

# The Apport crash report format

Version 0.2

Martin Pitt <martin.pitt@ubuntu.com>

July 16, 2012

## Contents

|          |   |          |
|----------|---|----------|
| <b>1</b> | <b>Introduction</b>                         | <b>2</b> |
| <b>2</b> | <b>File format</b>                          | <b>2</b> |
| 2.1      | Structure . . . . .                         | 2        |
| 2.2      | Keys . . . . .                              | 3        |
| 2.3      | Textual values . . . . .                    | 3        |
| 2.4      | Binary values . . . . .                     | 3        |
| 2.5      | Ordering . . . . .                          | 4        |
| 2.6      | Example . . . . .                           | 4        |
| <b>3</b> | <b>Standard keys</b>                        | <b>4</b> |
| 3.1      | Generic fields . . . . .                    | 5        |
| 3.2      | Process specific data fields . . . . .      | 5        |
| 3.3      | Signal crash specific data fields . . . . . | 6        |
| 3.4      | Package specific data fields . . . . .      | 6        |
| 3.5      | Kernel specific data fields . . . . .       | 7        |

# 1 Introduction

Apport is a system for automatic problem reporting and feedback, with the following features:

- intercept program crashes right when they happen the first time
- collect potentially useful information about the crash and the OS environment
- can be automatically invoked for unhandled exceptions in other programming languages (e. g. for Python)
- can be automatically invoked for other problems that can be detected mechanically (such as package installation/upgrade failures from update-manager)
- easy to understand UI that informs the user about the crash and instructs them on how to proceed,
- written in a very modular way: user interfaces (such as Gtk/Qt), crash databases (such as Launchpad/Bugzilla), packaging systems (apt/dpkg/rpm), are all factorized
- independent of a particular desktop environment, Linux flavour, etc.
- very robust due to exhaustive test suite coverage
- includes tools for post-processing crashes, such as post-mortem generation of symbolic stack traces, tools to create and work in chroots with only user privileges (using **fakeroot** and **fakechroot**)

The Apport home page<sup>1</sup> has some more information.

All components of apport (crash interception, enrichment with information, UI presentation, crash database up/download, crash post-processing) work on a common report file format. This allows adopters of Apport to use only some parts and combine it with existing project specific solutions like as Gnome's bug-buddy, and get the option to eventually merge such systems.

This document describes the structure of the report files and the pre-defined data fields.

## 2 File format

### 2.1 Structure

Apport report files consist of key/value pairs encoded with the standard RFC822<sup>2</sup> header format. Key name and value are separated by a dot and a space (" : ").

There must not be any blank lines and no lines that start with a non-whitespace character and do not start with a key name and a colon.

---

<sup>1</sup><https://wiki.ubuntu.com/Apport>

<sup>2</sup><http://www.ietf.org/rfc/rfc0822.txt>

## 2.2 Keys

Key names consist of numbers (0 – 9), English letters (a – z and A – Z) and dots (.).

## 2.3 Textual values

Single line textual values directly follow the key name, colon and dot without any further encoding or escaping. There is no line length limit.

In multi-line textual values, the line feed character (`\n`, ASCII Code 10) is escaped by appending a single space (ASCII code 32). In other words, every line of a multi-line value but the first one must be indented by a single space which is not part of the value.

## 2.4 Binary values

This is a compressed format intended for binary data such as memory dumps. It can optionally be used for long textual values like large log files if they should be compressed.

A binary value is introduced by the text “`base64`” and a line break following the key name, colon, and space. After that, the binary data is encoded as follows:

- Write a gzip header
- Initialize a zlib compressor object.
- Read a block of (at most) 1 MiB (1,048,576 Bytes) of binary data.
- Compress this block with the zlib compressor.
- Generate the base64-encoding of the compressed block
- Write a space and the base64-encoded block to the report file.
- If there is more source data to be encoded, go to 2.
- flush the zlib compressor, append the gzip trailer, base64-encode the tail and write it to the report file, again with a space prefix.

With this algorithm the binary encoding format obeys the same text line folding convention than the textual values.

## 2.5 Ordering

In order to keep the report files readable by humans, the following conventions should be met:

- The textual values should be at the top, the binary values at the bottom of the file. This eases their inspection in web browsers, even with partial downloads.
- Within each group (textual/binary), the keys should appear in ascending alphabetical order.

Software that processes Apport crash report files must not rely on those conventions. It is acceptable to not follow them for performance reasons.

## 2.6 Example

This table shows an example data set:

| Key     | Value  |
|---------|--|
| Short1  | Single line value                                |
| Date    | December 24, 2000                                |
| Long    | Multiple lines<br>with leading<br>space          |
| TestBin | ABABABABABABABABAB\0\0\0\0\0\0\0\0\0\0ZZZZZZZZZZ |

This would be encoded as:

```
Date: December 24, 2000
Long: Multiple lines
      with leading
      space
Short1: Single line value
TestBin: base64
eJw=
c3RyxIAMcBAFAG55BXk=
```

## 3 Standard keys

In order to provide a basic level of interoperability between all systems using the Apport report format, a number of standard key names and semantics are defined. This is particularly important for tools which automatically reprocess problem reports.

