? Feel free to contact me!

- `armijn@tjaldur.nl`
- `http://www.tjaldur.nl/`
- Binary Analysis Tool: `http://www.binaryanalysis.org/`