1 Debugging mod_perl C Internals

1.1 Description

This document explains how to debug C code under mod_perl, including mod_perl core itself.

For certain debugging purposes you may find useful to read first the following notes on mod_perl internals: Apache 2.0 Integration and mod_perl-specific functionality flow.

1.2 Debug notes

META: needs more organization

META: there is a new directive CoreDumpDirectory in 2.0.45, need to check whether we should mention it.

META: there is a new compile-time option in perl-5.9.0+: -DDEBUG_LEAKING_SCALARS, which prints out the addresses of leaked SVs and new_SV() can be used to discover where those SVs were allocated. (see perlhack.pod for more info)

META: httpd has quite a lot of useful debug info: http://httpd.apache.org/dev/debugging.html (need to add this link to mp1 docs as well)

META: profiling: need a new entry of profiling. + running mod_perl under gprof: Defining GPROF when compiling uses the moncontrol() function to disable gprof profiling in the parent, and enable it only for request processing in children (or in one_process mode).

META: Jeff Trawick wrote a few useful debug modules, for httpd-2.1: mod_backtrace (similar to bt in gdb, but doesn't require the core file) and mod_whatkilledus (gives the info about the request that caused the segfault). http://httpd.apache.org/~trawick/exception_hook.html

1.2.1 Setting gdb breakpoints with mod_perl built as DSO

If mod_perl is built as a DSO module, you cannot set the breakpoint in the mod_perl source files when the *httpd* program gets loaded into the debugger. The reason is simple: At this moment *httpd* has no idea about mod_perl module yet. After the configuration file is processed and the mod_perl DSO module is loaded then the breakpoints in the source of mod_perl itself can be set.

The trick is to break at *apr_dso_load*, let it load *libmodperl.so*, then you can set breakpoints anywhere in the modperl code:

```
% gdb httpd
(gdb) b apr_dso_load
(gdb) run -DONE_PROCESS
[New Thread 1024 (LWP 1600)]
[Switching to Thread 1024 (LWP 1600)]
```

```
Breakpoint 1, apr_dso_load (res_handle=0xbfffb48c, path=0x811adcc
  "/home/stas/apache.org/modperl-perlmodule/src/modules/perl/libmodperl.so",
  pool=0x80e1a3c) at dso.c:138
141         void *os_handle = dlopen(path, RTLD_NOW | RTLD_GLOBAL);
(gdb) finish
...
Value returned is $1 = 0
(gdb) b modperl_hook_init
(gdb) continue
```

This example shows how to set a breakpoint at *modperl_hook_init*.

To automate things you can put those in the *.gdb-jump-to-init* file:

```
b apr_dso_load
run -DONE_PROCESS -d ‘pwd‘/t -f ‘pwd‘/t/conf/httpd.conf
finish
b modperl_hook_init
continue
```

and then start the debugger with:

```
% gdb /home/stas/httpd-2.0/bin/httpd -command \
‘pwd‘/t/.gdb-jump-to-init
```

1.2.2 Starting the Server Fast under gdb

When the server is started under gdb, it first loads the symbol tables of the dynamic libraries that it sees going to be used. Some versions of gdb may take ages to complete this task, which makes the debugging very irritating if you have to restart the server all the time and it doesn’t happen immediately.

The trick is to set the `auto-solib-add` flag to 0:

```
set auto-solib-add 0
```

as early as possible in *~/.gdbinit* file.

With this setting in effect, you can load only the needed dynamic libraries with *sharedlibrary* gdb command. Remember that in order to set a breakpoint and step through the code inside a certain dynamic library you have to load it first. For example consider this gdb commands file:

```
.gdb-commands
------------
file ~/httpd/prefork/bin/httpd
handle SIGPIPE pass
handle SIGPIPE nostop
set auto-solib-add 0
b ap_run_pre_config
run -d ‘pwd‘/t -f ‘pwd‘/t/conf/httpd.conf \
-DONE_PROCESS -DAPACHE2 -DPERL_USEITHREADS
sharedlibrary mod_perl
b modperl_hook_init
# start: modperl_hook_init
```

```
continue
# restart: ap_run_pre_config
continue
# restart: modperl_hook_init
continue
b apr_poll
continue

# load APR/PerlIO/PerlIO.so
sharedlibrary PerlIO
b PerlIOAPR_open
```

which can be used as:

```
% gdb -command=.gdb-commands
```

This script stops in *modperl_hook_init()*, so you can step through the mod_perl startup. We had to use the *ap_run_pre_config* so we can load the *libmodperl.so* library as explained earlier. Since httpd restarts on the start, we have to *continue* until we hit *modperl_hook_init* second time, where we can set the breakpoint at *apr_poll*, the very point where httpd polls for new request and run again *continue* so it'll stop at *apr_poll*. This particular script passes over modperl_hook_init(), since we run the `continue` command a few times to reach the *apr_poll* breakpoint. See the Precooked gdb Startup Scripts section for standalone script examples.

