

Code Coverage using Gcov

How Gcov works ?

Gcov [GCOV01] is a tool part of the GNU CC suite used for code coverage analysis.

The easiest way to get started with gcov, on a modern Unix operating system, is to do the following:

- 1- `./configure --with-your-options CFLAGS="-static -fprofile-arcs -ftest-coverage"`
- 2- `make`
- 3- `./your_binary`
- 4- `gcov main.c`

`dot.bb` file contains a list of source files (including headers), functions within those files and line numbers corresponding to each basic block in the source file.

`dot.bbg` file contains a list of the program flow arcs for each function which in combination with the `.bb` file enables gcov to reconstruct the program flow.

At runtime, the counter vector entries are incremented every time an instrumented basic block is entered then the program dumps the counter information into the `dot.da` at the time of exit (it populates the file with the size of the vector and the counters of the vector itself).

This is further documented in [GCOV01] Section 8.4: « Brief description of gcov data files ».

Notes :

(a) One can notice that those files have changed since gcc-3.4 (CC by default on FreeBSD 6.x).

Only one file is now created at build time: `dot.gcda`, when exiting, another one called `dot.gcno` is created, containing the results.

(b) Another interesting change appears in gcc-3.3 where `__bb_fork_func` has been renamed `__gcov_flush`

To start with, we recommend that you statically link your application and, preferably, build it without optimisations.

Note :

dynamic libraries and constructor function `__bb_init_func` and `__bb_fork_func` :

If you want to avoid such an error: «Undefined symbol "`__bb_init_func`"», use static binaries.

Explanation: when compiling using Gcov special flags "`-fprofile-arcs`" and "`-ftest-coverage`", the linking of dynamic libraries may not perform as expected.

Then, in our case, we need to move some files generated by gcc on the build host (`dot.bb`, `dot.bbg` and the source code) to the custom firewall.

Another point: Gcov requires the source tree be the exactly the same as the one on the build host.

From the `gcov(1)` manpage:

« gcov should be run with the current directory the same as that when you invoked the compiler. Otherwise it will not be able to locate the source files. »

Hopefully, you do not have to worry about where, on the filesystem, you run the binary because it is statically linked and the PATH to where the binary has been built is hardcoded during the compilation process. Using `strings(1)` against an instrumented binary will confirm that.

Nevertheless, the PATH needs to be restructured on the coverage/testing host **before** you launch it,

otherwise it won't be able to create the dot.da files at the end of its execution.

Example: arc profiling: Can't open output file /home/update/hping3-aphal-pre2/sendrawip.da

Having to relocate those files is somewhat ugly and may be avoid as soon as we switch to gcc-3.4 which introduce cross-profiling features [GCOV02].

After its invocation, gcov products dot.gcov files containing the original source code.

The first row of this file is used to indicate the number of times the function has been called during the tests.

Lines starting with the ##### string indicate lines that have never been executed (ie: not covered by the regression test) and the ones starting with the – string are lines without code.

A sample gcov output file:

```
 -: 0:Source:testssl.c
 -: 0:Object:testssl.bb
 -: 1:#include <stdio.h>
 -: 2:
 -: 3:#define OPENSSEL_THREAD_DEFINES
 -: 4:#include <openssl/opensslconf.h>
 -: 5:
 -: 6:
1: 7:int main() {
 -: 8:#if defined(THREADS)
 -: 9:     printf("SSL has threads\n");
 -: 10:#else
1: 11:     printf("SSL has no threads\n");
call 0 returns 100%
 -: 12:#endif
 -: 13:
 -: 14:}
```

As you can notice, gcov only knows about binary in the first time, thus we get 100% coverage with this trivial example even if one printf() is not called (removed at compilation by the C pre-processor).

This is logical but needs to be well understood especially for some portability cases since we can get different coverage metrics from one OS to another because of this.

Developers should read [GCOV01] §8.3: « Using gcov with GCC Optimization ».

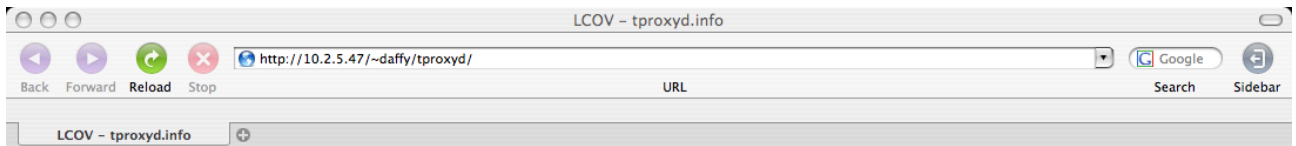
User Interface

Because gcov only creates ASCII text files, the latest stage was to use some parsing tools to generate human readable reports. As usual we do not want to reinvent the wheel and finally choose lcov [LTP01] from the Linux Testing Project for this part.

Lcov automates the process of extracting the coverage data using Gcov and producing HTML results based on that data.

