

Debugging PigeonDeliver services and modules

Carlo Contavalli

ccontavalli@commedia.it

Revision History

Revision 1.0.0 2005/11/19
First document revision

Debugging a running daemon is not easy. When there are chroots involved, change of privileges or modules being loaded, things may even get harder. In this document, we will try to give some basic knowledge and useful tricks to debug PigeonDeliver services and modules.

1. Debugging modules using the Postfix SAPI

As to debug any other module, it is best to first test it alone, without other components possibly interfering with its own code.

Using the Postfix SAPI, it is thus a good idea to first test the module alone, without any other PigeonDeliver modules, by forcing Postfix to use the module directly as a service.

Along the whole document, we will be testing the mailAntivirus module, one of the standard modules provided with PigeonDeliver.

1.1. Testing the module alone

In order to test your module alone, you need to:

configure the master.cf file

Specify a line with the name of the service in the first column, and pd-postfix in the last column, like in the following:

```
mailAntivirus unix - - - - - pd-postfix -D
```

Note that the `-D` parameter after `pd-postfix` tells the postfix library to attach a debugger as soon as this process is started, while the first column tells `pd-postfix` and the postfix master they will have to deal with requests directed to the `mailAntivirus` service.

configure a debugger in the `main.cf` file

Well, whenever postfix needs to load a service with the `-D` parameters, a debugger is first loaded and attached to the process.

The problem is usually how to then attach the debugger to a console. There are many possible choices. For example, you could use a serial line or, as suggested in the postfix manual, use `X` and attach `gdb` to an `xterm`. However, I don't use `X` often, and I prefer to attach `gdb` to "screen" sessions. In order to do so, you need: `screen` installed on your system, some practice in using it (please, read the manual or some tutorial around the internet), and to specify the following line in the `main.cf`:

```
debugger_command =
  PATH=/usr/bin:/usr/X11R6/bin
  screen -d -S $process_name.$process_id
    -m gdb $daemon_directory/$process_name $process_id & sleep 5
```

configure the `main.cf` file to force delivery using the specified module

The main parameters are:

```
service = mailAntivirus
default_transport = mailAntivirus
```

The first one is a PigeonDeliver specific parameter, which forces the loading of the `mailAntivirus` module under *any* circumstance.

The second parameter tells postfix to use the `mailAntivirus` module as the default transport service.

Make also sure not to have any transport map specified, and not to have any other "dangerous" configuration. My own debugging configuration file simply looks something like:

```
service = mailAntivirus
default_transport = mailAntivirus
```

```
myhostname = localhost
mydomain = pippo.it
```

```
debugger_command =
  PATH=/usr/bin:/usr/X11R6/bin
  screen -d -S $process_name.$process_id
    -m gdb $daemon_directory/$process_name $process_id & sleep 5
```

disable any kind of external delivery

now, make sure your debugging server will not connect to the outside world, trying to deliver the junk emails you will use for debugging.

An easy way to do so is to edit the master.cf file and comment out all the lines related to "external services", like the "smtp" line and the "lmtpl" service. As an example, to avoid any kind of outgoing smtp connection, you can comment out the line:

```
#smtp unix - - - - - smtp
```

Since I often work on workstations where there is a real smtp server up and running, I usually also change the line:

```
smtp inet n - - - - smtpd
    in something like
dsmtpl inet n - - - - smtpd
```

in order to have a different service in the first column, so, my server, will not listen on port 25 (dsmtpl must be defined in /etc/services).

start to deliver something and have fun

You can use some of the mails in test/chat/ to try to deliver emails. As an example, to try to deliver a single mail, just run something like:

```
socket localhost dsmtpl < test/chat/input00
or
socat STDIO TCP4:localhost:dsmtpl < test/chat/input00
or
nc localhost dsmtpl < test/chat/input00
```

I personally got in love with socat and socket, even if I can't explain the incomprehensible (to me) popularity of nc (netcat).

1.2. Real debugging for real men

When gdb is started by the postfix API, if you used the gdb_command suggested in the previous section, you should end up with a brand new screen session for your debugger.

Just run "su" to become root, and a good and nice "screen -ls" to get the listing of all the screen sessions available. You should see something like:

```
There are screens on:
    9859.pd-postfix.9856      (Detached)
    24200.vc-3.joshua       (Attached)
2 Sockets in /var/run/screen/S-root.
```

well, the pd-postfix session above should contain our own gdb. In my case, running something like "screen -RR" or "screen -R 9859" should get me to the gdb prompt. But, hey, should already know how to use screen! Just to make sure you know at least a couple commands: "ctrl+a k" (ctrl+a followed by k, without any control key pressed), will kill your debugger, "ctrl+a d" will allow you to leave your debugger alone and get back to the shell...

Well, once you reattach to your screen session, you should be somewhere like:

```
[...]  
Reading symbols from /var/spool/postfix/lib/libnss_files.so.2...done.  
Loaded symbols for /var/spool/postfix/lib/libnss_files.so.2  
0xb7cbfb1e in waitpid () from /lib/tls/libc.so.6  
(gdb)
```

which means, somewhere in postfix or your own libc API. Well, now, however, you are in gdb, and you should figure out what to do by yourself (why did you bother using gdb in the first place??).

Usually, when using gdb, the first step is to get to the piece of code you suspect to have problems, take a deep look to what it is doing, and then correct any error you may find.

The next few sections will explain some tricks you may use to get your gdb to the correct place at the right time.