When gdb stops at the function *apr_poll* it's a time to start the client, that will issue a request that will exercise the server execution path we want to debug. For example to debug the implementation of `APR::Pool` we may run:

```
% t/TEST -run apr/pool
```

which will trigger the run of a handler in *t/response/TestAPR/pool.pm* which in turn tests the `APR::Pool` code.

But before that if we want to debug the server response we need to set breakpoints in the libraries we want to debug. For example if we want to debug the function `PerlIOAPR_open` which resides in *APR/PerlIO/PerlIO.so* we first load it and then we can set a breakpoint in it. Notice that gdb may not be able to load a library if it wasn't referenced by any of the code. In this case we have to load this library at the server startup. In our example we load:

```
PerlModule APR::PerlIO
```

in *httpd.conf*. To check which libraries' symbol tables can be loaded in gdb, run (when the server has been started):

```
gdb> info sharedlibrary
```

which also shows which libraries are loaded already.

Also notice that you don't have to type the full path of the library when trying to load them, even a partial name will suffice. In our commands file example we have used `sharedlibrary mod_perl` instead of saying `sharedlibrary mod_perl.so`.

If you want to set breakpoints and step through the code in the Perl and APR core libraries you should load their appropriate libraries:

```
gdb> sharedlibrary libperl
gdb> sharedlibrary libapr
gdb> sharedlibrary libaprutil
```

Setting *auto-solib-add* to 0 makes the debugging process unusual, since originally gdb was loading the dynamic libraries automatically, whereas now it doesn't. This is the price one has to pay to get the debugger starting the program very fast. Hopefully the future versions of gdb will improve.

Just remember that if you try to *step-in* and debugger doesn't do anything, that means that the library the function is located in wasn't loaded. The solution is to create a commands file as explained in the beginning of this section and craft the startup script the way you need to avoid extra typing and mistakes when repeating the same debugging process again and again.

Under threaded mpms (e.g. worker), it's possible that you won't be able to debug unless you tell gdb to load the symbols from the threads library. So for example if on your OS that library is called *libpthread.so* make sure to run:

```
sharedlibrary libpthread
```

somewhere after the program has started. See the Precooked gdb Startup Scripts section for examples.

Another important thing is that whenever you want to be able to see the source code for the code you are stepping through, the library or the executable you are in must have the debug symbols present. That means that the code has to be compiled with *-g* option for the gcc compiler. For example if I want to set a breakpoint in /lib/libc.so, I can do that by loading:

```
gdb> sharedlibrary /lib/libc.so
```

But most likely that this library has the debug symbols stripped off, so while gdb will be able to break at the breakpoint set inside this library, you won't be able to step through the code. In order to do so, recompile the library to add the debug symbols.

If debug code in response handler you usually start the client after the server was started, when doing this a lot you may find it annoying to need to wait before the client can be started. Therefore you can use a few tricks to do it in one command. If the server starts fast you can use sleep():

```
% ddd -command=.debug-modperl-init & ; \
sleep 2 ; t/TEST -verbose -run apr/pool
```

or the `Apache::Test` framework's `-ping=block` option:

```
% ddd -command=.debug-modperl-init & ; \
t/TEST -verbose -run -ping=block apr/pool
```

which will block till the server starts responding, and only then will try to run the test.

1.2.3 Precooked gdb Startup Scripts

Here are a few startup scripts you can use with gdb to accomplish one of the common debugging tasks. To execute the startup script, simply run:

```
% gdb -command=.debug-script-filename
```

They can be run under gdb and any of the gdb front-ends. For example to run the scripts under `ddd` substitute `gdb` with `ddd`:

```
% ddd -command=.debug-script-filename
```

- **Debugging mod_perl Initialization**

  The *code/.debug-modperl-init*:

```
# This gdb startup script breaks at the modperl_hook_init() function,
# which is useful for debug things at the modperl init phase.
#
# Invoke as:
# gdb -command=.debug-modperl-init
#
# see ADJUST notes for things that may need to be adjusted

# ADJUST: the path to the httpd executable if needed
file ~/httpd/worker/bin/httpd
handle SIGPIPE nostop
handle SIGPIPE pass
set auto-solib-add 0

define myrun
    tbreak main
    break ap_run_pre_config
    #  ADJUST: the httpd.conf file's path if needed
    #  ADJUST: add -DPERL_USEITHREADS to debug threaded mpms
    run -d `pwd`/t -f `pwd`/t/conf/httpd.conf -DONE_PROCESS -DAPACHE2
    continue
end

define modperl_init
    sharedlibrary mod_perl
    b modperl_hook_init
    continue
end

define sharedap
    # ADJUST: uncomment next line to debug threaded mpms
    #sharedlibrary libpthread
    sharedlibrary apr
    sharedlibrary aprutil
    #sharedlibrary mod_ssl.so
    continue
end

define sharedperl
```

```
    sharedlibrary libperl
end

# start the server and run till modperl_hook_init on start
myrun
modperl_init

# ADJUST: uncomment to reach modperl_hook_init on restart
#continue
#continue

# ADJUST: uncomment if you need to step through the code in apr libs
#sharedap

# ADJUST: uncomment if you need to step through the code in perlib
#sharedperl
```

startup script breaks at the `modperl_hook_init()` function, which is useful for debugging code at the modperl's initialization phase.

