Debugging across pipes and sockets with `strace`

Niklas Hambüchen, *FP Complete*

mail@nh2.me

niklas@fpcomplete.com

Scenario

Privileged-server / unprivileged-client

- Server that runs restricted set of commands (`ls` , `dmesg`) sent to it via a socket as root
- Client that sends `argv[0]` to the server and prints the output

Problem

- `ls` case works fine
- `dmesg` case just hangs without output

Further complication

- Assume this is hard to reproduce (only happens every 1000th time), so you really want to debug it on the currently hanging system, and not restart any processes.

Code

Command server `command-server.py`:

```
#!/usr/bin/env python2

from __future__ import print_function
import socket
import subprocess
from subprocess import PIPE

serversocket = socket.socket(socket.AF_INET, socket.SOCK_
serversocket.setsockopt(socket.SOL_SOCKET, socket.SO_REUS
serversocket.bind(("localhost", 1234))
serversocket.listen(5)
```

`command-server.py` continued:

```
def run_command_for_client(command, clientsocket):
  if command in ["ls", "dmesg"]:
    p = subprocess.Popen([command], stdout=PIPE)
    p.wait()
    out = p.stdout.read()
  else:
    out = b"command not allowed\n"
  clientsocket.sendall(out)

# Server loop
while True:
  (clientsocket, address) = serversocket.accept()

  command = clientsocket.recv(100).decode('utf-8')
  print("server got command: " + command)

  run_command_for_client(command, clientsocket)

  clientsocket.close()
```

Command client `command-client.py`:

```
#!/usr/bin/env python2

from __future__ import print_function
import socket
import sys

name = sys.argv[1]

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(("localhost", 1234))

sock.sendall(name.encode('utf-8'))

while True:
  data = sock.recv(100)
  if len(data) == 0:
    break
  sys.stdout.write(data)
```

Command wrappers:

`unprivileged-ls.py`

```
#!/usr/bin/env python2

from __future__ import print_function
import subprocess
from subprocess import PIPE
import sys


p = subprocess.Popen(['./command-client.py', "ls"],
                     stdout=PIPE)

sys.stdout.write(p.communicate()[0])
```

`unprivileged-dmesg.py` (same thing with `dmesg`)

```
....
p = subprocess.Popen(['./command-client.py', "dmesg"], ..
```

Program communication

```
./unprivileged-dmesg.py
    | |
    | | stdout pipe
    | |
./command-client.py
     |
     |  TCP socket
     |
./command-server.py
    | |
    | | stdout pipe
    | |
dmesg executable
```

Outputs

```
% ./command-server.py
server got command: ls
server got command: dmesg
```

```
% ./unprivileged-ls.py
command-client.py
command-server.py
unprivileged-dmesg.py
unprivileged-ls.py
```

```
% ./unprivileged-dmesg.py
[hangs]
```

> 95% of computer problems can be solved with `strace`.

me

strace part

`strace` (on the process that hangs)

```
% sudo strace -fp "$(pgrep -f 'unprivileged-dmesg')"
strace: Process 23572 attached
read(3,
```

Add `-y` (*prints paths associated with file descriptor arguments*):

```
% sudo strace -fp "$(pgrep -f 'unprivileged-dmesg')" -y
strace: Process 23572 attached
read(3<pipe:[139828372]>,
```

The pipe number it's trying to read from is `139828372`.

Let's chase down that pipe.

Chasing pipes with `lsof`

```
% lsof -n -P | grep --color '139828372'
COMMAND PID   .. FD TYPE DEVICE .. NODE NAME
python2 23572 .. 3r FIFO 0,12   .. 139828372 pipe
python2 23573 .. 1w FIFO 0,12   .. 139828372 pipe
```

`3r` means process `23572` has a read-end of the pipe open as file descriptior `3`. See:

```
% ls -l /proc/23572/fd/3
...             /proc/23572/fd/3 -> pipe:[139828372]
```

`1w` means process `23573` has a write-end of the pipe open as file descriptior `1`.

So the only possible producer to unblock our `read(3,` in `strace` is process is process `23573`.

Let's `strace` the process that has the wipe write end:

```
% sudo strace -fp 23573 -y
strace: Process 23573 attached
recvfrom(3<socket:[139825870]>,
```

It's blocked reading from a socket.

Let's chase down that socket.

Chasing sockets with `lsof`

```
% lsof -n -P | grep --color '139825870'
COMMAND   PID .. FD TYPE    DEVICE .. NODE NAME
python2 23573 .. 3u IPv4 139825870 .. TCP 127.0.0.1:39392
```

`TCP 127.0.0.1:39392->127.0.0.1:1234 (ESTABLISHED)`

Follow that TCP connection (potentially on a another machine, in our case `127.0.0.1` is the same machine):

```
% lsof -n -P | grep --color '\b1234\b'
COMMAND   PID    NODE NAME
python2 23270 .. TCP 127.0.0.1:1234 (LISTEN)
python2 23270 .. TCP 127.0.0.1:1234->127.0.0.1:39392 (EST
python2 23573 .. TCP 127.0.0.1:39392->127.0.0.1:1234 (EST
```

So process `23270` has the other end of that socket.

Let's `strace` it.

General appraoch

Find what a process is blocked reading from / writing to with `strace -y`.

