

The GPL Compliance Engineering Guide

v1.01 - 2008-10-20

Armijn Hemel <armijn@loohuis-consulting.nl>

Copyright © 2008 Loohuis Consulting. Verbatim copying and distribution of this entire article is permitted in any medium, provided this notice is preserved.

Table of Contents

Introduction.....	4
The consumer electronics business.....	4
How a product is developed.....	5
Technical analysis.....	5
Initial network scan.....	5
Firmware analysis.....	7
Embedded design 101.....	7
Boot sequence and boot loaders.....	7
Popular bootloaders.....	8
Compression techniques.....	8
File systems.....	8
squashfs.....	9
ext2/ext3.....	9
cramfs.....	10
jffs2.....	10
Executable files.....	10
Compilation 101.....	11
Executable formats.....	11
Tools.....	12
hexdump.....	13
file.....	13
strings.....	14
grep.....	14
bzip2/bzcat.....	14
gzip/zcat.....	14
lzma.....	14
cramfsswap.....	15
mtd-utils.....	15
unrar.....	15
md5sum/sha1sum/sha256sum/sha512sum.....	15
cabextract.....	16
unshield.....	16
Physical access.....	16
Serial console.....	16
Attaching a serial cable to a router.....	16
Accessing the serial port.....	20
JTAG.....	20
Software packages.....	20
Linux kernel modules.....	20
busybox.....	21
C libraries.....	22
Toolchain.....	22
Experiences.....	23
Physical compliance.....	23
Compliance engineering on Microsoft Windows.....	23
Common violations.....	24

Tools.....	24
Cygwin compliance engineering.....	24
Appendix A: GPL checklist.....	25
Appendix B: Reporting, handling and preventing violations.....	25
Reporting a violation.....	25
Handling a violation report.....	25
Preventing a violation.....	26
Appendix C: Commercial compliance engineering.....	26

Introduction

This is a guide explaining how to find license violations in a device. This guide will try to highlight useful practices from various perspectives, including a rough analysis of the device through network scans, extracting information from a firmware and physically altering a device to get access to it.

Before we can dive into the technical details, it is worth taking a look at the business processes of the consumer electronics industry, where most violations are found.

WARNING: Some things described in this guide might not be legal due to DMCA(-like) legislation in your country. If you want to be sure what you can and cannot do in your country, please consult a lawyer. This is not legal advice.

The consumer electronics business

The consumer electronics business can only be described as "weird" by an outsider (like myself). The business itself is very high volume and has very low margin. Competition in this market is very fierce. The consumers mostly look at one thing: price. Time to market, marketing, price and emotional attachment to a particular brand are what drives the market. The shelf life of a typical device is short: 1 to 1.5 years. Most of the sales of a new product happen during the first 3 months the device is on the market.

Making it to the shops a few days later than a competitor's product (which ironically often comes from the same factories) could mean the difference between having a profit or turning a loss. Raising the price by 10 cents per device could mean the same.

Compliance engineering and checking for licensing issues tends to endanger profit. First of all, it delays the release. Proper compliance engineering could take a few days (depending on the device), any questions have to go back to the factory, sources have to be shipped, and so on. With bad luck the factory won't or can't release all sources and it could take many months, before the device is compliant.

Compliance engineering in general is not cheap and the costs of it have to be split per device. A price of EUR 1200 for checking a device is realistic, given the hourly rates for a commercial embedded systems engineer. Still, for companies in this market this is a lot of money, especially if you keep in mind that many companies do so called test runs of hardware, which can be as low as 200 devices. If a product is selling, additional shipments are ordered at the factory. For GPL compliance the amount of devices does not matter, since distribution is distribution, but EUR 1200 divided by 200 equates to a sharp raise in the price of a device.

Companies often have to make a choice: ship incompliant software and risk a court case or face a huge loss resulting from missed sales. Some people have hinted that a court case is unlikely to happen and is probably a lot cheaper than the alternative. Various organisations, like the gpl-violations.org project and SFLC have started pushing for compliance a lot more in the past years, so this argument is likely to become invalid soon.

How a product is developed

The products we see in the shop are not developed by the company that has its name on the box. There are few Western companies selling devices in large quantities to end consumers that do their own development. Even these companies that do are unlikely to do all the work themselves.

There are often quite a few companies involved in the development of a product. The Western companies buy their devices in Asia, most often from a Taiwanese or sometimes a Korean company. Sometimes a custom casing is developed for the product, but more often a generic casing is adapted with the company logo on the casing. The manual and packaging are also adapted to taste and everything is shipped to the West. The Western companies do distribution, marketing, end user support, rebates, and so on.

The company where the devices are produced use a board design with a SDK, which they get from another upstream vendor, often the chip vendor. There can be additional layers in between. The engineers at the Taiwanese company, or any of the other layers, sometimes add some extra code, or make other changes using the SDK. The extra code might contain kernel drivers for various hardware components in the device, such as wireless network cards, or software firewalling modules.

These changes may be fully, partially or not at all integrated into the source archive from the SDK. If the sources are not or partially integrated the result is that the sources distributed as the "GPL sources" are not complete.

There is a lot of secrecy (or perceived secrecy) in the consumer electronics ODM world. When you ask for sources you will be told they are "very very secret", even though there is a good chance these sources contain GPL or LGPL licensed code.

Technical analysis

A technical analysis is the technical part of the GPL compliance engineering process. The goal of a technical analysis is to determine if there is GPL or LGPL licensed software on a device. A technical analysis can be performed in several ways. Often there is device, firmware, source tarball (or any combination thereof) that you are asked to check for compliance. Depending on the situation, a lot of work could be required to discover whether GPL violations exist, or to make sure there are none. This can range from dissecting a firmware and go as far as physical modification of a device to log in via a serial port onto the device, or beyond. This section summarizes my tools of choice to do this. It is far from complete and I am very certain I do not find all violations. Still, it is more than what most people at companies are able (or willing) to find. The more violations you catch, the more pressure we can put on a company to adopt better internal processes to prevent violations from happening at all in the future.

Initial network scan

If the device is networked, it is a good idea to start a scan from the network. Many operating systems have slight differences in the networking stack in how they respond to certain packets. Using heuristics, where a lot of specially crafted packets are sent to the device it is

possible to determine what a device runs. Scanning tools like nmap have a fingerprinting option, which is fairly accurate:

```
# nmap -P0 -O <ip address> -p 1-65535
```

This command does not first try to ping the device on the network (-P0), which is a considerable speedup, since many devices do not respond to pings. The fingerprinting option needs root privileges.