- **Debugging mod_perl's Hooks Registeration With httpd**

  Similar to the previous startup script, the *code/.debug-modperl-register*:

```
# This gdb startup script allows to break at the very first invocation
# of mod_perl initialization, just after it was loaded. When the
# perl_module is loaded, and its pointer struct is added via
# ap_add_module(), the first hook that will be called is
# modperl_register_hooks().
#
# Invoke as:
# gdb -command=.debug-modperl-register
#
# see ADJUST notes for things that may need to be adjusted

define sharedap
    sharedlibrary apr
    sharedlibrary aprutil
    #sharedlibrary mod_ssl.so
end

define sharedperl
    sharedlibrary libperl
end

### Run ###

# ADJUST: the path to the httpd executable if needed
file ~/httpd/prefork/bin/httpd
handle SIGPIPE nostop
handle SIGPIPE pass
set auto-solib-add 0

tbreak main

# assuming that mod_dso is compiled in
```

```
b load_module


# ADJUST: the httpd.conf file's path if needed
# ADJUST: add -DPERL_USEITHREADS to debug threaded mpms
run -d `pwd`/t -f `pwd`/t/conf/httpd.conf \
-DONE_PROCESS -DNO_DETACH -DAPACHE2

# skip over 'tbreak main'
continue

# In order to set the breakpoint in mod_perl.so, we need to get to
# the point where it's loaded.
#
# With static mod_perl, the bp can be set right away
#

# With DSO mod_perl, mod_dso's load_module() loads the mod_perl.so
# object and it immediately calls ap_add_module(), which calls
# modperl_register_hooks(). So if we want to bp at the latter, we need
# to stop at load_module(), set the 'bp modperl_register_hooks' and
# then continue.

# Assuming that 'LoadModule perl_module' is the first LoadModule
# directive in httpd.conf, you need just one 'continue' after
# 'ap_add_module'. If it's not the first one, you need to add as many
# 'continue' commands as the number of 'LoadModule foo' before
# perl_module, but before setting the 'ap_add_module' bp.
#
# If mod_perl is compiled statically, everything is already preloaded,
# so you can set modperl_* the breakpoints right away

b ap_add_module
continue

sharedlibrary mod_perl
b modperl_register_hooks
continue

#b modperl_hook_init
#b modperl_config_srv_create
#b modperl_startup
#b modperl_init_vhost
#b modperl_dir_config
#b modperl_cmd_load_module
#modperl_config_apply_PerlModule

# ADJUST: uncomment next line to debug threaded mpms
#sharedlibrary libpthread

# ADJUST: uncomment if you need to step through the code in apr libs
#sharedap

# ADJUST: uncomment if you need to step through the code in perlib
#sharedperl
```

startup script breaks at the `modperl_register_hooks()`, which is the very first hook called in the mod_perl land. Therefore use this one if you need to start debugging at an even earlier entry point into mod_perl.

Refer to the notes inside the script to adjust it for a specific *httpd.conf* file.

- **Debugging mod_perl XS Extensions**

  The *code/.debug-modperl-xs*:

```
# This gdb startup script breaks at the mpxs_Apache__Filter_print()
# function from the XS code, as an example how you can debug the code
# in XS extensions.
#
# Invoke as:
# gdb -command=.debug-modperl-xs
# and then run:
# t/TEST -v -run -ping=block filter/api
#
# see ADJUST notes for things that may need to be adjusted

# ADJUST: the path to the httpd executable if needed
file /home/stas/httpd/worker/bin/httpd
handle SIGPIPE nostop
handle SIGPIPE pass
set auto-solib-add 0

define myrun
    tbreak main
    break ap_run_pre_config
    # ADJUST: the httpd.conf file's path if needed
    # ADJUST: add -DPERL_USEITHREADS to debug threaded mpms
    run -d `pwd`/t -f `pwd`/t/conf/httpd.conf \
    -DONE_PROCESS -DNO_DETACH -DAPACHE2
    continue
end

define sharedap
    # ADJUST: uncomment next line to debug threaded mpms
    #sharedlibrary libpthread
    sharedlibrary apr
    sharedlibrary aprutil
    #sharedlibrary mod_ssl.so
    continue
end

define sharedperl
    sharedlibrary libperl
end

define gopoll
    b apr_poll
    continue
```

```
    continue
end

define mybp
    # load Apache/Filter.so
    sharedlibrary Filter
    b mpxs_Apache__Filter_print
    # no longer needed and they just make debugging harder under threads
    disable 2
    disable 3
    continue
end

myrun
gopoll
mybp

# ADJUST: uncomment if you need to step through the code in apr libs
#sharedap

# ADJUST: uncomment if you need to step through the code in perlib
#sharedperl
```

