ABRT Documentation

Release 2.14

ABRT Project

Feb 04, 2022

Contents

ABRT is a set of tools to help users detect and report application crashes. Its main purpose is to ease the process of reporting an issue and finding a solution.

The solution in this context might be a bug tracker ticket, a knowledge base article or suggestion to update a package to a version containing a fix.

CHAPTER 1

Quick links

- Source code: GitHub
- ABRT blog
- IRC channel: #abrt @ irc.libera.chat
- Fedora problem tracker
- Mailing list

CHAPTER 2

Contents

2.1 How ABRT works

When your application crashes, the crash event is handled by one of ABRT's language or runtime dependent hooks which forwards the crash information to *abrt daemon*. The daemon stores the data in `/var/spool/abrt` and runs a chain of events that for example collect more data about the system, produce backtrace from core dump or notifies a user. This process is followed by automatic or manual reporting to various supported targets like bugzilla or ABRT Analytics (*ABRT Analytics*).

2.1.1 Reporting

Currently, typical desktop reporting work-flow consists of generating so called *µReport* (micro report) designed to be completely anonymous so it can be sent and processed automatically avoiding costly bugzilla queries and manpower.

The ABRT Analytics (also known as ABRT server in the past) in this scenario works like a first line of defense — collecting massive amounts of similar reports and responding with tracker URLs in case of known problems.

If user is lucky enough to hit a unique issue not known by ABRT Analytics, reporting chain continues with reporting to bugzilla, more complex process which requires user having a bugzilla account and going through numerous steps including verification that the report doesn't contain sensitive data.

ABRT also features unattended reporting possibilities in form of email reports, upload functionality and integration with tools like Spacewalk or Foreman for large scale deployment scenarios.

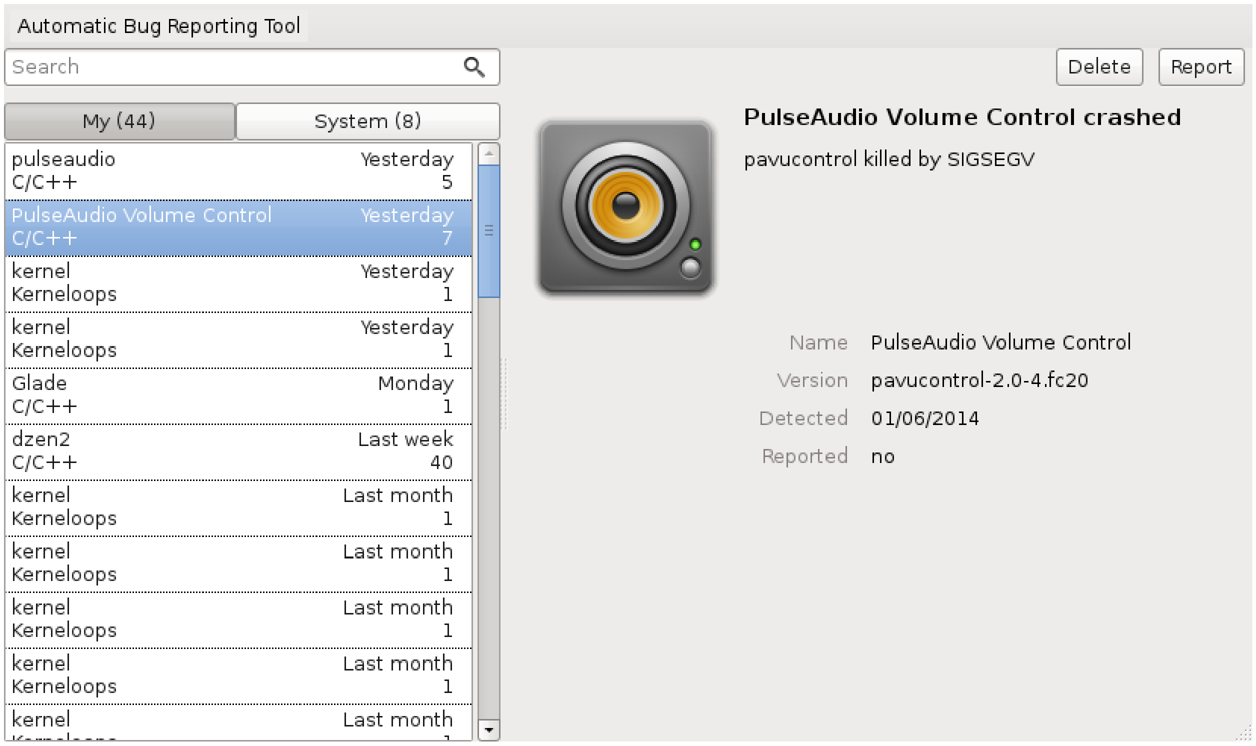
2.1.2 Components

abrt

The daemon and collection of tools for handling crashes and monitoring logs for errors. Takes care of storing and processing of the crashes caught by various hooks.

gnome-abrt

Simple GUI application for problem management and reporting.



pulseaudio C/C++	Yesterday 5
PulseAudio Volume Control C/C++	Yesterday 7
kernel Kerneloops	Yesterday 1
kernel Kerneloops	Yesterday 1
Glade C/C++	Monday 1
dzen2 C/C++	Last week 40
kernel Kerneloops	Last month 1
kernel Kerneloops	Last month 1
kernel Kerneloops	Last month 1
kernel Kerneloops	Last month 1
kernel	Last month

PulseAudio Volume Control crashed

pavucontrol killed by SIGSEGV

Name	PulseAudio Volume Control
Version	pavucontrol-2.0-4.fc20
Detected	01/06/2014
Reported	no

libreport

Libraries providing an API for reporting problems via different paths like email, bugzilla, ABRT Analytics, scp upload..

ABRT Analytics

Crash collecting server, also known as ABRT Analytics (or ABRT server in the past). Provides accurate statistics of incoming reports and acts as a proxy in front of bugzilla (or any other issue tracker) when it comes to automatic reporting of crashes. It's designed to receive anonymous *µReports* and to find clusters of similar reports among them. For reports that are known, user receives fast response containing links to ABRT Analytics's problem page, issue tracker or an entry from knowledge base, that contains number of well-known issues like usage of proprietary kernel modules or browser crashes caused by unsupported modules.

satyr

Satyr is a collection of low-level algorithms for program failure processing, analysis, and reporting supporting kernel space, user space, Python, and Java programs. Considering failure processing, it allows to parse failure description from various sources such as GDB-created stack traces, Python stack traces with a description of uncaught exception, and kernel oops message. Infromation can also be extracted from the core dumps of unexpectedly terminated user space processes and from the machine executable code of binaries. Considering failure analysis, the stack traces of failed processes can be normalized, trimmed, and compared. Clusters of similar stack traces can be calculated. In multi-threaded stack traces, the threads that caused the failure can be discovered. Considering failure reporting, the library can generate a failure report in a well-specified format, and the report can be sent to a remote machine.

retrace server

The retrace server provides a core dump analysis and backtrace generation service over a network using HTTP protocol.

2.1.3 Privacy

µReports consist of structured data suitable to be analyzed in a fully automated manner, though they do not necessarily contain sufficient information to fix the underlying problem. The reports are designed not to contain any potentially sensitive data to eliminate the need for review before submission.

2.2 Supported programming languages and software projects

2.2.1 Overview

Langauge/Project	Package
C	abrt-addon-ccpp
C++	abrt-addon-ccpp
Java	abrt-java-connector
Python	abrt-addon-python
Python 3	abrt-addon-python3
Ruby	rubygem-abrt
Linux (kernel oops)	abrt-addon-kerneloops
Linux (vmcore)	abrt-addon-vmcore
Linux (pstore)	abrt-addon-pstore
X.Org Server	abrt-addon-xorg

2.2.2 C/C++

ABRT installs its own core dump handler via `abrt-ccpp.service` which when started, overrides the default value of kernel's *core_pattern*. This causes C/C++ crashes to be handled by `abrtd` and by default prevents creation of `core.*` files in crashed process' current directory. More details available in *C/C++ hook* design section.

2.2.3 Java

ABRT Java Connector is a JVM agent which reports uncaught Java exceptions to ABRT. The agent registers several JVMTI event callbacks and has to be loaded into JVM using `-agentlib` command line parameter.

2.2.4 Python

Python hooks override the default `sys.excepthook` with custom function reporting uncaught Python 2 and Python 3 exceptions to `abrtd`.

2.2.5 Ruby

`rubygem-abrt` registers a custom handler using `at_exit` feature executed when the program ends which allows to check for possible unhandled exceptions.

2.2.6 Linux kernel

Kernel oops

By checking the output of kernel logs, ABRT is able to catch and process so called kernel oopses — non-fatal deviations from correct behavior of the Linux kernel. This functionality is provided by `abrt-addon-kerneloops` and `abrt-oops.service`.

Kernel panic

ABRT is able to process `vmcore` files (kernel core dumps) produced on fatal non-recoverable errors when kernel panics and reboot is required. If the kernel crash dumping mechanism is enabled[1] `vmcore` file is produced at the time of the crash. ABRT is then able to read and process the `vmcore` file from `/var/crash/` directory. This functionality requires installing `abrt-addon-vmcore` and enabling `abrt-vmcore.service`.

Persistent storage

`abrt-addon-pstoreoops` package contains plugin for collecting kernel oopses from platform dependent persistent storage (pstore)[2][3].

2.3 Installing ABRT

2.3.1 Server or headless machine

Use

`yum install abrt-cli`

to install basic support for crash collection and set of command line tools for crash management and reporting.

If you want to receive email notifications about crashes caught by abrt you need to install `libreport-plugin-mailx` package, which by default, sends notifications to `root` at local machine. Email destination can be configured in `/etc/libreport/plugins/mailx.conf` file.

If you prefer console notifications at login time, install `abrt-console-notification` package. Notification sample:

```
ABRT has detected 1 problem(s). For more info run: abrt-cli list --since 1394120788
```

2.3.2 Desktop

Use

`yum install abrt-desktop`

to install complete desktop support including notification applet (`abrt-applet`) and `gnome-abrt` — GUI for problem management and reporting.

[1] https://fedoraproject.org/wiki/How_to_use_kdump_to_debug_kernel_crashes
[2] http://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/ABI/testing/pstore
[3] http://mjg59.dreamwidth.org/23554.html

2.3.3 From source

Clone the repositories you're interested in:

```
git clone git@github.com:abrt/satyr.git
git clone git@github.com:abrt/abrt.git
git clone git@github.com:abrt/libreport.git
git clone git@github.com:abrt/faf.git
```

Proceed with installing dependencies, you can get list of required packages by running `./autogen.sh sysdeps` and directly install these with `yum` by running `./autogen.sh sysdeps --install`.

You can then build a project by running:

```
./autogen.sh
./configure
make
```

If build passes continue with `make rpm` which will create rpm packages in `./build` directory. This way is preferred over `make install` installation as your package manager keeps track of installed files allowing for easy removal.

2.4 Usage

2.4.1 Command line

When crash is detected on a headless machine abrt notifies user via email or console notification, see *Server or headless machine*.

To list all crashes on a machine run:

```
abrt-cli list
```

Example output:

```
id 58101e309c3e473d49b1a7d60868ab7023a62dd6
reason:         will_abort killed by SIGABRT
time:           Wed 17 Sep 2014 03:24:07 AM CEST
cmdline:        will_abort
package:        will-crash-0.7-4.fc21
uid:            1000 (mhabrnal)
count:          1
Directory:      /var/spool/abrt/ccpp-2014-09-17-02:51:07-5368

id 58101e309c3e473d49b1a7d60868ab7023a62dd6
reason:         will_abort killed by SIGABRT
time:           Wed 17 Sep 2014 02:51:07 AM CEST
cmdline:        will_abort
package:        will-crash-0.7-4.fc21
uid:            1000 (mhabrnal)
count:          1
Directory:      /var/spool/abrt/ccpp-2014-09-17-02:51:07-5368
Reported:       cannot be reported

id afdf8c42ddeb324b975f3510f6e976085b46c4fe
reason:         will_segfault killed by SIGSEGV
```

(continues on next page)

(continued from previous page)

```
time:           Tue 16 Sep 2014 11:12:09 PM CEST
cmdline:        will_segfault
package:        will-crash-0.7-4.fc21
uid:            1000 (mhabrnal)
count:          1
Directory:      /var/spool/abrt/ccpp-2014-09-16-23:12:09-4265
Reported:       https://retrace.fedoraproject.org/faf/reports/bthash/
↪102f484335e2df215da7a92d962d017e7d9edcc9
                https://bugzilla.redhat.com/show_bug.cgi?id=1124867
```

Displayed are three crashes collected by abrt. Each crash has an identifier and a directory that can be used for further manipulation using `abrt-cli`. The second crash (from the above example) cannot be reported. Run `abrt-cli info -d $ID` to see why the problem cannot be reported.

