Debugging Tricks with Apache HTTP Server 2.4

Jeff Trawick

`http://emptyhammock.com/`
`trawick@emptyhammock.com`

April 7, 2014

Get these slides...

http://emptyhammock.com/projects/info/slides.html

Table of Contents

Introduction — Who am I?

- I've worked at
  - several large corporations, for over two decades
  - my own one-person *company*, Emptyhammock, for the last two years
- I've worked on
  - several products which were primarily based on or otherwise included Apache HTTP Server
  - lower-level networking products
  - web applications
- I've debugged *many* customer and user problems over the years.

Introduction — What will we attempt to cover?

- Touch on all the basics.
- Describe all the new httpd 2.4 debugging features.
- Summarize the techniques which are different with httpd 2.2.

What kinds of issues encountered

- Crash
- Hang of server
- Stall of individual requests
- Termination
- Bad response time
- Limited concurrency without problem symptoms
- High CPU
- High memory
- High consumption of other pooled resources
- Incorrect output - wrong transformation
- Incorrect output - missing/bad protocol element

Non-problems

- Validate behavior of new software/configuration
- Understand steady-state behavior for baseline when something is wrong

Using tools to look inside the web server

- Logging (the information itself, the timestamp, information about other processing at about the same time)
- OS-level tools (view use of resources, whether discrete items like files or continuous like CPU)
- CPU-, code-level tools (determine what code is running frequently, what is running for the request, analyze memory references, walk through the processing of a request, etc.)

Careful with logging!

As you increase the level of logging, you increase the chances that private data will be logged.

- Passwords, session keys, etc.

Modules/log configurations of particular interest:

- mod_dumpio, mod_log_config when configured to log certain request or response header fields
- mod_log_forensic
- http (the built-in module) when configured at higher trace levels

Logging

- Error log

Error log records

- Configurable content
- Fields dropped when information is unavailable
- Third-party modules can implement their own fields

Typical message:

```
[Sun Oct 28 13:37:27.676386 2012] [:error]
[pid 14340:tid 140625844377344] [client 127.0.0.1:50837]
mod_wsgi (pid=14340): Target WSGI script
'/home/trawick/myhg/apache/documents/AC20
12EU/lookup.wsgi' does not contain WSGI
application 'application'.
```

Hiding error log fields

The ErrorLogFormat directive can limit which fields are logged, but you could implement post-processing to remove fields as appropriate for what you are debugging.

My own silly mechanism:

```
$ grep Accept-Ranges logs/error_log
[Thu Apr 03 07:26:49.605322 2014] [http:trace4] [pid 13680:tid 140130244732672] http_filters.c(83
$ grep Accept-Ranges logs/error_log | nots.pl
[http:trace4] [pid 13680:tid 140130244732672] http_filters.c(837): [client 192.168.1.207:60141]
$ grep Accept-Ranges logs/error_log | nots.pl | nomodlevel.pl
[pid 13680:tid 140130244732672] http_filters.c(837): [client 192.168.1.207:60141]   Accept-Ranges
$ grep Accept-Ranges logs/error_log | nots.pl | nomodlevel.pl | nopidtid.pl
http_filters.c(837): [client 192.168.1.207:60141]   Accept-Ranges: bytes
$ grep Accept-Ranges logs/error_log | nots.pl  | nomodlevel.pl  | nopidtid.pl | noclient.pl
http_filters.c(837):   Accept-Ranges: bytes
```

Detailed logging only for specified client IP

```
LogLevel info
<If "%{REMOTE_ADDR} =~ /127.0.0/">
LogLevel trace8
</If>
```

- Only works once request processing has reached a certain point. Connection-level issues which occur before that point won't be logged.
- *Prior to 2.4.4, this expression needed to be placed inside a Location container to be effective.*

Detailed logging for problematic requests

```
LogLevel info
<Location /problem/>
  LogLevel trace8
</Location>
```