startup script breaks at the `mpxs_Apache__Filter_print()` function implemented in *xs/Apache/Filter/Apache__Filter.h*. This is an example of debugging code in XS Extensions. For this particular example the complete test case is:

```
% ddd -command=.debug-modperl-xs & \
t/TEST -v -run -ping=block filter/api
```

When *filter/api* test is running it calls mpxs_Apache__Filter_print() which is when the breakpoint is reached.

- **Debugging code in shared objects created by `Inline.pm`**

  This is not strictly related to mod_perl, but sometimes when trying to reproduce a problem (e.g. for a p5p bug-report) outside mod_perl, the code has to be written in C. And in certain cases, Inline can be just the right tool to do it quickly. However if you want to interactively debug the library that it creates, it might get tricky. So similar to the previous sections, here is a gdb *code/.debug-inline*:

```
# save this file as .debug and execute this as:
# gdb -command=.debug
# or if you prefer gui
# ddd -command=.debug
#
# NOTE: Adjust the path to the perl executable
# also this perl should be built with debug enabled
file /usr/bin/perl

# If you need to debug with gdb a live script and not a library, you
# are going to have a hard time to set any breakpoint in the C code.
# the workaround is force Inline to compile and load .so, by putting
# all the code in the BEGIN {} block and call Inline->init from there.
#
# you also need to prevent from Inline deleting autogenerated .xs so
```

```
# you can step through the C source code, and of course you need to
# add '-g' so .so won't be stripped of debug info
#
# here is a sample perl script that can be used with this gdb script
#
# test.pl
# #-----#
# use strict;
# use warnings;
#
# BEGIN {
#     use Inline Config =>
#         #FORCE_BUILD => 1,
#         CLEAN_AFTER_BUILD => 0;
#
#     use Inline C => Config =>
#         OPTIMIZE => '-g';
#
#     use Inline C => <init;
#
# }
#
# my_bp();

tb main
# NOTE: adjust the name of the script that you run
run test.pl

# when Perl_runops_debug breakpoint is hit Inline will already load
# the autogenerated .so, so we can set the bp in it (that's only if
# you have run 'Inline->init' inside the BEGIN {} block

b S_run_body
continue
b Perl_runops_debug
continue

# here you set your breakpoints
b my_bp
continue
```

startup script that will save you a lot of time. All the details and a sample perl script are inside the gdb script.

1.3 Analyzing Dumped Core Files

META: need to review (unfinished)

When your application dies with the *Segmentation fault* error (which generates a `SIGSEGV` signal) and optionally generates a *core* file you can use `gdb` or a similar debugger to find out what caused the *Segmentation fault* (or *segfault* as we often call it).

1.3.1 Getting Ready to Debug

In order to debug the *core* file we may need to recompile Perl and mod_perl with debugging symbols inside. Usually you have to recompile only mod_perl, but if the *core* dump happens in the *libmodperl.so* library and you want to see the whole backtrace, you probably want to recompile Perl as well.

Recompile Perl with *-DDEBUGGING* during the ./Configure stage (or even better with *-Doptimize="-g"* which in addition to adding the `-DDEBUGGING` option, adds the *-g* options which allows you to debug the Perl interpreter itself).

After recompiling Perl, recompile mod_perl with `MP_DEBUG=1` during the *Makefile.PL* stage.

Building mod_perl with `PERL_DEBUG=1` will:

1. add '-g' to EXTRA_CFLAGS
2. turn on MP_TRACE (tracing)
3. Set PERL_DESTRUCT_LEVEL=2
4. Link against `libperld` if -e $Config{archlibexp}/CORE/libperld$Config{lib_ext}

If you build a static mod_perl, remember that during *make install* Apache strips all the debugging symbols. To prevent this you should use the Apache *--without-execstrip* `./configure` option. So if you configure Apache via mod_perl, you should do:

```
panic% perl Makefile.PL USE_APACI=1 \
  APACI_ARGS='--without-execstrip' [other options]
```

Alternatively you can copy the unstripped binary manually. For example we did this to give us an Apache binary called `httpd_perl` which contains debugging symbols:

```
panic# cp httpd-2.x/httpd /home/httpd/httpd_perl/bin/httpd_perl
```

Now the software is ready for a proper debug.

1.3.2 Creating a Faulty Package

META: no longer need to create the package, use `Debug::DumpCore` from CPAN. Need to adjust the rest of the document to use it.