To display detailed report about particular problem use:

```
abrt-cli list -d <ID_OR_PATH>
```

To report a problem via `abrt-cli` use:

```
abrt-cli report <ID_OR_PATH>
```

To delete a problem run:

```
abrt-cli remove <ID_OR_PATH>
```

For more details consult `man abrt-cli`.

Selecting a preferred text editor

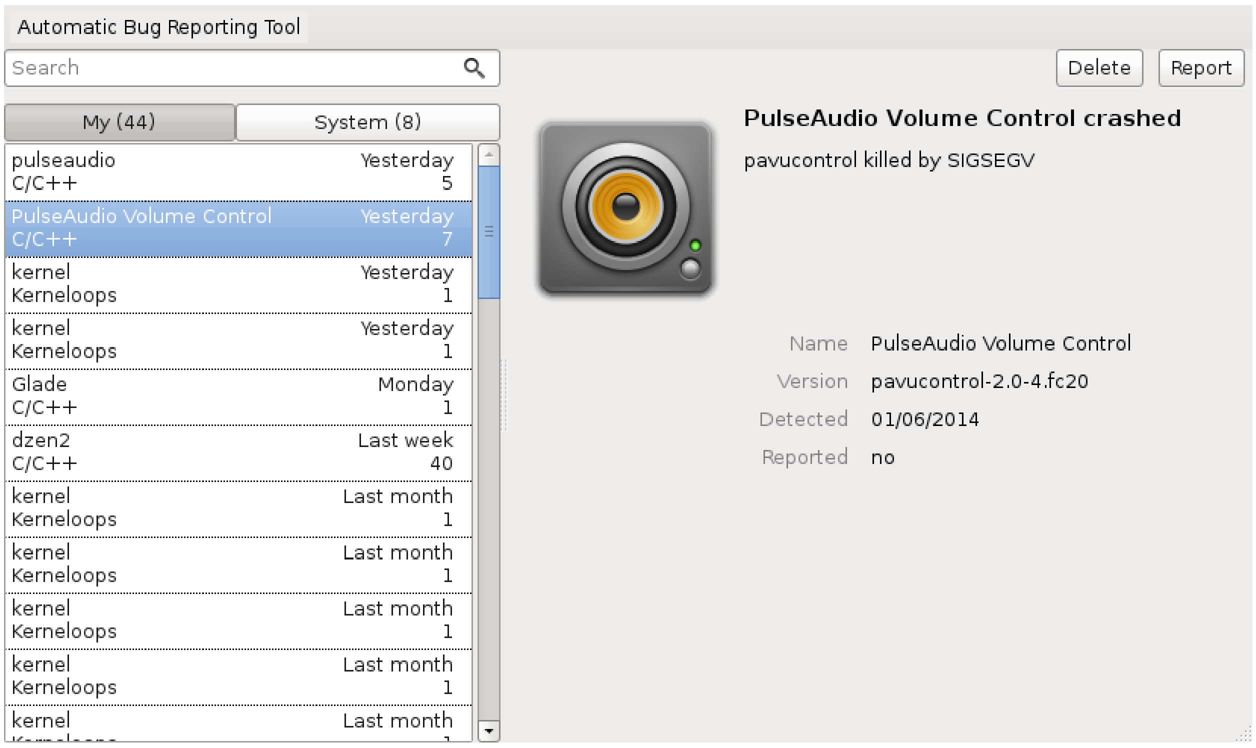
During the reporting process `abrt-cli report` will open a text editor. It uses the editor defined in `$ABRT_EDITOR` environment variable. If the variable is not defined, it checks the `$VISUAL` and `$EDITOR` variables. If none of these variables is set, `vi` is used.

You can set the preferred editor in your `.bashrc` configuration file. For example, if you prefer GNU Emacs, add the following line to the file:

```
export EDITOR=emacs
```

2.4.2 Graphical user interface

After a crash is handled by ABRT user is presented with a notification with options to ignore or report the problem. If user chooses to report the problem *gnome-abrt* application opens:



pulseaudio C/C++	Yesterday 5
PulseAudio Volume Control C/C++	Yesterday 7
kernel Kerneloops	Yesterday 1
kernel Kerneloops	Yesterday 1
Glade C/C++	Monday 1
dzen2 C/C++	Last week 40
kernel Kerneloops	Last month 1
kernel Kerneloops	Last month 1
kernel Kerneloops	Last month 1
kernel Kerneloops	Last month 1
kernel	Last month

PulseAudio Volume Control crashed

pavucontrol killed by SIGSEGV

Name	PulseAudio Volume Control
Version	pavucontrol-2.0-4.fc20
Detected	01/06/2014
Reported	no

Report button then starts a reporting wizard guiding user through reporting process.

2.4.3 Testing ABRT functionality

To make sure you won't miss a crash of your application you should verify that abrt works as expected.

Simplest way to do so is to crash `sleep` executable available everywhere:

```
sleep 10m &
kill -SIGSEGV %1
```

Sleep then produces segmentation fault and should be caught by abrt's C/C++ hook. If it's not working correctly consult *Debugging ABRT*.

will-crash

For testing the functionality of various hooks we've created a set of crashing executables called `will-crash`[1].

First, install the package:

```
yum install will-crash
```

Then run one of the crashing executables depending on which hook you would like to test, most commonly C/C++:

```
will_segfault
```

This executable segfaults immediately and should be caught by abrt. To get a list of other crashing executables run:

[1] http://github.com/sorki/will-crash

```
rpm -ql will-crash | grep bin
```

2.5 Debugging ABRT

Check steps described in *What to do when ABRT is not able to catch the crash of my application?*. If this won't solve your problem proceed with steps described on this page.

2.5.1 ABRT log messages

Look for failures in system log:

```
journalctl | grep abrt
```

or if your system doesn't support journal:

```
grep abrt /var/log/messages
```

Possible reasons for abrt not handling crashes might include:

- `abrtd` not running due to permission errors
- `DUP_OF_DIR: <dir>` your crash is a duplicate of previous crash
- `Executable '<exe>' doesn't belong to any package and ProcessUnpackaged is set to 'no'` — see *How to enable handling of unpackaged software*
- Your package is not GPG signed — see *How to enable handling of non-GPG signed software*
- Crashing executable has setuid bit set — see *How to enable dumping of setuid binaries*

2.5.2 abrtd

If `abrtd` malfunctions try stopping it and running it in foreground with verbose mode enabled:

```
systemctl stop abrtd
abrtd -vvv -d
```

If it hangs use `gdb` to attach to it and produce a backtrace:

```
gdb /usr/sbin/abrtd $( pidof abrtd ) -batch -ex 'bt full'
```

2.6 Interfacing with ABRT

2.6.1 Socket API

Socket API allows creation of new problems. It is used by Python and Java hooks to pass required data to `abrtd`.

Socket path:

```
/var/run/abrt/abrt.socket
```

First line has to contain HTTP header:

```
POST / HTTP/1.1\r\n\r\n
```

followed by `key=value` pairs delimited with `\0`. The server expects another `\0` at the end of the message.

Mandatory keys:

- `type` — *(string)* problem type, see *Supported problem types*.
- `pid` — *(integer)* process ID of the crashed procss, ranges from 0 to *PID_MAX* (`/proc/sys/kernel/pid_max`)
- `executable` — *(string)* path of the affected executable
- `backtrace` — *(string)*
- `reason` — *(string)* reason of the crash

To ensure the problem can be reported to Bugzilla via report-gtk or report-cli you have to add the following keys with the following contents:

- `duphash` — *(string)* duplicate hash. The hash is placed in Bugzilla's `Whiteboard` field in the format `abrt_hash:$duphash`. For C/C++, the content of `duphash` is the SHA-1 digest of the concatenation of the names of top six functions on the stacktrace. For Python exceptions, it is the SHA-1 digest of the stacktrace.
- `uuid` — *(string)* local identifier of the problem. The content can be the same as for `duphash`.

Optionally, the server accepts other elements listed in *Elements collected by ABRT*.

If there's no error, the server responds with:

```
HTTP/1.1 201 Created\r\n\r\n
```

or `400` status code in case of error.

Python hook may serve as an example of the socket API usage.

2.6.2 DBus API

Documentation for the DBus API at `org.freedesktop.problems` is available as part of the `abrt-dbus` package or online at http://jfilak.fedorapeople.org/ProblemsAPI/re01.html.

2.6.3 Python API

Documentation for the Python API is available in the `python3-abrt(5)` man page (part of the `python3-abrt` package).

2.7 Configuration

2.7.1 Configuration files

Upon installation, ABRT and libreport place their respective configuration files into the several directories on a system:

`/etc/libreport/` contains the `report_event.conf` main configuration file. More information about this configuration file can be found in *Event configuration*.

`/etc/libreport/events/` holds files specifying the default setting of predefined events.

`/etc/libreport/events.d/` keeps configuration files defining events.

`/etc/libreport/plugins/` contains configuration files of programs that take part in events.

`/etc/abrt/` holds ABRT specific configuration files used to modify the behavior of ABRT's services and programs. More information about certain specific configuration files can be found in *ABRT Specific Configuration*.

`/etc/abrt/plugins/` keeps configuration files used to override the default setting of ABRT's services and programs. For more information on some specific configuration files refer to *ABRT Specific Configuration*.

The configuration files expect Key-Value pairs separated by an equal sign. Quoting of the values is not supported.

2.7.2 ABRT Specific Configuration

Standard ABRT installation currently provides the following ABRT specific configuration files:

`/etc/abrt/abrt.conf` allows you to modify the behavior of the abrtd service.

`/etc/abrt/abrt-action-save-package-data.conf` allows you to modify the behavior of the abrt-action-save-package-data program.

`/etc/abrt/plugins/CCpp.conf` allows you to modify the behavior of ABRT's core catching hook.

The following configuration directives are supported in the `/etc/abrt/abrt.conf` file:

- `WatchCrashdumpArchiveDir = /var/spool/abrt-upload`

Enable this directive if you want *abrtd* to auto-unpack crashdump tarball archives (*.tar.gz*) which are located in the specified directory. In the example above, it is the `/var/spool/abrt-upload/` directory. Whichever directory you specify in this directive, you must ensure that it exists and it is writable for abrtd. The ABRT daemon will not create it automatically. If you change the default value of this option, be aware that in order to ensure proper functionality of ABRT, this directory **must not** be the same as the directory specified for the `DumpLocation` option.

Caution: Do not modify this option when using SELinux

Changing the location for crashdump archives will cause SELinux denials unless you reflect the change in respective SELinux rules first. See the *abrt_selinux(8)* manual page for more information on running ABRT in SELinux.

Remember that if you enable this option when using SELinux, you need to execute the following command in order to set the appropriate SELinux boolean allowing ABRT to write into the `public_content_rw_t` domain:

```
setsebool -P abrt_anon_write 1
```

- `MaxCrashReportsSize = size_in_mebibytes`

This option sets the amount of storage space, in mebibytes, used by ABRT to store all problem information from all users. The default setting is 5000 MiB. Once the quota specified here has been met, ABRT will continue catching problems, and in order to make room for the new crash dumps, it will delete the oldest and largest ones.

- `DumpLocation = /var/spool/abrt`

This directive specifies the location where problem data directories are created and in which problem core dumps and all other problem data are stored. The default location is set to the `/var/spool/abrt` directory. Whichever directory you specify in this directive, you must ensure that it exists and it is writable for *abrtd*. If you change the default value of this option, be aware that in order to ensure proper functionality of ABRT, this directory **must not** be the same as the directory specified for the `WatchCrashdumpArchiveDir` option.

Caution: Do not modify this option when using SELinux

Changing the dump location will cause SELinux denials unless you reflect the change in respective SELinux rules first. See the *abrt_selinux(8)* manual page for more information on running ABRT in SELinux.

