

# Apache 2.0 Modules: Development and Debugging

Part 2  
Cliff Woolley

# Build Apache

- **Build Apache with your module enabled**

```
root@deephought:/root/apache/httpd-2.1# cat config.nice
#!/bin/sh
#
# Created by configure

CFLAGS="-O0 -g -ggdb3"; export CFLAGS
"./configure" \
"--enable-modules=actions alias asis authn_file authn_dbm authn_anon authn_default
authz_host authz_groupfile authz_user authz_dbm authz_owner authz_default
auth_basic auth_digest autoindex bucketeer cache case_filter case_filter_in
cern_meta cgid charset_lite dav dav_fs dav_lock deflate dir disk_cache echo env
example expires ext_filter file_cache headers http ident imap include info
log_config logio mem_cache mime mime_magic negotiation optional_fn_export
optional_fn_import optional_hook_export optional_hook_import proxy
proxy_connect proxy_ftp proxy_http rewrite setenvif spelling ssl status suexec
unique_id userdir usertrack vhost_alias" \
"--with-ssl=/usr" \
"--prefix=/home/jcw5q/apachetest" \
"--with-mpm=worker" \
"--enable-maintainer-mode" \
"$@"
```

This particular configuration enables every single module that comes with Apache by default. It's what I normally use for testing. The way I see it from a developer's standpoint, if I always enable every single module, then I'll know when one fails to compile, and I'll have a higher chance of any problems being caught by the test suite. Also keep in mind that many of these modules are solely for testing purposes and don't do anything useful for a production server. So in reality you'll just pare this list down to the actual modules you want. One you might not know about but will probably want to hang onto for testing, though, is the bucketeer module. We can use it to manipulate the data structures that flow through Apache in order to try to hit edge cases in our modules. The test suite's mod\_include tests use this extensively, for example. Anyway, this is all well and good, but if a problem does arise, you'll quickly find a problem with the above configuration when you go to try to debug your server...

# Build Apache

- **Build Apache with your module enabled... in single threaded mode**

```
root@deephought:/root/apache/httpd-2.1# cat config.nice
#!/bin/sh
#
# Created by configure

CFLAGS="-O0 -g -ggdb3"; export CFLAGS
"./configure" \
"--enable-modules=actions alias asis authn_file authn_dbm authn_anon authn_default
authz_host authz_groupfile authz_user authz_dbm authz_owner authz_default
auth_basic auth_digest autoindex bucketeer cache case_filter case_filter_in
cern_meta cgid charset_lite dav dav_fs dav_lock deflate dir disk_cache echo env
example expires ext_filter file_cache headers http ident imap include info
log_config logio mem_cache mime mime_magic negotiation optional_fn_export
optional_fn_import optional_hook_export optional_hook_import proxy
proxy_connect proxy_ftp proxy_http rewrite setenvif spelling ssl status suexec
unique_id userdir usertrack vhost_alias" \
"--with-ssl=/usr" \
"--prefix=/home/jcw5q/apachetest" \
"--with-mpm=worker" \
"--enable-maintainer-mode" \
"$@"
```

...namely that the server is VERY hard to debug when it's running in multi-threaded mode. That means you're going to have to pick a single-threaded MPM... namely prefork. When you use prefork, you can run your server in `./httpd -X` mode, which runs only one child process. Then when you attach your debugger to the child process and connect to it and make a request, you know you'll be debugging the process that handles the request, and you won't have to guess at which thread you should go look at.

One gotcha, though. `mod_cgid` and `mod_cgi` do the same thing, except that the former is for use with multi-threaded MPMs, and the latter is for use with single-threaded MPMs. So when you change from worker to prefork, don't forget to change `cgid` to `cgi`!

# Build Apache

- **Build Apache with your module enabled... single threaded!**

```
root@deephought:/root/apache/httpd-2.1# cat config.nice
#!/bin/sh
#
# Created by configure

CFLAGS="-O0 -g -ggdb3"; export CFLAGS
"./configure" \
"--enable-modules=actions alias asis authn_file authn_dbm authn_anon authn_default
authz_host authz_groupfile authz_user authz_dbm authz_owner authz_default
auth_basic auth_digest autoindex bucketeer cache case_filter case_filter_in
cern_meta cgi charset_lite dav dav_fs dav_lock deflate dir disk_cache echo env
example expires ext_filter file_cache headers http ident imap include info
log_config logio mem_cache mime mime_magic negotiation optional_fn_export
optional_fn_import optional_hook_export optional_hook_import proxy
proxy_connect proxy_ftp proxy_http rewrite setenvif spelling ssl status suexec
unique_id userdir usertrack vhost_alias" \
"--with-ssl=/usr" \
"--prefix=/home/jcw5q/apachetest" \
"--with-mpm=prefork" \
"--enable-maintainer-mode" \
"$@"
```

Now we can go ahead and run our build.

