

Build Tracing

Armijn Hemel, MSc

Tjaldur Software Governance Solutions

About me

- computer science at Utrecht University (NL)
 - created first prototype of NixOS distribution, based on Nix package manager
- former board member at NLUUG & NixOS Foundation
- former core team [gpl-violations.org](https://www.gpl-violations.org)
- creator of (open source licensed) tools for binary analysis/software composition analysis & build analysis
- license compliance audits, defending against GPL trolls, M&A work, and so on

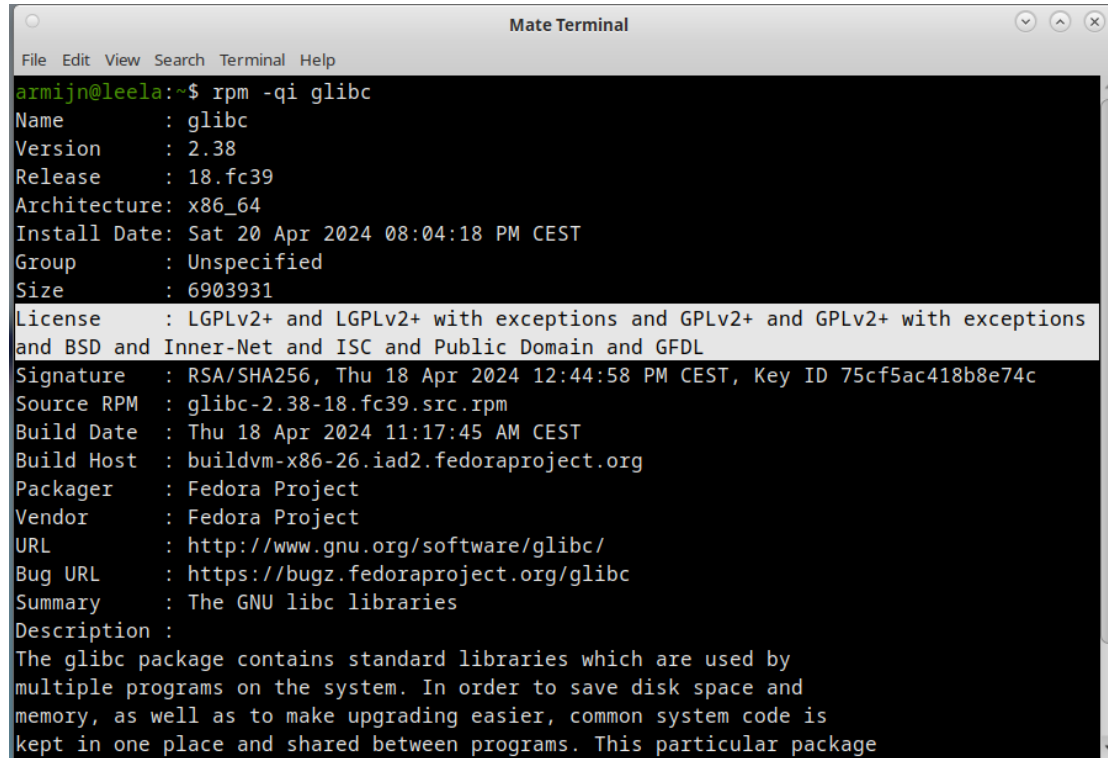
Possible views on software

- there are three possible views to look at software:
 - source code: what license/author/copyright do files or packages have?
 - build time: how were programs built and with which source code?
 - run time: how do programs combine/interact at run time?

Source code view on software

- the source code view has been the focus of SBOMs:
 - package level license
 - file level license
 - author/copyright
 - dependencies
 - etc.
- but it doesn't give us the full picture of how software is built, installed and used and can lead to confusion.

Example: glibc



```
Mate Terminal
File Edit View Search Terminal Help
armijn@leela:~$ rpm -qi glibc
Name       : glibc
Version    : 2.38
Release    : 18.fc39
Architecture: x86_64
Install Date: Sat 20 Apr 2024 08:04:18 PM CEST
Group      : Unspecified
Size       : 6903931
License    : LGPLv2+ and LGPLv2+ with exceptions and GPLv2+ and GPLv2+ with exceptions
and BSD and Inner-Net and ISC and Public Domain and GFDL
Signature  : RSA/SHA256, Thu 18 Apr 2024 12:44:58 PM CEST, Key ID 75cf5ac418b8e74c
Source RPM : glibc-2.38-18.fc39.src.rpm
Build Date : Thu 18 Apr 2024 11:17:45 AM CEST
Build Host  : buildvm-x86-26.iad2.fedoraproject.org
Packager    : Fedora Project
Vendor      : Fedora Project
URL         : http://www.gnu.org/software/glibc/
Bug URL     : https://bugz.fedoraproject.org/glibc
Summary     : The GNU libc libraries
Description :
The glibc package contains standard libraries which are used by
multiple programs on the system. In order to save disk space and
memory, as well as to make upgrading easier, common system code is
kept in one place and shared between programs. This particular package
```

GPL code in a library?!?!11!?