Remember that if you enable this option when using SELinux, you need to execute the following command in order to set the appropriate SELinux boolean allowing ABRT to write into the `public_content_rw_t` domain:

```
setsebool -P abrt_anon_write 1
```

The following configuration directives are supported in the `/etc/abrt/abrt-action-save-package-data.conf` file:

- `OpenGPGCheck = yes/no`

Setting the OpenGPGCheck directive to yes (the default setting) tells ABRT to only analyze and handle crashes in applications provided by packages which are signed by the GPG keys whose locations are listed in the /etc/abrt/gpg_keys file. Setting OpenGPGCheck to no tells ABRT to catch crashes in all programs.

- `BlackList = nspluginwrapper, valgrind, strace, [more_packages ]`

Crashes in packages and binaries listed after the BlackList directive will not be handled by ABRT. If you want ABRT to ignore other packages and binaries, list them here separated by commas.

- `ProcessUnpackaged = yes/no`

This directive tells ABRT whether to process crashes in executables that do not belong to any package. The default setting is *no*.

- `BlackListedPaths = /usr/share/doc/*, */example*`

Crashes in executables in these paths will be ignored by ABRT.

The following configuration directives are supported in the `/etc/abrt/plugins/CCpp.conf` file:

- `MakeCompatCore = yes/no`

This directive specifies whether ABRT's core catching hook should create a core file, as it could be done if ABRT would not be installed. The core file is typically created in the current directory of the crashed program but only if the `ulimit -c` setting allows it. The directive is set to *yes* by default.

- `SaveBinaryImage = yes/no`

This directive specifies whether ABRT's core catching hook should save a binary image to a core dump. It is useful when debugging crashes which occurred in binaries that were deleted. The default setting is *no*.

2.7.3 Configuring ABRT to Detect a Kernel Panic

ABRT can detect a kernel panic using the `abrt-vmcore` service, which is provided by the `abrt-addon-vmcore` package. The service starts automatically on system boot and searches for a core dump file in the `/var/crash/` directory. If a core dump file is found, `abrt-vmcore` creates the problem data directory in the `/var/spool/abrt/` directory and copies the core dump file to the newly created problem data directory. After the `/var/crash/` directory is searched through, the service is stopped until the next system boot.

To configure ABRT to detect a kernel panic, perform the following steps:

1. Ensure that the `kdump` service is enabled on the system. Especially, the amount of memory that is reserved for the `kdump` kernel has to be set correctly. You can set it by using the `system-config-kdump` graphical tool, or by specifying the `crashkernel` parameter in the list of kernel options in the `/etc/grub2.conf` configuration file.

2. Install and enable the `abrt-addon-vmcore` package using yum:

   ```
   yum install abrt-addon-vmcore
   systemctl enable abrt-vmcore
   ```

   This installs the `abrt-vmcore` service with respective support and configuration files.

3. Reboot the system for the changes to take effect.

Unless ABRT is configured differently, problem data for any detected kernel panic is now stored in the `/var/spool/abrt/` directory and can be further processed by ABRT just as any other detected kernel oops.

2.7.4 Desktop Session Autoreporting

Enabled autoreporting behavior

With the desktop session autoreporting enabled, ABRT automatically uploads *μReport* for user problems immediately upon their detection. If the abrt server (faf) knows the reported problem, the server provides additional information about the problem and ABRT informs the user that the detected problem is known via a notification bubble. The notification bubble offers showing the problem's web page, opening the problem in the ABRT GUI or simply ignoring the problem. If the problem is unknown, ABRT shows a notification bubble and the user can either start reporting process as usual or ignore the problem.

Disabled autoreporting behavior

With the autoreporting disabled, ABRT uploads an *μReport* for the detected problem after click on *"Report" button* in the notification bubble. If the detected problem is not known to abrt server, ABRT proceeds with reporting wizard.

Event though ABRT notifies about problems of other users, it never uploads uReports for these problems automatically. The other user's problems are always processed in the way of processing problems with the autoreporting disabled, which is described in the 2nd paragraph.

In case of unavailable network, ABRT will postpone notification of the detected problems until the network becomes available again. The list of postponed problems will be held only for a single user desktop session. The postponed problems may not be notified at all, if the network won't become available during the desktop session's lifetime.

Uploading uReports requires a writeable problem directory and in order to make the reporting more automatic and less confusing, ABRT might move the problem directory from the system dump location (usually `/var/spool/abrt/` directory) to `$HOME/.cache/abrt/spool/` directory without asking the user for a permission to do so. However, ABRT moves the directories only if the user has no rights for writing to the system dump location.

Enabling desktop session autoreporting

The desktop autoreporting can be enabled in various ways. The easiest way is to answer **Yes** in a dialogue asking for *enabling automatically submitted crash reports* which appears after clicking on *"Report" button* in the notification bubble. The second way is to open the `Automatic Bug Reporting Tool` application, open the application menu and click on the following option:

```
ABRT Configuration
```

and turn on option:

```
Automatically send uReport
```

The last but the most inconvenient way is to manually edit file:

```
$HOME/.config/abrt/settings/abrt-applet.conf
```

and add the following line:

```
AutoreportingEnabled = yes
```

2.7.5 System Autoreporting

ABRT can be configured to submit an *µReport* for each of the detected problems to the abrt server (faf) immediately upon their detection. The server provides the following information about the submitted problem:

- URLs to existing bug reports if any (Bugzilla bugs)
- short description text

System Autoreporting can be enabled by issuing the following command:

```
abrt-auto-reporting enabled
```

or via Augeas:

```
augtool set /files/etc/abrt/abrt.conf/AutoreportingEnabled yes
```

or by adding the following line to the `/etc/abrt/abrt.conf` configuration file:

```
AutoreportingEnabled yes
```

When System Autoreporting is enabled, Desktop Session Autoreporting is enabled too.

2.7.6 Shortened Reporting

Enables shortened GUI reporting where the reporting is interrupted after AutoreportingEvent is done. It means that the reporting is done when user clicks *"Report" button* on the notification bubble. Upon that, ABRT uploads an uReport describing handled problem, shows a notification bubble stating that the problem has been reported and finishes.

Shortened Reporting has no effect on the reporting process started from the GUI, because we wanted to allow advanced users to easily submit full bug report into Bugzilla. We believe that all users who care about detected crashes and open **Automatic Bug Reporting Tool** application to see them are advanced users.

```
Default value:  Yes but only if application is running in GNOME
desktop
```

To turn Shortened Reporting on open:

```
Automatic Bug Reporting Tool
```

go to the application menu and click:

```
ABRT Configuration
```

and turn on option:

```
Shortened Reporting
```

Or manually edit file:

```
$HOME/.config/abrt/settings/abrt-applet.conf
```

and add there the following line:

```
ShortenedReporting = yes
```

2.7.7 Automatic sensitive data filtering

ABRT keeps the global list of *sensitive words* in `/etc/libreport/forbidden_words.conf` so in order to change this list for all users, system administrator has to edit this file. There is also per-user list in `$HOME/.config/abrt/settings/forbidden_words.conf` (doesn't exist by default, so user has to create it). The format of the file is one word per line. Wildcards are **NOT** supported.

The forbidden words are sometimes a part of other words and these are usually not deemed as sensitive information. Offering such false positive sensitive words for review by user makes the process of removing sensitive data from reports hard and the real sensitive data may be missed. Therefore, ABRT has another list of words that are never considered as sensitive information. The list contains common words consisting from *the sensitive words*. The global list of *ignored words* is kept in file:

> `/etc/libreport/ignored_words.conf`

And the per-user list:

> `$HOME/.config/abrt/settings/ignored_words.conf`

2.7.8 Event configuration

Each event is defined by one rule structure in a respective configuration file. The configuration files are typically stored in the `/etc/libreport/events.d/` directory. These configuration files are loaded by the main configuration file, `/etc/libreport/report_event.conf`.

The `/etc/libreport/report_event.conf` file consists of include directives and rules. Rules are typically stored in other configuration files in the `/etc/libreport/events.d/` directory.

If you would like to modify this file, please note that it respects shell metacharacters (`*`, `$`, `?`, etc.) and interprets relative paths relatively to its location.

Each rule starts with a line with a non-space leading character, all subsequent lines starting with the space character or the tab character are considered a part of this rule. Each rule consists of two parts, a condition part and a program part. The condition part contains conditions in one of the following forms:

```
VAR=VAL,

VAR!=VAL

VAL~=REGEX
```

where:

- `VAR` is either the `EVENT` key word or a name of a problem data directory element such as `executable`, `package`, `hostname`, ... See *Elements collected by ABRT* for more.
- `VAL` is either a name of an event or a problem data element, and
- `REGEX` is a regular expression.

The program part consists of program names and shell interpretable code. If all conditions in the condition part are valid, the program part is run in the shell. The following is an event example:

```
EVENT=post-create date > /tmp/dt
        echo $HOSTNAME `uname -r`
```

This event would overwrite the contents of the `/tmp/dt` file with the current date and time, and print the hostname of the machine and its kernel version on the standard output.

Here is an example of a yet more complex event which is actually one of the predefined events. It saves relevant lines from the `~/.xsession-errors` file to the problem report for any problem for which the `abrt-ccpp` services has been used to process that problem, and the crashed application has loaded any *X11* libraries at the time of crash:

```
EVENT=analyze_xsession_errors analyzer=CCpp dso_list~=.*/libX11.*
        test -f ~/.xsession-errors || { echo "No ~/.xsession-errors"; exit 1; }
        test -r ~/.xsession-errors || { echo "Can't read ~/.xsession-errors"; exit 1;␣
↪}
        executable=`cat executable` &&
        base_executable=${executable##*/} &&
        grep -F -e "$base_executable" ~/.xsession-errors | tail -999 >xsession_errors␣
↪&&
        echo "Element 'xsession_errors' saved"
```

The set of possible events is not hard-set. System administrators can add events according to their need. Currently, the following event names are provided with standard ABRT and libreport installation:

`post-create` This event is run automatically by `abrtd` on newly created problem data directories. When the `post-create` event is run, `abrtd` checks whether the `UUID` identifier of the new problem data matches the `UUID` of any already existing problem directories. If such a problem directory exists, the new problem data is deleted. See *Deduplication* for more details on duplicate handling.

`analyze_name_suffix` where *name_suffix* is the adjustable part of the event name. This event is used to process collected data. For example, the `analyze_LocalGDB` runs the GNU Debugger (`GDB`) utility on a core dump of an application and produces a backtrace of a crash.

`collect_name_suffix` where name_suffix is the adjustable part of the event name. This event is used to collect additional information on a problem.

`report_name_suffix` where name_suffix is the adjustable part of the event name. This event is used to report a problem.

Additional information about events (such as their description, names and types of parameters which can be passed to them as environment variables, and other properties) is stored in the `/etc/libreport/events/event_name.xml` files. These files are used by both GUI and CLI to make the user interface more friendly. Do not edit these files unless you want to modify the standard installation.

Standard ABRT Installation Supported Events

Standard ABRT installation currently provides a number of default analyzing, collecting and reporting events. Some of these events are configurable using the `gnome-abrt` GUI application. The following is a list of default analyzing, collecting and reporting events provided by the standard installation of ABRT:

`analyze_VMcore` — Analyze VM core Runs `GDB` (the GNU debugger) on problem data of an application and generates a backtrace of the kernel. It is defined in the `/etc/libreport/events.d/vmcore_event.conf` configuration file.

`analyze_LocalGDB` — Local GNU Debugger Runs `GDB` (the GNU debugger) on problem data of an application and generates a backtrace of a crash. It is defined in the `/etc/libreport/events.d/ccpp_event.conf` configuration file.

`analyze_RetraceServer` — Generate backtrace remotely Uploads core dump to *retrace server* for remote backtrace generation. It is defined in the `/etc/libreport/events.d/ccpp_retrace_event.conf` configuration file.

`analyze_xsession_errors` — Collect .xsession-errors Saves relevant lines from the `~/.xsession-errors` file to the problem report. It is defined in the `/etc/libreport/events.d/ccpp_event.conf` configuration file.

`report_Logger` — Logger Creates a problem report and saves it to a specified local file. It is defined in the `/etc/libreport/events.d/print_event.conf` configuration file.

`report_Mailx` — Mailx Sends a problem report via the `mailx` utility to a specified email address.i It is defined in the `/etc/libreport/events.d/mailx_event.conf` configuration file.

`report_Uploader` — Report uploader Uploads a tarball (*.tar.gz*) archive with problem data to the chosen destination using the FTP or the SCP protocol. It is defined in the `/etc/libreport/events.d/uploader_event.conf` configuration file.

`report_uReport` — *µReport* uploader Uploads *µReport* to faf server.

