Debugging

Joseph Hallett

January 18, 2023

Whats all this about?

Writing programs is hard

- We should have strategies and *tools* for when things go wrong

Lets point you towards some!

An example program

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(int argc, char *argv[]) {
  char message[128];
  size_t message_len = 256;
  char timestamp[128];
  time_t t;
  struct tm *tmp;
  FILE *file = fopen(argv[1], "a+");

  printf("Type your log: ");
  getline(&message, &message_len, stdin);

  t = time(NULL);
  tmp = localtime(&t);
  strftime(timestamp, 256, "%C", tmp);

  fprintf(file, "%s: %s\n", timestamp, message);
  return 0;
}
```

Lets compile!

```
make journal
cc journal.c -o journal journal.c: In function ‘main’: journal.c:14:11: warning: passing argument 1 of ‘getline’
from incompatible pointer type [-Wincompatible-pointer-types]
In file included from journal.c:1: /usr/include/stdio.h:645:45: note: expected ‘char * restrict’ but argument is
of type ‘char ()[128]’
```

And when we run...

```
./journal <<<"Hello World!"
Segmentation fault (core dumped)
```

Okay, lets try and debug

```
   # gdb ./journal
Reading symbols from ./journal...
(No debugging symbols found in ./journal)
(gdb) run <<<"hello"
Starting program: /home/joseph/Repos/Talks/COMS10012-Software-Tools/Debugging/journal <<<"hello"
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".

Program received signal SIGSEGV, Segmentation fault.
__vfprintf_internal (s=0x0, format=0x402026 "%s: %s\n", ap=ap@entry=0x7fffffffde50, mode_flags=mode_flags@entry=0) at vfpr
722^^I ORIENT;
(gdb) bt
#0  __vfprintf_internal (s=0x0, format=0x402026 "%s: %s\n",
    ap=ap@entry=0x7fffffffde50, mode_flags=mode_flags@entry=0)
    at vfprintf-internal.c:722
#1  0x00007ffff7e2360a in __fprintf (stream=<optimized out>,
    format=<optimized out>) at fprintf.c:32
#2  0x000000000040125f in main ()
```

Lets make it a *little* easier

```
  cc -Og -g journal.c -o journal
  gdb ./journal
  (gdb) run <<<"hello"
Starting program: /home/joseph/Repos/Talks/COMS10012-Software-Tools/Debugging/journal <<<"hello"
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".

Program received signal SIGSEGV, Segmentation fault.
__memcpy_avx_unaligned_erms () at ../sysdeps/x86_64/multiarch/memmove-vec-unaligned-erms.S:333
Downloading 0.01 MB source file /usr/src/debug/glibc-2.36.9000-19.fc38.x86_64/string/../sysdeps/x86_64/multiarch/memmove-v
333^^I^^Imovl^^I%ecx, -4(%rdi, %rdx)
(gdb) bt
#0  __memcpy_avx_unaligned_erms ()
    at ../sysdeps/x86_64/multiarch/memmove-vec-unaligned-erms.S:333
#1  0x00007ffff7e496ac in __GI___getdelim (
    lineptr=lineptr@entry=0x7fffffffdff0, n=n@entry=0x7fffffffdfe8,
    delimiter=delimiter@entry=10, fp=0x7ffff7fa5aa0 <_IO_2_1_stdin_>)
    at iogetdelim.c:111
#2  0x00007ffff7e237d1 in __getline (lineptr=lineptr@entry=0x7fffffffdff0,
    n=n@entry=0x7fffffffdfe8, stream=<optimized out>) at getline.c:28
#3  0x00000000004011d6 in main (argc=<optimized out>, argv=<optimized out>)
    at journal.c:14
```

Looks like it all went wrong on line 14 of journal.c...

```
(gdb) b journal.c:14
Breakpoint 2 at 0x4011ba: file journal.c, line 14.
(gdb) run <<<"hello"
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/joseph/Repos/Talks/COMS10012-Software-Tools/Debugging/journal <<<"hello"
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".