# Build Apache

- Build Apache with your module enabled...

```
root@deephought:/root/apache/httpd-2.1# make
...
/bin/sh /root/apache/httpd-2.1/srclib/apr/libtool --silent --mode=compile gcc -
Wall -Wmissing-prototypes -Wstrict-prototypes -Wmissing-declarations -pthread
-O0 -g -ggdb3 -DHAVE_CONFIG_H -DLINUX=2 -D_REENTRANT -D_XOPEN_SOURCE=500 -
D_BSD_SOURCE -D_SVID_SOURCE -D_GNU_SOURCE -I../include -I../include/arch/unix
-c apr_cpystirn.c && touch apr_cpystirn.lo
/bin/sh /root/apache/httpd-2.1/srclib/apr/libtool --silent --mode=compile gcc -
Wall -Wmissing-prototypes -Wstrict-prototypes -Wmissing-declarations -pthread
-O0 -g -ggdb3 -DHAVE_CONFIG_H -DLINUX=2 -D_REENTRANT -D_XOPEN_SOURCE=500 -
D_BSD_SOURCE -D_SVID_SOURCE -D_GNU_SOURCE -I../include -I../include/arch/unix
-c apr_sprintf.c && touch apr_sprintf.lo
/bin/sh /root/apache/httpd-2.1/srclib/apr/libtool --silent --mode=compile gcc -
Wall -Wmissing-prototypes -Wstrict-prototypes -Wmissing-declarations -pthread
-O0 -g -ggdb3 -DHAVE_CONFIG_H -DLINUX=2 -D_REENTRANT -D_XOPEN_SOURCE=500 -
D_BSD_SOURCE -D_SVID_SOURCE -D_GNU_SOURCE -I../include -I../include/arch/unix
-c apr_strnatcmp.c && touch apr_strnatcmp.lo
...
```

Whoa, what's all this junk?? There are pages and pages and pages of it. With Apache 1.3, the build was very clean – if a warning or error came up, it was easy to spot. That's no longer the case, since we switched to libtool (which is a GNU product that helps you portably create libraries and link to them). Bummer, right? Well, there's an easy workaround...

# Install Apache

- **Build Apache with your module enabled...**

```
root@deephought:/root/apache/httpd-2.1# make >/dev/null  
root@deephought:/root/apache/httpd-2.1#
```

```
root@deephought:/root/apache/httpd-2.1# make install
```

```
root@deephought:/root/apache/httpd-2.1# chown -R jcw5q \  
/home/jcw5q/apachetest
```

Aha! If we just tack on a `>/dev/null` , then all of the uninteresting “normal” stuff from the build gets thrown away, but any warnings or errors that crop up will be kept. In this case I had no warnings or errors... that’s a good thing! `_DO_` pay attention to warnings. You might not have noticed the “`—enable-maintainer-mode`” flag that we passed to configure in the previous step, but basically that’s there to turn on all kinds of useful warnings. Our usual mantra is: “if you get a warning, you did something wrong.”

Then you make install, and Apache will be installed wherever your “`—prefix`” was set to in the configure step. If you’ll look back, you’ll notice that I have it set to `/home/jcw5q/apachetest` . In order to get the test suite work in a minute, we’re going to have to run it as some other user than root (just because it’s quirky and hard to get to work if you do it as root). So I’m going to run it as the user “`jcw5q`”. Thus it’s convenient to install Apache into a subdirectory of `jcw5q`’s homedir (and set `jcw5q` as the owner of it).

# Run the test suite

- **Get the test suite working...**

```
root@deephought:/root/apache/httpd-2.1# cd ~jcw5q  
root@deephought:/home/jcw5q# cd httpd-test/perl-framework
```

Look at the README!

Now take a step back and install the modules we need...