`report_EmergencyAnalysis` — Upload problem directory to faf Uploads a tarball to faf server for further analysis. Used in case of reporting failure when standard reporting methods fail. It is defined in the `/etc/libreport/events.d/emergencyanalysis_event.conf` configuration file.

2.7.9 Workflow configuration

report-gtk and report-cli are tools that report application crashes and other problems caught by abrtd daemon, or created by other programs using libreport. To do so, they invoke so called *EVENTs*. There are two ways to specify an EVENT to be performed. Either it can be specified as a command line argument (option `-e EVENT`) to report-gtk/report-cli, or it can be defined in a *workflow file* located in `/usr/share/libreport/workflows/`. Which of these workflow files will be used is defined in *workflow configuration files* in `/etc/libreport/workflows.d/`.

Every EVENT which is used in a workflow must have its corresponding XML description file in `/usr/share/libreport/events/`. The format of these files is described in the `report_event.conf(5)` man page.

Workflow file

Each XML file in `/usr/share/libreport/workflows/` must conform to the following DTD:

```
<!ELEMENT workflow    (name+,description+,priority?,events*)>
<!ELEMENT name        (#PCDATA)>
<!ATTLIST name         xml:lang CDATA #IMPLIED>
<!ELEMENT description (#PCDATA)>
<!ATTLIST description  xml:lang CDATA #IMPLIED>
<!ELEMENT priority =  (#PCDATA)>
<!ELEMENT events =    (event)+>
<!ELEMENT event =     (#PCDATA)>
```

`name` User-facing name of the workflow.

`description` User-facing description of the workflow.

`priority` Priority of the workflow. Higher number means a more visible place in the user interface. If none is provided, 0 is used. The value is a signed integer.

`events` List of events executed in the workflow.

event Name of the event to be executed. You may also use a wildcard (*) at the end of the name to select multiple events with a common prefix. If an event is not applicable to the problem data or if it is not defined, the process continues with next event sibling.

Workflow configuration file

The configuration file contains rules governing which of the workflows shall be executed under the given conditions. Each rule starts with a line with a non-space leading character. Each rule consists of two parts, the name of a workflow and an optional condition in following format:

```
EVENT=<WORKFLOW_NAME> [CONDITION]
```

The latter part consists of a space-separated list of conditions in one of the following forms:

```
VAR=VAL,

VAR!=VAL, or

VAL~=REGEX
```

where:

- `VAR` is a problem data directory element (such as `executable`, `package`, `hostname`, etc. – see *Elements collected by ABRT* for more),
- `VAL` is a problem data element, and
- `REGEX` is a regular expression.

Loading procedure

report-gtk or report-cli looks into the `/etc/libreport/workflows.d/` directory and goes trough all rules in each of the configuration files, checking if the specified conditions of each rule are satisfied.

If there is only one workflow whose conditions are satisfied, its specification is loaded from `/usr/share/libreport/workflows/<WORKFLOW_NAME>.xml` and each of its events is executed. If multiple workflows match the conditions, the user is given a choice wich of them should be executed.

Example workflow

To illustrate the process, we provide an example of creating a workflow for mailx, the POSIX mail utility. The first step is to create a workflow configuration file in `/etc/libreport/workflows.d/` with the following content:

```
EVENT=workflow_mailx analyzer=CCpp
```

This instructs the reporter to look for a `workflow_mailx.xml` file in `/usr/share/libreport/workflow/` whenever the analyzer is CCpp (the crash analyzer for C and C++).

In the second step, we create the required workflow file, `workflow_mailx.xml`, with the following content:

```
<?xml version="1.0" encoding="UTF-8"?>
<workflow>
    <name>Send the problem data via mailx</name>
    <description>Analyze the problem locally and send information via mailx</
↪description>
```

(continues on next page)

(continued from previous page)

```
    <priority>-99</priority>

    <events>
        <event>report_Mailx</event>
    </events>
</workflow>
```

It instructs the reporter (report-gtk or report-cli) to run the event `report_Mailx` as the only step of this workflow.

The third step is to create the EVENT configuration file `report_Mailx.xml` which corresponds to the `report_Mailx` EVENT from the `workflow_mailx.xml` configuration file described above. The content of this file may be as follows:

```
<?xml version="1.0" encoding="UTF-8" ?>
<event>
    <name>Mailx</name>
    <description>Send via email</description>

    <requires-items></requires-items>
    <exclude-items-by-default>count,event_log,reported_to,coredump,vmcore</exclude-
↪items-by-default>
    <exclude-items-always></exclude-items-always>
    <exclude-binary-items>no</exclude-binary-items>
    <include-items-by-default></include-items-by-default>
    <minimal-rating>0</minimal-rating>
    <gui-review-elements>yes</gui-review-elements>

    <options>
        <option type="text" name="Mailx_Subject">
            <label>Subject</label>
            <allow-empty>no</allow-empty>
            <description>Message subject</description>
            <default-value>[abrt] detected a crash</default-value>
        </option>
        <option type="text" name="Mailx_EmailFrom">
            <label>Sender</label>
            <allow-empty>no</allow-empty>
            <description>Sender's email</description>
        </option>
        <option type="text" name="Mailx_EmailTo">
            <label>Recipient</label>
            <allow-empty>no</allow-empty>
            <description>Recipient's email</description>
        </option>
        <option type="bool" name="Mailx_SendBinaryData">
            <label>Send Binary Data</label>
            <description>Send binary files like coredump</description>
            <default-value>no</default-value>
        </option>
    </options>
</event>
```

2.7.10 Adjusting plugin configuration

ABRT reports problems to various destinations. Almost every reporting destination require some configuration. For instance, Bugzilla requires login and password and URL to an instance of the Bugzilla service. Some configuration details can have default values (e.g. Bugzilla's URL) but others don't have sensible defaults (e.g. login).

ABRT lets user provide configuration through text configuration files, such as `/etc/libreport/events/report_Bugzilla.conf`. All text configuration files consist of key/value pairs.

The event text configuration can be stored in one of these files:

- `/etc/libreport/events/somename.conf` - for system scope configuration
- `$XDG_CACHE_HOME/abrt/events/somename.conf` - for user scope configuration [XDG]

These files are the bare minimum necessary for running events on the problem directories. ABRT GUI and CLI tools will read configuration data from these files and pass it down to events they run.

However, in order to make GUI interface more user-friendly, additional information can be supplied in XML files in the same directory, such as `report_Bugzilla.xml`. These files can contain the following information:

- user-friendly event name and description (*Bugzilla*, *Report to Bugzilla bug tracker*).
- the list of items in problem directory which are required for event to succeed.
- default and mandatory selection of items to send or not send.
- whether GUI should prompt for data review.
- what configuration options exist, their type (string, boolean, etc), default value, prompt string, etc. This lets GUI to build the appropriate configuration dialogs.

ABRT's GUI saves configuration options in `gnome-keyring` or `ksecrets` and passes them down to events, overriding data from text configuration files.

You can obtain a set of keys for a particular event by executing of the following command:

```
xmllint --xpath "/event/options/option/@name" $EVENT_XML_FILE | sed 's/name="\([^ ]*\)
↪"/\1\n/g'
```

The mapping between event XML definition files and event configuration files:

Event name	Definition file	Configuration file
Bugzilla	report_Bugzilla.xml	report_Bugzilla.conf
Logger	report_Logger.xml	report_Logger.conf
Analyze C/C++ Crash	analyze_CCpp.xml	analyze_CCpp.conf
Local GNU Debugger	analyze_LocalGDB.xml	analyze_LocalGDB.conf
Retrace Server	analyze_RetraceServer.xml	analyze_RetraceServer.conf
Analyze VM core	analyze_VMcore.xml	analyze_VMcore.conf
Collect GConf configuration	collect_GConf.xml	collect_GConf.conf
Collect system-wide vim configuration files	collect_vimrc_system.xml	collect_vimrc_system.conf
Collect your vim configuration files	collect_vimrc_user.xml	collect_vimrc_user.conf
Collect .xsession-errors	collect_xsession_errors.xml	collect_xsession_errors.conf
Post report	post_report.xml	post_report.conf
Kerneloops.org	report_Kerneloops.xml	report_Kerneloops.conf
Mailx	report_Mailx.xml	report_Mailx.conf
Report uploader	report_Uploader.xml	report_Uploader.conf
uReport	report_uReport.xml	report_uReport.conf

By default the ABRT complains about missing configuration if any of mandatory options is not configured. Mandatory option is option not marked as 'Allow empty'. Run the following command to obtain the list of mandatory options:

```
xmllint --xpath "/event/options/option[allow-empty!='yes']/@name" $EVENT_XML_FILE \
        | sed 's/name="\([^ ]*\)"/\1\n/g'
```

2.8 Documentation for package maintainers

2.8.1 Collecting more data for your package

If a program developer (or package maintainer) requires some specific information which ABRT is not collecting, they can write a custom ABRT hook which collects the required data for his program (package). Such hook can be run at a different time during the problem processing depending on how "fresh" the information has to be. It can be run:

1. at the time of the crash
2. at the time when user decides to analyse the problem (usually run `gdb` on it)
3. at the time of reporting

All you have to do is create a `.conf` and place it to `/etc/libreport/events.d/` from this template:

```
EVENT=<EVENT_TYPE> [CONDITIONS]
   <whatever command you like>
```

The commands will execute with the current directory set to the problem directory (e.g: `/var/spool/abrt/ccpp-2012-05-17-14:55:15-31664`)

If you need to collect the data at the time of the crash you need to create a hook that will be run as a `post-create` event.

WARNING: post-create events are run with root privileges!

This sample hook will save the time to the file `dateofcrash` at the time when the crash is detected by ABRT:

```
EVENT=post-create component=binutils
        date > dateofcrash
```

A more realistic example: save the `smart` data when one of tools from `udisks` crashes:

```
EVENT=post-create component=udisks
        which skdump >/dev/null 2>&1 || exit 0
        for f in /dev/[sh]d[a-z]; do
            test -e "$f" || continue
            skdump "$f"
            echo
        done >smart_data
```

If you want to collect the data at the time when the problem is being reported, then you need to use a `collect_` event type. Example:

```
EVENT=collect_syslog
        executable=`cat executable` &&
        base_executable=${executable##*/} &&
        grep -F -e "$base_executable" /var/log/messages | tail -999 >syslog
```

2.8.2 Testing how ABRT handles crashes of your application

To make sure abrt is able to handle the crash of your application and generate backtraces correctly, you can try to crash it by sending it a *SIGSEGV* signal (only in case of C/C++ application):

```
kill -SIGSEGV $( pidof yourapp )
```

Abrt should catch the crash and produce a problem directory correctly. It should be possible to report the crash to ABRT Analytics and bugzilla.

2.8.3 Using custom bugzilla template

ABRT allows package maintainers to override default Bugzilla bug formatting for their packages by providing a component specific template file for abrt. The file must be structured according to *Bugzilla plugin output configuration* and must be stored in

- `/etc/libreport/plugins/bugzilla_format_$component.conf` for initial bug formatting and
- `/etc/libreport/plugins/bugzilla_formatdup_$component.conf` for additional bug comments

Template files references ABRT problem elements listed on *Elements collected by ABRT* page.

Bugzilla plugin output configuration

Bugzilla reporter plugin accepts the `-F` option (default: `/etc/libreport/plugins/bugzilla_format.conf`), which allows user to modify the contents of the newly created bugs. Lines in this file have the following format:

```
# comment
%summary:: summary format
section:: element1[,element2]...
the literal text line to be added to Bugzilla comment. Can be empty.
```

Summary format is a line of text, where `%element%` is replaced by text element's content, and `[[...%element%...]]` block is used only if `%element%` exists. `[[...]]` blocks can nest.

Sections can be:

- `%attach::` a list of elements to attach.
- text, double colon (`::`) and a list of comma-separated elements.

Elements can be:

- problem directory element names, which get formatted as

  ```
  <element_name>: <contents>
  ```

  or

  ```
  <element_name>:
  :<contents>
  :<contents>
  :<contents>
  ```

- problem directory element names prefixed by `%bare_`, which is formatted as-is, without `<element_name>:` and starting colons
- `%oneline`, `%multiline`, `%text` wildcards, which select all corresponding elements for output or attachment
- `%binary` wildcard, valid only for `%attach` section, instructs to attach binary elements
- problem directory element names prefixed by "-", which excludes given element from all wildcards

Nonexistent elements are silently ignored. If none of elements exists, the section will not be created.

