

Introducing the Binary Analysis Tool

Armijn Hemel, MSc
Tjaldur Software Governance Solutions

November 13, 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
- ▶ owner Tjaldur Software Governance Solutions

Binary Analysis Tool

Binary Analysis Tool (or: BAT) is a lightweight tool under an open source license that automates binary analysis.

- ▶ demystify compliance engineering by codifying knowledge
- ▶ make it easier to have reproducible results
- ▶ common language for binary analysis
- ▶ only analyses binaries, but draws no legal conclusions

Although BAT is a generic framework for binary analysis my focus is on software license compliance.

Important: a license violation is not a technical issue, but a legal issue. Technical measures are only used to obtain evidence.

BAT 15 was released on October 10 2013.

Why analyse binaries?

- ▶ software is often supplied in binary form by vendors (on a device/CD/DVD/flash chip/download/app). Sometimes source code is supplied and if you're lucky it might match the binary.
- ▶ even between companies (for example in a supply chain) software is often shipped as binaries

Shipping software as source code is the *exception*.

If you pass on binary software that you get from upstream (for example 3rd party software in a product) you *have* to know what you ship! This means you have to analyse the binaries.

Binary analysis

A binary usually looks like a blob with random data. Often there is a structure, with embedded file systems or compressed files that can “easily” be recognised.

Analysis steps

Steps to determine if a binary contains a particular source code:

1. extract binary files from blobs (firmwares, installers, etc.) recursively (if needed)
2. extract identifiers (strings, function names, variable names, etcetera) from binary files and compare these to (publicly available) source code
3. use other information like file names, presence of other files, package databases, etcetera, for circumstantial evidence

“Ducktyping”

“If it looks like a duck and quacks like a duck, it is probably a duck”

If you can relate many strings, function names, variable names, and so on from a binary file to source code it becomes statistically hard to deny (re)use of a certain software.

Often it is possible to match hundreds or even thousands of strings, function/method names or variable names.

Drawbacks of manual inspection

Checking can be done by hand using standard Linux tools, but there are drawbacks:

- ▶ limited by the knowledge of the engineer
- ▶ time consuming (so expensive)
- ▶ easy to overlook things

So you really want to automate this! BAT can do this for you.

Place of BAT in an open source compliance process

BAT is not meant as a replacement of a source code scanner, because it focuses on different problems.

BAT is useful when:

- ▶ you get binaries, but no source code and want to know what could be in there
- ▶ you get binaries and source code, but don't know if binaries and sources match
- ▶ you want to know how binaries interact (for example linking), which a source code scanner cannot tell you

Inner workings of BAT

1. discovery of offsets of known file systems and compressed files, verification of file types and unpacking of found file systems and compressed files (recursively)
2. check each unpacked file, like identifier search or ELF linking verification
3. reporting, generating pictures, etcetera

BAT modules

BAT is extremely modular and it comes with several modules:

- ▶ unpacking over 30 file systems and compressed files
- ▶ report on common properties (file type, size, etcetera)
- ▶ search for license markers and identifiers
- ▶ advanced string identifier search
- ▶ dynamic ELF linking verification
- ▶ kernel module analysis
- ▶ many more

Advanced identifier search/ranking

Most advanced check in BAT extracts string constants, function/method names, variable names, etcetera, from binaries and compares them with a large database of strings, function/method names, etcetera extracted from source code:

Currently over 170,000 packages from GNU, GNOME, KDE, Samba, Debian, Savannah, FedoraHosted, Linux kernel, Maven, F-Droid, . . .

Database is not part of BAT, but only available as a subscription, or you can “roll your own”.

Algorithm has been published at the Mining Software Repositories 2011 conference and most scripts to create the database are open.

Inner workings of BAT ranking

BAT ranking algorithm uses a database where data is extracted from *source code*:

- ▶ string constants (using `xgettext` and regular expressions for some Linux kernel code)
- ▶ function names (C) and method names (Java) (using `ctags`)
- ▶ variable names and Linux kernel symbols (C), field names and class names (Java) (using `ctags`)
- ▶ licenses (if enabled) (using Ninka and FOSSology)
- ▶ Linux kernel module info (using regular expressions)
- ▶ various characteristics of the file (SHA256 checksum, etc.)

Core of algorithm uses string constants (bulk of the usable information in the binary), rest of information is used to verify/strengthen the findings.

String constant example

```
...  
} else {  
    printf("%s %s: status is %x, should never happen\n",  
          inst->prog, inst->device, status);  
    status = EXIT_ERROR;  
}  
...
```

String extraction from binaries

From each binary that has not yet been discarded (graphics, video, audio, resources files and text files are not interesting so ignored) string constants are extracted using `strings`.

String constants are used for fingerprinting because they are not discarded by the compiler.