The output of the command looks like this:

```
# nmap -P0 -O 10.0.1.1
```

```
Starting Nmap 4.20 ( http://insecure.org ) at 2007-09-16 01:16 CEST
Interesting ports on gateway.local (10.0.1.1):
Not shown: 1692 closed ports
PORT      STATE SERVICE
22/tcp    open  ssh
25/tcp    open  smtp
53/tcp    open  domain
139/tcp   open  netbios-ssn
445/tcp   open  microsoft-ds
MAC Address: 00:00:00:00:00:00
Device type: general purpose
Running: FreeBSD 6.X
OS details: FreeBSD 6.1-RELEASE through 6.2-BETA3 (x86)
Uptime: 36.216 days (since Fri Aug 10 20:06:29 2007)
Network Distance: 1 hop
```

```
OS detection performed. Please report any incorrect results at http://insecure.org/nmap/submit/ .
Nmap finished: 1 IP address (1 host up) scanned in 21.304 seconds
```

While powerful this method is not failsafe. There are software packages, so called scrubbers, which will hide a lot of these details and make fingerprinting relatively useless. There are embedded devices that use obfuscation techniques like scrubbers (the Thomson Speedtouch is an example of this), but many devices do not deploy such techniques, so for now it can be regarded as a good indication what is running, until all devices use scrubbers.

Apart from fingerprinting this command also scans all TCP ports. Sometimes there is a debug port, or telnet port to which you can connect to gain shell access to the device itself and inspect it. Banner strings for programs (webserver, FTP server, etcetera) can also help in determining what is running on a particular device.

Please note, the fact that a portscanner reports that a device runs on another operating system than Linux does not mean you can skip further compliance engineering. As said earlier, the device might make use of scrubbing.

There are also plenty of GPL license violations on devices that don't run Linux. There are for example devices that run a very basic proprietary operating system, but also include some GPL licensed code, which is linked into one big binary blob. Getting these devices license compliant is a challenge.

Firmware analysis

A reliable method of finding GPL licensed code in a device is by grabbing the firmware of the device from the download site or CD and dissecting it to reveal all bits and pieces of what is in the firmware. There is no standard recipe for dissecting firmware, since there are many ways the firmware of a device can be structured. However, the underlying methodologies as outlined in this document can be used for many devices. Before these methodologies are explained there is a short explanation of how devices work and why the design influences the layout of the firmware.

Embedded design 101

There are a few basic design steps you will need to know for proper reverse engineering. These are:

- boot sequence and boot loaders
- file systems
- compression techniques
- executable formats

A good book to read to get some general understanding about embedded Linux is "Building Embedded Linux Systems", published by O'Reilly.

<http://www.oreilly.com/catalog/belinuxsys/>

Boot sequence and boot loaders

When a device starts the CPU executes certain commands to initialize the whole device. One of the things that is hardcoded in the CPU is the memory location of an instruction that should be loaded first. This instruction is often the first instruction of the bootloader. The bootloader is a program which sets up the rest of the system.

The bootloader itself resides on flash memory at a fixed offset. This offset varies greatly between CPUs, boards and vendors.

As an example, a fairly typical layout for the flash chip of a device with the AR7 chipset could look like this:

```
mtd0    0x900a0000, 0x903f0000
mtd1    0x90010000, 0x900a0000
mtd2    0x90000000, 0x90010000
mtd3    0x903f0000, 0x90400000
```

These regions are used by the bootloader. In this case the bootloader itself resides on "mtd2" and this is the location that the CPU uses to find the bootloader. The bootloader reads the other locations from nvrAm to find the kernel and the root file system and load and start accordingly.

The offsets between the file systems can often be seen in the firmware. To create the right offsets between different parts of the firmware something called padding is used. This often

consists of zeroes, or other padding characters (0xff seems to be popular too). This makes it easy to recognize the different parts, since there will be a lot of these padding characters together (up to several thousand in some cases).

Popular bootloaders

There are a few GPL licensed bootloaders that are popular in current embedded products. In compliance engineering these are often overlooked.

Bootloader	platforms	comments
PPCBoot	PowerPC	discontinued, but still used
ARMboot	ARM	
u-boot	various	
RedBoot	various	originally from eCos, modified GPL license

To find out if these bootloaders are used it is often necessary to access the device through the serial port.

Compression techniques

Sometimes file systems (squashfs, ext2) or a kernel image can be found directly, but they are often in compressed form in the firmware and have to be decompressed first. During start up of the device the boot loader decompresses the kernel and/or file systems in memory, before actually launching the OS.

Common used compression methods are gzip and bzip2. A bit rarer, but increasing in popularity is the use of LZMA and 7z.

File systems

There are a couple of file systems in use on embedded Linux devices. They can be divided into two categories. The file systems in the first category load the file system from flash and uncompress it into normal memory. The file systems in the second category don't load into memory, but use more flash to reduce wear levelling on the flash memory. Both have their advantages and disadvantages.

Commonly used file systems are:

- squashfs, increasingly with LZMA compression instead of zlib compression
- ext2fs
- cramfs (Compressed ROM File System), both big endian/little endian
- romfs
- jffs2

Most of these file systems can be mounted over loopback on a recent Linux system (like

Fedora 9), but for some there are other ways too.

squashfs

Squashfs is a read only file system for Linux. It is a popular choice in embedded devices. It can be found in a firmware file by looking for the string 'sqsh' (big endian format) or 'hsqs' (little endian format).

The squashfs file system (with zlib compression) can also be unpacked as a normal user, using "unsquashfs" from the squashfs-tools package:

```
$ unsquashfs -d rootdir -i /path/to/squashfs-image
```

This command unpacks the squashfs image in the directory "rootdir". This method is actually preferable to mounting over loopback, since it won't create device files if you run it as a normal user and prevent you from mistakes later on, such as trying to grep through tty files.

Unpacking a squashfs file system with LZMA compression is possible in some cases, but not in all cases. The reason for this is that there are quite a few versions of LZMA in use, which are not always compatible. The Squashfs LZMA version at <http://www.squashfs-lzma.org/> for example uses different magic and it can't work with many Squashfs filesystems that are actually used on embedded devices.

It is not possible to detect LZMA compression using the command "file", since the signature is usually not different from an uncompressed squashfs file system. When you try to mount it and it fails, you might see this in dmesg, which is a clear indication another compression technique than gzip has been used:

```
SQUASHFS: Mounting a different endian SQUASHFS filesystem on loop0
SQUASHFS error: zlib_inflate returned unexpected result 0xffffffffd, srclength 8192,
avail_in 160, avail_out 8192
SQUASHFS error: sb_bread failed reading block 0x4b0
SQUASHFS error: Unable to read cache block [12bf5c:3d6]
SQUASHFS error: Unable to read inode [12bf5c:3d6]
```

The OpenWrt project builds a version of "unsquashfs" with LZMA support by default (called "unsquashfs-lzma"), since June 2008. With this tool it is possible to extract Squashfs 3.0 filesystems that use LZMA compression. Older versions of Squashfs can't be uncompressed with it. It is expected that this will be possible in newer versions, as soon as Squashfs with LZMA compression is accepted in the mainline kernel.

ext2/ext3

The default file system on most Linux systems is the ext2 file system, or the journalling variant ext3. These can simply be mounted on the majority of systems over loopback. Some kernels don't have support for this file system built in (rarely), or sometimes you have no root access to mount an ext2 file system over loopback. In such cases the e2tools package provides a barebones way to access an ext2 file system from userspace:

```
$ e2ls ramdisk_e1
bin          boot          default      dev          etc          home
```

```

image.cfs    lib          lost+found  mnt          proc          root
sbin        sys           tmp         usr          var           web

$ e2ls ramdisk_e1:etc
TZ           fstab        ftpaccess   ftpaccess.default
ftpconversions  ftpmaxnumber hotplug     inittab
nsswitch.conf rc.d         samba

```

cramfs

Another popular (though less and less seen) is the cramfs file system. It can be fairly easily recognized by searching for the string "Compressed ROMFS". There are two versions: one for big endian systems (PowerPC, SPARC, big endian MIPS) and little endian systems (x86, little endian MIPS).