Example:

```
%summary:: [abrt] %package%[[: %reason%]]

This bug was automatically created by ABRT.

Description of problem:: %bare_comment

Version-Release number of selected component:: %bare_package
Additional info:: \
 -analyzer,-count,-duphash,-uuid,-abrt_version,\
 -username,-hostname,\
 %oneline

Bogosection:: nonexistent_element_name

Backtrace:: %bare_backtrace

%attach:: -comment,-reason,-reported_to,-event_log,%multiline,coredump
```

Note that empty lines are significant. In the above example, there is no empty line between *Version-Release number of selected component* and *Additional info* sections, which will result in these two sections having no empty line between them in newly created bug's description too.

Default configuration files

- Initial bug formatting: https://github.com/abrt/libreport/blob/master/src/plugins/bugzilla_format.conf
- Additional bug comments formatting: https://github.com/abrt/libreport/blob/master/src/plugins/bugzilla_formatdup.conf

2.9 Documentation for developers

2.9.1 Contributing

Send us patches to our mailing list at http://lists.fedorahosted.org/mailman/listinfo/crash-catcher

or create a pull request for the corresponding repository in the abrt GitHub organization.

Where appropriate, your submission should come with a test — either unit test or as a part of our integration test suite. See *Writing new integration test* for more details.

2.9.2 Nightly builds

Nightly builds and repositories for Fedora and RHEL are avaialable at https://copr.fedorainfracloud.org/groups/g/abrt/coprs/

2.9.3 Ignoring common functions on the stack

To improve clustering of similar crashes done by *ABRT Analytics*, backtraces are first normalized to skip common functions like `_start` from glibc or `__kernel_vsyscall` from Linux kernel.

Such functions are listed in satyr/lib/normalize.c.

2.9.4 Writing man pages

Abrt man pages are written in AsciiDoc, a minimal text markup language. The `asciidoctor` tool generates the formatted manpage files. Man pages in AsciiDoc format are stored in each project's `doc/` directory. For example, man pages for `libreport` are placed in libreport/doc/.

See the Asciidoctor documentation for more information on writing man pages in AsciiDoc. The AsciiDoc Writer's Guide is a good resource for learning AsciiDoc syntax, although not all features are supported by the man page generator.

The existing files in abrt/doc/ can be used as templates for creating new man pages.

2.10 Documentation for administrators

2.10.1 Configuring Centralized Crash Collection

In some scenarios, it's not desired to process crashes on affected machines. In this case, ABRT offers upload functionality that can upload problems caught by ABRT to remote machine via different protocols like scp, http or ftp.

Server side

To allow clients to upload problem directories to your server via scp follow these steps:

1. Install `abrt-addon-upload-watch`:

```
yum install abrt-addon-upload-watch
```

2. Create `abrt-upload` user:

```
useradd abrt-upload
```

3. Allow `abrt-upload` user write access to `/var/spool/abrt-upload`:

```
setfacl -m u:abrt-upload:-wx /var/spool/abrt-upload
```

4. Set password for `abrt-upload` user or generate a pair of keys to be used instead of a password. To generate authentication keys run:

```
su - abrt-upload # become abrt-upload user
ssh-keygen -f ~/.ssh/id_dsa -N '' # create new keypair with empty passphrase
ln -s ~/.ssh/id_dsa.pub ~/.ssh/authorized_keys # allow loging in with newly␣
↪created keys
```

4. In `/etc/abrt/abrt.conf` uncomment `WatchCrashdumpArchiveDir = /var/spool/abrt-upload` option.

5. Enable `abrt-upload-watch.service`:

```
systemctl start abrt-upload-watch.service
systemctl enable abrt-upload-watch.service
```

Server is now ready to accept reports from clients.

Client side

On the client this functionality is provided by `libreport-plugin-reportuploader` package which contains `reporter-upload` executable.

Complete the following steps on every client system which will use centralized crash reporting:

1. Install `libreport-plugin-reportuploader`:

   ```
   yum install libreport-plugin-reportuploader
   ```

2. Modify the `/etc/libreport/plugins/upload.conf` configuration file so that the `reporter-upload` plugin knows where to copy the saved crash reports in the following way:

   ```
   URL = scp://USERNAME:PASSWORD@SERVERNAME/var/spool/abrt-upload/
   ```

   If you've chosen to use authentication keys instead of passwords, copy `id_dsa` file created in previous step to `/root/id_dsa`. In this case your URL will look like this:

   ```
   URL = scp://USERNAME@SERVERNAME/var/spool/abrt-upload/
   ```

   If you're using passwords, make sure that `/etc/libreport/plugins/upload.conf` is only readable by root:

   ```
   chmod 600 /etc/libreport/plugins/upload.conf
   ```

3. Allow automatic uploads:

   ```
   echo 'EVENT=notify reporter-upload' >> /etc/libreport/events.d/uploader_event.conf
   ```

4. Observe logs on both machines and try to crash something. You can either use `will_segfault` from `will_crash` package or just crash the `sleep` binary with the following commands:

   ```
   sleep 100 &
   kill SIGSEGV %1
   ```

   If everything is configured correctly, problem directory should be transfered to server machine and the server should process it.

Troubleshooting

If uploading doesn't work make sure it's possible to upload problem directories manually by running:

```
reporter-upload -vvv -d /var/spool/abrt/<PROBDIR>
```

2.10.2 Spacewalk

Spacewalk is able to collect and report crash statistic from clients using abrt. More information and configuration can be found on spacewalk wiki.

2.11 Design

2.11.1 Daemon

`abrtd` is a daemon that watches for application crashes. When a crash occurs, it collects the problem data (core file, application's command line, . . .) and takes action according to the configuration and the type of application that crashed.

By default it uses inotify interface[3] to monitor the dump location (`/var/spool/abrt/`) for new directories created by C/C++ hook and a *Socket API* (`/var/run/abrt/abrt.socket`) used by other hooks like *Python hook*.

The reason for using socket instead of direct filesystem access is security. When a Python script throws unhandled exception, *Python hook* catches it, running as a part of the broken Python application. The application is running with certain SELinux privileges, for example it can not execute other programs, or to create files in `/var/spool/abrt` or anything else required to properly fill a problem directory. Adding these privileges to every application would weaken the security. The most suitable solution for the Python application is to open a socket where `abrtd` is listening, write all relevant data to that socket, and close it. `abrtd` handles the rest of the processes.

2.11.2 C/C++ hook

When C/C++ application crashes kernel uses *core_pattern* to handle the crash. Abrt overrides default core_pattern with a pipe to `abrt-hook-ccpp` executable that stores core dump in abrt's dump location and notifies daemon about new crash. It also stores number of files from `/proc/<PID>/` that might be useful for debugging — `maps`, `limits`, `cgroup`, `status`. Format and meaning of these files is described in the documentation of the Linux kernel[1].

To enable C/C++ hook use:

```
systemctl enable --now abrt-ccpp
```

core_pattern

Variable used to specify a core dump file name template. If the first character of the pattern is `|`, the kernel will treat the rest of the pattern as a command to run. The core dump will be written to the standard input of that program instead of to a file.

By default, `/proc/sys/kernel/core_pattern` contains `core` string and kernel produces `core.*` files in crashed process' current directory.

Abrt's C/C++ hook overrides this with:

```
|/usr/libexec/abrt-hook-ccpp %s %c %p %u %g %t e
```

which results in kernel calling `abrt-hook-ccpp`. Detailed description can be found in the documentation of the Linux kernel[2].

debuginfo

To be able to get full featured GDB backtrace from a core dump file, debuginfo data must be available on the local file system. These data are usually provided in the form of installable packages, how-

[3] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/plain/Documentation/filesystems/inotify.txt
[1] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/plain/Documentation/filesystems/proc.txt
[2] https://www.kernel.org/doc/html/latest/admin-guide/sysctl/kernel.html#core-pattern

ever, ABRT needs to allow non-privileged users to analyze the core dump file and report the obtained backtrace to bug tracking tool. Hence, ABRT maintains its own debuginfo directory `/var/cache/abrt-di` where all users can download and unpack the required debuginfo packages through `/usr/libexec/abrt-action-install-debuginfo-to-abrt-cache` command line utility.

Upon a new core dump file detection ABRT generates a list of build-ids (`XXYYY..YYY`) using `eu-unstrip -n --core=coredump`. When a user decides to report the core dump file, the ABRT debuginfo tool goes through that list and remembers those build-ids for which the file `XX/YYY..YYY.debug` exists either in the system directories (`/usr/lib/debug/.build-id` or `/usr/lib/.build-id`) or in the ABRT debuginfo directory. Finally, packages that provide the debug files are looked up in `*debug*` repositories, downloaded and unpacked to the ABRT debuginfo directory.

2.11.3 Python hook

The `python3-abrt-addon` package provides an exception handler for Python 3 applications.

The Python interpreter automatically imports the `abrt.pth` file installed in `/usr/lib64/python3.7/site-packages/`. This file in turn imports `abrt_exception_handler.py` which overrides Python's default `sys.excepthook` with a custom handler that forwards unhandled exceptions to `abrtd` via its *Socket API*.

Automatic import of site-specific modules can be disabled by passing the `-S` option to the Python interpreter:

```
python -S file.py
```

2.11.4 Oops watcher

Kernel oopses are detected by watcher `abrt-dump-journal-oops`, typicaly this process runs as a daemon and watches systemd-journal. When kernel oops logs appears, watcher extracts them and creates problem dir, which is further processed by post-create event handler for type Kerneloops.

2.11.5 Xorg watcher

Xorg crashes are detected by watcher `abrt-dump-journal-xorg`. Mechanism is same as in oops watcher, systemd-journal is watched and Xorg crashes are extracted in time of their occurence. In addition xorg watcher can be configured to search for next Xorg crashes, config file is located in `/etc/abrt/plugins/xorg.conf`.

2.11.6 Events

A problem life cycle is driven by events in ABRT. For example:

- *Event 1* — a problem data directory is created.
- *Event 2* — problem data is analyzed.
- *Event 3* — a problem is reported to Bugzilla.

When a problem is detected and its defining data is stored, the problem is processed by running events on the problem's data directory. For event configuration how-to, refer to *Event configuration*.

Standard ABRT installation currently supports several default events that can be selected and used during problem reporting process. Refer to *Standard ABRT Installation Supported Events* to see the list of these events.

Only following three events are run automatically by ABRT:

`post-create` runs after the problem directory creation

notify runs after the processing chain is finished to notify user about new problem

notify-dup similar to `notify` for duplicate problems. See *Deduplication*.

2.11.7 Deduplication

When ABRT catches new crash it compares it to the rest of the stored problems to avoid storing duplicate crashes.

It first checks if there is `core_bactrace` or `uuid` item in the problem directory we are processing.

If there is a `core_backtrace`, it iterates over all other dump directories and computes similarity to their core backtraces (if any). If one of them is similar enough to be considered duplicate, event processing is stopped and only `notify-dup` event is fired.

If there is an `uuid` item (and no core backtrace), simple comparison of `uuid` hashes is used for duplicate detection.