(Mostly) HTTP layer logging at different levels

```
[core:trace5] Request received from client: GET / HTTP/1.1
[http:trace4] Headers received from client:
[http:trace4]   Connection: keep-alive
[http:trace4]   Cache-Control: max-age=0
[http:trace4]   User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/535.19 (KH...
[http:trace4]   Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
[http:trace4]   Accept-Encoding: gzip,deflate,sdch
[http:trace4]   Accept-Language: en-US,en;q=0.8
[http:trace4]   Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
[http:trace4]   If-None-Match: \\"2d-4b1922bade1c0\\"
[http:trace4]   If-Modified-Since: Sat, 12 Nov 2011 23:41:03 GMT
[http:trace3]  Response sent with status 304, headers:
[http:trace5]   Date: Tue, 06 Nov 2012 12:18:57 GMT
[http:trace5]   Server: Apache/2.4.4-dev (Unix) OpenSSL/1.0.0e mod_wsgi/3.4 Python...
```

mod_log_debug

- Configurable debug logging mechanism using new `LogMessage` directive.
- Different ways to think of it:
  - Generate custom trace or error messages for processing of interest to you.
  - Track interesting values as they change (or not) during request processing.
- Conditional expression support with access to dynamic values is provided by the new *ap_expr* support.
  - `http://httpd.apache.org/docs/2.4/expr.html`

mod_log_debug – sample configuration

```
# Log some module's request note at all phases
# of processing (but only if set)
<Location />
  LogMessage "%{note:mod_your_debug}" hook=all \
    "expr=-T %{note:mod_your_debug}"
</Location>

# Log when a location is requested as a subrequest
<Location /app/dash/>
  LogMessage "subrequest to /app/dash/" \
    hook=type_checker "expr=-T %{IS_SUBREQ}"
</Location>

# Log when a particular error is encountered
LogMessage "Timeout from %{REMOTE_ADDR}" \
    "expr=%{REQUEST_STATUS} = 408"
```

mod_dumpio

- This is a way to trace the raw, unencrypted data exchange into the error log.
- A packet trace is usually preferable, but in some environments it is simpler to modify the httpd configuration to enable this module than it is to capture packets.
  - Also, if the person analyzing diagnostic data won't have access to server keys, a packet trace can't be used to understand most application-layer issues.

mod_dumpio configuration

```
LogLevel info dumpio:trace7
DumpIOInput On
DumpIOOutput On
```

mod_dumpio output

You can see I/O operations on input as well as input and output data.

```
dumpio_in [getline-blocking] 0 readbytes
dumpio_in (data-HEAP): 20 bytes
dumpio_in (data-HEAP): GET /dir/ HTTP/1.1\r\n
dumpio_in [getline-blocking] 0 readbytes
dumpio_in (data-HEAP): 22 bytes
...
dumpio_in (data-HEAP): Connection: keep-alive\r\n
```

extraneous information removed

Catching requests which do not finish

(presumably due to a child process crash, though you could potentially identify hung requests if you don't use mod_status)

```
LoadModule log_forensic_module modules/mod_log_forensic.so
ForensicLog logs/forensic.log
```

This logs the start and end of the request along with all of the request headers.

```
+UJggYn8AAQEAAAs1da4AAAAA|GET / HTTP/1.1|Host...
-UJggYn8AAQEAAAs1da4AAAAA
```

- `check_forensic` will scan the log and determine which requests didn't finish cleanly.
- Compare with mod_whatkilledus, described later.

Where did that error message come from?

- module id in error log:
  ```
  [core:info] [pid 4373:tid 140043736946432] ...
  AH00128: File does not exist: ...
  ```
- whoops, missing module id:
  ```
  ... [:info] [pid 8889:tid 140363200112416] mod_wsgi
  (pid=8889): Initializing Python.
  ```
  In this case it is obviously mod_wsgi, but it isn't always that easy. (FWIW, the fix is in mod_wsgi issue 292.)
  ```
  ... [:error] [pid 14883:tid 140625458312960] 1
  ... [:error] [pid 14883:tid 140625458312960] 2
  ... [:error] [pid 14883:tid 140625458312960] 3
  ... [:error] [pid 14883:tid 140625458312960] 4
  ```
  (That was mod_wsgi logging stderr from a script.)

Where did that error message come from?

The module id in the error log records is your hint on controlling the log level to see or hide the message.

core:info	Configure a specific LogLevel for module `core` to see or hide this.
:error	No module is available, so this log message can't be controlled with a module-specific LogLevel.

mod_backtrace feature to identify message source

Consider this if no module id is available or you need to know the caller of a utility function that logged a message.