- If it's a **file** `strace -y` will show it inline.
- If it's a **pipe**, look up its number in `lsof`.
  Find PID that has the other end, strace that one.
- If it's a **socket**, look up its number in `lsof`.
  Find PID (or host IP) that has the other end, strace that one.

strace'ing the process that has the other end of the TCP socket (PID `23270` is `python2 ./command-server.py`):

```
% sudo strace -fp 23270 -y
strace: Process 23270 attached
wait4(32084,
```

Blocked on `wait4()`, so that's probably this code from the server:

```
    p = subprocess.Popen([name], stdout=PIPE)
    p.wait()
```

We've found the bug, because this is incorrect code, as the Python `Popen()` documentation says on `wait()`:

> This will deadlock when using stdout=PIPE ... and the child process generates enough output to a pipe such that it blocks waiting for the OS pipe buffer to accept more data.
> Use Popen.communicate() when using pipes to avoid that.

`dmesg` creates more output than fits in the pipe buffer.

This was easy to identify because there's only one `wait()` invocation in the server.

But how would we find the location of the problem if there were hundreds of `wait()` invocations in the server software, across many files?

In general, after you've chased down the problematic process using a series of straces, how do you find the userspace location issuing the blocking syscall?

> 95% of computer problems can be solved with `strace`.
> For the remaining 4% there's `gdb`.

me

GDB part

Inspecting Python with GDB

From this deleted StackOverflow question and here we learn:

Switch to the frame in the stack which has function:

```
PyEval_EvalFrameEx (or eval_frame)  # For Python < 3
```

To get the **file name**:

```
x/s ((PyStringObject*)f->f_code->co_filename)->ob_sval
```

To get the **function name**:

```
x/s ((PyStringObject*)f->f_code->co_name)->ob_sval
```

To get the **line number**:

```
print f->f_lineno
```

GDB preparation

If your GDB shows

```
Reading symbols from /usr/bin/python2.7...
   (no debugging symbols found)...done.
```

then install debugging symbols, e.g.

```
sudo apt-get install python2.7-dbg
```

GDB and blocking syscalls

When a process is blocked on a syscall, GDB drops you into a shell at that syscall, and you can ask for the `backtrace` to see how the program got there.

GDB run

```
% sudo gdb -p $(pgrep -f command-server)
Attaching to process 23270

0x00007f61090aff2a in __waitpid (pid=2946, stat_loc=0x7ff

(gdb) backtrace
#0  0x00007f61090aff2a in __waitpid (pid=2946, ...)
#1  0x0000000000576dd6 in posix_waitpid.lto_priv ()
#2  0x00000000004c30ce in ext_do_call (...)
#3  PyEval_EvalFrameEx ()
#4  0x00000000004b9ab6 in PyEval_EvalCodeEx ()
#5  0x00000000004c1e6f in fast_function (...)
#6  call_function (...)
#7  PyEval_EvalFrameEx ()
#8  0x00000000004c136f in fast_function (...)
...
```

OK, stuck in `__waitpid()`. We are in stack frame `#0`.

Let's go `up` to `PyEval_EvalFrameEx` ...

```
(gdb) up
#1  0x0000000000576dd6 in posix_waitpid.lto_priv (...)

(gdb) up
#2  0x00000000004c30ce in ext_do_call (...)

(gdb) up
#3  PyEval_EvalFrameEx ()
```

```
(gdb) x/s ((PyStringObject*)f->f_code->co_filename)->ob_s
0x7f6109388eac: "/usr/lib/python2.7/subprocess.py"

(gdb) print f->f_lineno
$1 = 473

(gdb) x/s ((PyStringObject*)f->f_code->co_name)->ob_sval
0x7f610932ac94: "_eintr_retry_call"
```

We're at the bottom of the Python standard library, in a wrapper that loops around the `wait()` syscall (code of subprocess.py:473). Let's go futher `up` until we're in our applications's code.

```
(gdb) up
#4  0x00000000004b9ab6 in PyEval_EvalCodeEx ()
(gdb) up
#5  0x00000000004c1e6f in fast_function (...)
(gdb) up
#6  call_function (...)
(gdb) up
#7  PyEval_EvalFrameEx ()

(gdb) x/s ((PyStringObject*)f->f_code->co_filename)->ob_s
0x7f610933fa2c: "/usr/lib/python2.7/subprocess.py"

(gdb) up
#8  0x00000000004c136f in fast_function (...)
(gdb) up
#9  call_function ()
(gdb) up
#10 PyEval_EvalFrameEx ()

(gdb) x/s ((PyStringObject*)f->f_code->co_filename)->ob_s
0x7f61093924b4: "./command-server.py"
```

This is the first/lowest stack frame that's in our code.

```
(gdb) x/s ((PyStringObject*)f->f_code->co_name)->ob_sval
0x7f879c0d4454: "run_command_for_client"

(gdb) print f->f_lineno
$1 = 13
```

So we've tracked down the precise location in our python program's userpace:

It's the `wait()` call in `command-server.py`, in the function `run_command_for_client()` which starts at line `13`.

Summary

1. Investigate issues on the running system via syscalls using `strace`.
2. Chase through pipes, sockets and across machines, with `strace` and `lsof`.
3. Find the origin of the final syscall in your userspace program using `gdb` or a similar debugger.

You can do this to debug hard-to-reproduce problems in production, and knowing very little about the programs you are debugging.

System calls are tye universal inspection point on Linux.

Thanks!