2.11.8 Elements collected by ABRT

Commonly available elements:

Property	Meaning	Example
`executable`	Executable path of the component which caused the problem. Used by the server to determine `component` and `package` data.	`'/usr/bin/time'`
`type`	Problem typem, see *Supported problem types*.	`'Python'`
`component`	Component which caused this problem.	`'time'`
`hostname`	Hostname of the affected machine.	`'fiasco'`
`os_release`	Operating system release string.	`'Fedora release 17 (Beefy Miracle)'`
`uid`	User ID	`1000`
`username`	User name	`'jeff'`
`architecture`	Machine architecture string	`'x86_64'`
`kernel`	Kernel version string	`'3.6.6-1.fc17.x86_64'`
`package`	Package string	`'time-1.7-40.fc17.x86_64'`
`time`	Time of the occurrence (unixtime)	`datetime.datetime(2012, 12, 2, 16, 18, 41)`
`count`	Number of times this problem occurred	`1`
`pkg_name`	Package name	`'time'`
`pkg_epoch`	Package epoch	`0`
`pkg_version`	Package version	`'1.7'`
`pkg_release`	Package release	`'40.fc17'`
`pkg_arch`	Package architecture	`'x86_64'`
`uuid`	Unique problem identifier computed as a hash of the first three frames of the backtrace	`'c55e3deb95d46553fdbefb1bc1d020e89a`

Elements dependent on problem type:

Property	Meaning	Example	Applicable
`abrt_version`	ABRT version string	`'2.0.18.84.g211c'`	Crashes caught by ABRT
`cgroup`	cgroup (control group) information for crashed process	`'9:perf_event:/\n8:blkio:/\n...'`	C/C++
`core_backtrace`	Machine readable backtrace with no private data		C/C++, Python, Ruby, Kerneloops
`backtrace`	Original backtrace or backtrace produced by retracing process		C/C++ (after retracing), Python, Ruby, Xorg, Kerneloops
`dso_list`	List of dynamic libraries loaded at the time of crash		C/C++, Python
`exploitable`	Likely crash reason and exploitable rating		C/C++
`maps`	Copy of `/proc/<pid>/maps` file of the problem executable		C/C++
`cmdline`	Copy of `/proc/<pid>/cmdline` file	`'/usr/bin/gtk-builder-convert'`	C/C++, Python, Ruby, Kerneloops
`coredump`	Core dump of the crashing process		C/C++
`environ`	Runtime environment of the process		C/C++, Python
`open_fds`	List of file descriptors open at the time of crash		C/C++
`pid`	Process ID	`'42'`	C/C++, Python, Ruby
`proc_pid_status`	Copy of `/proc/<pid>/status` file		C/C++
`limits`	Copy of `/proc/<pid>/limits` file		C/C++
`var_log_messages`	Part of the `/var/log/messages` file which contains crash information		C/C++
`suspend_stats`	Copy of `/sys/kernel/debug/suspend_stats`		Kerneloops
`reported_to`	If the problem was already reported, this item contains URLs of the services where it was reported		Reported problems
`event_log`	ABRT event log		Reported problems
`dmesg`	Copy of `dmesg`		Kerneloops

Supported problem types

Supported values for `type` element:

- `CCpp`
- `java`
- `Kerneloops`
- `selinux`
- `Python`
- `Python3`
- `Ruby`
- `xorg`

2.12 μReport

μReport (*microreport*) is a JSON object representing a problem: binary crash, kerneloops, SELinux AVC denial... These reports are designed to be small, machine readable and completely anonymous which allows us to use them for automated reporting. These reports allow us to keep track of bug occurrences while not necessarily providing enough information for our engineers to fix the bug. For that, you still may have to send a full bug report.

2.12.1 What exactly does a μReport contain?

Each μReport generally contains a stack trace, or multiple stack traces in the case of multi-threaded C/C++ and Java programs. The stack trace only describes the call stack of the program at the time of the crash and does not contain contents of any variables.

Every μReport also contains identification of the operating system, versions of the RPM packages involved in the crash, and whether the program ran under a root user.

There are also items specific to each crash type: - for C/C++ crashes, these are: path to the executable and signal delivered to the program - for Python exceptions, there is the type of the exception (without the error message, which may contain sensitive data) - for kernel oopses, these are: list of loaded kernel modules, list of taint flags, and full text of the kernel oops

2.12.2 Anonymity

μReport **MUST NOT** contain any private data. μReports are considered public and may be accessed through the web UI or web API, forwarded to other instances or archived. Clients need to think about the data they are sending. These may contain passwords, credit card numbers, private keys etc.

Example: Why not to include the message from python exception? Think about the following:

```
ValueError: invalid literal for int() with base 10: 'my_secret_password'
```

2.12.3 Specification

The object contains a numeric field `ureport_version`, which defines the expected contents. If the field is missing, the version is considered 0.

μReport0 and former

First μReports were not versioned at all because of quick evolution. μReport0 is the codename for the last of these (no known client has ever sent *ureport_version = 0*). All former μReports will fail parsing. Although μReport0 is still supported (it can be turned into μReport1), it is obsolete and should not be used in client applications.

μReport1

μReport1 format was designed for the needs of ABRT (at that moment processing C/C++ crashes, unhandled Python exceptions and kerneloopses). The background idea is that all the reports contain a stacktrace and some metadata about the environment (OS version, CPU architecture, kernel version...). Only supported fields are allowed. Some of them are mandatory, some are not. Any μReport containing excessive fields or missing mandatory fields will fail parsing.

Supported fields

- `ureport_version` *(int)* MUST be set to `1` in order to be parsed and processed by μReport1 workflow
- `type` *(string)* either of `userspace` `python` `kerneloops`
- `reason` *(string)* a short message describing what happend
- `uptime` *(int)* uptime of the machine
- `component` *(string)* affected component
- `executable` *(string)* affected executable
- `architecture` *(string)* CPU architecture
- `crash_thread` *(int)* ID of the crash thread
- `kernel_taint_state` *(string)* kernel tainted flags
- `proc_status` placeholder
- `proc_limits` placeholder
- `oops` *(string)* the raw kerneloops as written into syslog
- `user_type` *(string)* either of `root` `nologin` `local` `remote`
- `installed_package` *(obj)* maps to appropriate RPM attributes
  - `name` *(string)*
  - `epoch` *(int)*
  - `version` *(string)*
  - `release` *(string)*
  - `architecture` *(string)*
- `running_package` *(obj)* same as `installed_package`
- `related_packages` *(list)* list of following objects:
  - `installed_package` *(obj)* same as `installed_package` above
  - `running_package` *(obj)* same as `installed_package` above
- `os` *(obj)* operating system
  - `name` *(string)*
  - `version` *(string)*
- `reporter` *(obj)* identification of the client application
  - `name` *(string)*
  - `version` *(string)*
- `core_backtrace` *(list)* list of frames (following objects):
  - `thread` *(int)* thread number
  - `frame` *(int)* frame number
  - `path` *(string)* associated file
  - `buildid` *(string)* build-id of the file
  - `offset` *(int)* offset within the file

  - `funcname` *(string)* function name
  - `funchash` *(string)* function hash

- `os`state` *(obj)*
  - `suspend` *(string)* either `yes` or `no`
  - `boot` *(string)* either `yes` or `no`
  - `login` *(string)* either `yes` or `no`
  - `logout` *(string)* either `yes` or `no`
  - `shutdown` *(string)* either `yes` or `no`

- `selinux` *(obj)*
  - `mode` *(string)* either of `enforcing` `permissive` `disabled`
  - `context` *(string)* the affected context
  - `policy`package` *(obj)* same format as `installed_package`

Example:

```
{
        "ureport_version": 1,
        "type": "python",
        "reason": "TypeError",
        "uptime": 1,
        "component": "faf",
        "executable": "/usr/bin/faf-btserver-cgi",
        "installed_package": { "name": "faf",
                               "version": "0.4",
                               "release": "1.fc16",
                               "epoch": 0,
                               "architecture": "noarch" },
        "related_packages": [ { "installed_package": { "name": "python",
                                                        "version": "2.7.2",
                                                        "release": "4.fc16",
                                                        "epoch": 0,
                                                        "architecture": "x86_64" } } ],
        "os": { "name": "Fedora", "version": "16" },
        "architecture": "x86_64",
        "reporter": { "name": "abrt", "version": "2.0.7-2.fc16" },
        "crash_thread": 0,
        "core_backtrace": [
          { "thread": 0,
            "frame": 1,
            "buildid": "f76f656ab6e1b558fc78d0496f1960071565b0aa",
            "offset": 24,
            "path": "/usr/bin/faf-btserver-cgi",
            "funcname": "<module>" },
          { "thread": 0,
            "frame": 2,
            "buildid": "b07daccd370e885bf3d459984a4af09eb889360a",
            "offset": 190,
            "path": "/usr/lib64/python2.7/re.py",
            "funcname": "compile" },
          { "thread": 0,
            "frame": 3,
```

(continues on next page)

(continued from previous page)

```
            "buildid": "b07daccd370e885bf3d459984a4af09eb889360a",
            "offset": 241,
            "path": "/usr/lib64/python2.7/re.py",
            "funcname": "_compile" }
        ],
        "user_type": "root",
        "selinux": { "mode": "permissive",
                     "context": "unconfined_u:unconfined_r:unconfined_t:s0",
                     "policy_package": { "name": "selinux-policy",
                                         "version": "3.10.0",
                                         "release": "2.fc16",
                                         "epoch": 0,
                                         "architecture": "noarch" } },
        "kernel_taint_state": "G    B       "
}
```

Problems

- Although μReport1 was designed to be independent on the operating system, it is closely related to Fedora.
- New types of problems are appearing, μReport1 is not generic enough to handle differences (SELinux AVC denial does not contain stacktrace, kerneloops does not contain executable. . .)
- All reports are handled in the same way. This either results into special-casing in code (`if type != "python"` etc.) or into lower-quality results (clustering kerneloops and C/C++ with the same parameters does not work well).

μReport2

The format of μReport2 is derived from the new pluginable model of FAF project. It only contains a common metadata and the identification of [OS plugin](https://github.com/abrt/faf/wiki/Operating-System-Plugins) and [Problem plugin](https://github.com/abrt/faf/wiki/Problem-Plugins). Only the common part is parsed by a global parser. *os* and *packages* are handled by OS plugin, *problem* is handled by Problem plugin. All excessive fields are ignored.

Supported fields

- `ureport_version` *(int)* must be set to `2` in order to be parsed and processed by μReport2 workflow
- `problem` *(obj)* problem, passed to Problem plugin
  - `type` *(string)* must match to the `name` of any installed Problem plugin
  - anything else
- `os` *(obj)* operating system - passed to OS plugin
  - `name` *(string)* must match the `name` of any installed OS plugin
  - `version` *(string)* must match to the version in storage
  - `arch` *(string)* CPU architecture
  - anything else
- `packages` *(list)* list of affected packages, the nature of "package" is defined by the OS plugin
- `reason` *(string)* a short (human-readable) message describing what happend

- `reporter` *(obj)* identification of the client application
  - `name` *(string)*
  - `version` *(string)*

C/C++ crash example:

```
{
  "ureport_version": 2,

  "reason": "Program /usr/bin/sleep was terminated by signal 11",

  "os": {
    "name": "fedora",                       # OS plugin
    "version": "18",
    "architecture": "x86_64"
  },

  "problem": {
    "type": "core",                         # problem plugin

    "executable": "/usr/bin/sleep",

    "signal": 11,

    "component": "coreutils",

    "user": {
      "local": true,
      "root": false
    },

    "stacktrace": [
      {
        "crash_thread": true,

        "frames": [
          {
            "build_id": "5f6632d75fd027f5b7b410787f3f06c6bf73eee6",
            "build_id_offset": 767024,
            "file_name": "/lib64/libc.so.6",
            "address": 251315074096,
            "fingerprint": "6c1eb9626919a2a5f6a4fc4c2edc9b21b33b7354",
            "function_name": "__nanosleep"
          },
          {
            "build_id": "cd379d3bb5d07c96d491712e41c34bcd06b2ce32",
            "build_id_offset": 16567,
            "file_name": "/usr/bin/sleep",
            "address": 4210871,
            "fingerprint": "d24433b82a2c751fc580f47154823e0bed641a54",
            "function_name": "close_stdout"
          },
          {
            "build_id": "cd379d3bb5d07c96d491712e41c34bcd06b2ce32",
            "build_id_offset": 16202,
            "file_name": "/usr/bin/sleep",
            "address": 4210506,
```

(continues on next page)

(continued from previous page)

```
            "fingerprint": "562719fb960d1c4dbf30c04b3cff37c82acc3d2d",
            "function_name": "close_stdout"
          },
          {
            "build_id": "cd379d3bb5d07c96d491712e41c34bcd06b2ce32",
            "build_id_offset": 6404,
            "fingerprint": "2e8fb95adafe21d035b9bcb9993810fecf4be657",
            "file_name": "/usr/bin/sleep",
            "address": 4200708
          },
          {
            "build_id": "5f6632d75fd027f5b7b410787f3f06c6bf73eee6",
            "build_id_offset": 137733,
            "file_name": "/lib64/libc.so.6",
            "address": 251314444805,
            "fingerprint": "075acda5d3230e115cf7c88597eaba416bdaa6bb",
            "function_name": "__libc_start_main"
          }
        ]
      }
    ]
  },

  "packages": [
    {
      "name": "coreutils",
      "epoch": 0,
      "version": "8.17",
      "architecture": "x86_64",
      "package_role": "affected",
      "release": "8.fc18",
      "install_time": 1371464601
    },
    {
      "name": "glibc",
      "epoch": 0,
      "version": "2.16",
      "architecture": "x86_64",
      "release": "31.fc18",
      "install_time": 1371464176
    },
    {
      "name": "glibc-common",
      "epoch": 0,
      "version": "2.16",
      "architecture": "x86_64",
      "release": "31.fc18",
      "install_time": 1371464184
    }
  ],

  "reporter": {
    "version": "0.3",
    "name": "satyr"
  }
}
```