Depending on which system you work on these file systems might need to be byteswapped from big endian to little endian, or vice versa. The cramfsswap tool can be used for this.

This byte swapping will not always work, since some devices (notably with the bcm63xx chipset) have a patched cramfs implementation, but it is often enough to extract at least the directory hierarchy and names of the files on the device, which will often give you more information about what is actually on the device.

jffs2

The jffs2 file system is special, since it can't be mounted directly over loopback. It first needs to be written to a special device in memory, which can then be mounted as a normal file system. For this some dark kernel voodoo magic is needed.

The jffs2 file system comes in two flavours: little endian and big endian. Big endian file systems can't be mounted on little endian file systems and vice versa. It might be necessary to convert the endianness of the file system with a program such as jffs2dump before you can access its contents.

```

modprobe mtdcore
modprobe jffs2
modprobe mtdram
modprobe mtblock
modprobe mtdchar
dd if=/path-to-jffs2-file of=/dev/mtd0
mount -t jffs2 /dev/mtblock0 /tmp/mnt/

```

This should be enough to mount the file system. The default size that the mtd device can hold is 4 MB. Sometimes there are bigger jffs2 file systems than that and you have to supply a size parameter when loading the mtdram module:

```
modprobe mtdram total_size=8196
```

This will create a ramdisk sized 8 megabytes.

Executable files

Executable files are usually the "real" programs on a device. There are two types of

executable files:

- scripts
- compiled programs

Scripts can be GPL licensed too, but since they tend to be human readable anyway this is often regarded as not having the highest priority.

The focus of GPL compliance engineering is mostly on compiled programs, which have been transformed from a human readable format into a machine readable format by a process called "compilation".

Compilation 101

Compilation is the process of turning a piece of human readable code into a machine executable program. It starts with someone writing a program in a programming language like C or C++. The compiler analyses the program (this is called "parsing") and translates it into object files. The object files are then linked into an executable, or into general purpose libraries, so they can be used by multiple programs.

There are two types of linking. The first one is static linking, where all functionality that is needed is compiled into one standalone binary file. This includes (parts of) the system C library and all other libraries that are needed to make the program run. Static linking is done at compile time.

Dynamic linking works differently. The linking phase is postponed until the program is actually executed. A program called "dynamic linker" combines the program with the libraries that need to be loaded to make the program run.

License wise there are no differences between the two (an often made mistake), but the reverse engineering process might differ.

Executable formats

There are a few types of executable formats you can find on an embedded device:

- ELF with/without gzip compression, stripped and not stripped
- Binary Flat format (bFLT) with/without gzip compression

The ELF format is the most common format. Most of the time the binaries will be "stripped", which means that all the debugging information has been removed from the file. If you are lucky the binary has not been stripped and all this information will still be there. This gives more clues about what is actually in the file.

A rare form of the ELF format is where the programs are compressed with gzip, after the ELF header. To get to the contents of the file you first have to extract the contents from the file. This is done in the same way as you would extract a file system which has been compressed with gzip.

The ELF format is an industry standard. There are a lot of tools which can be used to inspect ELF binaries from all kinds of platforms. The GNU binutils collection contains a few tools for

doing exactly this: readelf and objdump.

One of the interesting sections in the ELF format is the so called 'dynamic section'. In this section the dynamically linked libraries are listed:

```
$ objdump -x <file> | grep NEEDED
```

There is a lot more functionality that readelf and objdump offer, but the bulk of violations are not discovered that way.

Another format is the Binary Flat Format, or bFLT. It is the default on uClinux based systems and not used on normal Linux systems. This format is more space efficient than ELF, but also contains less information which can be used to identify strings inside programs. There are also fewer tools available with which you can inspect the binaries (other than just dumping the strings). As with the ELF format, there is a special variant which uses gzip compression that has to be unpacked first.

Tools

The toolbox of a reverse engineer contains a lot of tools, mostly the standard GNU utils. The minimal set of tools would be:

- binutils
- hexdump
- file
- ldd (part of a C library, like GNU libc)
- strings
- grep
- any editor that can read in binary files, such as vi or emacs
- dd
- gunzip/zcat
- bunzip2/bzcat
- lzma
- tar
- cpio
- a recent Linux system with support for loopback file systems
- cramfsswap, to translate between big endian/little endian cramfs file systems
- squashfs-tools
- mtd-utils
- e2tools

- p7zip
- unrar
- base64
- md5sum/sha1sum/sha256sum for comparing files and fingerprinting files
- cabextract
- unshield

Most tools are fairly straightforward to use, but a few deserve a bit more attention:

hexdump

The hexdump utility is a very valuable tool for reverse engineering. It displays the contents of a file, with offsets and ASCII translations, if the '-C' option is used. It outputs via standard output, so you will need a pager, such as 'more' or 'less' to catch its output. Example output would look like this:

```
00012da0  20 64 6f 6e 65 2c 20 62  6f 6f 74 69 6e 67 20 74  | done, booting t|
00012db0  68 65 20 6b 65 72 6e 65  6c 2e 0a 00 1f 8b 08 00  |he kernel.....|
00012dc0  3b b2 2a 44 02 03 ec bd   7d 7c 54 57 b9 36 bc f6  |;. *D....}|TW.6..|
```

With this you can quickly spot interesting text ("kernel") and the gzip header ("1f 8b 08") immediately following it. Padding can be easily be spotted because hexdump "compresses" this information for you by using '*':

```
00007ee0  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |.....|
*
00010000  27 05 19 56 af a1 29 38   44 2a b2 3f 00 0b 11 b8  |'..V..)8D*.?....|
```

file

The "file" tool quickly lets you determine what a file might contain. It does so by looking at the first so many bytes and comparing that with known signatures from the so called "magic" file, which on a Linux system can usually be found in /usr/share/magic.

```
$ file main-fs
main-fs: Squashfs filesystem, big endian, version 3.0, 9319589 bytes, 1498 inodes,
blocksize: 65536 bytes, created: Fri Aug 10 14:33:39 2007
```

Often firmware will just show up as "data":

```
$ file zImage
zImage: data
```

This is because a filesystem often has a header or other bytes (padding) put in front of it. Using "file" is not a 100% foolproof method. Sometimes a match for a file system or compressed file is found, while in reality there is no file system or compressed file there. Also "file" won't detect every file system. Before you use it, always try to have the latest version of the magic database installed on your system.

strings

The "strings" tool comes in handy when you want to extract readable strings from a binary file. The strings you extract from binaries are often gibberish, but the readable parts you can get out of a binary are often very helpful and contain function names, literal output written by programs (for example `kprintf()` statements), and so on. These strings, combined with a search engine or knowledge base of known strings, can reveal a lot.

grep

The "grep" tool is great for quickly finding strings in files (even binaries) that can be important. "Copyright" (with and without capitalization), "Free Software", "License", "GPL" and "General Public License" are good strings to search for. If you specify the command line option "-i" your searches will be case insensitive and quite a bit slower. I usually search for "icense", or "opyright", omitting the first character, which may or not be capitalized. It often saves me a few minutes waiting.

