Debugging (1)

- writing programs is hard
- we need strategies and tools for when programs go wrong
  - or things to guide us to working programs

An example program to debug

```
#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;
}
```

- when we compile we get various warnings

and when we run it:

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

gdb (GNU debugger)

- follows the chain of function calls
- to consistenyl view the state of the stack
- to view the computers internals as your program is running

You can set breakpoints with the b command:

```
b program.c:14
```

- the program name followed by the line you wish to set breakpoint at, in this case line 14
  - you will find out what line the program is crashing on by using gdb
- so breakpoints allow you to pause the program and examine its current state
  - e.g. the state of the registers
  - memory
  - other critical data
- this all helps pin point where the program is breaking

Lets attempt to debug it using gdb:

```
   # 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 ()
```

- we run gdb with: `gdb ./journal`
- the we run it with our arguments: `(gdb) run <<<"hello"` - the <<< was mentioned in pipes, it feeds a string in as input to the program
- in the rest of the output we see that the program crashes as soon as it recieves: `SIGSEGV, Segmentation fault`


```
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 vfp
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 ()
```


- in the additional information it sayus

getting more information

- if we complile with some extra flags for debugging: `cc -Og -g journal.c -o journal`
  - `-g` enables debug symbols
  - `-Og` says “when you are compiling, dont optimise for speed or safety, instead compile to make the program as **easy to debug as possible**

  we then run the program using gdb again:

```
  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
```

- the #0 #1 #2 #3 give additional info
- #3 tells us that the program is crashing on line 14 of journal.c

Now within gdb, lets examine what is happening at line 14...

Setting a breakpoint

- use the `(gbd) b` command to set a breakpoint at a given line e.g.

```
(gdb) b journall.c: 14
```

- when we run we hit the breakpoint before triggering the seg fault
- we can see below we are at:

```
14^^I getline(&message, &message_len, stdin);
```

```
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)
```

- so it is crashing when reading in this message

lets use the `inspect` command for a further look:

```
(gdb) inspect message
```

so why is it seg faulting?

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.

- when we wrote the initial program code we didnt give message a size using `malloc` but instead a static length of 128:

```
char message[128];
```

lets fix the source code and run again

```c
#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
```

when running we still get a seg fault:

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

so lets run it using gdb again:

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

now the program is crashing at line 20: `at journal.c: 20`

```
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)
```

Strace

- this tool lets you trace what **system calls** a program uses
  - i.e. what functions of the operating system it calls as part of its program
- variance of this for most OS
  - on OpenBSD: **ktrace** and **kdump**
  - on MacOS/FreeBSD: **dtruss** and **dtrace**

Strace gives you **a lot of output**

- running on our earlier program:

```
$ 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"...,
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_PAT
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\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\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)
```

- this displays every system call your program makes when it is run
- the left side are all the functions that happen when your program is being **loaded into memory**
- the right side is what is happening **whilst the program is running**

Strace lets you use **regular expressions** to filter what system calls you view

- could also use **grep**

Ltrace

- another tool is **ltrace**
  - ltrace is **Linux specific** tracing tool
  - it traces ***library* calls**
    - where strace traces calls to the operating system (system calls) ltrace traces calls to third party libraries

Running on our program:

```
$ 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 +++
```

- we see that fopen is returning nil `= nil` when it should be returning a pointer to the open file
- the `fprintf(nil...)` says that a null pointer is getting through to that fprintf on line 20, indicating where our program may be going wrong

Using strace with the `-e` flag which limites the output of strace

```
$ 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)
```

- we can see that the `(No such file or directory)` is likely causing the issues, i.e. the file you are attempting to open does not currently exist

Lets print some error messages using perror from the errno.h C library:

```c
#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
```

- now we're getting some helpful error messages

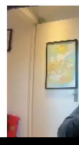
Valgrind

- tool for detecting memory leaks
- tells you if every memory allocation has a matching free
- running it with the previous program:

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
```



- we know that we are leaking memory: `definitely lost: 120 bytes...`

Best practice for aiding debugging is defensive programming

- **never ignore compiler warnings**
  - use werror flag
- **always check assumptions**
- **always check function return codes**
- use MORE warning flags if possible

e.g. for C programs:

`-Wall -Wextra --std=c11 -pedantic` these make the compiler incredibly picky about your code

There are also more tools called **linters** which can be even more picky during compilation

- aka **static analysis tools**

- they give you more warnings than normal

BPF tools

- Linux has a new intrumentation framework called eBPF
  - it lets you get lots of detail about what programs are doing
  - **HIGHLY** Linux specific