Python exception example:

```
{
  "ureport_version": 2,

  "reason": "ImportError in /usr/bin/faf:55",

  "os": {
    "name": "fedora",                          # OS plugin
    "version": "18",
    "architecture": "x86_64"
  },

  "problem": {
    "type": "python",                          # problem plugin

    "component": "faf",

    "exception_name": "ImportError",

    "user": {
      "local": true,
      "root": false
    },

    "stacktrace": [
      {
        "file_name": "/usr/lib64/python2.7/site-packages/sqlalchemy/dialects/
↪postgresql/psycopg2.py",
        "file_line": 312,
        "line_contents": "psycopg = __import__('psycopg2')",
        "function_name": "dbapi"
      },
      {
        "file_name": "/usr/lib64/python2.7/site-packages/sqlalchemy/engine/strategies.
↪py",
        "file_line": 64,
        "line_contents": "dbapi = dialect_cls.dbapi(**dbapi_args)",
        "function_name": "create"
      },
      {
        "file_name": "/usr/lib64/python2.7/site-packages/sqlalchemy/engine/__init__.py
↪",
        "file_line": 338,
        "line_contents": "return strategy.create(*args, **kwargs)",
        "function_name": "create_engine"
      },
      {
        "file_name": "/usr/lib/python2.7/site-packages/pyfaf/storage/__init__.py",
        "file_line": 213,
        "line_contents": "self._db = create_engine(config['storage.connectstring'])",
        "function_name": "__init__"
      },
      {
        "file_name": "/usr/lib/python2.7/site-packages/pyfaf/storage/__init__.py",
        "file_line": 199,
        "line_contents": "db = Database(debug=debug, dry=dry)",
        "function_name": "getDatabase"
```

(continues on next page)

(continued from previous page)

```
      },
      {
        "file_name": "/usr/bin/faf",
        "file_line": 29,
        "line_contents": "db = getDatabase(debug=cmdline.sql_verbose, dry=cmdline.dry_
↪run)",
        "function_name": "main"
      },
      {
        "file_name": "/usr/bin/faf",
        "file_line": 55,
        "special_function": "module",
        "line_contents": "main()"
      }
    ]
  },

  "packages": [
    {
      "name": "faf",
      "epoch": 0,
      "version": "0.9",
      "architecture": "noarch",
      "package_role": "affected",
      "release": "1.fc18"
    }
  ],

  "reporter": {
    "version": "0.3",
    "name": "satyr"
  }
}
```

μReport attachment

Currently used to attach Bugzilla ticket number to previously reported μReport identified by `bthash`:

```
{
  "type": "RHBZ",
  "bthash": "5f6632d75fd027f5b7b410787f3f06c6bf73eee6",
  "data": "123456"
}
```

2.13 Debugging crashes reported by abrt

2.13.1 Signals

- Signal 5 (*SIGTRAP*) = “General crash”
- Signal 6 (*SIGABRT*) = SIGABRT is commonly used by `libc` and other libraries to abort the program in case of critical errors. For example, `glibc` sends an SIGABRT in case of a detected double-free or other heap corruptions.

- Signal 11 (*SIGSEGV*) = Segmentation fault, bus error, or access violation. It is generally an attempt to access memory that the CPU cannot physically address.

2.14 Integration test suite

At the time of writing this document, we have 60 integration tests which tests abrt, libreport and satyr functionality in real world scenarios.

These tests are run nightly for each of the currently supported releases of Fedora and Red Hat Enterprise Linux. You can find the results on our public mirror.

2.14.1 Running the test suite

Caution: Don't run the test suite on your machine **directly**, use virtual machine or machine dedicated for the testing.

In your virtual machine or dedicated machine:

- clone the ABRT repository: `git clone git://git.fedorahosted.org/abrt.git`
- go to `abrt/tests/runtest/`
- run `./run`

This will run all the tests in order specified in `./aux/test_order`. When the testing is done, the output can be found in `/tmp/abrt-testsuite/`.

To override the destination directory or the other options, you can edit `./aux/config.sh`.

Running single test

To run only one of the tests, you have to run it via `./aux/runner.sh`. This script prepares your system by restarting required services and cleaning up possible stale files.

For example, to run `dbus-api` test, you have to pass the whole path to `runtest.sh` to `runner.sh` script:

```
./aux/runner.sh dbus-api/runtest.sh
```

Running a subset of the tests

To run only part of the tests, edit `./aux/test_order` file and comment out the tests you don't want to run.

For example, when using the test suite to test already installed packages, you don't need to build and install git versions. In this case, you can comment out `abrt-nightly-build` test and all the `\*-make-check` tests. These are now disabled by default.

2.14.2 Writing new integration test

To create a new test you have to:

- create a directory reflecting its name

- create two files in that directory:
  - executable `runtest.sh` file
  - `PURPOSE` file
- add the test to `./aux/test_order`

The tests are written in bash using BeakerLib library. BeakerLib Quick Reference [PDF] is what you need.

It's always a good idea to copy one of the existing tests as a base for your test. Good candidates are:

- `run-abrtd` which basically serves as an example,
- `ccpp-plugin` if you need to test crash handling (if you require crash directory),
- `dbus-api` if you need to test DBus functionality.

Complex tests

Please provide an explanation for complex tests. Either explain certain sections of the test in comments in `runtest.sh` or create `README` file inside the directory of the test.

2.14.3 Output directory structure

To avoid painful digging in several distinct files logging is now divided into several logical levels:

`$OUTPUT_ROOT` variable is defined in `aux/config.sh` which represents the root for all logs.

For each stage of the test suite run there is corresponding directory in root directory. It contains *stage.log* file which contains output of script governing the stage.

```
/tmp/abrt-testsuite
├── format/
├── post/
├── pre/
├── report/
├── results
└── test/
```

Stages:

- **`PRE`** install required packages, collect log files, collect system information
- **`TEST`** run all the tests
- **`FORMAT`** format the results for REPORT stage
- **`REPORT`** report the results
- **`POST`** collect logs, cleanup

For `TEST` stage there is an additional subdirectory for each test case:

```
/tmp/abrt-testsuite/test/
├── abrt-make-check
├── abrt-nightly-build
├── abrt-should-return-rating-0-on-fail
├── blacklisted-package
...
```

Each directory contains several files:

```
/tmp/abrt-testsuite/test/systemd-init/
├── dmesg
├── avc
├── fail.log
├── full.log
├── messages
└── protocol.log
```

Only `full.log` is mandatory. It contains *stdout* and *stderr* of the test run. `protocol.log` only contains the protocol generated by BeakerLib. If the test fails with FATAL error, `protocol.log` is not generated. In case of other failures, these are extracted to `fail.log` along with line numbers pointing to lines in `full.log`.

`dmesg`, `messages` and `avc` each contains log file messages written during the test run.

2.15 Frequently asked questions

2.15.1 Where does ABRT store the crashes?

ABRT stores its problem directories in the `/var/spool/abrt` directory. Note that the directory itself is readable by `root` only by default.

2.15.2 What type of crashes can ABRT handle?

See *Supported programming languages and software projects*.

2.15.3 What to do when ABRT is not able to catch the crash of my application?

- Make sure that following services are running:
  - `abrtd`
  - `abrt-ccpp`

  ```
  $ systemctl status abrtd abrt-ccpp
  ```

- If one of them is not running you can use the following command to restart both of them:

  ```
  $ sudo systemctl restart abrtd abrt-ccpp
  ```

- If the above doesn't help, consult `journactl` or `/var/log/messages` for error logs.
- By default ABRT won't handle crashes produced by 3rd party (unpackaged) software, for this to work read *How to enable handling of unpackaged software*.

2.15.4 How to enable handling of unpackaged software

- Edit `/etc/abrt/abrt-action-save-package-data.conf` and change `ProcessUnpackaged = no` to `ProcessUnpackaged = yes`

  ```
  # sed -i 's/ProcessUnpackaged = no/ProcessUnpackaged = yes/' /etc/abrt/
  abrt-action-save-package-data.conf
  ```

2.15.5 How to enable handling of non-GPG signed software

- Edit `/etc/abrt/abrt-action-save-package-data.conf` and change `OpenGPGCheck = yes` to `OpenGPGCheck = no`

  ```
  # sed -i 's/OpenGPGCheck = yes/OpenGPGCheck = no/' /etc/abrt/
  abrt-action-save-package-data.conf
  ```

2.15.6 How do I list crashes handled by ABRT?

- Use either GUI application: `$ gnome-abrt`
- or command line tool: `$ abrt-cli list`

  See *Usage* for more details.

2.15.7 What is μReport?

- μReport (*microreport*) is a JSON object representing a problem: binary crash, kerneloops, SELinux AVC denial, etc. These reports are designed to be small and completely anonymous which allows us to use them for automated reporting.

  See *μReport* page for more details.

2.15.8 What is tainted kernel and why is my kernel tainted?

The Linux kernel maintains a *taint state* which indicates whether something happened to the running kernel that might caused a kernel error.

Common reasons include:

- proprietary kernel module was loaded (P flag)
- previous kernel error (kerneloops) occurred (D flag).
- previous kernel warning (`GW` flags).
- Both cases `D` flag and `GW` flags mean that kernel data structures may be corrupted. Therefore the current error is not necessary a real error, it could be a random consequence of the previous error.
- To get rid of the tainted kernel, you need to reboot your machine or stop loading proprietary modules.
- ABRT respects these flags and won't allow reporting if one or more are in effect because kernel developers are usually not able to fix issues when the kernel is tainted.

Complete list of taint flags:

```
P - Proprietary module has been loaded.
F - Module has been forcibly loaded.
S - SMP with CPUs not designed for SMP.
R - User forced a module unload.
M - System experienced a machine check exception.
B - System has hit bad_page.
U - Userspace-defined naughtiness.
D - Kernel has oopsed before
A - ACPI table overridden.
W - Taint on warning.
C - modules from drivers/staging are loaded.
```

(continues on next page)

(continued from previous page)

```
I - Working around severe firmware bug.
O - Out-of-tree module has been loaded.
```

Source: http://git.kernel.org/?p=linux/kernel/git/torvalds/linux.git;a=blob;f=kernel/panic.c

2.15.9 How do I create a private bugzilla ticket?

ABRT can create reports with restricted access which means the access to the report is limited to a group of trusted people. Please note that the restriction differs between various bug trackers and even if you mark something as restricted it still can leak to public, so if you are not sure, then don't report anything!

To create a private bugzilla ticket, you have to specify the list of groups to restrict the access to. The tricky part is that it has to be the internal id of the group from bugzilla database. To ease the pain, here is the list of the private group ids for supported bugzillas:

Bugzilla server	group name	group ID to use in the settings
http://bugzilla.redhat.com	Fedora Contrib (Bug accessible by Fedora Contrib members)	fedora_contrib_private
http://bugzilla.redhat.com	Private Group (Bug accessible only by the maintainer)	private

2.15.10 How do I enable screencasting?

To enable screencasting in abrt you have to install fros package with plugin matching your desktop environment. Currently there are only 2 plugins available: `fros-gnome` and `fros-recordmydesktop`. Gnome plugin works only with Gnome 3, `recordmydesktop` should work with the most of other desktop environments. To install the plugin run one of the following commands (depending on your desktop environment):

```
yum install fros-gnome
yum install fros-recordmydesktop
```

2.15.11 Why ABRT Analytics collects tainted kernel oopses?

ABRT Analytics collects tainted oopses because each received oops is forwarded to http://oops.kernel.org/ and kernel people want to see **every** oops and **not only untainted** ones.

2.15.12 Why is my backtrace unusable?

Unusable backtrace is usually caused by damaged core dump, missing debug information or usage of unsupported coding technique (i.e. JavaScript in GNOME3).

These cause that the generated backtrace has low information value for developers because function names are replaced with `'??'` string which is place holder for unavailable function name. In order to provide valuable crash reports, ABRT will not let you create a Bugzilla bug for such a backtrace.

You can use ABRT to send the unusable backtrace to maintainers via `Email reporter`, but this is on your own responsibility.

2.15.13 How to enable dumping of setuid binaries

By default kernel won't dump set-user-ID or otherwise protected/tainted binaries. To change this behavior you need to change `fs.suid_dumpable` kernel variable.