Be warned, many file systems contain special device files or symbolic links to /tmp or other parts of your own file system. If you're not careful you might be grepping on your whole computer, or 'grep' might be stuck on a special device file. A good idea is to first filter out the right files with for example "find" and then grep through them.

bzip2/bzcat

Data compressed with bzip2 can be easily found by searching for the string "BZh" inside the firmware image.

gzip/zcat

A gzip header as used in most devices starts with the header "1f 8b 08" (hexadecimal). Using "hexdump -C" these can be easily found. If you look with vi or vim, then these three characters are formatted as "`^_<8b>`" by vim. Files can be easily unpacked by using zcat and redirecting output to a file:

```
$ zcat infile > outfile
```

lzma

A technique that is becoming increasingly popular is LZMA compression. Its claim is that it offers better compression than other compression techniques. There is support for LZMA decompression in various bootloaders and it is fairly popular for Squashfs file systems as a replacement for zlib compression.

An example from a file that uses LZMA compression:

```
00000020 4c 69 6e 75 78 20 4b 65 72 6e 65 6c 20 49 6d 61 |Linux Kernel Ima|
00000030 67 65 00 00 00 00 00 00 00 00 00 00 00 00 00 |ge.....|
00000040 5d 00 00 00 02 00 b0 2a 00 00 00 00 00 00 00 6f |].....*.....o|
```

The LZMA compressed file can be recognized by the sequence '5d 00 00', at hex offset 0x040. Unpacking can be done by the 'lzma' tool. It does not support offsets, so in this case

the first 64 bytes should be removed, after which it can be unpacked:

```
$ lzma -cd infile > outfile
```

Another tool that is convenient is 'lzmainfo', which gives a lot of information about a file compressed with LZMA:

```
$ lzmainfo lzma-file
```

```
lzma-file
Uncompressed size:          3 MB (2797568 bytes)
Dictionary size:           32 MB (2^25 bytes)
Literal context bits (lc):  3
Literal pos bits (lp):     0
Number of pos bits (pb):   2
```

cramfsswap

The cramfsswap utility is a tool that changes the endianness of a cramfs (Compressed ROM file system). A cramfs file system can be in both little endian and big endian form. If you want to mount a cramfs file system on a system, the cramfs file system has to be in the same endianness as the host system. Sometimes it is necessary to first convert between little endian and big endian, or vice versa.

mtd-utils

The mtd-utils package contains various tools to work with flash memory devices. One of the most useful tools for compliance engineer is jffs2dump. With jffs2dump you can inspect the structure file systems and change endianness.

A rule of thumb is that if you dump the contents of the jffs2 file (using -c) and you get a lot of warnings, but no real data, you should supply one of the options -b (big endian) or -l (little endian), depending on the endianness of your own system.

unrar

The RAR archiving format is very popular on Windows, because of its (claimed) superior compression rates. A lot source archives are distributed in this format. To unpack these files on Linux/Unix you should use the "unrar" program. This program is distributed as freeware and there is currently no free software alternative.

md5sum/sha1sum/sha256sum/sha512sum

Fingerprinting tools, like md5sum and other tools from the SHA family, come in handy for identifying files. They work by taking the contents of a file and creating a cryptographic checksum. Two files that are identical will have the same checksum.

Since the MD5 and SHA1 algorithms are known to have "collisions" (two files can have the same fingerprint) it is advised to use sha256sum or sha512sum instead.

cabextract

The "cabextract" utility is a program to extract cabinet (.cab) archives. This is an archive format which is commonly used on Microsoft Windows. Since GPL license violations are not limited to just Unix(-like) platforms, but also can occur on Microsoft Windows this is a useful tool to have. Some ActiveX components, for example to make an IP camera work with Internet Explorer, are distributed as a cabinet archive.

unshield

The "unshield" utility is used to extract InstallShield cabinet files. They are a variant of the .cab archives that can be extracted with "cabextract".

Physical access

The final part of compliance engineering work is getting physical access to a device. Sometimes the bootloader is not shipped in a firmware update and can only be accessed through a serial console or JTAG. Often a GPL licensed bootloader is used on a device. If you don't perform a check using a serial port, it can easily be missed.

Serial console

Many devices have a serial port , or a serial port can be attached to it without too much effort. A serial port is used during development of the device. The firmware of the device often lets you log in on the device via the serial port when you connect to it through a serial cable, or gives you a root shell on the device directly. This is not always guaranteed to work. In some devices no output is sent to the serial port during booting, or once the device has booted.

Attaching a serial cable to a router

You can log onto the serial port by using a cable, which attaching one side to the serial port on the router and the other port to the PC, either a serial port on the motherboard, or a serial-USB converter.

WARNING: Many routers work on 3.3 Volts, while a serial port on a PC works on 12 Volts. You need a special cable which can shift between the two voltages or you risk blowing up the device.

There are special kits (MAX 232) to make so called "level shifters", that take care of the voltage difference. Old Siemens phone cables also work. Some online shops sell RS 232 shifters:

http://www.sparkfun.com/commerce/product_info.php?products_id=449

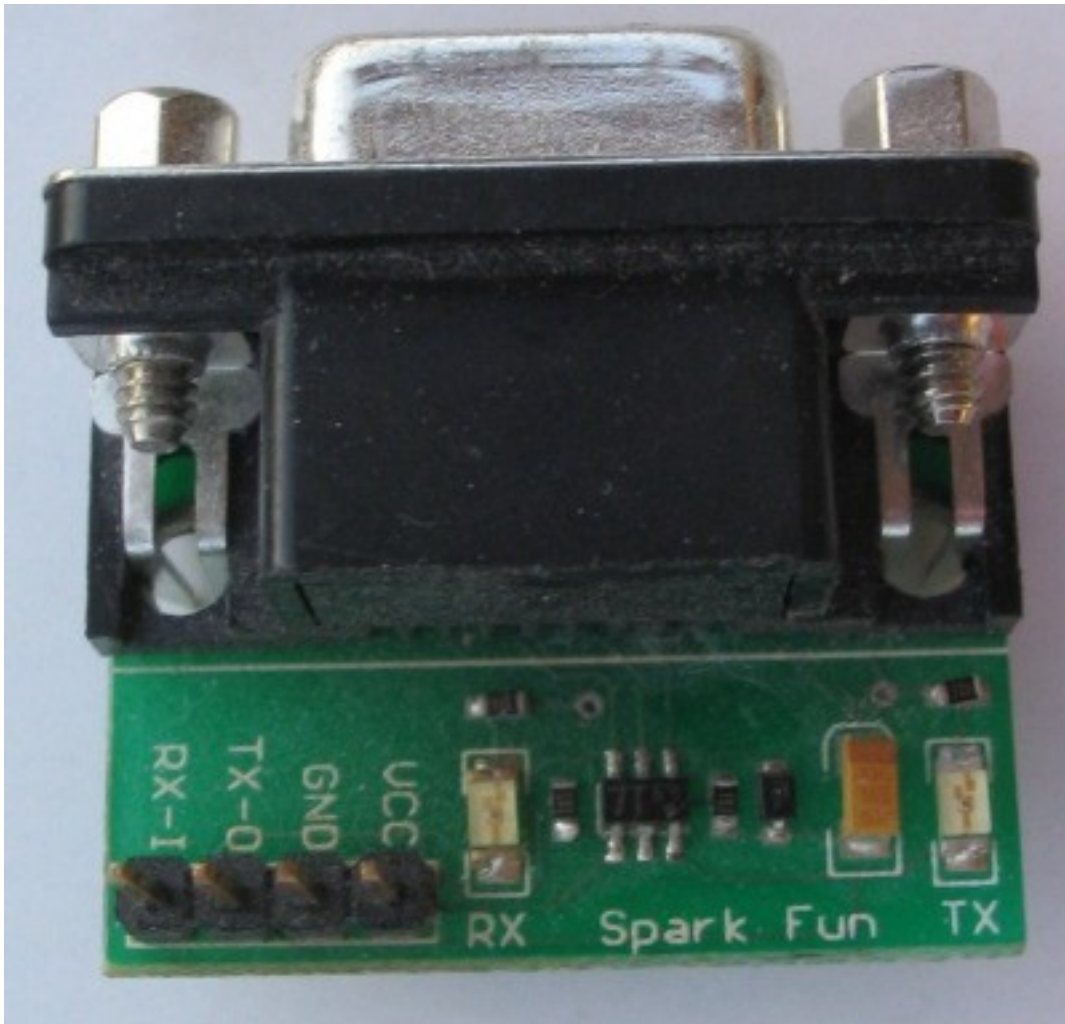


Illustration 1: Pre-made level shifter.

Before you can connect the onboard serial port to a PC you often have to solder header pins onto the solder pads for the serial port. Often there are four or more solder pads next to each other on a device, sometimes even labeled as "serial", "COM1", or equivalent. These header pins can be obtained at any decent electronics store, for a few cents per pin.

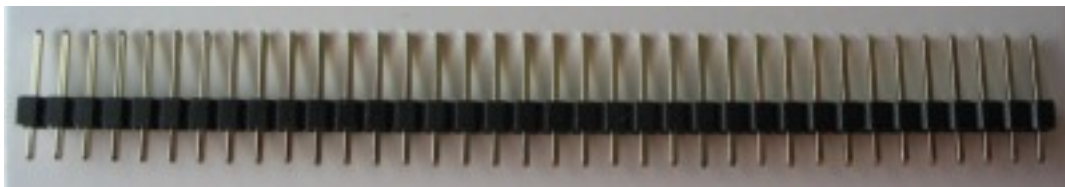


Illustration 2: A row of 36 header pins.

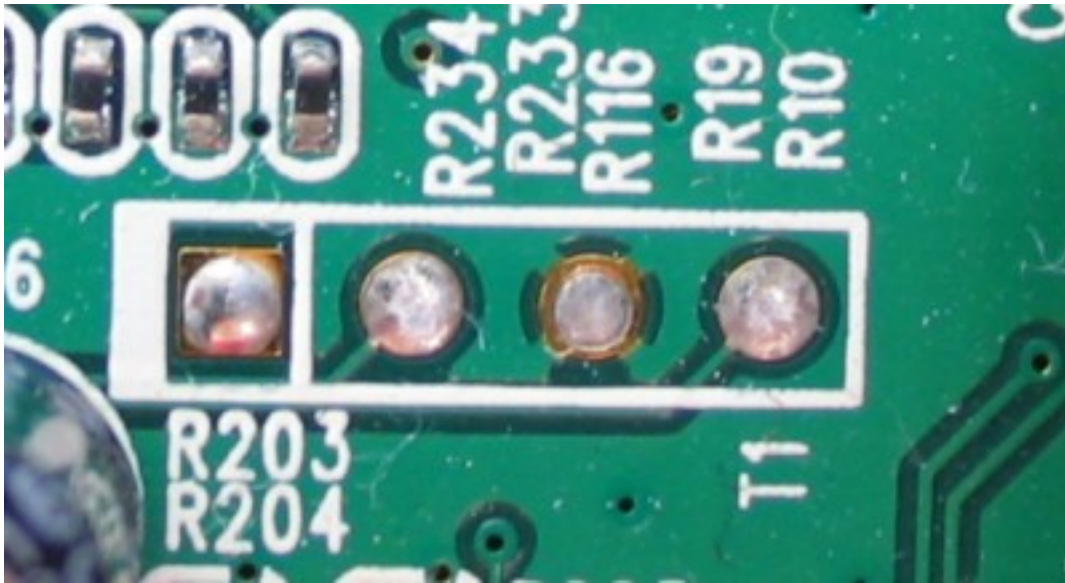


Illustration 3: Solder pads for a serial port on a device, without header pins

Some vendors, notably AVM in their Fritz!Box devices, try to hide these solder pads to make physical access to the device harder. Luckily most vendors simply don't care and in some of the devices you can already find pin headers soldered onto the solder pads.

