What's in your containers? Tracing the origin of binaries

foo.h

foo.c

/bin/gcc

foo.o

/bin/ld

foo.exe
Philippe Ombredanne

- Enthusiast FOSS developer: AboutCode, strace, SPDX (and Python, Eclipse, JBoss, many more)
- On a mission to make it easier to reuse FOSS safely
- CTO at nexB Inc. a software company helping software teams understand where their code comes from (and its licensing, vulnerability, quality, etc) with a combo of:
  - FOSS tools
  - a commercial enterprise Dashboard
Why should you care?

- Building and deploying vulnerable software every day.
- Why? We do not know what is in our binaries.
- Systems assembled from 1000's FOSS packages
- Containers ARE custom Linux stacks
- Vulnerable packages will sneak in

The first line of defense is to NOT ship code with publicly known, exploitable vulnerabilities
The problem: back to source

• Given some binaries:
  ○ where do they come from?
  ○ which known FOSS package are they built from?

• Then given some known provenance, is there a known vulnerability? And also, licensing? other attributes?

• Some binary = Elf, Mach-O, WinPE, statically or dynamically linked
Techniques to get there

● Off topic
  ○ Reversing, disassembly, emulation and other human assisted analysis approaches used to dissect malware

Rather:

● Mostly automated tracing of code origin
● Forward tracing (sources to binaries)
● Backward tracing (binaries back to sources), static analysis
What's in a container?

- Image is built from base and Dockerfile

Dockerfile

FROM xxx
ADD xyz
RUN foo
CMD bar

Base image file

Dockerfile

Image file

Docker “build”

Running container

Docker “run”
At runtime, the container sees files from the top layer down.

The orange file "masks" the red file at the same path in lower layer. Red file is present in lower layer but not visible.

Each Dockerfile instruction line creates a new layer.

Dockerfile
FROM xxx
ADD xyz
COPY baz
RUN foo
CMD bar

Dockerfile The FROM xxx base image and its layers become the bottom layers (0 to 3)

New layers (4-6) built from the Dockerfile become the top layers
Techniques to get there

- **Forward tracing**
  - *Dynamic build tracing*

- **Backward tracing**
  - Package managers, manifests, dependencies
  - Path matching
  - Compiler conventions (e.g. Java)
  - *Signatures and Fingerprints*
  - Symbols and debug symbols
  - Shared string content
Build tracing: the problem

- Static analysis is not an exact science
- Difficult to obtain what is needed from dev teams: debug builds are not the norm
- Builds are complex and hard to modify
- Hard to conclude that something is NOT built
- Built != Deployed in a product
The ideal solution

- Should be very easy on the dev team
- NO CHANGE to the build and config
- Should provide 20/20 vision in analysis
- Should be 100% accurate
Syscalls to the rescue!

- The “machine” language of the Linux kernel
- 100% accurate
- Everything that touches a file, access the network ends up in a syscall
- Very low level e.g.:
  - open/read/write file
  - spawn process and run executable
TraceCode

- Run a build under “strace” a system call tracer for Linux
- Collect a trace of EVERY syscalls
- Process the trace
- ... and rebuild a graph of the file transformation that took place during the build
- Query the graph for fun and profit!
TraceCode (2)

- Completely agnostic wrt. toolchain or programming language
- Does not require *ANY* change to the build process
- Only need to run a traced build under strace
- Can provide 20/20 vision in the build process
TraceCode graph
Credits

- strace rocks!
- TraceCode is implementing this paper

"Discovering Software License Constraints: Identifying a Binary's Sources by Tracing Build Processes"
By Sander van der Burg, Julius Davies, Eelco Dolstra, Daniel M. German, Armijn Hemel.
Via package managers

● e.g. Run dpkg/apk/rpm "in or on" each layer
  ○ In requires to run the container... not great
  ○ On is just accessing statically the installed package DBs. No container needed

