Visualizing Linux kernel module linking graphs

Armijn Hemel
Tjaldur Software Governance Solutions

June 3, 2015

About Armijn

- using Open Source software since 1994
- MSc Computer Science from Utrecht University (The Netherlands)
- core team `gpl-violations.org` from 2005 - May 2012
- Tjaldur Software Governance Solutions since May 2011
- creator of Binary Analysis Tool

Today’s topic

This talk is about visualizing dependencies between kernel modules and the main Linux kernel and finding “smells”

- compliance issues
- configuration bugs
- upstreaming opportunities
- . . .

Linux kernel modules

Linux loadable kernel modules (LKM) are pieces of code that can be loaded into a Linux kernel at runtime to provide extra functionality:

- hardware support
- extra firewalling protocols
- file systems
- ...

Modules: legal issues

Some people/companies have used the LKM mechanism in Linux to load proprietary code into the Linux kernel. There is no consensus about whether or not this is allowed.

Some mechanisms were added to "curb" this behaviour:

- "tainted"
- license field (`MODULE_LICENSE`)
- GPL only kernel symbols (`EXPORT_SYMBOL_GPL`)

GPL only kernel symbols

The `EXPORT_SYMBOL_GPL` macro was created to flag symbols that should only be made visible to modules that have their license set to GPL, or a GPL-compatible license.

This seems to work fairly well, but it is still largely a social contract: the source code can be changed to change `EXPORT_SYMBOL_GPL` to `EXPORT_SYMBOL` (and some people have), but this is very controversial.

Visualizing kernel linking graphs

Visualizing kernel linking graphs is useful for a number of reason:

- seeing if a proprietary module uses GPL kernel modules, possibly several layers deep
- seeing if a module uses unknown symbols (upstreaming opportunities)
- ...

For this the following is needed:

- symbols needed by a module
- symbols exported by a module or the main Linux kernel image
- symbol names and type extracted from source code
- extra meta information from the module (license, version, dependency information)

Extracting needed symbols

Linux kernel modules are ELF files. When loaded into the Linux kernel undefined symbols are (or should be) resolved. These symbols can be found in the ELF symbols section of the file, example:

```
$ readelf -W --syms fat.ko  | grep UND
```

Extracting exported symbols (modules, kernel in ELF format)

Symbols are stored in a section called `ksymtab_strings` and can easily be extracted:

```
$ readelf -W -p__ksymtab_strings fat.ko
```

After some postprocessing (easy to do with a bit of scripting) you get the list of symbols exported to the outside world.

Extracting exported symbols (kernel not in ELF format)

Symbols in a Linux kernel are stored in a list with NUL-terminated strings. Extract symbols by looking for a symbol that exists in (virtually) every kernel, like `loops_per_jiffy` and then walk the list forwards and backwards.

After some postprocessing (easy to do with a bit of scripting) you get the list of symbols exported to the outside world.

Extracting symbols from source code

Extracting symbols from source code can easily be done using ctags, for example (edited for clarity):

```
$ ctags -f - -x fatent.c | grep EXPORT
fat_free_clusters variable    634 fatent.c
              EXPORT_SYMBOL_GPL(fat_free_clusters);
```

These can then be stored in a database with their type (GPL symbol, or “regular” symbol).

Extract metainformation from kernel modules

Some more information is needed for more sanity checking. This information can be obtained using `modinfo`:

- version: `modinfo -F version`
- license: `modinfo -F license`
- dependency information: `modinfo -F depends`

Creating a graph (simplified)

For each collection of Linux kernel images and Linux kernel modules (for example: router firmware) do:

1. extract exported symbols from the kernel image and kernel modules
2. extracted needed symbols from the kernel modules (not the Linux kernel, as it is always the end point of the graph)
3. search for each symbol which module provides the symbol and record the dependency
4. create edges between the kernel, modules and their dependencies

Of course, there can be multiple kernel images on a system, which creates extra challenges.

Decorating a graph

Not every symbol is the same:

- “regular” symbols
- GPL kernel symbols
- unknown symbols

These can be recorded for each edge and there could be a different edge per symbol type.

Dependency information can also be recorded as either there, or “missing”.

The license field can also be used to check if a proprietary driver uses GPL kernel symbols and decorate the node in the graph is something is wrong.

Using a database (1)

For the Binary Analysis Tool (BAT) I made a database with a lot of characteristics of source code that I can use when scanning binaries. I also recorded information about Linux kernel code:

- names of symbols
- Linux kernel version
- declared module license
- license in source code file
- etc.

Using a database (2)

When processing symbols I query the database for the following information:

- type of symbol
- Linux kernel version in case the symbol exists as both normal and GPL only to decide which to choose

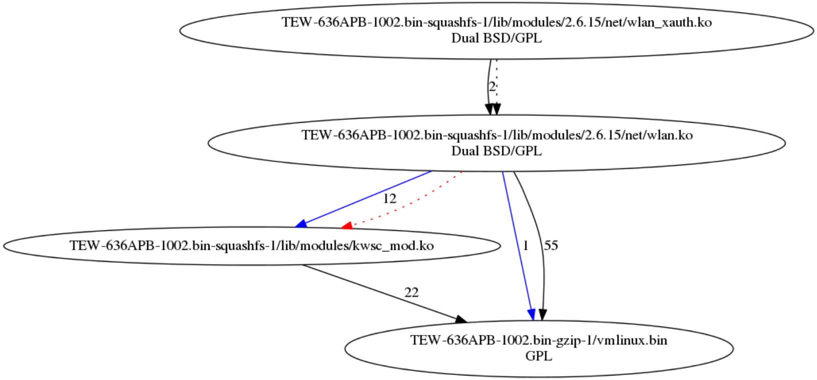
This also circumvents the cases where vendors have redefined the symbol from `EXPORT_SYMBOL_GPL` to `EXPORT_SYMBOL`.

Examples (1)

Let's look at a few examples. The edges can have different colours:

- black solid line: "regular" symbols
- red solid line: GPL only symbols
- blue solid line: unknown symbols
- black dotted line: declared dependency
- red dotted line: dependency not declared

Examples (2)


TEW-636APB-1002.bin-squashfs-1/lib/modules/2.6.15/net/wlan_xauth.ko
Dual BSD/GPL
2
TEW-636APB-1002.bin-squashfs-1/lib/modules/2.6.15/net/wlan.ko
Dual BSD/GPL
12
TEW-636APB-1002.bin-squashfs-1/lib/modules/kwsc_mod.ko
1
55
22
TEW-636APB-1002.bin-gzip-1/vmlinux.bin
GPL

Examples (3)

Let’s walk through a few more examples.

Future work

- sanity checking license field: it could be that some companies have changed the license field and it actually does not correspond with the license field that is in the actual source code.
- sanity checking dependencies: research whether or not there are screwups in the declared dependencies (cycles, etc.), although so far most cases I have seen are fairly clean
- your ideas!

Availibility

The code to make these graphs is available as part of the Binary Analysis Tool (open source licensed). The database for creating these graphs can either be purchased from my company, or you can create your own database using the scripts in the Binary Analysis Tool.

Questions?

Contact

Any more questions? Feel free to contact me!

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