Breakpoint 2, main (argc=<optimized out>, argv=<optimized out>) at journal.c:14
14^^I getline(&message, &message_len, stdin);
(gdb) inspect message
$3 = "@\000\000\000\000\000\000\000\000\200", '\000' <repeats 14 times>, "\006\000\000\000\216\000\000\000\f\000\000\000\b
(gdb) inspect message_len
$4 = 256
(gdb) d
Delete all breakpoints? (y or n) y
(gdb)
```

If in doubt... read the manual

In `man 3 getline`:

> *getline() reads an entire line from stream, storing the address of the buffer containing the text into *lineptr. The buffer is null-terminated and includes the newline character, if one was found.*
>
> *If *lineptr is set to NULL before the call, then getline() will allocate a buffer for storing the line. This buffer should be freed by the user program even if getline() failed.*
>
> *Alternatively, before calling getline(), *lineptr can contain a pointer to a malloc(3)-allocated buffer *n bytes in size. If the buffer is not large enough to hold the line, getline() resizes it with realloc(3), updating *lineptr and *n as necessary.*

Well we're passing a *statically* allocated buffer... lets fix that.

A new *example program

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(int argc, char *argv[]) {
  char *message = NULL;
  size_t message_len;
  char timestamp[128];
  time_t t;
  struct tm *tmp;
  FILE *file = fopen(argv[1], "a+");

  printf("Type your log: ");
  getline(&message, &message_len, stdin);

  t = time(NULL);
  tmp = localtime(&t);
  strftime(timestamp, 256, "%C", tmp);

  fprintf(file, "%s: %s\n", timestamp, message);
  return 0;
}
```

```
cc -g -Og journal2.c -o journal2
```

And now when we run...

```
$ ./journal2 <<<"hello"
Segmentation fault (core dumped)

 # gdb ./journal2
(gdb) run <<<"hello"
Starting program: /home/joseph/Repos/Talks/COMS10012-Software-Tools/Debugging/journal2 <<<"hell

Program received signal SIGSEGV, Segmentation fault.
0x00007ffff7e2de82 in __vfprintf_internal () from /lib64/libc.so.6
Missing separate debuginfos, use: dnf debuginfo-install glibc-2.36.9000-19.fc38.x86_64
(gdb) bt
#0  0x00007ffff7e2de82 in __vfprintf_internal () from /lib64/libc.so.6
#1  0x00007ffff7e2360a in fprintf () from /lib64/libc.so.6
#2  0x0000000000401225 in main (argc=<optimized out>, argv=<optimized out>) at journal2.c:20
(gdb)
```

...well, we got further...

We could continue with `gdb`

GDB is an extremely powerful debugging tool

- Its also *really* hard to use
- See *Computer Systems B* next year, or *Systems and Software Security* at Masters level
- If you're on a Mac or BSD box check out `lldb`
- Or for a proper tutorial the documentation it refers you to *every time you open it*.

It is *well worth your time to learn…*

- But *this course* is about *Software Tools* and I want to show you *more* of them

Strace

The `strace` tool lets you trace what systemcalls a program uses

- On OpenBSD see `ktrace` and `kdump`
- On MacOS/FreeBSD see `dtruss` and `dtrace`

Lets run it!

```
$ strace ./journal2 <<<'Hello'
execve("./journal2", ["./journal2"], 0x7ffd3ef71360 /* 36 vars */) = 0
brk(NULL)                               = 0x154f000
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffc01b1b610) = -1 EINVAL (Invalid argument)
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=74509, ...}, AT_EMPTY_PATH) = 0
mmap(NULL, 74509, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7ff1ca8ef000
close(3)                                = 0
openat(AT_FDCWD, "/lib64/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0P|\2\0\0\0\0\0"..., 832)...
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"...,
newfstatat(3, "", {st_mode=S_IFREG|0755, st_size=2232840, ...}, AT_EMPTY_PATH) ...
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f...
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"...,
mmap(NULL, 1961264, PROT_READ, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0x7ff1ca70e00...
mmap(0x7ff1ca734000, 1409024, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED|MAP_DE...
mmap(0x7ff1ca88c000, 339968, PROT_READ, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3,...
mmap(0x7ff1ca8df000, 24576, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DEN...
mmap(0x7ff1ca8e5000, 32048, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANO...
close(3)                                = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f...
arch_prctl(ARCH_SET_FS, 0x7ff1ca8ee640) = 0
set_tid_address(0x7ff1ca8ee910)         = 34069
set_robust_list(0x7ff1ca8ee920, 24)     = 0
rseq(0x7ff1ca8eef60, 0x20, 0, 0x53053053) = 0
mprotect(0x7ff1ca8df000, 16384, PROT_READ) = 0
mprotect(0x403000, 4096, PROT_READ)     = 0
mprotect(0x7ff1ca933000, 8192, PROT_READ) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=9788*1024, rlim_max=RLIM64_INFINITY}...
munmap(0x7ff1ca8ef000, 74509)           = 0
getrandom("\x9e\xe1\xb9\x13\x31\x2c\x9e\xee", 8, GRND_NONBLOCK) = 8
brk(NULL)                               = 0x154f000
brk(0x1570000)                          = 0x1570000
openat(AT_FDCWD, NULL, O_RDWR|O_CREAT|O_APPEND, 0666) = -1 EFAULT (Bad address)
newfstatat(1, "", {st_mode=S_IFCHR|0620, st_rdev=makedev(0x88, 0x5), ...}, AT_EMPTY_PATH
newfstatat(0, "", {st_mode=S_IFIFO|0600, st_size=0, ...}, AT_EMPTY_PATH) = 0
read(0, "hello\n", 4096)                = 6
openat(AT_FDCWD, "/etc/localtime", O_RDONLY|O_CLOEXEC) = 3
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=3664, ...}, AT_EMPTY_PATH) = 0
newfstatat(3, "", {st_mode=S_IFREG|0644, st_size=3664, ...}, AT_EMPTY_PATH) = 0
read(3, "TZif2\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\10\0\0\0\10\0\0\0\0"..., 4096) = 366
lseek(3, -2329, SEEK_CUR)               = 1335
read(3, "TZif2\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\10\0\0\0\10\0\0\0\0"..., 4096) = 232
close(3)                                = 0
--- SIGSEGV {si_signo=SIGSEGV, si_code=SEGV_MAPERR, si_addr=0xc0} ---
+++ killed by SIGSEGV (core dumped) +++
Segmentation fault (core dumped)
```