- mod_backtrace has the capability of adding a backtrace to error log messages in certain conditions.

  ```
  ErrorLogFormat ... [%{/AH00128/}B] ...
  ```

- If the search string appears in the message, a mini-backtrace will appear as an additional field in the error log record.

  ```
  ... [0x4453dd<ap_run_handler<ap_invoke_handler<
  ap_process_async_request<ap_process_request] ...
  AH00128:...
  ```

- `http://emptyhammock.com/projects/httpd/diag/`

Looking from the outside

- examining resource use
- tracing activity

Resource use

- top/iostat/vmstat/etc. (even ps)

System call trace

- strace/truss/dtruss

Higher level tools

- Brendan Gregg's DTrace Toolkit, at `http://www.brendangregg.com/dtracetoolkit.html`

  The DTrace Toolkit has been around for a while and contains a number of analysis and reporting scripts based on DTrace.
- sysdig, at `http://www.sysdig.org/`

  sysdig was just announced last week. I haven't played with it much yet; the Lua scripts, *chisels*, appear to operate at roughly the same layer as the scripts in the DTrace Toolkit, and a lower-level command provides the basic collection features.

A simple DTrace Toolkit example...

```
$ sudo /usr/share/dtrace/toolkit/procsystime -n httpd
^C

Elapsed Times for processes httpd,

         SYSCALL          TIME (ns)
...
         accept4           25569461
           close           29081544
            stat           36630193
          munmap           41668446
          writev           48378858
        shutdown           71471901
     gettimeofday          97454962
           write         1000753076
          select         8131189175
        _umtx_op        22781598217
          kevent        32804433802
```

A simple sysdig example...

```
$ sudo sysdig -n 1000 -c topfiles_bytes proc.name=httpd

Bytes     Filename
------------------------------
9.27KB    /home/trawick/inst/24-64/logs/forensic.log
960B      /home/trawick/inst/24-64/logs/access_log
494B      /home/trawick/inst/24-64/manual/mod/module-dict.html
```

(I forgot that I had enabled mod_log_forensic...)

DTrace one-liners, on several platforms

Jeff, this is where you view the document in the browser.

Looking inside the process with a debugger

Basic information: Backtraces

- gdb
  - Most platforms (even Windows, using MinGW gdb on MinGW build of httpd)
  - Basic use:

    ```
    gdb /path/to/httpd pid-or-corefile
    (gdb) thread apply all bt full
    ```

    (but other commands may be useful too)
- pstack
  - Solaris (I learned through bad experiences to pretend that pstack isn't available on Linux)
  - Use:

    ```
    pstack pid-or-corefile
    ```

    (but pflags and pldd information is also good)

Getting more debugging information

- The backtraces (with variables if available) are most important, but more information is available if you ask for it.
- gdb, more details:

```
(gdb) info sharedlibrary
(gdb) info threads
(gdb) thread apply all bt full
(gdb) thread apply all x/i $pc
```

- Solaris /proc tools:

```
# pstack 13579
# pldd 13579
# pflags 13579
# pmap 13579
```

httpd-specific gdb tricks

```
(gdb) source /path/to/2.4.x/.gdbinit
(gdb) dump_table r->headers_in
[0] 'Host'='127.0.0.1:8080' [0x7f8094003cb6]
[1] 'Connection'='close' [0x7f8094003cd4]
(gdb) dump_brigade b
dump of brigade 0x7f8094007320
   | type     (address)    | length | data addr  | contents          | rc
------------------------------------------------------------------------------
 0 | FILE     (0x7f8094000b08) | 45     | 0x7f8094000ba8 | [**unp... | 1
 1 | EOS      (0x7f8094000c48) | 0      | 0x00000000 |            | n/a
end of brigade
```

Use of these macros requires some familiarity with the httpd implementation or module programming interface.

Example output

Jeff, this is where you show ubuntu64.core.collect.gdbout and solaris10.core.pstackout.

Umm, what does that stuff mean?

- Recognize normal behavior
- Determine where crash likely occurred
- Determine definitively where crash occurred

(similar issues for hang)

Umm, what does that stuff mean? (cont.)