Next stage is to create a package that aborts abnormally with the *Segmentation fault* error. We will write faulty code on purpose, so you will be able to reproduce the problem and exercise the debugging technique explained here. Of course in a real case you will have some real bug to debug, so in that case you may want to skip this stage of writing a program with a deliberate bug.

We will use the `Inline.pm` module to embed some code written in C into our Perl script. The faulty function that we will add is this:

```
void segv() {
    int *p;
    p = NULL;
    printf("%d", *p); /* cause a segfault */
}
```

For those of you not familiar with C programming, *p* is a pointer to a segment of memory. Setting it to `NULL` ensures that we try to read from a segment of memory to which the operating system does not allow us access, so of course dereferencing `NULL` pointer causes a segmentation fault. And that's what we want.

So let's create the `Bad::Segv` package. The name *Segv* comes from the `SIGSEGV` (segmentation violation signal) that is generated when the *Segmentation fault* occurs.

First we create the installation sources:

```
panic% h2xs -n Bad::Segv -A -O -X
Writing Bad/Segv/Segv.pm
Writing Bad/Segv/Makefile.PL
Writing Bad/Segv/test.pl
Writing Bad/Segv/Changes
Writing Bad/Segv/MANIFEST
```

Now we modify the *Segv.pm* file to include the C code. Afterwards it looks like this:

```
package Bad::Segv;

use strict;
BEGIN {
    $Bad::Segv::VERSION = '0.01';
}

use Inline C => <<'END_OF_C_CODE';
  void segv() {
      int *p;
      p = NULL;
      printf("%d", *p); /* cause a segfault */
  }

END_OF_C_CODE

1;
```

Finally we modify *test.pl*:

```
use Inline SITE_INSTALL;

BEGIN { $| = 1; print "1..1\n"; }
END {print "not ok 1\n" unless $loaded;}
use Bad::Segv;

$loaded = 1;
print "ok 1\n";
```

Note that we don't test Bad::Segv::segv() in *test.pl*, since this will abort the *make test* stage abnormally, and we don't want this.

Now we build and install the package:

```
panic% perl Makefile.PL
panic% make && make test
panic% su
panic# make install
```

Running *make test* is essential for `Inline.pm` to prepare the binary object for the installation during *make install*.

META: stopped here!

Now we can test the package:

```
panic% ulimit -c unlimited
panic% perl -MBad::Segv -e 'Bad::Segv::segv()'
Segmentation fault (core dumped)
panic% ls -l core
-rw-------  1  stas stas 1359872  Feb 6  14:08 core
```

Indeed, we can see that the *core* file was dumped, which will be used to present the debug techniques.

1.3.3 Getting the core File Dumped

Now let's get the *core* file dumped from within the mod_perl server. Sometimes the program aborts abnormally via the SIGSEGV signal (*Segmentation Fault*), but no *core* file is dumped. And without the *core* file it's hard to find the cause of the problem, unless you run the program inside `gdb` or another debugger in first place. In order to get the *core* file, the application has to:

- have the effective UID the same as real UID (the same goes for GID). Which is the case of mod_perl unless you modify these settings in the program.

- be running from a directory which at the moment of the *Segmentation fault* is writable by the process. Notice that the program might change its current directory during its run, so it's possible that the *core* file will need to be dumped in a different directory from the one the program was started from. For example when mod_perl runs an `Apache::Registry` script it changes its directory to the one in which the script source is located.

- be started from a shell process with sufficient resource allocations for the *core* file to be dumped. You can override the default setting from within a shell script if the process is not started manually. In addition you can use `BSD::Resource` to manipulate the setting from within the code as well.

  You can use `ulimit` for `bash` and `limit` for `csh` to check and adjust the resource allocation. For example inside `bash`, you may set the core file size to unlimited:

```
panic% ulimit -c unlimited
```

or for `csh`:

```
panic% limit coredumpsize unlimited
```

For example you can set an upper limit on the *core* file size to 8MB with:

```
panic% ulimit -c 8388608
```

So if the core file is bigger than 8MB it will be not created.

- Of course you have to make sure that you have enough disk space to create a big core file (mod_perl *core* files tend to be of a few MB in size).

Note that when you are running the program under a debugger like `gdb`, which traps the `SIGSEGV` signal, the *core* file will not be dumped. Instead it allows you to examine the program stack and other things without having the *core* file.

So let's write a simple script that uses `Bad::Segv`:

```
core_dump.pl
------------
use strict;
use Bad::Segv ();
use Cwd()

my $r = shift;
$r->content_type('text/plain');

my $dir = getcwd;
$r->print("The core should be found at $dir/core\n");
Bad::Segv::segv();
```

In this script we load the `Bad::Segv` and `Cwd` modules. After that we acquire the request object and send the HTTP header. Now we come to the real part--we get the current working directory, print out the location of the *core* file that we are about to dump and finally we call Bad::Segv::segv() which dumps the *core* file.