Too much output!

`strace` lets you use regexp to filter what syscalls you look at

- ...or you could just use `grep`...

```
$ strace -e '/open.*' ./journal2 <<<hello
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/lib64/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, NULL, O_RDWR|O_CREAT|O_APPEND, 0666) = -1 EFAULT (Bad address)
openat(AT_FDCWD, "/etc/localtime", O_RDONLY|O_CLOEXEC) = 3
--- SIGSEGV {si_signo=SIGSEGV, si_code=SEGV_MAPERR, si_addr=0xc0} ---
+++ killed by SIGSEGV (core dumped) +++
Segmentation fault (core dumped)
```

Oh yeah... we forgot an arg

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(int argc, char *argv[]) {
  char *message = NULL;
  size_t message_len;
  char timestamp[128];
  time_t t;
  struct tm *tmp;
  FILE *file = fopen(argv[1], "a+"); /* line 11 */

  printf("Type your log: ");
  getline(&message, &message_len, stdin);

  t = time(NULL);
  tmp = localtime(&t);
  strftime(timestamp, 256, "%C", tmp);

  fprintf(file, "%s: %s\n", timestamp, message); /* line 20 */
  return 0;
}
```

Lets fix that...

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(int argc, char *argv[]) {
  char *message = NULL;
  size_t message_len;
  char timestamp[128];
  time_t t;
  struct tm *tmp;

  if (argc < 2) { printf("Usage %s path/to/log\n", argv[0]); exit(1); };
  FILE *file = fopen(argv[1], "a+"); /* line 11 */

  printf("Type your log: ");
  getline(&message, &message_len, stdin);

  t = time(NULL);
  tmp = localtime(&t);
  strftime(timestamp, 256, "%C", tmp);

  fprintf(file, "%s: %s\n", timestamp, message); /* line 20 */
  return 0;
}
```

Now when we run!

```
./journal3 documents/log.txt <<<hello
Segmentation fault (core dumped)
```

Lets try `ltrace` this time (no equivalent on other platforms)...

- It traces *library* calls

ltrace and a bit more strace

```
$ ltrace ./journal3 documents/log.txt <<<hello
fopen("documents/log.txt", "a+")                                        = nil
printf("Type your log: ")                                               = 15
getline(0x7fffebcc0fc8, 0x7fffebcc0fc0, 0x7f4bfcf40aa0, 0)              = 6
time(nil)                                                               = 1674045150
localtime(0x7fffebcc0f38)                                               = 0x7f4bfcf47640
strftime("20", 256, "%C", 0x7f4bfcf47640)                               = 2
fprintf(nil, "%s: %s\n", "20", "hello\n" <no return ...>
--- SIGSEGV (Segmentation fault) ---
+++ killed by SIGSEGV +++

$ strace -e openat ./journal3 documents/log.txt <<<hello
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/lib64/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "documents/log.txt", O_RDWR|O_CREAT|O_APPEND, 0666) = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/localtime", O_RDONLY|O_CLOEXEC) = 3
--- SIGSEGV {si_signo=SIGSEGV, si_code=SEGV_MAPERR, si_addr=0xc0} ---
+++ killed by SIGSEGV (core dumped) +++
Segmentation fault (core dumped)
```

Lets fix that...

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <errno.h>

int main(int argc, char *argv[]) {
  char *message = NULL;
  size_t message_len;
  char timestamp[128];
  time_t t;
  struct tm *tmp;

  if (argc < 2) { printf("Usage %s path/to/log\n", argv[0]); exit(1); };
  FILE *file = fopen(argv[1], "a+"); /* line 11 */
  if (file == NULL) {
    perror("Failed to open log");
    exit(2);
  }

  printf("Type your log: ");
  getline(&message, &message_len, stdin);

  t = time(NULL);
  tmp = localtime(&t);
  strftime(timestamp, 256, "%C", tmp);

  fprintf(file, "%s: %s\n", timestamp, message); /* line 20 */
  return 0;
}
```

Now when we run...

```
$ ./journal4 <<<hello
Usage ./journal4 path/to/log

$ ./journal4 documents/log.txt <<<hello
Failed to open log: No such file or directory

$ ./journal4 /etc/passwd <<<hello
Failed to open log: Permission denied

$ ./journal4 /dev/stdout
Type your log: hello
20: hello
```

20?!

From `man 3 strftime`:

> %c *The preferred date and time representation for the current locale. (The specific format used in the current locale can be obtained by calling* $nl_{langinfo}(3)$ *with* D_{TFMT} *as an argument for the %c conversion specification, and with* ERA_{DTFMT} *for the %Ec conversion specification.) (In the POSIX locale this is equivalent to %a %b %e %H:%M:%S %Y.)*
>
> %C *The century number (year/100) as a 2-digit integer.* (SU) *(The %EC conversion specification corresponds to the name of the era.) (Calculated from* tm_{year}.)

Debugging tools can't catch poorly written code!

But other tools can catch things...

Thinking back to when we fixed up `getline`... it said it would allocate the memory for the line

- ...did we ever free it?

```
$ valgrind ./journal4 /dev/stdout <<<hello
==36111== Memcheck, a memory error detector
==36111== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==36111== Using Valgrind-3.20.0 and LibVEX; rerun with -h for copyright info
==36111== Command: ./journal4 /dev/stdout
==36111==
20: hello

Type your log: ==36111==
==36111== HEAP SUMMARY:
==36111==     in use at exit: 592 bytes in 2 blocks
==36111==   total heap usage: 13 allocs, 11 frees, 13,684 bytes allocated
==36111==
==36111== LEAK SUMMARY:
==36111==    definitely lost: 120 bytes in 1 blocks
==36111==    indirectly lost: 0 bytes in 0 blocks
==36111==      possibly lost: 0 bytes in 0 blocks
==36111==    still reachable: 472 bytes in 1 blocks
==36111==         suppressed: 0 bytes in 0 blocks
```

Wrap up

In this lecture we've cone over the *very basics* of several debugging tools

- `strace`, `ltrace`, `valgrind` and `gdb` will help deal with most of the bugs you encounter

But so will good defensive programming strategies

- *Always* check the return code of functions
- *Always* check assumptions
- *Always* fix your compiler warnings

...actually get more warnings!

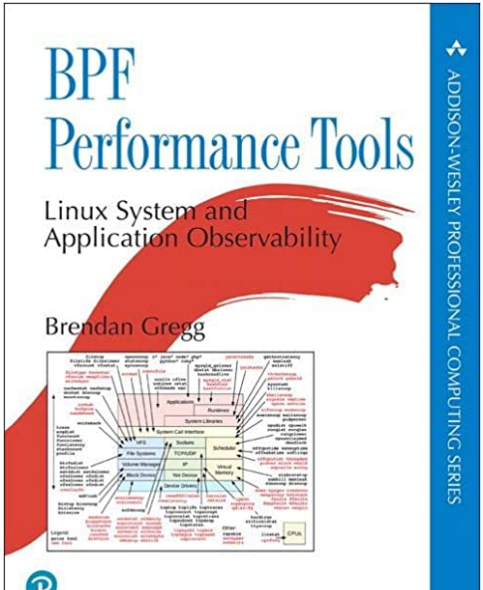
Compiling with the `-Wall -Wextra --std=c11 -pedantic` will make the compiler really picky about your C code...
But there are *other tools* called *linters* that can get even more picky

C/C++	Clang Static Analyser, Rats
Java	FindBugs
Haskell	hlint
Python	pylint, mypy

Other tools for C/C++ can add extra runtime checks

ASan	Address Sanitizer; checks for pointer shenangians
UBSan	Undefined Behaviour Sanitizer; checks for C gotchas

BPF tools



Linux has a (reasonably) new insrumentation framework called *eBPF*

- It lets you get *loads* of detail about what programs are doing
- Highly Linux specific
- I need to learn it `:-(`