To read the value use:

```
sysctl fs.suid_dumpable
```

To change the value use:

```
sysctl fs.suid_dumpable=0
```

Possible values are:

0. (*default*) — traditional behaviour. Any process which has changed privilege levels or is execute only will not be dumped.
1. (*debug*) — all processes dump core when possible. The core dump is owned by the current user and no security is applied. This is intended for system debugging situations only. Ptrace is unchecked. This is insecure as it allows regular users to examine the memory contents of privileged processes.
2. (*suidsafe*) — Any binary which normally would not be dumped (see "0" above) is dumped readable by root only. This allows the user to remove the core dump file but not to read it. For security reasons core dumps in this mode will not overwrite one another or other files. This mode is appropriate when administrators are attempting to debug problems in a normal environment.

   Additionally, since Linux 3.6, /proc/sys/kernel/core_pattern must either be an absolute pathname or a pipe command, as detailed in core(5). Warnings will be written to the kernel log if core_pattern does not follow these rules, and no core dump will be produced.

Source: http://man7.org/linux/man-pages/man5/proc.5.html

2.16 Examples

2.16.1 Automatic crash reporting through e-mails

This example deals with configuration of libreport for sending full crash reports including *backtrace* via an e-mail service from an autonomous system. We usually encounter such systems in automatic testing environments. Let's configure ABRT to send notification e-mails to *devel@lists.project.org*.

libreport's default configuration already supports sending of notification e-mails to root user on a local system. This functionality is stored in `/etc/libreport/events.d/mailx_event.conf` file that contains the following definition of `notify` and `notify-dup` events:

```
EVENT=notify
    # do not rely on the default config nor on the config file
    Mailx_Subject="[abrt] a crash has been detected" \
    Mailx_EmailFrom="ABRT Daemon <DoNotReply>" \
    Mailx_EmailTo="root@localhost" \
    reporter-mailx --notify-only

EVENT=notify-dup
    # do not rely on the default config nor on the config file
    Mailx_Subject="[abrt] a crash has been detected again" \
    Mailx_EmailFrom="ABRT Daemon <DoNotReply>" \
```

(continues on next page)

(continued from previous page)

```
    Mailx_EmailTo="root@localhost" \
    reporter-mailx --notify-only
```

Note: The subject, sender and recipient are passed through the environment variables because *reporter-mailx* does not support these options in its command line arguments and putting it to its configuration file would break the general reporting workflow.

Detected problem types can be divided into two groups based on availability of *backtrace* at the time of detection. The first group consists of Python, Kernel oops, Ruby and Java problem types where *backtrace* is available at the time of detection. The second group consists of CCpp (process crashes, C/Cpp) where *backtrace* must be extracted from a dump file.

The default configuration of `notify` event is almost usable for the problem types having backtrace available at the time of detection. We can reuse the default configuration with some tweaks. We need to update recipient's address and remove `--notify-only` string from *reporter-mailx*'s command line arguments as it forces *reporters-mailx* to compose the message from a minimal subset of problem data in order to not waste too much disk space on a local system where all the excluded data are easily accessible anyway. We also need to configure libreport to not use it for CCpp problems because we will provide its own event definition.

The default `notify-dup` event is almost usable for all problem types because it is run for consecutive occurrences of a single problem (`notify` event is run for the first occurrence and should create all the required data). We only need to update recipient's address because `--notify-only` argument make sense for `notify-dup` event.

Since we are going to use *repoter-mailx* in several places and we do not want to repeat ourself, we first move the static e-mail preferences (sender and recipient) to `/etc/libreport/plugins/mailx.conf`:

```
Subject = [abrt] a crash has been detected
EmailFrom = ABRT Daemon <DoNotReply>
EmailTo = devel@lists.project.org
```

And the updated default configuration lines follow:

```
EVENT=notify analyzer!=CCpp
    # full notify e-mail
    Mailx_Subject="[abrt] a crash has been detected" \
    reporter-mailx

EVENT=notify-dup
    # full notify-dup e-mail
    Mailx_Subject="[abrt] a crash has been detected again" \
    reporter-mailx --notify-only
```

`"analyzer!=CCpp"` tells libreport to run the lines above only if contents of *analyzer* file does not equal to `"CCpp"` string.

We are done with the default configuration and now we are going to define `notify` event for CCpp. To be able to generate full GDB backtrace, we need all relevant debug data. On Fedora like systems, ABRT can provide the required debug data by extracting build ids from *coredump* file and unpacking rpm packages providing paths made from the extracted build ids.

CCpp *notify* event lines follow:

```
EVENT=notify analyzer=CCpp
    # CCpp notify event sending e-mail with full gdb backtrace
    # extract build ids from coredump file and save them in 'build_ids' file
```

(continues on next page)

(continued from previous page)

```
    abrt-action-analyze-core --core=coredump -o build_ids >>event_log 2>&1 &&
    # download and unpack rpm package for each id in 'build_ids' file
    abrt-action-install-debuginfo -y >>event_log 2>&1 &&
    # call gdb to generate backtrace with local variables, call arguments
    # disassembly, etc. and save it in 'backtrace' file
    abrt-action-generate-backtrace >>event_log 2>&1 &&
    # generate 'duphash', 'crash_function' and 'backtrace_rating'
    abrt-action-analyze-backtrace >>event_log 2>&1
    #
    # send e-mail
    Mailx_Subject="[abrt] a crash has been detected" \
    reporter-mailx
```

Along with other problem data, the notification e-mail will also contain *event_log* (useful for ABRT debugging), *backtrace_rating* (backtrace quality measure value based on ration of resolved and unresolved frames) and *duphash* which ABRT uses to identify a crash across all supported bug-tracking systems and all machines (e.g. ABRT puts `"abrt_hash:$(cat duphash)"` string to Whiteboard field of Bugzilla bug reports.)

Tip 1:

You can add `reporter-bugzilla -h $(cat duphash) >>bugzilla_bug_id` command to include existing Bugzilla bug report IDs for the currently processed crash. This works on Fedora only.

Tip 2:

You can make e-mail's subject more informative. The following script is not bullet proof but produces really nice e-mail subject for C/Cpp problems:

```
Mailx_Subject="[abrt] $(cat package || cat executable): $(cat crash_function && echo
↪"():") $(cat reason || (cat executable && echo " crashed"))"
```

2.16.2 Getting core files from systemd-coredumctl

By default, ABRT detects crashes of native programs (C, C++) by its core dump helper which is registered in `/proc/sys/kernel/core_pattern`. Unfortunately, there can be only one core pattern helper at the time so the ABRT core dump helper cannot coexists with *systemd-coredumctl*.

However, the ABRT core dumper helper can be turned off and the ABRT *systemd-coredumpctl* watcher can be used to make ABRT notified of crashes of native programs.

Everything you need to do is to disable *abrt-ccpp.service* which replaces the *core_pattern* configured via *sysctl* with the ABRT core pattern helper. If the service is running, the *core_pattern* should start with `|/usr/libexec/abrt-hook-ccpp`.

```
systemctl stop abrt-ccpp.service
systemctl disable abrt-ccpp.service
```

Once you stop and disable *abrt-ccpp.service*, the *core_pattern* variable should start with `|/usr/lib/systemd/systemd-coredump`. If it does not, please check if the file `/usr/lib/sysctl.d/50-coredump.conf` exist and ensure that there is no other file setting `kernel.core_pattern` (see `man 5 sysctl.d`).

The last two things you need to do is to enable and start *abrt-journal-core.service*.

```
systemctl enable abrt-journal-core.service
systemctl start abrt-journal-core.service
```

The current version of *abrt-journal-core.service* needs to make copies of data that ABRT needs to be able to open a report in a bug tracking tool and leaves the *systemd-coredumpctl* data untouched. That means that you cannot use the ABRT journal core service to clean *systemd-coredumpctl* and you will end up having two copies of core files, one in *systemd-coredumpctl*'s storage and one in a sub-directory of `/var/spool/abrt/`.

2.16.3 Collecting extra log files for crashes of a particular package

ABRT tries to provide maintainers with as much information as possible. Good source of problem details can be log files. Hence, upon detection of a crashed process, ABRT goes through the system logs and copy lines looking related to the crashed process to a file called `var_log_messages` in the problem directory. The file can be found attached to bug reports opened by ABRT/libreport.

However, application may not opt in for logging to system logs for various reasons. In such case, ABRT can be configured to copy log files to problem data directory and libreport will automatically attach them to bug reports.

Let's assume we have a package called `foo` and we want to copy log files that are created in user's `/var/run/` directory. The application runs several concurrent processes and each of the processes writes debug messages to its own log file denoted by process' PID.

To get these log files, we need to create a new libreport EVENT configuration file and instruct ABRT to run it after a crash of foo's executable appears. By default, ABRT runs `post-create`, `notify`, and `notify-dup` EVENTs upon new problem detection. We cannot use `post-create` because at that time package of crashed executable might not be known. `notify-dup` is not suitable because the event is executed for re-appearing problems, hence, the log files should be already captured. Therefore we must define new `notify` EVENT.

```
EVENT=notify pkg_name=foo
    # Copy log files of crashed process to foo.log
    cp /var/run/$(cat uid)/foo.$(cat pid).log foo.log
```

The `pkg_name=foo` string tells ABRT to run the lines above for problems in any of executable shipped by the `foo` package. You can add as many such conditions as you need. The lines below the first line are interpreted by */bin/sh* and current working directory contains all problem data captured by ABRT.

The code must be placed in a file in the `/etc/libreport/events.d/` directory. Packages often follow the `${package name}_event.conf` rule for these files.

2.16.4 Ignore particular binaries on hook level in older version of ABRT

Ignoring crashes of specific binaries on hook level was introduced in RHEL 7.3 in ABRT version 2.1.11-36, in RHEL 6.8 in version 2.0.8-37, and in Fedora in ABRT version 2.8.1. Option `IgnoredPaths` in conf file `/etc/abrt/plugins/CCpp.conf`.

Is it possible to ignore crashes on hook level even in older version of ABRT?

Yes, it is. You can write your own *core_pattern* script which filters/ignores binaries and works as a wrapper for abrt-hook-ccpp which cannot do the filtering. For more deatils, see the kernel documentation[1].

[1] https://www.kernel.org/doc/html/latest/admin-guide/sysctl/kernel.html#core-pattern

Example how to do this:

Create the *core_pattern* script (for instance `/etc/my_abrt_ccpp_hook.sh`) with following contents:

```
#!/bin/bash

# kernel.core_pattern = |/usr/libexec/abrt-hook-ccpp %s %c %p %u %g %t %e
#                                                     $1 $2 $3 $4 $5 $6 $7
# You want to filter %e - executable filename (without path prefix), so parameter $7

# Is it the particular binary you want to ignore?
if [[ $7 == "EXECUTABLE_YOU_WANT_TO_IGNORE" ]]
then
    # Do what you want
else
    # In other cases use ABRT's hook in standard way
    cat /dev/stdin |/usr/libexec/abrt-hook-ccpp $1 $2 $3 $4 $5 $6 $7
fi
```

Set the new *kernel.core_pattern* using *sysctl* (basically, change `/usr/libexec/abrt-hook-ccpp` to `/etc/my_abrt_ccpp_hook.sh`):

```
# sysctl kernel.core_pattern
kernel.core_pattern = |/usr/libexec/abrt-hook-ccpp %s %c %p %u %g %t %e
# sysctl kernel.core_pattern="|/etc/my_abrt_ccpp_hook.sh  %s %c %p %u %g %t %e"
kernel.core_pattern = |/etc/my_abrt_ccpp_hook.sh %s %c %p %u %g %t %e
```

Is it possible to ignore crashes from programs with specific UIDs?

Yes, it is. You can modify files in */etc/libreport/events.d/* and add your own snippets of bash code to them. In event *post-create* (see for example */etc/libreport/events.d/python3_event.conf* for Python 3 crashes or */etc/libreport/events.d/ccpp_event.conf* for C and C++ crashes) you have access to contents of */proc/PID/status* file of the process that crashed.

Example how to filter crashes from programs with UIDs lower than 1000 in C and C++ programs: ~~~~~~~~~~~~~~~~~~~~~~~~

Open */etc/libreport/events.d/ccpp_event.conf* and add this snippet in the *EVENT=post-create* section:

```
# Parse Uid from  proc_pid_status
uid=`grep '^Uid:' proc_pid_status | sed 's/^Uid:[[:space:]]*\([0-9]*\).*/\1/'`

if [ 1000 -lt "$uid" ]; then
    # If Uid is less then 1000 abrt will remove the problem directory
    exit 1
fi
```