● Cross-check the installed packages files with original package

● Get corresponding sources from there

● Other ways: path matching, detect manifests
From binaries back to source

- Debug symbols: DWARFs and similar
  - Mostly a reverse path matching
- Build logs: unstructured, brittle, messy
- Signatures:
  - A subset of a binary used to identify a binary
- Fingerprints:
  - A hash/checksum of some subset or features of a binary
Fingerprints

- Simplest is a checksum aka. message digest
  - SHA1, MD5 on the whole file content
  - but brittle

- Ideas:
  - Only fingerprint a subset of the content
  - Use "sketches" fingerprints (aka. Locality Sensitive Hashing)
Fingerprints: selection

- Select subset of the bytes (e.g. every 4th chunk of 16 bytes)
  - Several heuristics: rsync/ssdeep, Winnowing, Hailstorm
  - Eventually make this selection context-free such that the same sequence of bytes will always be selected anywhere

- Or select symbols: functions and structure names
- Or select literals in source code
Fingerprints: Fuzzy hashing

- Think about it like image resizing
Fingerprints: LSH hashing

- Use sketches such as Simhash, Minhash, TLSH
- Small content change triggers a small checksum change
- Minhash, example:
  - Take a rolling window of bytes
  - Select windows and hash
  - Sum each bit column
  - Keep 0 or 1 if above or below average
Signatures

- Some unique byte string(s) in some binary
- Basic technique used to identify malware
- Often hand crafted, weaker with polymorphic malware
- But here we are not talking malware! but known packages
- Yet everything that has been learned the hard way for malware recon applies here too
Signatures: selecting

- Symbols: from ELFs and similar
- Or shared content:
  - multiple builds for the same sources (say from Fedora and Debian), many arches
  - Find unique byte substrings present in all
  - Discard bytes substrings present in many (not discriminant)
  - Simple pair-wise string comparison (e.g. using suffix trees or suffix arrays/FM index)
  - Done once at indexing time
Signatures: matching

- Multi-levels to tackle scale
- Two levels of signatures:
  - to identify a range of package versions for a binary
  - then an exact version within that range
- Two levels of matching:
  - Bloom filters front for fast presence check
  - Then automatons/Aho-corasick or finite state transducers. Tested with ClamAV.
Signatures: matching

- Key data structure is the Aho-Corasick automaton
  - Used in IDS, Virus scanners
  - Fast, stable, good worst case complexity
- Signatures can be compiled as a ClamAV database (or VirusTotal YARA rules)
- Here we are not matching malware but known binaries
Signatures: experiments

● Input: index of known libraries built on Linux/many arches
  ○ Unique shared binary strings with a suffix tree
  ○ Generated ClamAV "virus" database :)

● Input: an unknown iOS banking app, statically linked

● Output
  ○ Found exact versions of OpenSSL, libjpeg and libtiff in the statically linked iOS Mach-O
  ○ These were from a pre-built, vendor-provided library for optical check recognition
Beyond...

- Knowing a binary provenance gets you a long way...
- But how do you know if this is vulnerable?
- Scattered, sometimes sketchy vulnerabilities data; which package is impacted is not always there (NVD/CVE/CPE)
- Most data aggregated about FOSS is not open
- Cybersecurity needs open data for open source!
- FOSS and security communities should unite
  - shameless plug: launching soon http://vulnerati.org as a coalition of willing open source projects, CERTs and more.
Tools

● TraceCode - build tracing
  ○ https://github.com/nexB/tracecode-build/
  ○ Written in Python, Apache-licensed.

● TraceCode - binary signatures
  ○ Upcoming
  ○ Will be on https://github.com/nexB/
Thank you!
Credits

Special thanks to all the people who made and released these awesome free resources:

○ Presentation template by SlidesCarnival
○ Photographs by Unsplash
○ Icons from openclipart.org

And all the software authors that made TraceCode and AboutCode tools possible.