GPL compliance engineering- part 1

Armijn Hemel
Tjaldur Software Governance Solutions

October 31, 2013

Subjects

Today you will learn:

- backgrounds of GPL compliance engineering
- identifying compressed files and file systems within a bigger binary blob
- extracting and verifying compressed files and file systems

What is GPL compliance engineering?

GPL compliance engineering is finding out what is inside a firmware blob to see if any GPL licensed code was used, plus seeing if license conditions are met.

Supply chain

It is very uncommon for a company to do all the work on a product. Parts (including software) are bought from a *supply chain* and combined and redistributed.

Supply chains can be long and often one or more companies screw up and distribute components license incompliant, exposing companies further down in the chain to significant legal risk.

Example: in consumer electronics the companies at the end of the supply chain carry the most/all risk, but have no say about the choices made by the other companies.

A warning before we start

It is important to keep in mind that the methods outlined below are only for finding evidence. Whether or not someone violates a license is a legal question, not a technical one. It is easy to jump to incorrect conclusions. Interpretation of legal results should be done by legal professionals, not engineers.

GPL compliance engineering process

- extract programs from blobs (firmwares, installers, etc.)
- extract human readable strings from binaries and compare this to (publicly available) source code
- (if possible) extract function names from binaries and compare this to (publicly available) source code
- use other information like file names, other files, etc. for circumstantial evidence

If you can relate enough (unique) strings and/or function names from a binary file to source code it becomes statistically hard to deny (re)use of software.

What’s in this blob?

```
00000000  50 4b 03 04 14 00 00 00  08 00 29 52 57 3c fa c0  |PK........)RW<..|
00000010  03 a7 26 9e 16 01 f4 ae  19 01 15 00 00 00 76 31  |..&...........v1|
00000020  2e 31 2e 31 2e 31 37 5f  53 4d 43 5f 61 6c 6c 2e  |.1.1.17_SMC_all.|
00000030  65 78 65 ec 3a 6d 78 53  55 9a f7 26 69 9a 42 ca  |exe.:mxSU..&i.B.|
00000040  0d d0 38 65 69 30 60 50  94 96 56 43 91 98 06 03  |..8ei0`P..VC....|
00000050  92 18 9f e1 e3 d6 c8 4d  03 f4 03 69 6b b8 a3 88  |.......M...ik...|
00000060  78 2f 83 da 76 c3 a6 d9  6d 7a 37 0f 38 8b 33 ae  |x/..v...mz7.8.3.|
00000070  33 ce d0 89 ee 8a f8 38  ae 3a 88 1f 30 61 c2 52  |3......8.:..0a.R|
00000080  3a ea 33 ac e3 02 0e 3c  b3 38 ea ee e9 a4 ce d4  |:.3....<.8......|
00000090  85 2d 01 0b 77 df f7 dc  f4 03 1c 67 66 9f fd db  |.-..w......gf...|
000000a0  ab 37 f7 9c f7 bc e7 fd  38 e7 bc 5f a7 ac 5c bb  |.7......8.._..\.|
000000b0  8b d1 33 0c 63 80 57 55  19 e6 00 a3 3d 5e e6 cf  |..3.c.WU....=^..|
000000c0  3f 67 e1 9d 72 fd 5b 53  98 d7 8b de 9f 7d 80 5d  |?g..r.[S.....}.]|
000000d0  f1 fe ec fb 22 9b 1e b5  6f d9 fa f0 03 5b 37 3c  |...."...o....[7<|
000000e0  64 df b8 61 f3 e6 87 25  fb fd 2d f6 ad f2 66 fb  |d..a...%..-...f.|
000000f0  a6 cd f6 e5 ab 83 f6 87  1e 6e 6e 59 50 5c 3c c9  |.........nnYP\<.|
00000100  91 a7 d1 fc c1 99 4b f6  d7 5e dd 3b f2 5e da f5  |......K..^.;.^..|
00000110  f2 de 6a f8 ae 7e e9 cd  bd f3 e0 9b fa c9 3b 7b  |..j..~........;{|
00000120  17 d2 fe 81 bd 9b e0 fb  eb 5d fb f6 56 52 dc d7  |.........]..VR..|
00000130  f6 7e 1f be 37 ee 7a 73  ef 2d f0 fd af 9f be be  |.~..7.zs.-......|
00000140  77 36 7c ef dd b4 31 82  74 46 64 e4 7d 0c b3 82  |w6|...1.tFd.}...|
00000150  35 30 43 1b fd 9e 31 b9  39 76 32 6b 64 98 2a 96  |50C...1.9v2kd.*.|
00000160  61 9a f4 14 76 a1 1b 7e  2c a8 38 ab 69 6f d1 fa  |a...v..~,.8.io..|
00000170  86 fc 9c 91 2f b3 c7 a0  8d c1 a3 a3 bf 96 7c df  |..../.........|.|
00000180  32 0a b7 8c 5b a3 c8 3d  2c b3 07 1b c7 59 e6 85  |2...[..=,....Y..|
00000190  5f e8 fe 82 55 fd 0b 1f  90 d3 a0 ff fa c1 05 52  |_...U..........R|
```

Binary analysis

A firmware seems to be a blob with random data. Often there is a structure. Steps needed for binary analysis are:

1. find file systems and compressed files
2. unpack them
3. research unpacked files, goto 1 if needed

Obtaining binary files

Downloadable firmwares are not enough for a *complete* analysis of a device:

- not all software that is on the device is in the firmware (bootloader, recovery partition, etc.)
- firmware might be encrypted
- no firmware might be available for download

Via other methods (serial port, JTAG, security holes) you might need to gain access to the device to get access to the full contents of the flash chip.