So now we jump over to the jcw5q user's homedir and we're going to jump down into the perl-framework subdirectory of httpd-test (which I'll assume you already have downloaded and put in the right place). But be sure you look at the README file in there if it's your first time using the test suite. In particular, there are some Perl modules you'll need to install.

# Run the test suite

- **Get the test suite working...**

```
root@deephought:/root# perl -MCPAN -e 'install Bundle::ApacheTest'
CPAN: Storable loaded ok
Going to read /root/.cpan/Metadata
  Database was generated on Tue, 14 Oct 2003 05:45:58 GMT
CPAN: LWP::UserAgent loaded ok
Fetching with LWP:
  ftp://cpan.uky.edu/pub/CPAN/authors/id/S/ST/STAS/Apache-Test-1.04.tar.gz
CPAN: Digest::MD5 loaded ok
Fetching with LWP:
  ftp://cpan.uky.edu/pub/CPAN/authors/id/S/ST/STAS/CHECKSUMS
CPAN: Compress::Zlib loaded ok
Checksum for /root/.cpan/sources/authors/id/S/ST/STAS/Apache-Test-1.04.tar.gz ok
Scanning cache /root/.cpan/build for sizes
Apache-Test-1.04/
Apache-Test-1.04/lib/
Apache-Test-1.04/lib/Apache/
...
```

Conveniently, one of the `httpd-test` developers (Stas Bekman) has already bundled up the names of all of the modules we need as “`Bundle::ApacheTest`”, so if you tell perl to use CPAN and install the `Bundle::ApacheTest`, it will go out and install all the Good Stuff you’re going to need for `httpd-test` to work correctly.

# Run the test suite

- **Get the test suite working...**

```
root@deephought:/root/apache/httpd-2.1# cp .gdbinit \  
~jcw5q/httpd-test/perl-framework
```

```
root@deephought:/root/apache/httpd-2.1# su - jcw5q  
jcw5q@deephought:~$ cd httpd-test/perl-framework
```

```
jcw5q@deephought:~/httpd-test/perl-framework$ perl Makefile.PL \  
-apxs /home/jcw5q/apachetest/bin/apxs
```

```
jcw5q@deephought:~/httpd-test/perl-framework$ make
```

NOW we're ready to get started with the test suite. Be sure to copy the `.gdbinit` file out of the `httpd` distribution into your `perl-framework` directory... it will come in handy later. Then go ahead and login as the test user and change back down into the `perl-framework` directory. Then the first thing to do is tell the test suite where Apache is installed. We do that by giving it the path to `apxs` (which is in the same directory as `httpd`); then we run `make`.

# Run the test suite

- **Finally we can RUN the test suite.**

```
jcw5q@deephought:~/httpd-test/perl-framework$ t/TEST
/home/jcw5q/apachetest/bin/httpd -d /home/jcw5q/httpd-test/perl-framework/t -f
/home/jcw5q/httpd-test/perl-framework/t/conf/httpd.conf -DAPACHE2
using Apache/2.1.0-dev (prefork MPM)

waiting for server to start: ..
waiting for server to start: ok (waited 1 secs)
server localhost:8529 started
...
apache/404.....ok
apache/acceptpathinfo....ok
apache/byterange.....ok
apache/chunkinput.....ok
apache/contentlength.....ok
...
```

To run the tests, you just type `t/TEST`, and off they go. It's good to just run the default set of tests right off the bat to make sure you have everything installed correctly. Ideally, all of the tests that come with the suite will pass, but occasionally one of them (correctly) fails. As of this writing, for example, a few of the `mod_include` tests and some of the `mod_ssl` tests do not pass on Apache 2.0, indicating bugs in the server. As long as you don't get any really catastrophic errors, you'll be okay.

# Run the test suite

- **Add your own tests:**
  - Look at existing tests for examples (see `t/modules/`\*)
  - Just add a `.t` file to `t/modules/` and the test suite will see it.
  - The test should throw as many different kinds of requests at your module would ever be expected to handle
    - Start with the basics
    - Think defensively

# Run the test suite

- **Add your own tests:**
  - You can use additional Perl modules (eg, client libraries) to help you out if you need
  - If there's any configuration you need, you can add it to `t/conf/extra.conf.in`
    - You'll need to "make clean" and then start over from the "perl Makefile.PL ..." step.