What's going on?

- in glibc there are pure GPL-2.0 licensed files, but these are individual tools not part of the *library* part of glibc. Tool example: “ldconfig”
- in many packages you can find files that are not part of what is installed on a machine/device:
 - build files (GNU autotools, custom scripts, etc.)
 - translation files
- these might only be used during build time and not be shipped in an installable package, or in a separate package, or not included in a build.

Build time view

- the source code view is correct, but it misses some information, namely “what source went into a *particular* binary?”
- knowing for each binary which source code files were used, allows you to:
 - zoom in on license/security issues
 - reduce problem space and triage more effectively
- note: SPDX actually has a “build profile” but it is not granular enough for information related to individual binaries

How to find out what went into a specific binary?

- two ways:
 - fingerprinting - only needs a binary, but is not very accurate
 - build tracing - very accurate, but needs access to source code and the build process (you need to be able to rebuild)

Fingerprinting

- extract symbols (function names, strings) and map to a database of same information extracted from known source code
- risk of missing files that were used (example: static ELF linking means far fewer symbols available for fingerprinting)
- makes no sense if you have source code and can rebuild!
- I make (open source licensed) tools for binary analysis fingerprinting. Binary Analysis Next Generation (BANG)
<https://github.com/armijnhemel/binaryanalysis-ng/>

Tracing a build

- instrument the *unmodified* build (with strace or BPF) and record which files are being opened, created, renamed, and so on, and create a build graph to track which files were used to create a binary.
- This works really well:
 - ASE2014 conference: “Tracing Software Build Processes to Uncover License Compliance Inconsistencies”
 - Authors - Sander van der Burg, Eelco Dolstra, Shane McIntosh, Julius Davies, Daniel M. German, Armijn Hemel

Example: FFmpeg

- FFmpeg has switches to include/exclude GPL licensed code
- by just looking at the source code it is very hard to find out what will be used (build script parsing is not the way)
- by tracking the files that are opened in the build you can very easily see which files are used
- we used build tracing to uncover license inconsistencies in FFmpeg (which were then fixed by FFmpeg). These were very hard to spot otherwise.

Example projects using tracing

- some projects using (a form of) tracing
 - Eclipse Adoptium
 - TraceCode
 - https://github.com/armijnhemel/tracing_software_packages
 - build-recorder: <https://github.com/eellak/build-recorder>
 - ESSTRA: <https://github.com/sony/esstra/>

Build tracing project

- I am making proper tools (work in progress, still early days) to make this all a bit easier:
- https://github.com/armijnhemel/tracing_software_packages
- this work is funded through NGI Zero Core, a fund established by NLnet with financial support from the European Commission's Next Generation Internet program.

Run time view

- the run time view could include:
 - all run time dependencies
 - configuration
- personally I think that this is impossible to capture in SBOMs for run time environments:
 - dynamic ELF loading, dynamic Java class loading, WebAssembly, JavaScript, etc.
- Nix closures/expressions probably capture more information!
 - please talk to me to learn more

Build tracing specifics: how it works (1)

- these instructions are for Linux
 - standard Linux tools (strace), with a bit of custom glue (open source licensed)
 - no BPF
- only a subset of system calls are traced:
 - file related (open, close, stat + friends)
 - process related (vfork, clone + friends)
 - file descriptor related (dup, pipe, tee + friends)

Build tracing specifics: how it works (2)

- trace output is written to a file per PID
- files are parsed to create a build graph:
 - which process creates which other process?
 - inputs and outputs
- a build graph can be traversed to answer the question:
 - which process created which binary and which inputs were involved?
- let's look at some example trace files

Build tracing specifics: how it works (3)

- making sure your code is complete and corresponding:
 - copy files that were opened/stat'ed
 - rebuild
 - compare the binaries
- let's look at another example (Linux kernel 6.11)
 - original number of files
 - copied number of files

Some statistics

- Trace files can get BIG and there are MANY:
 - Linux kernel 6.11 build trace files are ~1.4 GiB (or 3 GiB when tracing all system calls)
 - ~25,000 trace files
- PID wrapping can be an issue, but should not be a problem on modern systems (max PID: 4M+)
- tracing makes a build slower

What to do and not to do when tracing

- don't trace **every** build: ideally you should only have to trace a build once
- only retrace when code changes significantly
- rebuild to see if you are “complete and corresponding”
- do not base your license analysis only on the copied files (there could be license references somewhere else), but use it to have a better understanding of what you are actually building

Tracing software packages project

- https://github.com/armijnhemel/tracing_software_packages
- currently:
 - tracing a build and writing files
 - copying files
- planned:
 - graph traversal
 - data backends (scancode, vulnerablecode)
- welcome to contribute (code, funding)

Conclusion

- Build tracing can be a very effective way to get more granular information
- If you need more information: armijn@tjaldur.nl
- https://github.com/armijnhemel/tracing_software_packages
- this work is funded through NGI Zero Core, a fund established by NLnet with financial support from the European Commission's Next Generation Internet program.