Obtaining firmwares via these methods is out of scope for this course, we use a complete firmware.

Firmware used in this course

We use one firmware in this course:

- Trendnet TEW-636ABP
- corresponding source code

These are on the USB stick. Please note: the firmware and source code are GPL incompliant, so please don't redistribute.

Compliance engineering

- discovery of offsets
- unpacking of file systems
- analysis of binaries

Discovery of offsets

Two clear indicators where compressed files or file systems can start:

- known identifiers
- padding

Displaying files in a readable format, for example with `hexdump`, is very helpful for manual analysis.

hexdump

hexdump is a great tool for displaying files in hexadecimal format.

The switch -C displays three columns:

1. hexadecimal offset in the file
2. hexadecimal value of bytes
3. human readable representation of bytes

```
hexdump -C $file | less
```

```
00000000  48 44 52 30 00 10 21 00  0a 07 eb 3d 00 00 01 00  |HDR0..!....=....|
00000010  1c 00 00 00 04 09 00 00  00 b4 07 00 1f 8b 08 00  |................|
00000020  00 00 00 00 02 03 a5 57  4d 6c 5b 59 19 3d be ef  |.......WMl[Y.=..|
```

Known identifiers

Many compressed files and file systems start, or end, with identifiers, for example:

- the string `PK` for ZIP files
- the string `ELF` for ELF files
- the hexadecimal sequence `0x1f 0x8b 0x08` for gzip compression

Many identifiers are well documented, for example in `/usr/share/magic` or file specifications. In practice only a few file systems and compressed files are in widespread use.

Some identifiers are very generic and false positives are likely to occur so you always need some verification step.

Padding in firmwares

Firmwares are typically written to flash memory. The bootloader on devices is often configured with offsets for:

- start of file systems
- start of bootloader
- location of (compressed) kernel
- etcetera

There is often not enough data to completely fill the allocated space, so *padding* is used, usually NUL characters.

```
0007b130  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

You can look for padding in a firmware file and search around it for identifiers.

Recognizing padding

`hexdump` will compress duplicate lines and replace them with `*`:

```
0007b130  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
0007b400  68 73 71 73 06 02 00 00  e0 05 62 0f 3f 00 00 00  |hsqs......b.?...|
```

Large amounts of NUL characters often indicate a partition boundary.

For example, in the example above the padding is followed by a SquashFS file system.

Or, we can visualise it!

Excercise 1: discover offsets

Copy the firmware (`TEW-636APB-1002.bin`) to a temporary directory and do the following:

1. use `hexdump -C` to display the contents of the file. You might want to use a pager like `less`, or redirect output to a file for later inspection.
2. report likely offsets for gzip compressed file and SquashFS file system you found.

Extracting files

After finding the right offsets you need to carve out the file system or compressed file from the firmware. There are a few ways to do this and depends on personal preference:

- editor that can process binary files (vim, emacs, etcetera)
- dd

Example with `dd`:

```
dd if=/path/to/firmware of=/path/to/outfile
bs=$offset skip=1
```

Please note that `$offset` has to be in decimal, not in hexadecimal, so you might need to convert it first.

Example: gzip

Firmware has a gzip file at offset `0x1C`. Remove the bytes in front:

```
dd if=/path/to/firmware of=/path/to/outfile bs=28
skip=1
```

and verify using `file` that the file is actually a gzip file:

```
$ file /path/to/outfile
/path/to/outfile: gzip compressed data, from Unix,
    max compression
```

Then you can unpack using `zcat`:

```
$ zcat /path/to/outfile > /path/to/outfile2
gzip: /path/to/outfile: decompression OK,
    trailing garbage ignored
```

Excercise 2: extract, verify and unpack files

Extract the following:

- gzip compressed file at offset `0x0030060`
- SquashFS file system (find the offset yourself). Use `unsquashfs` to extract this file system.

Automated extraction using the Binary Analysis Tool

Manually extracting files is a lot of work and it is easy to miss files. The Binary Analysis Tool automates extraction:

- search for identifiers
- extraction and verification

BAT works recursively and can unpack nested files and file systems.

Running BAT

First make sure BAT plus all necessary dependencies are installed. Then run:

```
bruteforce.py -c /path/to/configuration -b
/path/to/firmware -o /path/to/resultfile >
/path/to/redirect/stdout
```

The mandatory parameters are as follows:

- -c is the path to a configuration file for BAT
- -b is the path to the binary file that needs to be scanned
- -o is the path to an output file (TAR file which will contain unpacked file tree, plus results)

By default a XML result file is written to standard output. This can be redirected to a file for later inspection.

Using the BAT viewer to interpret results

To review results the BAT viewer can be used. The BAT viewer (called `batgui`) reads output the TAR output files generated by `bruteforce.py`.

`batgui` is written in wxPython.

A word of warning: result files can become rather large and unpacking results can be slow.

Exercise 3: run BAT and interpret results

Run BAT on the test firmware and confirm:

- you found a gzip compressed kernel and a squashfs file system

Conclusion

In this course you have learned about:

- backgrounds of the Binary Analysis Tool
- identifying compressed files and file systems within a bigger binary blob
- extracting and verifying compressed files and file systems
- configuring and using the Binary Analysis Tool to extract and verify compressed files and file systems

In the next course we will dig into per file analysis.