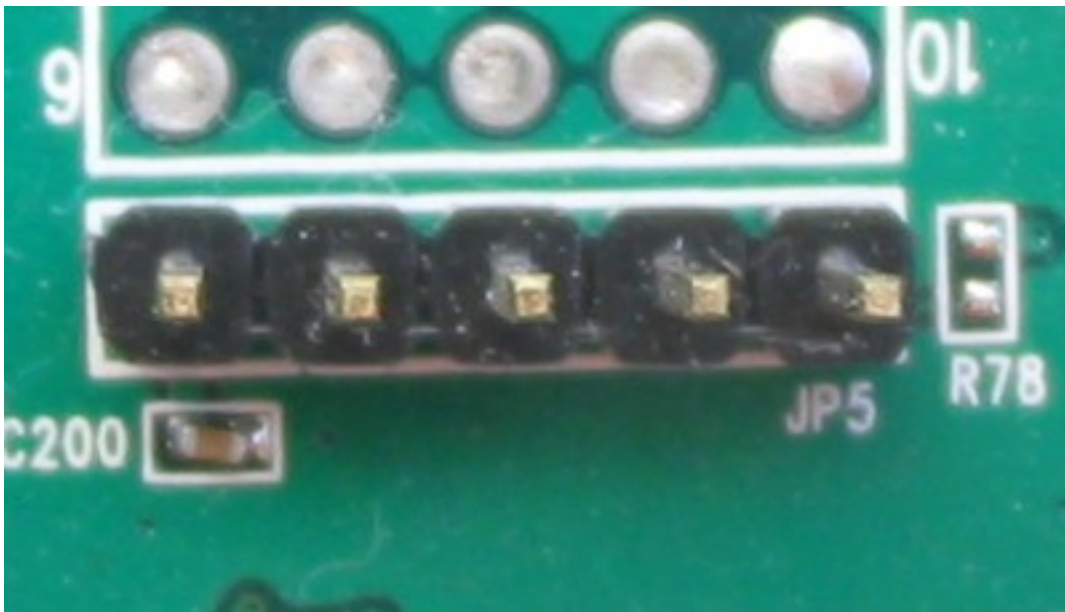


Illustration 4: Serial port with header pins pre-soldered.

Of the solder pads usually only four are needed:

- GND (ground)
- Tx (transmit)
- Rx (receive)

- VCC

These can be mapped to equivalent ports on a serial port on the PC. Different boards have different layouts, often varying between models and revisions.

A serial port can be discovered by using a multimeter to measure the voltage on the pins or solder pads.

- GND: 0 volts
- VCC: 3.3 volts
- Tx/Rx: variable

Many boards use a default layout and people have already made the effort of finding out where the ports on a wide variety of boards are located. The OpenWrt wiki is a great (though somewhat chaotic) resource for this. It is advised to always verify with a multimeter if the information is correct.

The next step is having a proper cable. A cheap solution is to use CD-ROM audio cables. Some of these have connectors that can easily be reused. The connectors look like this:

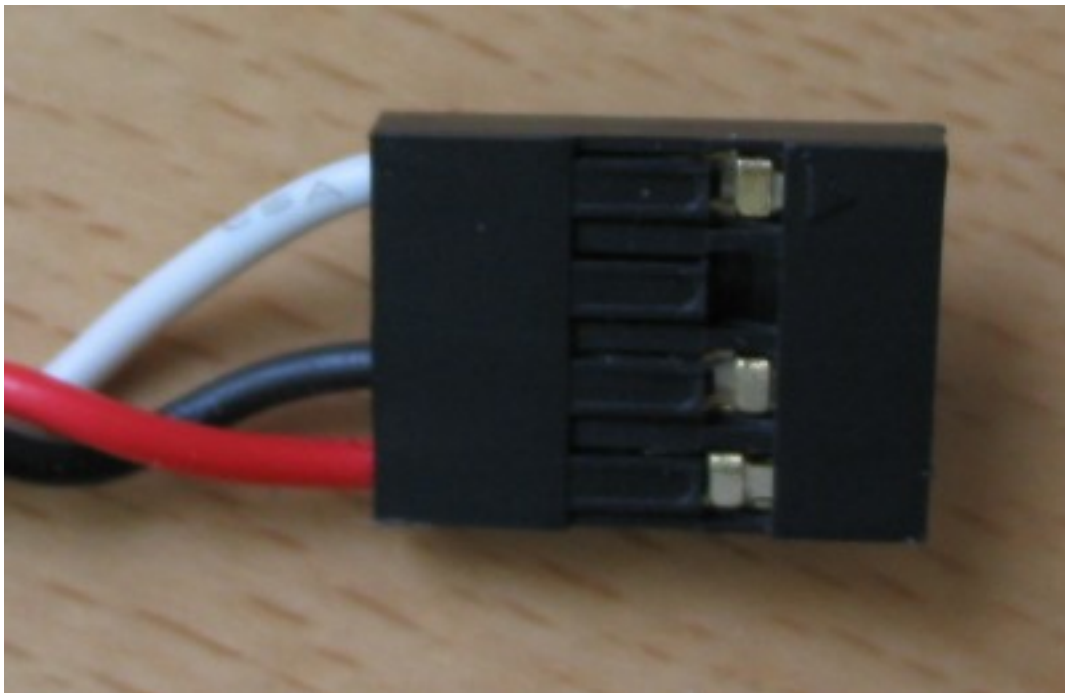


Illustration 5: CD-ROM audio cable.

The black clips that are holding the connectors can easily be lifted, so the cable, plus connector, can be removed by simply pulling the cable. The connectors can then be attached to the pins on the serial port.

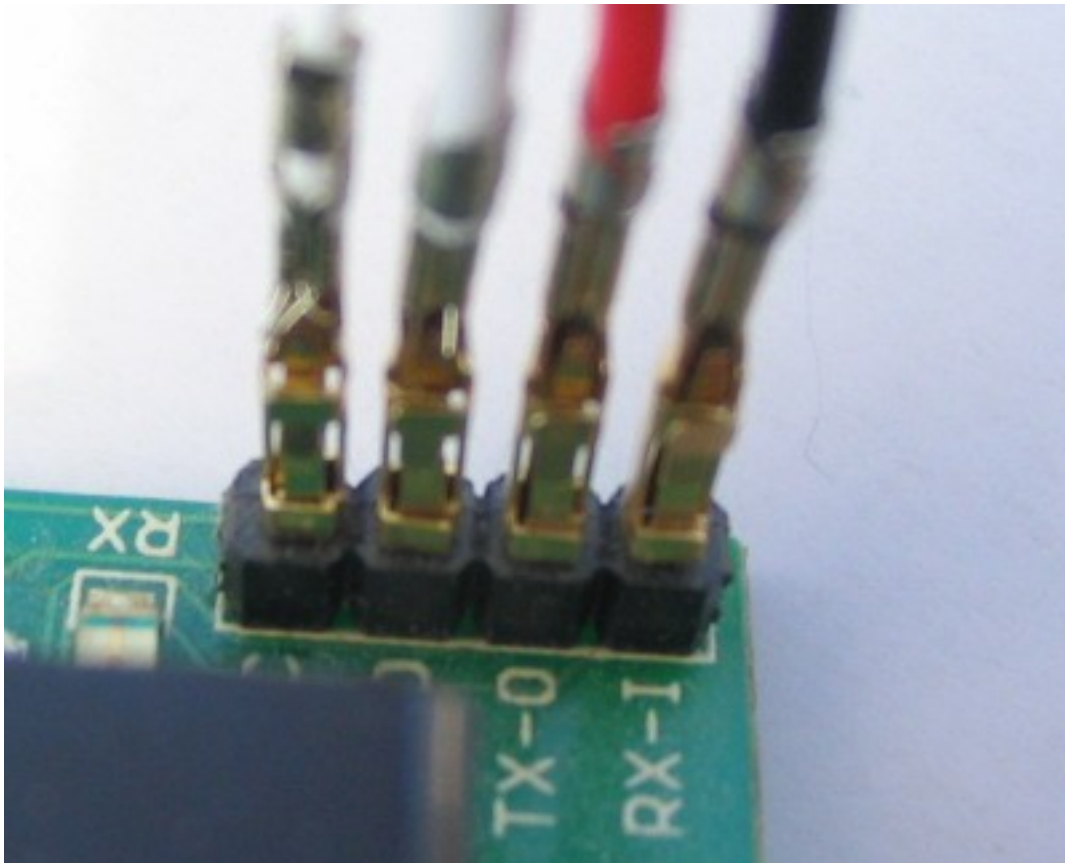


Illustration 6: Modified CD-ROM audio cable attached to header pins.