This tool is licenced under the terms of the GPL v2 and deals well with large projects (for example tproxyd is linked with step less than 22 libraries !).

```
> geninfo --no-checksum --directory appdir --capture --output-filename tproxyd.info
> genhtml -o /export/lcov/tproxyd tproxyd.info
```



LTP GCOV extension - code coverage report

Current view: **directory**




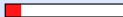
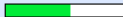









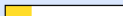

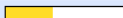
Test: **tproxyd.info**

Date: **2006-03-15**

Code covered: **16.1 %**

Instrumented lines: **21005**

Executed lines: **3373**

Directory name	Coverage
/usr/home/davidg/ROOT_DIR_V6_2_0_BP_L/UNIX/lib/libnasm	 27.6 % 883 / 3205 lines
/usr/home/davidg/ROOT_DIR_V6_2_0_BP_L/UNIX/lib/libnbase	 24.0 % 600 / 2503 lines
/usr/home/davidg/ROOT_DIR_V6_2_0_BP_L/UNIX/lib/libnconv	 40.6 % 54 / 133 lines
/usr/home/davidg/ROOT_DIR_V6_2_0_BP_L/UNIX/lib/libncrypto	 12.9 % 524 / 4067 lines
/usr/home/davidg/ROOT_DIR_V6_2_0_BP_L/UNIX/lib/libnpattern	 54.6 % 53 / 97 lines
/usr/home/davidg/ROOT_DIR_V6_2_0_BP_L/UNIX/lib/libnsrp	 0.0 % 0 / 910 lines
/usr/include	 0.0 % 0 / 10 lines
lib/libfw	 8.5 % 93 / 1090 lines
lib/libfwldap	 0.0 % 0 / 2329 lines
lib/libfwlog	 17.2 % 238 / 1383 lines
lib/libfwpki	 0.0 % 0 / 1361 lines
lib/libfwproxy	 16.1 % 270 / 1681 lines
lib/libfwthread	 0.0 % 0 / 232 lines
lib/libnasm/FreeBSD	 55.1 % 366 / 664 lines
lib/libnbase/FreeBSD	 21.8 % 44 / 202 lines
lib/libnfw	 10.5 % 73 / 695 lines
src/tproxyd	 39.5 % 175 / 443 lines

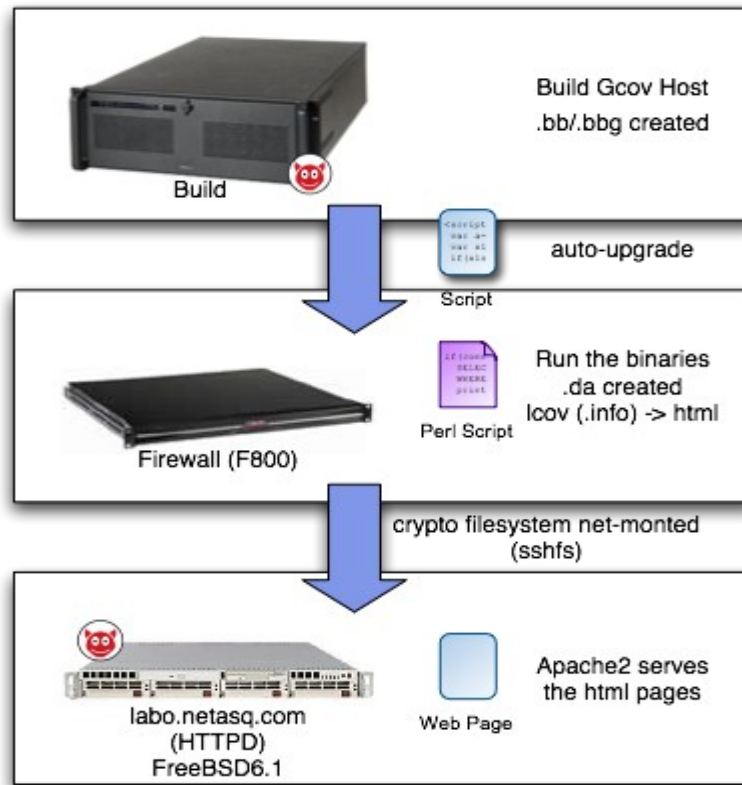
Generated by: [LTP GCOV extension version 1.4](#)

Another project similar to lcov is ggcov [GGCOV01], it implements some features missing in lcov like the ability to quickly know which functions in the source code were never called.

This feature and the doxygen documentation could be a great help when writing unit tests for an increasing code coverage metric.

Last but not least, the real benefit of code coverage analysis is that it can be used to analyze and improve the coverage provided by a test suite.

In these terms, code coverage is necessary but not sufficient.



Coverage specific flow

[GCOV01] http://gcc.gnu.org/onlinedocs/gcc-3.0/gcc_8.html

[GCOV02] http://gcc.gnu.org/onlinedocs/gcc/Cross_002dprofiling.html#Cross_002dprofiling

[LTP01] <http://ltp.sourceforge.net/coverage/lcov.readme.php>

[GGCOV01] <http://ggcov.sourceforge.net/>

[20] <http://www-128.ibm.com/developerworks/linux/library/l-stress/>

[21] <http://archive.linuxsymposium.org/ols2003/Proceedings/All- Reprints/Reprint- Larson- OLS2003.pdf>