Some preprocessing steps can be used to increase quality of the strings extracted (to avoid false positives and get better scan results).

Scoring (1)

Each binary file is sorted into a family of languages:

- ▶ C (C/C++/QML/etc. + unknown binaries)
- ▶ Java (JDK/Dalvik/Scala/etc.)
- ▶ C#
- ▶ ActionScript

Reason is that strings that are very insignificant in one family could be very significant in another and vice versa.

Drawback: language embedding (specifically .NET) is at odds with this. For most systems (Java, embedded Linux) this is actually not much of an issue.

Scoring (2)

Each string constant is compared to the database. If a match is found in the database a score is assigned to that string.

The score for a unique string (single package) is the length of the string.

If it is not unique the score very rapidly drops depending on in how many different packages it can be found.

If there is *cloning* the string is assigned to a package using an algorithm that picks the most promising package.

Cloning

Sometimes the wrong package will be detected, but this reflects how open source works!

- ▶ software reuse: code is “cloned” between packages. Some packages are forked and incorporated (partially) into others (in Java more so than in C code).
- ▶ packages are renamed for various reasons. For example in Debian: `httpd` is renamed to `apache2`, Firefox is `Iceweasel`, and so on.

BAT tries to take alternatives and “cloning” into account.

Some BAT statistics

- ▶ quad core Intel(R) Core(TM) i7-3770 CPU (3.40GHz) with Hyper Threading, so 8 threads in total
- ▶ 6 GiB DDR3 RAM
- ▶ Samsung 840 PRO SSD
- ▶ SATA3 disk
- ▶ stock Fedora 19, with one package from RPM Fusion, /tmp on tmpfs disabled
- ▶ BAT 15

Hardware costs (late 2012): about 930 Euro

All tests were done on a freshly booted system. Database resides on SSD. Firmwares are unpacked on SSD, extra tmpfs to speed up LZMA unpacking, compress unpacking and DEX scanning. Final result tarball written to SATA disk, unpacked data excluded.

Example: Trendnet TEW-636ABP

Filename: TEW-636APB-1002.bin

Size of the file is 4.0 MiB (complete flash dump). Total data after BAT unpacked it is 15 MiB.

In total 262 files were scanned:

- ▶ 89 ELF files
- ▶ 48 GIF files
- ▶ rest: empty files, HTML files, JavaScript, shell scripts and other files

Total scan time: 23s

Example: ASUS O!Play Air

Filename: HDP_R3_FW_128_PAL.zip

Size of the file is 24 MiB. Total data after BAT unpacked it is 272 MiB.

In total 1351 files were scanned:

- ▶ 96 ELF files
- ▶ 90 PNG files
- ▶ 346 XML files
- ▶ 95 ASCII text files
- ▶ rest: data files, scripts, other text files, others

Total scan time: 1m37s

Example: ASUS PadFone 2

Filename: JP_PadFone2_10_4_11_13UpdateLauncher.zip

Size of the file is 780 MiB. Total data after BAT unpacked it is 3.5 GiB.

In total 56,000+ files were scanned:

- ▶ 36971 PNG files
- ▶ 9 GIF files
- ▶ 658 ELF files
- ▶ 54 Android classes.dex files
- ▶ 129 Android odex files
- ▶ 191 Android resource files

Total scan time: 21m

Demo: Trendnet TEW-636ABP

Demo is walking through the results since a scan would take too long on this netbook.

Database challenges

There are challenges in creating a database:

- ▶ *huge* amounts of data (and not getting less)
- ▶ package names and file names are very important in BAT. DVCS like Git make software development more fluid.
- ▶ cloning of software between packages.

These challenges are not exclusive to BAT.

Database quality

Results of scanning are dependent on quality of the database: if only BusyBox is included everything will look like BusyBox.
Making a good database is not easy:

- ▶ What to include?
- ▶ What to exclude?
- ▶ When is a package a new package?