Accessing the serial port

When the serial cable has been properly attached to the router it can be accessed using a serial communication program. The most popular one on Linux is called 'minicom'. Not all serial ports use the same speed (or 'baud rate'). Popular baud rates are 9600, 38400, 57600 and 115200.

JTAG

Some devices can only be accessed through JTAG.

Software packages

A lot of packages are common on most devices. Depending on the package concerned, different techniques will have to be employed to find violations. A few common ones are described below.

Linux kernel modules

One of the grey areas of Linux kernel licensing has been kernel modules. There are a lot of

kernel modules which are not licensed as GPL. In the past there has been ongoing discussions whether or not the modules should be GPL licensed or not.

One of the first measurements to clarify the status is the use of a "license" macro in kernel modules:

```
# strings rt2500.o | grep license  
license=GPL
```

Modules that have set this macro will have access to more internals of the kernel. Licensing of modules that have this macro set should never be an issue.

With regard to other modules opinions differ. Greg Kroah-Hartman, one of the leading Linux kernel developers, told me in a personal e-mail on 14 October 2007:

[I]t's quite simple, me, and my lawyers feel that there is NO way to have a Linux kernel module that is not under the GPLv2. To do so otherwise violates the license of the kernel, and my copyrights. But it's not only me that says this, Novell and IBM have publicly stated this in the past, as well as HP (well, they kind of murmured it, but have said so in person.) Red Hat also states this, as well as a number of key Linux kernel contributors and holders of copyright on the kernel.

The Linux Foundation also issued a statement on closed source drivers and modules on June 23 2008:

http://www.linuxfoundation.org/en/Device_driver_statement

Appendix C of the book "Building Embedded Linux Systems", published by O'Reilly, also has 11 pages dedicated to how kernel developers see the legal status of binary kernel modules. Although the mails are dated (in the time period 1999-2002) and the authors of the mails are not legal professionals, they do provide an insight into the subject.

busybox

Busybox is a program that tries to add a lot of functionality of programs into one, while leaving out the more advanced features of many of the GNU tools. It is the Swiss army knife of embedded computing and nearly default on embedded devices. It works by making a symlink from a program, such as "cat" to the busybox binary. Depending on as which program it is invoked it will behave differently.

By default not all functionality is built into busybox. Using a configuration (much like the configuration for the Linux kernel, using a curses based interface) it is determined at compile time which functionality is built into busybox. The configuration commonly resides in a file called ".config" and will be written there the first time the configuration is run.

Options that are enabled are set like this in the configuration file:

```
CONFIG_CAT=y
```

Options that are disabled are set as:

```
# CONFIG_CHGRP is not set
```

Inside the busybox binary you can find the string "Currently defined functions", which is followed by a list of functions, which maps more or less directly to the configuration of busybox.

In general, the symlinks to the busybox binary and the functions defined in the binary should match the busybox configuration, or else it is a violation of the license.

C libraries

A Linux system is not complete without the so called C library, which contains functionality every program on the system, apart from the Linux kernel itself, is using one way or another. There are three C libraries on Linux that are popular: glibc, uClibc and dietlibc. Both glibc and uClibc are LGPL licensed and dietlibc is GPLv2 licensed, but sources are often missing.

Toolchain

An often overlooked part in the compliance process is the so called toolchain.

Most code is normally translated from human readable code to machine readable code, by using a compiler. A compiler parses, checks and translates the source code and generates machine readable code for the platform it was told to generate code for. In most cases, that is the same platform it is running on. So, for example, on my PC I compile a program with the standard compiler that Fedora 9 ships. The output of the compilation process will be a program that can run on my PC. If I would be developing for another platform, based on the MIPS or ARM architecture (or another platform, or another operating system) I would have to instruct my compiler to generate code that will run on that platform, because programs for my Intel x86 based PC will for example not run on a box that uses a MIPS CPU and runs NetBSD. For this you need a special setup of compiler, plus assembler and linker (found in GNU binutils) that can generate code for a specific platform and a C library to turn it into a working executable. This is commonly referred to as 'the toolchain'. This is not something the standard compilers on standard Linux distributions do by default.

The task of building a cross compiler is not trivial and quite tricky to get right (it even gets a lot more fun when you try to cross compile a cross compiler).

Some vendors, such as Broadcom, have adapted the GNU Compiler Collection (GCC) and GNU binutils to take advantage of/use specific characteristics of their CPU. Without these extensions to the compiler you will never be able to create a new program and run it on a machine with code generated with that compiler (it might not be as black and white as this, but it will make it a lot harder).

It is an ongoing debate whether or not the toolchain itself should be shipped. Some people say it should be, since without it it is very difficult and sometimes even impossible to build a new executable for a device without having access to the exact cross compiler that was used for building the software on the device. Other people say that because only the result of the toolchain is distributed, the toolchain does not need to be distributed.

It is beyond any doubt that if a toolchain is available in binary form in the GPL sources for a

device and it contains GPL or LGPL code (gcc, binutils, glibc, dietlibc or uClibc) the licenses should be adhered to. This is often an issue, since the toolchain is distributed as part of a software development kit (SDK), of which the sources are often not available to the vendors, that supply to the end company.

Experiences

Experience from several years looking through several hundreds of source archives has learned there are a few easy targets to look for in GPL compliance. These targets can serve as a very simple litmus test for GPL compliance. One easy target is the toolchain. Often a binary-only toolchain is shipped in a GPL archive, without sources. For other tools, like the ones to create an actual file system (mksquashfs with or without LZMA compression, mkfs.jffs2, genromfs, mkcramfs) the sources are missing quite often too.

Another common violation is lack of bootloader sources (if a GPL licensed bootloader is used on the device) and add-on packages which were not part of the original SDK the vendor got from upstream.