- Perplexing (?) problem: Show that output to an httpd developer and they can quickly determine the important parts (i.e., pick the interesting thread)
  - or determine that there's nothing interesting, which can be just as important
- Users typically report the least interesting thread from the core dump, which wastes their time and ours.
- Some sort of automatic annotation/explanation would be useful.
  - Descriptions of normal activity
  - Bug numbers for backtraces that match known problems
  - *et cetera*

Demo

Jeff, this is where you go to `http://emptyhammock.com/projects/httpd/explore/`*. Try loading PR53870.pstackout and ubuntu64.core.collect.gdbout.*

What if you build the code differently

- Improving general debuggability of the generated code by affecting code generation or symbols
- Enabling optional run-time checks
- Enabling third-party exception hooks
- Enabling third-party tracing of API hooks

Different code generation for debugging

- Adding symbols, not stripping executable
- Disabling in-lining of functions for better diagnosablity
- Disabling other optimization so that more variables can be checked
- Options like `-funwind-table` for tools like mod_backtrace to work on ARM

(huge YMMV, with architecture, OS, compiler, and compiler/linker flags as variables)

General debug capabilities not built in by default

- Hook tracing
- DTrace probes in the server (DTrace provider *ap*)
- Exception hooks

Hook tracing

- httpd hooks are what allow different modules to handle or otherwise affect processing of the different phases of execution.
- A module that needs to take part in a particular aspect of connection or request processing uses a special hook macro to save a callback pointer.
- At the point where httpd core passes control to modules, it invokes a special hook macro to continue calling module callbacks until a failure occurs, a module elects to handle the request, or all callbacks have been serviced (depending on the hook).
- By tracing what happens inside the hook invocation, some types of failures can be quickly tracked to a particular module.

Hook tracing (cont.)

- httpd now provides a way for third-party code to run during the hook macros at the following points:
  - Start of the hook execution
  - About to call a particular module's hook function
  - Returned from that module's hook function
  - End of the hook execution
- Code inserted into the calling of different modules' handler functions can determine what module's handler took ownership of this phase of request processing and/or caused the request to fail.
- More generally, if some mysterious error occurs at any phase of processing, such as the notorious 500 with no log message, hook tracing could pinpoint the module.

Enabling hook tracing

- Configure argument `--enable-hook-probes` causes `ap_hook_probes.h` to be included in files with hook definitions, making special macros active.
- `ap_hook_probes.h` isn't part of httpd, so it needs to be copied into `include` or located via `CPPFLAGS`.
- Any code invoked by the macros in `ap_hook_probes.h` has to be compiled into the server, so this can be handled by statically linking a module into the server if the desired logic can't be implemented completely in a macro.

Enabling hook tracing (cont.)

- Build mechanism for including this code

```
export CPPFLAGS=-I/path/to/module
./configure --enable-hook-probes \
--with-module=debugging:/path/to/module/mod_foo.c \
--other-args
```

- After httpd is built, `httpd -l` will show `mod_foo.c` as built-in (like `core.c` and a few others).

mod_hook_ar — Experimental hook tracer

- Must be built into the server as with other hook trace code.
- Sets a request note to information about the active module while a hook is active.
- Sets a request note to information about the failing module if a hook returns an error.
- Logging the `RequestFailer` note in the access log:

```
127.0.0.1 ..."GET /cgi-bin/printenva" \
   404 215 mod_cgid.c/404/handler
```

- Can log the name of the `ActiveModule` note in the case of a crash:

```
... [pid 30568:tid 140369329334016] Crash state: \
   mod_crash.c/handler
```

- Download from `http://emptyhammock.com/downloads/`

Possible directions with hook tracers

- How much performance degradation?
- Can this be used to implement DTrace probes?
- Can a built-in module provide a simple API for loadable hook debug modules?
- Will someone write a script to help with generating the right set of macros based on the hooks that need to be instrumented?

(if indeed this is interesting to anyone)

DTrace probes

- httpd-specific probes enabled via `--enable-dtrace` was the goal for 2.4, but only part of the code was committed, and it hasn't been kept up to date with new hooks.
- Someone needs to take interest in getting it working on one of the several platforms with DTrace.
- Existing DTrace providers can certainly help understand httpd processing.
- The pid provider provides great info but it is problematic with httpd because you have to specify a particular process id.

Has anyone tried to use mod_dtrace with 2.4?

Exception hooks

- `sig_coredump()` is the handler for fatal signals with httpd on Unix since the httpd 1.3 days.
- It changes to the configured core dump directory and re-throws the signal, causing the process to exit; at this point the system (possibly) creates a core file.
- If the `--enable-exception-hook` configure option was specified, `sig_coredump()` will also call exception hooks.
- This allows third-party modules to clean up some resource or save diagnostic information in the event of a crash.