Before we run the script we make sure that the shell sets the *core* file size to be unlimited, start the server in single server mode as a non-root user and generate a request to the script:

```
panic% cd /home/httpd/httpd_perl/bin
panic% limit coredumpsize unlimited
panic% ./httpd_perl -X
    # issue a request here
Segmentation fault (core dumped)
```

Our browser prints out:

```
The core should be found at /home/httpd/perl/core
```

And indeed the core file appears where we were told it will (remember that `Apache::Registry` scripts change their directory to the location of the script source):

```
panic% ls -l /home/httpd/perl/core
-rw------- 1 stas httpd 3227648 Feb 7 18:53 /home/httpd/perl/core
```

As you can see it's a 3MB *core* file. Notice that mod_perl was started as user *stas*, which had write permission for directory */home/httpd/perl*.

1.3.4 Analyzing the core File

First we start `gdb`:

```
panic% gdb /home/httpd/httpd_perl/bin/httpd_perl /home/httpd/perl/core
```

with the location of the mod_perl executable and the core file as the arguments.

To see the backtrace you run the *where* or the *bt* command:

```
(gdb) where
#0  0x4025ea08 in XS_Apache__Segv_segv ()
   from /usr/lib/perl5/site_perl/5.6.0/i386-linux/auto/Bad/Segv_C_0_01_e6b5959d800f515de36a7e7eeab28b39/Segv_C_0_01_e6b5959d800f515de36a7e7eeab28b39.so
#1  0x40136528 in PL_curcopdb ()
   from /usr/lib/perl5/5.6.0/i386-linux/CORE/libperl.so
```

Well, you can see the last commands, but our perl and mod_perl are probably without the debug symbols. So we recompile Perl and mod_perl with debug symbols as explained earlier in this chapter.

Now when we repeat the process of starting the server, issuing a request and getting the core file, after which we run `gdb` again against the executable and the dumped *core* file.

```
panic% gdb /home/httpd/httpd_perl/bin/httpd_perl /home/httpd/perl/core
```

Now we can see the whole backtrace:

```
(gdb) bt
#0  0x40323a30 in segv () at Segv_C_0_01_e6b5959d800f515de36a7e7eeab28b39.xs:9
#1  0x40323af8 in XS_Apache__Segv_segv (cv=0x85f2b28)
    at Segv_C_0_01_e6b5959d800f515de36a7e7eeab28b39.xs:24
#2  0x400fcbda in Perl_pp_entersub () at pp_hot.c:2615
#3  0x400f2c56 in Perl_runops_debug () at run.c:53
#4  0x4008b088 in S_call_body (myop=0xbffff788, is_eval=0) at perl.c:1796
#5  0x4008ac4f in perl_call_sv (sv=0x82fc2e4, flags=4) at perl.c:1714
#6  0x807350e in perl_call_handler ()
#7  0x80729cd in perl_run_stacked_handlers ()
#8  0x80701b4 in perl_handler ()
#9  0x809f409 in ap_invoke_handler ()
#10 0x80b3e8f in ap_some_auth_required ()
#11 0x80b3efa in ap_process_request ()
#12 0x80aae60 in ap_child_terminate ()
#13 0x80ab021 in ap_child_terminate ()
```

```
#14 0x80ab19c in ap_child_terminate ()
#15 0x80ab80c in ap_child_terminate ()
#16 0x80ac03c in main ()
#17 0x401b8cbe in __libc_start_main () from /lib/libc.so.6
```

Reading the trace from bottom to top, we can see that it starts with Apache calls, followed by Perl syscalls. At the top we can see the segv() call which was the one that caused the Segmentation fault, we can also see that the faulty code was at line 9 of *Segv.xs* file (with MD5 signature of the code in the name of the file, because of the way `Inline.pm` works). It's a little bit tricky with `Inline.pm` since we have never created any *.xs* files ourselves, (`Inline.pm` does it behind the scenes). The solution in this case is to tell `Inline.pm` not to cleanup the build directory, so we can see the created *.xs* file.

We go back to the directory with the source of `Bad::Segv` and force the recompilation, while telling `Inline.pm` not to cleanup after the build and to print a lot of other useful info:

```
panic# cd Bad/Segv
panic# perl -MInline=FORCE,NOCLEAN,INFO Segv.pm
Information about the processing of your Inline C code:

Your module is already compiled. It is located at:
/home/httpd/perl/Bad/Segv/_Inline/lib/auto/Bad/Segv_C_0_01_e6b5959d800f515de36a7e7eeab28b39/Segv_C_0_01_e6b5959d800f515de36a7e7eeab28b39.so

But the FORCE_BUILD option is set, so your code will be recompiled.
I'll use this build directory:
/home/httpd/perl/Bad/Segv/_Inline/build/Bad/Segv_C_0_01_e6b5959d800f515de36a7e7eeab28b39/

and I'll install the executable as:
/home/httpd/perl/Bad/Segv/_Inline/lib/auto/Bad/Segv_C_0_01_e6b5959d800f515de36a7e7eeab28b39/Segv_C_0_01_e6b5959d800f515de36a7e7eeab28b39.so

The following Inline C function(s) have been successfully bound to Perl:
  void segv()
```

