Filing useful bug reports

This is a page that we hope will help you file useful bug reports. This is always going to be work in progress, please make changes and additions as necessary.

Basic information

Before filing a new bug report, please spend a minute or two to see if there are already a bug filed identical or similar to your problem. You are also highly encouraged to reproduce and test your problem with the latest development release, or even better, a build from SVN trunk. With that said, here are a few bullets on information that each new bug should include:

- Version of Traffic Server used
- Platform (Linux, 64-bit, compilers used etc.)
- Any relevant configuration changes you've made from the default configurations (particularly for records.config).
- If possible, how to reproduce the bug.

Non-crashing problems

For these types of bugs, where Traffic Server isn't behaving as you expect, it's vital that you document the following:

- How to reproduce the behavior, this includes full client headers, server headers (from the origin) etc.
- What the expected result should be.
- What the actual result is from Traffic Server

Crashers

Crashers are difficult to debug, and some familiarity with the Gnu Debugger (gdb) is required to file any useful bug reports. We'll try to give a brief overview here how to do this, but you are encouraged to do some googling, and find some good on-line introductions to using gdb. The goal with GDB is two folded:

1. Get a stack trace from the crash. This is done using the **bt** command in gdb.
2. Collect information about parameter and other variables near the crasher. This is done using the **print** command in gdb.

Building Traffic Server for optimal debugging

The default build, aka a release build, of ATS has debug information, but it's also an optimized build. This makes it more difficult to debug using such a binary, because code gets reorganized, variables and parameters gets optimized out, and so on. For best debugging, you should build Traffic Server in a debug-build mode. For example:

```
$ ./configure --enable-debug
$ gmake
```

Demangle C++ symbols

When you get a stack trace (from gdb or the internal trace messages), please demangle the symbols by filtering it through the Unix command line utility ***c++filt***. This makes the crash reports much easier to analyze, and provides a consistent format in all bug reports (so we can search and cross reference).

Debugging memory related crashers

The environment variable MALLOC_CHECK_ can be useful in debugging certain malloc/free related bugs, such as double free (see: http://www.gnu.org/software/libc/manual/html_node/Heap-Consistency-Checking.html). Setting MALLOC_CHECK to 2 would cause an immediate abort and when properly configured to dump core this can be a useful mechanism for debugging these bugs.

Setting up GDB

For best use of GDB, the following commands should be placed in your .gdbinit config, or done manually after you start gdb:

```
set pagination 0
handle SIGPIPE nopass nostop noprint
```

The second configuration is absolutely vital, or you will like see SIGPIPE errors in write() while debugging, and these are not "crashers" even though they look like it.

Starting Traffic Server directly in gdb

The easiest way to debug ATS is simply starting it directly under gdb. For example:

```
$ sudo gdb bin/traffic_server
...
Reading symbols from /usr/local/bin/traffic_server...done.
(gdb) run
```

This will start up traffic_server inside gdb. You might get warnings telling you to install more detailed debug information, if so, you are encouraged to do so (just follow your Linux distributions instructions).

Attaching gdb to a running Traffic Server instance

In a production environment, where you want to start Traffic Server in the normal way, you must attach gdb to the already running traffic_server process. This is easy, you just need to find the PID of the running traffic_server process (e.g. 12345), and then tell gdb to attach to that PID. For example:

```
$ sudo gdb /usr/local/bin/traffic_server 12345
Loaded symbols for /lib64/libselinux.so.1
0x0000003f2dee1293 in epoll_wait () at ../sysdeps/unix/syscall-template.S:82
82    T_PSEUDO (SYSCALL_SYMBOL, SYSCALL_NAME, SYSCALL_NARGS)
(gdb) continue
Continuing.
```

It's important here that you either use a **.gdbinit** as above, or manually enter the required commands (see above), before you enter the **continue** command.

Debugging a core file

With some luck, you might have gotten a core file from a running ATS system. If so, you can debug traffic_server without affecting the production system, and simply attach GDB to that core file. For example:

```
$ sudo gdb /usr/local/bin/traffic_server /tmp/core.12345
...
(gdb) bt
```

Getting core files

Getting a core file can sometimes be a bit complicated, and even when properly setup, you might not always get one. This section will explain for various platforms how to properly setup your system to allow it to generate core files.

For all systems, set the proxy.config.stack_dump_enabled config value to 0 in records.config. It is 1 by default which means that the traffic_server process will grab the SIGSEGV signal and print a stack trace rather than dumping the core. You must set this configuration variable to 0, so the traffic server process allows the signal to pass on and dump the core.

```
CONFIG proxy.config.stack_dump_enabled INT 0
```

Linux

For an application to be able to generate a core file, it must have write permission to the directory where it should dump the core. This is by default $PWD, and for Traffic Server, this is not a directory that is typically writeable. Therefore, you should tell the kernel to generate core files in a different directory. The following example shows how to tell the system to generate core files like `/tmp/core.12345`:

```
$ sudo sysctl kernel.core_pattern=/tmp/core.%p
```

Additional parameters are available for naming, see `man 5 core` or here.

On older Linux systems you may need to use the following instead.

```
$ sudo sysctl kernel.core_uses_pid=1
$ sudo sysctl kernel.core_pattern=/tmp/core
```

You can add this to your `/etc/sysctl.conf` file. For more information please see the man page for core (i.e. `man 5 core`). Note that if you are currentlying using ABRT this will effectively disable that feature. You may instead be able to access the crash through ABRT rather than changing the core pattern.

In addition to being able to write to a directory, there are also a couple of resource limits that must be increased. Before starting traffic_server you must make sure that the process has unlimited core file size, and file size. You have a few options here, including modifying root's hard limits via `/etc/security/limits.conf` (but be aware that affects all of root's processes). Easiest is to do it from command line, e.g.

```
$ sudo -s
# ulimit -c unlimited
# ulimit -f unlimited
# /usr/local/bin/trafficserver start
```

If your traffic_server is started by root (via `init` or similar mechansim) it will call setuid/setgid. For Traffic Server versions before 3.1.1 (see TS-936) this will prevent a core file from being generated. To work around this you should set `/proc/sys/fs/suid_dumpable` to `1`.

```
# echo 1 > /proc/sys/fs/suid_dumpable
```

Note that this is a global setting with security implications (as a setuid program may contain sensitive information which ends up in the core file).