Other functionality in BAT

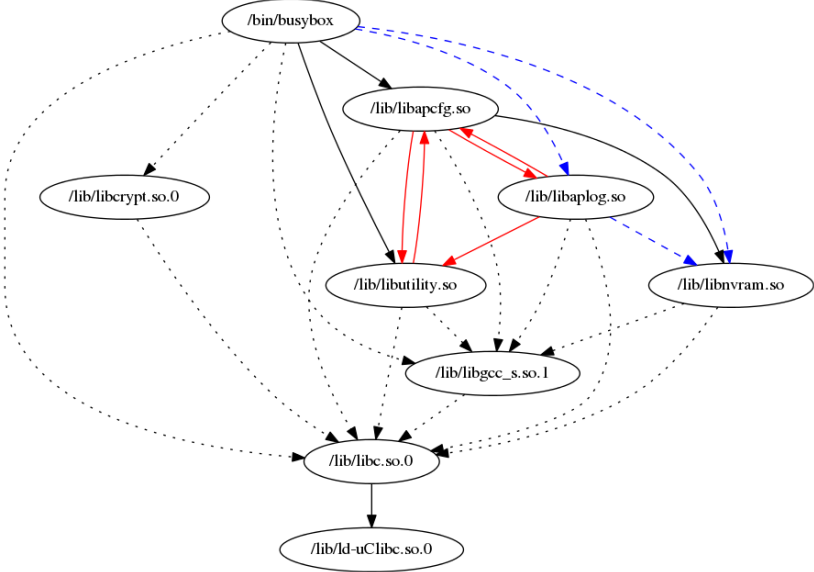
- ▶ finding duplicate files in firmwares
- ▶ finding leftover kernel modules (mismatching kernel version numbers)
- ▶ ELF dynamic linking dependency inspection
- ▶ (experimental) matching of results of binaries with source code archives
- ▶ and more

ELF dynamic linking

Actual license of a binary is only determined at *run time*. This is a largely unresearched area.

BAT can give more information about how ELF binary files interact.

ELF dynamic linking dependency graph



Challenges

- ▶ theory versus practice
- ▶ more data
- ▶ disk I/O
- ▶ new file systems
- ▶ compiler settings
- ▶ LZMA unpacking
- ▶ increasing size of firmwares
- ▶ encryption/obfuscation
- ▶ reporting

Challenge: theory versus practice

I often hear “How hard can it be?”

- ▶ proper scanning takes a lot of bookkeeping
- ▶ you have to deal with many exceptions and weird data (example: dangling JFFS2 inodes, corrupt archives, etc.)
- ▶ non-standard/modified archives/file systems (SquashFS with LZMA)

Challenge: more data

There is more and more software written, published and reused.
This leads to larger databases to query.

Example: the Linux kernel has grown massively in the last 10 years.

Example: Android apps

Challenge: disk I/O

I/O can take quite a bit of total scanning time. Mostly this is:

- ▶ querying the database
- ▶ LZMA unpacking (temporary files)

Many tricks are already used in BAT to decrease disk I/O. Smart system setup seriously helps (using SSD, enough memory for tmpfs, etcetera).

Challenge: file systems

New file systems that are used, or variants of file systems are sometimes difficult to unpack:

- ▶ YAFFS2 with different settings: lots of different versions float around
- ▶ UBI & UBIFS: `unubi` was removed from `mtddutils`
- ▶ ext4 with extents and no ext2 compatibility
- ▶ countless variations of SquashFS
- ▶ vendor specific tweaks to normal file systems (for whatever reason)

Challenge: compiler settings

Some compiler settings move strings (currently extracted from `.data`, `.rodata` and similar sections from the ELF file) to other sections inside the ELF file, but I don't know (yet) when.

Also, sometimes function names are not found in the dynamic section.

Detecting these compiler settings from the generated byte code is future work.

Challenge: LZMA unpacking

LZMA does not have a single “magic” header. Lots of different valid headers are possible (962,072,674,304 combinations possible for the first 5 bytes).

However, only a few are in widespread use.

Some filtering is done (based on information found “in the wild” and various implementations), but the only way to be complete is by trying to unpack every possible valid file. This can be very time consuming.

Being complete would cost too much time.

Challenge: increasing size of firmwares

Firmware updates of 1 GiB or more are no longer an exception.

Android file systems often contain two separate user lands (one with Google's tools, one with tools that actually work).

After after unpacking there are easily over 50,000 files to scan.

Bigger sizes amplify challenges mentioned earlier.

Challenge: encryption/obfuscation

Some vendors encrypt or obfuscate firmwares, ranging from obvious (XOR) to real encryption (AES).

BAT can handle a few XOR implementations right now (hard coded, experimental feature, disabled by default). Cryptanalysis is currently not on the roadmap, but could be interesting (technically and legally).

Challenge: reporting

BAT is pretty new technology and it is unclear what the most effective way to interpret the information generated by BAT is.

This needs more eyeballs and review: for *me* it is clear, but I want to know opinions of others.

Challenge: perfection

“Perfect is the enemy of good”

Working with binaries means working with incomplete information and 100% accuracy is impossible to achieve...but I try.

Upcoming functionality in BAT (next few months)

- ▶ overcoming or avoiding challenges mentioned earlier
- ▶ better GUI
- ▶ deployment as a webservice
- ▶ recreation of kernel configuration from binary kernel image and kernel modules (prototype works quite well)

Looking further into the future

There is a lot of information that can be extracted from binaries. Right now only a tiny fraction of the information is used. Generated code for example is not (yet) used.

There are other data sources that are still untapped like security information bug reports, and so on. I will integrate more of these in the future.

I have ideas for at least another few years of coding.

Questions?

Contact

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