1.3. General approach for debugging

Well, as you may well already have clear in your mind, you should start debugging using something like gdb only when there are bugs you cannot find by simply reading the code and/or trying to understand what is going on.

Usually, you should already have an idea in which method of your module the error lies.

Well, if you already have an idea of where your error lies, you can skip this section and go on to the next one.

If your application is getting a signal or crashing, or something else quite easy to see from a debugger, a good approach is just to run the application (use the run command), until it either crashes, gets the signal, or a catchpoint/watchpoint you previously set is hit.

When such an event happens, you can then simply: get a copy of the mail that caused the error (use `postcat -q <nameofqueueid>`), get a copy of the user configuration, get a backtrace by typing "bt" and by first using the "share" command.

As soon as you know where the bug lies, you can easily try to figure out how that condition verified, or try to follow the code under gdb until that point.

"until that point"... until that point from where? When started, gdb and the %ip register is somewhere not well specified in my memory... how do I get to "that point"? step by step? which functions should I skip? How should I enter my module? Well, those are all good questions we will try to answer in the next section.

1.4. Useful places for breakpoints

If you have problems in your `_load` or `_init` methods, you should probably place a breakpoint in `dscm_postfix_preinit`, probably around `main.c:160`, where the module has just been loaded. Just do something like:

```
br main.c:160
```

and then "continue" until the breakpoint is hit. When you are there, use "share" to load shared symbols and either:

- "step" ("s") into the `dscm_module_load` request to see why exactly your module is not being loaded (keep under control the `fpath` and `mname` variables).
- "next" ("n") after the `dscm_module_load` function, run the "share" command (read the next sections!) and place a breakpoint on any of the methods you are interested into.

If you need to debug your `del_init`, `del_perform` or any termination method, you can just place a "breakpoint" in the `dscm_postfix_deliver` function, with something like:

```
br dscm_postfix_deliver
```

and then go on with debugging, using the "continue" command until gdb hits the breakpoint.

If you don't need to debug the first request processed by the module, you can also simply run the "continue" command and then press "ctrl+c" to get attention from gdb.

2. General notes

2.1. raise and signal 6 terminations

Whenever you see a backtrace starting with `raise`, the problem is probably either a failed assertion or a "panic" call made somewhere by the code.

In the first case, a backtrace should give you full details about the problem. In the second case you should be able to find further details in the system logs.

2.2. When step (s) doesn't work

Depending on the version of gdb, it may sometimes decide not to step into functions, even when asked to do so with the 'step' command.

Usually, this happens when there are shared libraries or modules involved (uhm .. sounds like the PigeonDeliver situation, doesn't it?). Well, in order to force gdb to step into functions you really want to debug, you can set a breakpoint on that function, using the 'br' command.

There probably are smarter ways to achieve this same result. However, using a 'br' *has always* worked (for me).

2.3. Signal handling

Let's say you are happily debugging your code, quite close to finding your own bug. Well, at a certain point a signal is received. Your gdb session immediately jumps to that signal handler.

Well, sometimes this can be really useful, some others it may get really annoying, at least for two reasons: if the signal is completely unrelated to your program, you need to concentrate on how to get back to your code and go on debugging, which can be a waste of time. Additionally, gdb reports signals that your application may already be ignoring or set to the default handler.

To get information on how gdb will handle your signals, you can use:

```
info signals
or
info handle
```

To change the default behavior for a given signal, you can, as an example, use:

```
handle SIGPIPE nostop print ignore
```

which means:

nostop

we do not want to debug the signal handler, or the code nearby. Do not give us back the prompt whenever that signal is received.

print

tell us you did get the signal.

ignore

do not tell the application you did get that signal, since the application does not care about it.

It is important to note that gdb has different defaults for different signals. So, whenever you are dealing or want to deal with signals, make sure to change those defaults to a value that may suite your needs.

2.4. Shared modules

Well, many pigeondeliver components are made as shared objects. Sometimes, gdb fails to load the symbols for shared objects.

In order to load those symbols, you can simply run the command:

```
share
```

Note that until you run the above command, all external symbols that haven't yet been loaded will be considered exactly like libraries, which means: "step" will not enter the function, "breakpoints" will be almost ignored, "list" and many other commands will not be able to show any information about the code.

If you get the error "Error while reading shared library symbols: xxxx: No such file or directory.", gdb is probably having troubles in figuring out the real path of the library, probably because of postfix chroots. Read the next section.

2.5. gdb and postfix chroots

Sometimes, even with the above command gdb will not be able to load the modules symbol table.

Before issueing that command, or in case that command fails, you may want to try to specify:

```
set solib-search-path=/var/spool/postfix/:
```

this command should instruct gdb to look for libraries (or other shared object files) into /var/spool/postfix/, even if the path of the library is absolute.

If you run the "share" command one more time, it should now succeed with all the libraries and modules.

2.6. PigeonDeliver and core dumps

A great effort has been taken into making PigeonDeliver save core dumps whenever desired and allowed.

Depending on the system being used and its own security infrastructure, and considering that PigeonDeliver may be made of suid processes and that changes of euid and egid are quite frequent, the core dump may or may not be saved in case of abnormal termination.

If you are using linux, both `pcntl` and `setrlimit` are called whenever deemed necessary to force core creation. However, in order to have a useful name for your core dump file, we strongly suggest you to add something like:

```
echo "%h.%p.%s.%e.%t.crash" > /proc/sys/kernel/core_pattern
```

to your init files, or something like:

```
kernel.core_pattern = %h.%p.%s.%e.%t.crash
```

into your `sysctl.conf` file.