A tricky source of violations, which is hard to explain to vendors, is when "extra software" is shipped in the GPL sources that is not present on the device. It often happens that a certain software stack for a particular board is used for developing various types of devices for various vendors. Traces of different devices, with different software, can show up in the GPL sources for a device, for example in the form of a file system with pre-compiled binaries, that was accidentally left in. While technically not interesting if you only want to tweak the software, this is a source for license violations. It is hard to explain to vendors, because in their eyes all the software that is on the device is in the GPL tarball, in a GPL compliant way.

Physical compliance

The physical compliance requirements various licenses have are often overlooked. Compliance engineering is not complete without an inspection of the documentation that is shipped with a device.

The GPL and LGPL licenses require that a copy of the license is shipped with the device, either physically (for example, as part of the manual) or on a documentation CD-ROM. Quite often a device is not shipped with either of them, or just the GPL, even if LGPL licensed code is in used (which is the case in all Linux based devices).

Compliance engineering on Microsoft Windows

Most GPL violations seem to be occurring on embedded systems running Linux. While it is true that there are a lot of violations on embedded Linux, there appear to be plenty of violations on Microsoft Windows too, but not in the code that is part of Windows itself. The reason that these violations are fairly unknown is that they have never been a focal point for compliance engineering, mostly due to lack of research into this area.

Common violations

A common report is of shareware programs, like CD/DVD burning programs, or music players, that are being distributed in a GPL in-compliant way. The 'creators' of those programs tend to be rather immune to requests for the source code and keep happily violating the GPL and LGPL licenses.

Other reported violations are Cygwin, which is used in management software for various expensive routers. An interesting area of research for violations is in ActiveX components that are shipped with for example IP cameras or routers. The ActiveX components are on the device itself and are downloaded by the web browser from the device to get some extra functionality, such as viewing data, or controlling a camera. This is software too and it should also be checked for violations.

Tools

A common archiving format for Windows executables is 'cabinet archives', which often have the .cab file extension. On Unix systems the "cabextract" and "unshield" tools can be used to extract these files, depending on the type of file ("unshield" can only be used on InstallShield cabinet files).

On Windows different file formats are used than on Linux. On Linux the ELF executable format is primarily used, but on Windows the PE executable format is used. Binaries in PE format keep their data in a different form, in such a way that tools like "strings" are usually not successful for extracting interesting data. A PE decompiler or disassembler would be needed to extract this information better.

Cygwin compliance engineering

Cygwin is a program which provides a real POSIX compliant system for Microsoft Windows. Cygwin is GPL licensed. A lot of GNU packages have been ported to Cygwin. Packages that need Cygwin to run have included a DLL (dynamic-link library) with POSIX compatibility code in it. This code is licensed under the GPL license.

A Windows program can be easily detected using the 'file' command:

```
$ file a_program.exe
a_program.exe: PE32 executable for MS Windows (console) Intel 80386 32-bit
```

The contents of a file can be checked with the 'strings' program:

```
$ strings a_program.exe | grep cygwin
cygwin_internal
cygwin1.dll
_cygwin_crt0
__cygwin_crt0_common@8
_cygwin_premain3
_cygwin_premain2
_cygwin_premain1
_cygwin_premain0
__cygwin_crt0_bp
_cygwin_internal
```

```
_cygwin1_dll_iname  
__head_cygwin1_dll  
__imp__cygwin_internal
```

This is a clear indication that Cygwin is used.

Appendix A: GPL checklist

This is a small checklist for making sure a device is GPL compliant

1. Check the bootloader for GPL compliance. Use the list from the 'bootloader' section. If one of the bootloaders mentioned there is used, hunt down the sources.
2. Check if the device is GPL and LGPL compliant.
3. Check if the sources that are shipped are GPL and LGPL compliant (complete and not shipping more than necessary).
4. Check the documentation shipped with the device if it complies with GPLv2 section 1 and LGPLv2 section 1.

Appendix B: Reporting, handling and preventing violations

Reporting a violation

Reporting a violation to a corporation is not that difficult, but is often not done properly. Many times accusations and suspicions are voiced on public mailing lists. This is not always a good idea. Many mailing lists and websites are indexed and archived by search engines. A false report about a violation that is actually not a violation might be indexed. A false report can border on libel.

Also, always make sure you have your facts right. Loudly claiming that a device is not GPL compliant, while later on there actually is a written offer included with a device will only be a waste of time for all parties involved. If you are not absolutely sure there is a violation, don't accuse companies. So for example, if you have not checked if there was a written offer with the software or device, you can't claim there is a violation in a device.

Also, try to get copyright holders involved, since they are the only ones who can take legal action if necessary (or at least: have the highest chance of succeeding).

Handling a violation report

The other side of the coin is how to handle a violation as a company when a report comes in. From experience I can tell that complete silence is not a good strategy, even if you are actively working on fixing the violation. Always confirm that you have received the report and that you are looking into it. If the violation report was done in public, try to move the discussion to a non-public place as soon as possible. Mud slinging in public will only result in bad publicity for all parties involved.

Just confirming that a violation has occurred and then doing nothing is unacceptable.

Eventually the reporter will be disgruntled and chances are high he will go public. Checking the violation is a must. It is OK to ask the reporter where he thinks the violation is, it is not fair to offload your own compliance engineering work on the reporter.

Reporters might appear angry and this is usually because they are. In the free software community violating a license is frowned upon a lot. Many people have invested a lot of time writing software and licensed it under the GPL for a reason. They don't like the license terms being breached.

Not every reporter knows the full implication of the license. Make sure you do. Follow the GPL checklist, as described in Appendix A. Also, if necessary, employ or commission an engineer who can do compliance engineering for you.

Preventing a violation

The best way to fix a violation is to have it never occur at all. If you don't make your software or devices yourself you should make sure your suppliers get things right. An effective way to ensure that they do this is to give responsibility for code compliance to your suppliers in contracts you sign with them. Bearing all costs of resolving the GPL violation and perhaps a penalty, is an effective way to push costs to the source of the violation. Since margins in the embedded market are so low, this should be a good way to try to ensure there will be less violations in your product. In the end, contract language will be like an insurance: unnecessary most of the time, but very handy if it is there.

The Freedom Task Force of the Free Software Foundation Europe has made available boiler plate contract language, which can be used as a basis for your own contracts.

Appendix C: Commercial compliance engineering

This documentation was made by Armijn Hemel at Loohuis Consulting, while doing research for gpl-violations.org.

Loohuis Consulting is specialized in tailor made hosting, development, training and consultancy.

Loohuis Consulting is one of the few companies in the world to offer GPL compliance engineering as a service. The increased use of Free Software requires an understanding of the licenses in use as well as best practice in deployment, deployment processes and compliance. Loohuis Consulting employees have practical experience in this field, especially with regards to embedded devices.

Loohuis Consulting is also one of the leading experts on Universal Plug and Play security. Our employees are pioneers in undertaking security audits on devices using Universal Plug and Play and have written award winning papers and given numerous presentations on the subject.

For more information please visit the Loohuis Consulting website:

<http://www.loohuis-consulting.nl/>