Implementations can add additional fields at will, especially if these are mainly aimed at human examination. Field names which will be processed mechanically should be added to this standard document eventually.

### 3.1 Generic fields

The following keys apply to all types of problem reports. They classify the problem type and give information about the date, operating system and user environment.

**ProblemType:** (required) Classification of the problem type; Currently defined values are `Crash`, `Kernel`, `Package` (for failed install/upgrade of a software package), and `Bug` (for general bug reports)

**Date:** (required) Date and time of the problem report in ISO format (see `asctime(3)`)

**Uname:** (required) Output of `uname -srm`

**OS:** (optional) Name of the operating system. On LSB compliant systems, this can be determined with `lsb_release -si`.

**OSRelease:** (optional) Release version of the operating system. On LSB compliant systems, this can be determined with `lsb_release -sr`.

**Architecture:** (optional) OS specific notation of processor/system architecture (e. g. `i386`)

**UserGroups:** (optional) System groups of the user reporting the problem; for privacy reasons this should only include IDs smaller than 500, no groups which belong to other real users.

### 3.2 Process specific data fields

The following fields describe interesting properties of a particular process. This always applies to ProblemTypes `Crash` and also to `Bug` if the bug is reported against a running process (as opposed to just a package).

**ExecutablePath:** (required) Contents of `/proc/pid/exe` for ELF files; if the process is an interpreted script, this is the script path instead

**InterpreterPath:** (required for scripts) Contents of `/proc/pid/exe` if the process is an interpreted script

**ProcEnviron:** (required) A subset of the process' environment, from `/proc/pid/env`; this should only show some standard variables that do not disclose potentially sensitive information, like `$SHELL`, `$PATH`, `$LANG`, and `$LC_*`.

**ProcCmdline:** (required) Contents of `/proc/pid/cmdline`

**ProcStatus:** (required) Contents of `/proc/pid/status`

**ProcMaps:** (required) Contents of `/proc/pid/maps`

**ProcAttrCurrent:** (optional) Contents of `/proc/pid/attr/current`; this contains the process' security context if there is a Linux Security module enabled that makes use of that interface (e.g. SELinux, AppArmor).

### 3.3 Signal crash specific data fields

The following fields describe properties of a process that crashed due to a signal. This applies to `ProblemType Crash` if a core dump is available. Note that `Crash` is also used for unhandled exceptions of programs written in scripting languages, in which case there is no core dump.

**CoreDump:** (optional) core dump (binary value); this can also be a 'minidump' format or any other useful image of the stack.

**Stacktrace:** (optional) Stack trace (e. g. produced by gdb's `bt full` command or minidump processor)

**ThreadStacktrace:** (optional) Threaded stack trace (e. g. produced by the gdb command `thread apply all bt full` or minidump processor)

**StacktraceTop:** (optional) First five frames of **Stacktrace** with the leading addresses and local variables removed; this is intended to be evaluated for automatic duplicate detection

**Registers:** (optional) Register dump (e. g. produced by gdb's `info registers` command)

**Disassembly:** (optional) Disassembly of the code leading to the crash (e. g. produced by gdb's `x/16i $pc` command)

Note that every crash report must contain **CoreDump** or a symbolic **Stacktrace** in order to be useful at all. The recommended approach is to include the stack trace for the initial report, and drop it once it has been recombined with debug symbols to produce a full **Stacktrace**.

### 3.4 Package specific data fields

The following fields describe properties of a package and its dependencies. This applies to `ProblemTypes Crash`, `Package`, and `Bug` if the bug applies to a particular package (as opposed to being a generic OS bug).

**Package:** (required) Package name and version, separated by space

**PackageArchitecture:** (required if different from **Architecture**) Processor architecture the package was built for; there are some architectures (like `x86_64` or `sparc64`) which support multiple package architectures

**Dependencies:** (required) Package names and versions of all transitive dependencies of the package; one line per package

**SourcePackage:** (optional) The name of the corresponding source package

Optionally, the name and version in **Package** and **Dependencies** can be followed by a list of modified files in that package, enclosed in brackets. Example:

**Package:** `bash 3.2-1`

**Dependencies:** `libreadline5 5.2-3 [modified: /lib/libreadline.so.5]`

`libc6 2.5-1 [modified: /etc/ld.so.conf]`

### 3.5 Kernel specific data fields

The following fields describe properties of a kernel oops/crash. This applies to `ProblemType Kernel`.

**ProcVersion:** (required) Contents of `/proc/version`

**ProcCpuinfo:** (required) Contents of `/proc/cpuinfo`

**ProcModules:** (required) Contents of `/proc/modules`

**ProcCmdline:** (required) Contents of `/proc/cmdline`

**Dmesg:** (required) Output of `dmesg`

**LsPciVV:** (optional) Output of `lspci -vv`

**LsPciVVN:** (optional) Output of `lspci -vvn`