Simplifying compliance by trusting upstream

Armijn Hemel
Tjaldur Software Governance Solutions

May 30, 2013

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
- volunteer at `gpl-violations.org`
- Tjaldur Software Governance Solutions since May 2011
- Binary Analysis Tool

Current day issues in consumer electronics

There are plenty of GPL license issues in current day consumer electronics:

- no source code
- partial source code
- wrong source code
- no license text/written offers

This has lead to license enforcement in various countries (Germany, France, US).

Origin of license issues: supply chain

Supply chains can be long: multiple companies are involved in creating a single product.

ODMs and (some) chipset vendors are notoriously sloppy: code (source *and* binary) is pulled from many places (upstream projects, other suppliers, homebrew) and distributed without verifying license compliance.

Various reasons:

- market pressure
- time to market
- ignorance
- "competitive advantage"
- low chance of getting caught

Trust, but verify

While business relationships should be built on trust it never hurts to *verify*.

The large amount of files in source code releases makes it hard, even impractical, to do a full source code audit in little time.

Some numbers

- BusyBox 1.21.0: 1700+ files (771 C files, 97 header files)
- Linux kernel 2.6.32.9 (used in Android Froyo): 30,000+ files
- Linux kernel 3.0.31 (used in Android Jelly Bean): 36,000+ files
- Linux kernel 3.8.4: 41,000+ files, (17997 C files, 15041 header files, 1306 assembler files)
- Qt 5.0.1: 62,000+ files, (674 C files, 12013 C++ files, 12930 header files)

For an Android phone you are easily looking at millions of files in total.

More numbers

Lines of code (including comments and whitespace) in *just* the C files:

- BusyBox 1.21.0: 255,687 lines of code
- Linux kernel 3.8.4: 12,165,051 lines of code

Learning from the LTSI project

At LinuxCon Europe 2011 the LTSI project was launched. It introduced the"Yaminabe project" to detect duplicate work. Yaminabe used a supersimple algorithm:

- treat all known files from upstream kernel as uninteresting
- focus on differences

Using this method it was clear that around 95% of sources in the kernel that vendors shipped was identical to upstream and could be ignored.

This same method can also be used for compliance scanning!

My compliance method

One very simple assumption:

Upstream projects (Linux kernel, BusyBox, Samba, . . .) are "trusted": whatever they publish is safe. Instead, focus on the files that are *not* in upstream projects!

This is not about finding copied code in files: use a clone detector for that. However, my approach can be used as a very effective pre-filter.

Why trust upstream projects?

- they are the copyright holders
- they chose the license
- they can sue for copyright infringement
- code from upstream software projects is often used (largely) unmodified

Using releases from upstream projects as the "reference standard" and only looking at differences vastly simplifies compliance.

Not every upstream source is trustworthy: you have to decide for yourself.

High trust example: Linux kernel

- every commit is "signed off" by multiple developers: high levels of scrutiny
- for (almost) every line of code there is a very clear history
- code that is not clear license wise has been in the kernel for 20 years and often copyrighted by very visible developers (Linus Torvalds, Ted Ts'o, etcetera)
- high turnover of code: old code is gradually being replaced by code that is clearer license wise

High trust example: BusyBox

- there have been very visible legal cases about BusyBox
- extremely high levels of scrutiny: they cannot afford a licensing mistake themselves

Of course you still need to distribute everything license compliant.

High trust example: GNU project

- there have been very visible legal cases about GNU project software
- extremely high levels of scrutiny: they cannot afford a licensing mistake themselves
- they are the “moral standard”

Low/no trust example: Maven 2 Central Repository

Maven is a build tool for Java, that can use repositories for storing/getting binary artefacts. The Maven 2 Central Repository is awful:

- used as a dumping ground for Java code
- tons of copies of the same code, with only slight modifications, unclear copyrights, unclear origins
- thousands of companies trust it blindly!

Copyright time bomb waiting to explode!

Others

- Samba: high trust
- GNOME: high trust (on their own code)
- KDE: high trust (on their own code)
- OpenWrt: low trust
- Debian: low trust
- Fedora: low trust

Rule of thumb: upstream sources that repackage source code from others (like Linux distributions) should have “low trust”. Original authors should have “high trust”. Roles are not always clear and different situations might require different levels of trust.

Approach

1. create a database from source code from trusted upstream. Record SHA256 checksum, plus possibly strings, function names, etcetera
2. unpack source code archive from untrusted supplier
3. compute SHA256 checksums for each source code file from the untrusted supplier
4. compare to checksums in database
5. ignore files that can be found in the database
6. further analyse/scan the files that are different

That's it!

Effectiveness

This turns out to be *extremely* effective:

- scans are fast: around 10 minutes on my test machine (8 threads, SSD) for a recent kernel. This includes a license scan with Ninka and FOSSology for files that could not be found in any upstream Linux kernel version.
- Yaminabe project: scanning kernel releases of various Android handsets showed that between 5% and 10% of files of the Linux kernel differ. That is a reduction of the problem space with 90% or more!

Audits are no longer impossible, but become very doable. I have used this successfully for multiple customers.

Caveats

- it requires a "leap of faith"
- not every upstream source is worth trusting in every circumstance
- creating a complete enough database of good quality is *very* very time consuming
- database has to be kept up to date
- does not check for *all* compliance requirements (configuration files, etcetera)

Even if you would not trust upstream code, it is a great approach to *prioritize* potential issues and risks.

Method in practice

I have used this method in practice:

- audits for Open Source Automation Development Lab (OSADL)
- other customers

Example case study: Linux based router

Source code archive from a router made by a well known Chinese router manufacturer.

Device uses many standard components, like:

- Linux kernel
- BusyBox
- U-Boot

Linux kernel

24987 files in the Linux kernel were scanned.

Of those slightly over 900 (about 4%) could not be found in any kernel downloaded from `kernel.org`. 7 files could be found in kernels from other sources.

- version numbers, id, author fields commonly from CVS/RCS metadata by Perforce (likely)
- proprietary files (mostly chipset manufacturer, some device manufacturer), likely problematic licensewise.
- actual changed source code for (perceived) bugs

Over 600 files were just changes made by version control system, meaning only a handful of files were *actually* changed or new!

BusyBox

442 files were scanned, 62 (14%) not in database:

- 17 files with Perforce changes
- (modified) copy of bridge-utils
- 1 proprietary chipset vendor file
- 1 proprietary device manufacturer file
- changed files with patches for perceived bugs

U-Boot

2989 files were scanned, 395 (13%) not in database, 10 not found in official U-Boot upstream:

- many chipset vendor proprietary files
- many files with Perforce changes
- several non-upstream files with different licenses and copyrights
- 1 file undistributable for commercial use

Undistributable file in U-Boot in router sources

`u-boot/fs/jffs2/compr_lzari.c` contains the following license:

```
LZARI.C (c)1989 by Haruyasu Yoshizaki, Haruhiko Okumura,
and Kenji Rikitake.

All rights reserved. Permission granted for
non-commercial use.
```

Luckily this file is not used in the binary!

Tooling

Scripts to create database are already present in Binary Analysis Tool (BAT).

Basic script to scan will be released as part of BAT 15 (already present in version control).

Fancier programs with more functionality will be offered as a service.

Questions?

Contact

Any more questions? Feel free to contact me!

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