# Run the test suite

- Run the test suite with your new test.

```
jcw5q@deephought:~/httpd-test/perl-framework$ t/TEST
...
modules/access.....ok
modules/alias.....ok
modules/autoindex.....ok
modules/autoindex2.....ok
modules/cgi.....ok
modules/dav.....ok
modules/deflate.....ok    <- looks good so far... but...
modules/dir.....ok
modules/env.....ok
modules/expires.....ok
...
*** server localhost:8529 shutdown
!!! oh nuts, server dumped core
!!! for stacktrace, run: gdb /home/jcw5q/apachetest/bin/httpd -core \
    /home/jcw5q/httpd-test/perl-framework/t/core
```

Everything looks good at first, but then when we get down to the end, we discover that at some point the server dumped core. Uh-oh, that's not good. So we fire up gdb.

# Debug the server

- **Fire up gdb:**

```
jcw5q@deephought:~/httpd-test/perl-framework$ gdb ~/apachetest/bin/httpd t/core
GNU gdb 5.3
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-slackware-linux"...
Core was generated by `~/home/jcw5q/apachetest/bin/httpd -d /home/jcw5q/httpd-
test/perl-framework/t -f'.
Program terminated with signal 11, Segmentation fault.
...
#0  0x382f4a6e in ?? ()
(gdb)
```

We could just cut-and-paste the command that the test suite suggests we run... here I'm using a shorthand version that also works. Once we get to the prompt, gdb will tell us what the function at the top of the call stack is. Obviously not too helpful in this case, so we're going to have to dig.

# Debug the server

- **Get a backtrace:**

```
#0 0x382f4a6e in ?? ()
(gdb) bt
#0 0x382f4a6e in ?? ()
#1 0x40171382 in brigade_cleanup (data=0x8277f58) at apr_brigade.c:72
#2 0x4024ef81 in run_cleanups (cref=0x8274360) at apr_pools.c:1994
#3 0x4024e502 in apr_pool_destroy (pool=0x8274350) at apr_pools.c:763
#4 0x4024e41c in apr_pool_clear (pool=0x8264be8) at apr_pools.c:723
#5 0x080e6266 in child_main (child_num_arg=1) at prefork.c:563
#6 0x080e65d2 in make_child (s=0x81543a0, slot=1) at prefork.c:742
#7 0x080e6838 in perform_idle_server_maintenance (p=0x814d0f0)
   at prefork.c:877
#8 0x080e6c7c in ap_mpm_run (_pconf=0x814d0f0, plog=0x81a12e0, s=0x81543a0)
   at prefork.c:1073
#9 0x080ed5a6 in main (argc=6, argv=0xbffff774) at main.c:674
#10 0x40336d06 in __libc_start_main () from /lib/libc.so.6
```

Ahh, now we're getting somewhere. Okay, well, we know we've been working with buckets, and we see here that we got a segfault cleaning up a brigade somewhere. But since no particular request failed, under normal circumstances what we'd do here is set a breakpoint on our filter (which we suspect as the problem, since there was no segfault the last time we ran the test ;), and then we would run the test again, stepping into our module and trying to catch the segfault. As you can imagine, this is very tedious, but sometimes it's your only choice.

# Debug the server

- **Good places to set breakpoints (if you have to):**
  - `core_output_filter`
  - `default_handler`
  - `ap_pass_brigade`

As a side note, if you do find yourself having to step through the handling of a particular request because you have no clue which module is causing the problem, these are some particularly handy strategic points where you can stick your breakpoints as a first guess. The `core_output_filter` is what passes all your response data out to the network, so all of the buckets churned out by various filters that end up in the response will go through here. It's quite nice to break at the beginning of the `core_output_filter` and do a `dump_brigade` on the brigade that gets passed to it (I'll talk about `dump_brigade` later). If that doesn't help too much, you can break on the `default_handler` (or whatever other handler you may be using), which will hand control to you at the *beginning* of the processing of the request ... you can step through it from there if you need to. `ap_pass_brigade()` is what gets your data from one filter to the next, so it's also a good place to stop, checking the contents of the bucket brigades each time; that can help you pin down which filter is causing the problem if you don't know ahead of time.