It tells us that the code was already compiled, but since we have forced it to recompile we can look at the files after the build. So we go into the build directory reported by `Inline.pm` and find the *.xs* file there, where on line 9 we indeed find the faulty code:

```
9: printf("%d",*p); // cause a segfault
```

Notice that in our example we knew what script has caused the Segmentation fault. In a real world the chances are that you will find the *core* file without any clue to which of handler or script has triggered it. The special *curinfo* `gdb` macro comes to help:

```
panic% gdb /home/httpd/httpd_perl/bin/httpd_perl /home/httpd/perl/core
(gdb) source mod_perl-x.xx/.gdbinit
(gdb) curinfo
9:/home/httpd/perl/core_dump.pl
```

We start the `gdb` debugger as before. *.gdbinit*, the file with various useful `gdb` macros is located in the source tree of mod_perl. We use the `gdb` source() function to load these macros, and when we run the *curinfo* macro we learn that the core was dumped when */home/httpd/perl/core_dump.pl* was executing the code at line 9.

These are the bits of information that are important in order to reproduce and resolve a problem: the filename and line where the faulty function was called (the faulty function is Bad::Segv::segv() in our case) and the actual line where the Segementation fault occured (the printf("%d",*p) call in XS code). The former is important for problem reproducing, it's possible that if the same function was called from a different script the problem won't show up (not the case in our example, where the using of a value dereferenced from the NULL pointer will always cause the Segmentation fault).

1.3.5 Obtaining core Files under Solaris

There are two ways to get core files under Solaris. The first is by configuring the system to allow core dumps, the second is by stopping the process when it receives the SIGSEGV signal and "manually" obtaining the core file.

1.3.5.1 Configuring Solaris to Allow core Dumps

By default, Solaris 8 won't allow a setuid process to write a core file to the file system. Since apache starts as root and spawns children as 'nobody', core dumps won't produce core files unless you modify the system settings.

To see the current settings, run the coreadm command with no parameters and you'll see:

```
% coreadm
     global core file pattern:
       init core file pattern: core
            global core dumps: disabled
       per-process core dumps: enabled
      global setid core dumps: disabled
per-process setid core dumps: disabled
     global core dump logging: disabled
```

These settings are stored in the */etc/coreadm.conf* file, but you should set them with the coreadm utility. As super-user, you can run coreadm with -g to set the pattern and path for core files (you can use a few variables here) and -e to enable some of the disabled items. After setting a new pattern, enabling global, global-setid, and log, and rebooting the system (reboot is required), the new settings look like:

```
% coreadm
     global core file pattern: /usr/local/apache/cores/core.%f.%p
       init core file pattern: core
            global core dumps: enabled
       per-process core dumps: enabled
      global setid core dumps: enabled
per-process setid core dumps: disabled
     global core dump logging: enabled
```

Now you'll start to see core files in the designated cores directory and they will look like *core.httpd.2222* where httpd is the name of the executable and the 2222 is the process id. The new core files will be read/write for root only to maintain some security, and you should probably do this on development systems only.

1.3.5.2 Manually Obtaining core Dumps

On Solaris the following method can be used to generate a core file.

1. Use truss(1) as *root* to stop a process on a segfault:

```
panic% truss -f -l -t \!all -s \!SIGALRM -S SIGSEGV -p <pid>
```

or, to monitor all httpd processes (from bash):

```
panic% for pid in ‘ps -eaf -o pid,comm | fgrep httpd | cut -d’/’ -f1‘;
do truss -f -l -t \!all -s \!SIGALRM -S SIGSEGV -p $pid 2>&1 &
done
```

The used truss(1) options are:

- `-f` - follow forks.
- `-l` - (that’s an el) includes the thread-id and the pid (the pid is what we want).
- `-t` - specifies the syscalls to trace,
- !all - turns off the tracing of syscalls specified by `-t`
- `-s` - specifies signals to trace and the `!SIGALRM` turns off the numerous alarms Apache creates.
- `-S` - specifies signals that stop the process.
- `-p` - is used to specify the pid.

Instead of attaching to the process, you can start it under truss(1):

```
panic% truss -f -l -t \!all -s \!SIGALRM -S SIGSEGV \
       /usr/local/bin/httpd -f httpd.conf 2>&1 &
```

2. Watch the *error_log* file for reaped processes, as when they get SISSEGV signals. When the process is reaped it’s stopped but not killed.
3. Use gcore(1) to get a *core* of stopped process or attach to it with gdb(1). For example if the process id is 662:

   ```
   %panic gcore 662
   gcore: core.662 dumped
   ```

   Now you can load this *core* file in gdb(1).
4. `kill -9` the stopped process. Kill the truss(1) processes as well, if you don’t need to trap other segfaults.