Example exception hook module — mod_whatkilledus

- Like mod_log_forensic, this module saves information about the client request in an early request processing hook.
- Unlike mod_log_forensic, the info is kept in memory during the life of the request, and only logged if a crash occurs.
- Also, if mod_backtrace is loaded it will capture a backtrace for the crashing thread.

mod_whatkilledus report

```
**** Crash at 2012-09-06 14:48:23
Process id:  23368
Fatal signal: 11

...
/home/trawick/inst/24-64/bin/httpd:ap_run_fatal_exception+0x5b 0x4305
...
/home/trawick/inst/24-64/modules/mod_crash.so:0x7fecbd59e986
/home/trawick/inst/24-64/modules/mod_crash.so:0x7fecbd59ead8
/home/trawick/inst/24-64/bin/httpd:ap_run_handler+0x5b 0x45008e
/home/trawick/inst/24-64/bin/httpd:ap_invoke_handler+0x173 0x450966
/home/trawick/inst/24-64/bin/httpd:ap_process_async_request+0x264 0x4
/home/trawick/inst/24-64/bin/httpd:0x468dc4
/home/trawick/inst/24-64/bin/httpd:0x468fb3
/home/trawick/inst/24-64/bin/httpd:ap_run_process_connection+0x5b 0x4
...
```

mod_whatkilledus report (cont.)

```
Request line (parsed):
GET :10080 /crash/
Request headers:
Host:127.0.0.1%3a10080
User-Agent:ApacheBench/2.3
Accept:*/*

Client connection:
127.0.0.1:44883->127.0.0.1:10080  (user agent at 127.0.0.1:44883)
```

mod_whatkilledus notes

- mod_whatkilledus and mod_backtrace can actually work well on Windows, with great backtraces if the web server .pdb files are available. *Uhhh, I don't have mod_backtrace working for 64-bit httpd on Windows yet.*
- The original versions of mod_whatkilledus and mod_backtrace worked somewhat differently:
  - mod_backtrace and mod_whatkilledus acted independently.
  - Neither supported Windows, and mod_backtrace supported fewer Unix-y platforms.
  - mod_whatkilledus had no mechanism to filter out sensitive information.
- `http://emptyhammock.com/projects/httpd/diag/`

Comparison with httpd 2.2 — error log

- Error messages
  - No module id, pid, thread id, etc. unless the module generating the message adds it explicitly.
  - No control over the format.
  - No sub-second timestamps.
  - No traceXXX levels
    Some messages just aren't present, because even LogLevel debug would be too noisy, or separate log files are used (mod_rewrite) which have to be managed independently.
  - No per-module LogLevel, no per-dir LogLevel (which is what allows per-client LogLevel)
    Custom scripting can be used to reduce the output to something readable, though nothing can be done about the volume, and that may necessitate a different scheme for rotating logs during problem determination.

Comparison with httpd 2.2 — other logs

- mod_log_debug isn't available.

Recap of Jeff's toys

- Explore, collect.py
- mod_backtrace and mod_whatkilledus
- mod_hook_ar
- pgfiles.py (not mentioned; shows open files for a process group, organized to show which files are shared by different processes)
- nots.pl, nomodlevel.pl, etc.

Available from

- `http://emptyhammock.com/projects/` and/or `http://emptyhammock.com/downloads/` (or ask Jeff directly for nots.pl et al)

httpd materials

- httpd debugging guide in the reference manual
  http://httpd.apache.org/dev/debugging.html
- module debugging guide from Cliff Wooley
  http://www.cs.virginia.edu/~jcw5q/talks/apache/apache2moddebugging.ppt
- httpd debugging guide from Prefetch Technologies
  http://prefetch.net/articles/debuggingapache.html

In PDF, click on the title or cut and paste the URL.

More general information (all from Joyent?)

- The DTrace Book (http://www.dtracebook.com/index.php/Main_Page)
- DTrace one-liners from Brendan Gregg (http://www.brendangregg.com/DTrace/dtrace_oneliners.txt)
- “And It All Went Horribly Wrong...” talk from Bryan Cantrill (http://www.joyent.com/content/06-developers/01-resources/13-and-it-all-went-horribly-wrong-debugging-production-systems/debugging-production-systems.pdf)

In PDF, click on the title or cut and paste the URL.