# Rebuild Apache

- **Enable bucket debugging**

```
root@deephought:/root/apache/httpd-2.1# cat config.nice
#!/bin/sh
#
# Created by configure

CFLAGS="-O0 -g -ggdb3 -DAPR_BUCKET_DEBUG"; export CFLAGS
"./configure" \
"--enable-modules=actions alias asis authn_file authn_dbm authn_anon authn_default
authz_host authz_groupfile authz_user authz_dbm authz_owner authz_default
auth_basic auth_digest autoindex bucketeer cache case_filter case_filter_in
cern_meta cgi charset_lite dav dav_fs dav_lock deflate dir disk_cache echo env
example expires ext_filter file_cache headers http ident imap include info
log_config logio mem_cache mime mime_magic negotiation optional_fn_export
optional_fn_import optional_hook_export optional_hook_import proxy
proxy_connect proxy_ftp proxy_http rewrite setenvif spelling ssl status suexec
unique_id userdir usertrack vhost_alias" \
"--with-ssl=/usr" \
"--prefix=/home/jcw5q/apachetest" \
"--with-mpm=prefork" \
"--enable-maintainer-mode" \
"$@"
```

Fortunately, in this case, it turns out that we have some additional tools that we can use to debug the buckets. It means, however, that we have to jump all the way back to step one and reconfigure, rebuild, and reinstall Apache. When you debug your own modules, you'll probably just want to skip straight to this point and enable the tools I'm about to show you so that you don't have to do this twice. I did it the hard way first just so you could see the difference. If you do find yourself going back and rebuilding, note that some kinds of changes you make to the configure options will require a "make distclean" before you re-run configure just to make sure that all of your changes take effect. Changing MPMs is one example of a time you'd want to be sure to distclean... I do it every time just to be safe.

# Bucket Debugging

- Use `-DAPR_BUCKET_DEBUG` to enable a set of handy runtime assertions in the buckets code:
  - In the bucket memory allocator, we check to make sure you don't double-free
  - In the bucket brigade macros (insert, concat, etc), we check that the buckets in the brigade all point to each other correctly

- gdb tools:

```
root@deephought: /root/apache/httpd-2.1# grep define .gdbinit
...
define dump_bucket
define dump_brigade
define dump_filters
...
```

Fortunately, in this case, it turns out that we have some additional tools that we can use to debug the buckets. It means, however, that we have to jump all the way back to step one and reconfigure, rebuild, and reinstall Apache. When you debug your own modules, you'll probably just want to skip straight to this point and enable the tools I'm about to show you so that you don't have to do this twice. I did it the hard way first just so you could see the difference. If you do find yourself going back and rebuilding, note that some kinds of changes you make to the configure options will require a "make distclean" before you re-run configure just to make sure that all of your changes take effect. Changing MPMs is one example of a time you'd want to be sure to distclean... I do it every time just to be safe.

# Debug the server

- Now that we have our bucket debugging compiled in, re-run the test:

```
jcw5q@deephought:~/httpd-test/perl-framework$ t/TEST -v t/modules/deflate.t
...
modules/deflate...1..4
# Running under perl version 5.008 for linux
# Current time local: Wed Oct 15 01:08:36 2003
# Current time GMT: Wed Oct 15 05:08:36 2003
# Using Test.pm version 1.24
testing default
ok 1
ok 2
ok 3
ok 4
ok
All tests successful.
Files=1, Tests=4, 5 wallclock secs ( 0.46 cusr + 0.06 csys = 0.52 CPU)
*** server localhost:8529 shutdown
!!! oh golly, server dumped core
!!! for stacktrace, run: gdb /home/jcw5q/apachetest/bin/httpd -core /home/jcw5q/httpd-test/perl-
framework/t/core
```

You can narrow down which tests are run by just listing the name of the test as an argument to `t/TEST`. Also the `-v` option turns on “verbose” debugging. It’s more verbose in some modules’ tests than in others... `grep` through some of them for “`t_cmp`” for examples of how to do it the right way. `t_cmp` is a really handy little function to which you pass the expected value, the received value, and a message, all of which it displays if you’re in verbose output mode. It even does the equality check to see if `expected==received` for you.

# Debug the server

- Now that we have our bucket debugging compiled in, re-run the test (narrow it down to save time):

```
jcw5q@deephought:~/httpd-test/perl-framework$ gdb ~/apachetest/bin/httpd t/core
...
(gdb) bt
#0 0x4034bacl in kill () from /lib/libc.so.6
#1 0x402d99ed in pthread_kill () from /lib/libpthread.so.0
#2 0x402d9d0b in raise () from /lib/libpthread.so.0
#3 0x4034b6fa in raise () from /lib/libc.so.6
#4 0x4034d127 in abort () from /lib/libc.so.6
#5 0x40170745 in check_not_already_free (node=0x8278638)
    at apr_buckets_alloc.c:175
#6 0x4017078e in apr_bucket_free (mem=0x8278648) at apr_buckets_alloc.c:190
#7 0x401715cb in apr_brigade_cleanup (data=0x827bf50) at apr_brigade.c:86
#8 0x4017155a in brigade_cleanup (data=0x827bf50) at apr_brigade.c:72
#9 0x4024ff81 in run_cleanups (cref=0x827e380) at apr_pools.c:1994
#10 0x4024f502 in apr_pool_destroy (pool=0x827e370) at apr_pools.c:763
...
```

Much more useful! Now we know that we definitely tried to free something that had already been freed. We did this while cleaning up a pool (it's often useful to know *which* pool ... Sander will explain how you go about figuring that out).

# Debug the server

- Figure out which bucket was being freed:

```
(gdb) frame 6
#6  0x4017078e in apr_bucket_free (mem=0x8278648) at apr_buckets_alloc.c:190
190      check_not_already_free(node);
(gdb) dump_bucket 0x8278648
bucket=EOS      (0x08278648) length=0      data=0x00000000
contents=[**unknown**]      rc=n/a
```

If we jump down to frame 6, which is the call to `apr_bucket_free()`, we can start playing around with the void “mem” pointer that was passed to it. Now I happen to know that `apr_bucket_free()` only really gets called on a few kinds of things: `apr_bucket` structures, a small handful of internal bucket structures, and some bucket buffers, so I would probably guess that `mem` in this case is an `apr_bucket`. Of course, you probably wouldn’t know that if I hadn’t just told you, but you could have figured it out by grepping through the buckets code. Alternatively, if we’re too unsure what’s going on in one stack frame, we can always try to start from some other stack frame. So let’s go look at frame 7.

# Debug the server

- Figure out which bucket was being freed:

```
(gdb) frame 7
#7  0x401715cb in apr_brigade_cleanup (data=0x827bf50) at apr_brigade.c:86
86      apr_bucket_delete(e);
(gdb) list
84      while (!APR_BRIGADE_EMPTY(b)) {
85          e = APR_BRIGADE_FIRST(b);
86          apr_bucket_delete(e);
87      }
```

Now we really start to see the picture. Notice that the segfault happened while clearing a pool, and none of the requests themselves failed. Sander will show us how to figure out that it was the “request” pool, but just take my word for it that that’s which one it was. The request pool is where we put all of our “scratch space” for handling the current request. We clear it after the request is done. Normally buckets are cleaned up as soon as you’re done with them, but as a safety precaution, any brigades that have buckets in them at the end of the request are automatically cleaned up. So here we are, cleaning up one of these brigades, and we discover that we’re deleting some bucket that’s already been deleted elsewhere. That’s odd. So either we destroyed (ie, freed) the bucket without removing it from this brigade, or it was in two brigades and got deleted (“deleted” meaning “destroyed and removed”) from the other brigade.

# Debug the server

- Figure out which bucket was being freed:

```
(gdb) dump_brigade b
dump of brigade 0x827bf50
  | type      (address)      | length | data addr | contents      | rc
-----|-----|-----|-----|-----|-----
0 | FLUSH      (0x08278878) | 0      | 0x00000000 | [**unknown**] | n/a
1 | 0Pc00(0x0826b35c) | 136753984 | 0x4017153c | [**unknown**] | n/a
2 | 0Pc00(0x0826b35c) | 136753984 | 0x4017153c | [**unknown**] | n/a
3 | 0Pc00(0x0826b35c) | 136753984 | 0x4017153c | [**unknown**] | n/a
...
(gdb) dump_bucket e
bucket=EOS      (0x08278648) length=0      data=0x00000000
contents=[**unknown**]      rc=n/a
```

So let's take a look at this brigade. We can use the "dump\_brigade b" command, which prints out the contents of b. b is obviously corrupted since we've already started deleting it, but we can also look directly at e by using the "dump\_bucket e" command. So the bucket that killed us was an EOS bucket. We're guaranteed to only get one of those per request, so I think now we have enough information to track this down. Let's go look at the part of our filter that handles EOS buckets.

# Debug the server

- Inspect the relevant code:

```
static apr_status_t deflate_out_filter(ap_filter_t *f,
                                      apr_bucket Brigade *bb)
{
    ...
    APR_BRIGADE_FOREACH(e, bb) {
        ...
        if (APR_BUCKET_IS_EOS(e)) {
            ...
            APR_BRIGADE_INSERT_TAIL(ctx->bb, e);

            /* Okay, we've seen the EOS.
             * Time to pass it along down the chain.
             */
            return ap_pass_brigade(f->next, ctx->bb);
        }
        ...
    }
}
```

Now here's the part that really takes a sharp eye. What we've done is to take our EOS bucket "e" and put it at the end of ctx->bb and pass that brigade down the chain. Fine. But where was e before we put it in ctx->bb? We can't just leave dangling references to it, which is exactly what we've done. This bucket was originally in the "bb" brigade that got passed in to us from the filter before us. We didn't actually *remove* bucket e from that brigade, we just inserted it into some new brigade. That's bad. In particular, brigade bb still has pointers to e, but e->next and e->prev no longer point to buckets in bb; they point to buckets in ctx->bb. If we ever ran a consistency check on bb (eg, by inserting something else into it), we'd find this out. But in this case we apparently never did anything else with bb until it was time to clean it up at the end of the request. And lo and behold, we went to clean up this bucket we THOUGHT was still in bb, but the filter we passed it to via ctx->bb already cleaned it up. We've found the bug! Now let's fix it.

# Debug the server

- Fix the bug:

```
static apr_status_t deflate_out_filter(ap_filter_t *f,
                                      apr_bucket brigade *bb)
{
    ...
    APR_BRIGADE_FOREACH(e, bb) {
        ...
        if (APR_BUCKET_IS_EOS(e)) {
            ...
            APR_BUCKET_REMOVE(e);
            APR_BRIGADE_INSERT_TAIL(ctx->bb, e);

            /* Okay, we've seen the EOS.
             * Time to pass it along down the chain.
             */
            return ap_pass_brigade(f->next, ctx->bb);
        }
        ...
    }
}
```

So all we do is stick in an `APR_BUCKET_REMOVE(e)` call before we insert `e` into `ctx->bb`. What that does is to unlink `e` from the brigade it's currently in, allowing us to do with it as we please without causing any other data structures to become inconsistent or corrupt. In fact, if you go and look at what `apr_bucket_delete(e)` does, it's actually just a macro that does "`APR_BUCKET_REMOVE(e); apr_bucket_destroy(e);`". That's really the same thing, if you think about it; if you're going to say "I no longer want this bucket; throw it out," you should fix up the brigade it's in not to point to it anymore before freeing its memory, which is what `apr_bucket_destroy` does.

## Additional references

- Learn by example from other filters:
  - `mod_case_filter`
  - `mod_bucketeer`
  - `mod_input_body_filter`
- Read the buckets documentation:
  - [http://apr.apache.org/docs/apr-util/group\\_\\_APR\\_\\_Util\\_\\_Bucket\\_\\_Brigades.html](http://apr.apache.org/docs/apr-util/group__APR__Util__Bucket__Brigades.html)
- Slides from last year:
  - <http://www.apache.org/~gregames/ap2filters.ppt>
  - <http://www.cs.virginia.edu/~jcw5q/talks/apache/bucketbrigades.ac2002.ppt>

So we're done with this little exercise. There are all kinds of things like this that you're going to have to get familiar with if you're writing filter code. I recommend you look at some of the example filters that we have in the "experimental" and "test" directories – `mod_case_filter` is nice to start with because it's VERY bare-bones and easy to follow. `Mod_bucketeer` is also worth a look, as are the filters that come with the `httpd-test` suite. From there, you can tackle the rest of `mod_deflate` if you'd like. I warn you against looking too closely at `mod_include`, however, as your eyes may be permanently crossed – the one that comes with Apache 2.0 is a horrible beast. In fact, it's SO hard to debug that it alone is a large part of the reason we decided to hold this tutorial in the first place. The one in Apache 2.1+ was rewritten by Andre Malo to address that particular problem, so have a look at that one if you're feeling adventurous, but it's still a lot to try to wrap your head around.

There are also slides from last year's ApacheCon talks on bucket brigades and filters, which were given by myself and Greg Ames, respectively; you can get those from the URL's above.

## Other debugging tools

- [Back to Sander...](#)