Obviously, this isn’t great to be doing on a production system since truss(1) stops the process after it dumps core and prevents Apache from reaping it. So, you could hit the clients/threads limit if you segfault a lot.

1.4 Debugging Threaded MPMs

1.4.1 Useful Information from gdb Manual

Debugging programs with multiple threads: http://sources.redhat.com/gdb/current/online-docs/gdb_5.html#SEC25

Stopping and starting multi-thread programs: http://sources.redhat.com/gdb/current/online-docs/gdb_6.html#SEC40

1.4.2 libpthread

when using:

```
set auto-solib-add 0
```

make sure to:

```
sharedlibrary libpthread
```

(or whatever the shared library is used on your OS) without which you may have problems to debug the threaded mpm mod_perl.

1.5 Defining and Using Custom gdb Macros

GDB provides two ways to store sequences of commands for execution as a unit: user-defined commands and command files. See: http://sources.redhat.com/gdb/current/onlinedocs/gdb_21.html

Apache 2.0 source comes with a nice pack of macros and can be found in *httpd-2.0/.gdbinit*. To use it issue:

```
gdb> source /wherever/httpd-2.0/.gdbinit
```

Now if for example you want to dump the contents of the bucket brigade, you can do:

```
gdb> dump_brigade my_brigade
```

where `my_brigade` is the pointer to the bucket brigade that you want to debug.

mod_perl 1.0 has a similar file (*modperl/.gdbinit*) mainly including handy macros for dumping Perl datastructures, however it works only with non-threaded Perls. But otherwise it's useful in debugging mod_perl 2.0 as well.

1.6 Expanding C Macros

Perl, mod_perl and httpd C code makes an extensive use of C macros, which sometimes use many other macros in their definitions, so it becomes quite a task to figure out how to figure out what a certain macro expands to, especially when the macro expands to different values in differnt environments. Luckily there are ways to automate the expansion process.

1.6.1 Expanding C Macros with `make`

The mod_perl *Makefile*'s include a rule for macro expansions which you can find by looking for the `c.i.` rule. To expand all macros in a certain C file, you should run `make filename.i`, which will create *filename.i* with all macros expanded in it. For example to create *apr_perlio.i* with all macros used in *apr_perlio.c*:

```
% cd modperl-2.0/xs/APR/PerlIO
% make apr_perlio.i
```

the *apr_perlio.i* file now lists all the macros:

```
% less apr_perlio.i
# 1 "apr_perlio.c"
# 1 "<built-in>"
#define __VERSION__ "3.1.1 (Mandrake Linux 8.3 3.1.1-0.4mdk)"
...
```

1.6.2 Expanding C Macros with `gdb`

With gcc-3.1 or higher and gdb-5.2-dev or higher you can expand macros in gdb, when you step through the code. e.g.:

```
(gdb) macro expand pTHX_
expands to:  PerlInterpreter *my_perl __attribute__((unused)),
(gdb) macro expand PL_dirty
expands to: (*Perl_Tdirty_ptr(my_perl))
```

For each library that you want to use this feature with you have to compile it with:

```
CFLAGS="-gdwarf-2 -g3"
```

or whatever is appropriate for your system, refer to the gcc manpage for more info.

To compile perl with this debug feature, pass `-Doptimize='-gdwarf-2 -g3'` to `./Configure`. For Apache run:

```
CFLAGS="-gdwarf-2 -g3" ./configure [...]
```

for mod_perl you don't have to do anything, as it'll pick the `$Config{optimize}` Perl flags automatically, if Perl is compiled with `-DDEBUGGING` (which is implied on most systems, if you use `-Doptimize='-g'` or similar.)

Notice that this will make your libraries **huge**! e.g. on Linux 2.4 Perl 5.8.0's normal *libperl.so* is about 0.8MB on linux, compiled with `-Doptimize='-g'` about 2.7MB and with `-Doptimize='-gdwarf-2 -g3'` 12.5MB. `httpd` is also becomes about 10 times bigger with this feature enabled. *mod_perl.so* instead of 0.2k becomes 11MB. You get the idea. Of course since you may want this only during the development/debugging, that shouldn't be a problem.

The complete details are at: http://sources.redhat.com/gdb/current/onlinedocs/gdb_10.html#SEC69

1.7 Maintainers

Maintainer is the person(s) you should contact with updates, corrections and patches.

Stas Bekman <stas (at) stason.org>

1.8 Authors

- Stas Bekman <stas (at) stason.org>

Only the major authors are listed above. For contributors see the Changes file.

Table of Contents: