

# Debugging

Jo Hallett

November 13, 2025



## 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\u2014your\u2014log:\u2014");
    getline(&message, &message_len, stdin);

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

    fprintf(file, "%s:\u2014%s\n", timestamp, message);
    return 0;
}
```

# Lets compile!

```
make journal
```

```
cc    journal.c -o journal
```

And when we run...

```
./journal <<<"Hello World!"
```

```
Segmentation fault (core dumped)
```

# Segfaults are annoying

Programs randomly crashing is extremely irritating

- ▶ ...but a fact of life
- ▶ ... we are programmers
- ▶ ... it is our job to fix them

Lets discuss some tools for spotting errors

**AddressSanitizer** Adds extra debugging checks

**GDB** the GNU Debugger

**Strace** Systemcall tracer

**Ltrace** Library tracer

**Valgrind** Memory error detector

## AddressSanitizer

Tool from the LLVM project (clang) to give more information about crashes **at runtime**.  
Other sanitizers exist:

**UndefinedBehaviorSanitizer** spots bad compiler practice

**ThreadSanitizer** spots bad multithreading practice

**LeakSanitizer** spots bad memory management

To use, compile with clang and add the `-fsanitize=address` flag

```
clang journal.c -fsanitize=address -o journal
```

## Downsides

- ▶ Your program will use more memory and be slower
- ▶ Your program may be more easy to hack ; - )

Lets go!

Looks like its a NULL pointer dereference?

```
$ ./journal <<<"hello"
AddressSanitizer:DEADLYSIGNAL
=====
==302530==ERROR: AddressSanitizer: SEGV on unknown address 0x0000000000c0 (pc 0x7f99bd58e470 bp 0x7ffcea5c2120)
==302530==The signal is caused by a READ memory access.
==302530==Hint: address points to the zero page.
#0 0x7f99bd58e470 (/lib64/libc.so.6+0x66470) (BuildId: 37e4ac6a7fb96950b0e6bf72d73d94f3296c77eb)
#1 0x453325 (/home/jh18636/journal+0x453325) (BuildId: becf911c680ef05574954cad1b054efee58f4fab)
#2 0x5115f0 (/home/jh18636/journal+0x5115f0) (BuildId: becf911c680ef05574954cad1b054efee58f4fab)
#3 0x7f99bd5627e4 (/lib64/libc.so.6+0x3a7e4) (BuildId: 37e4ac6a7fb96950b0e6bf72d73d94f3296c77eb)
#4 0x42975d (/home/jh18636/journal+0x42975d) (BuildId: becf911c680ef05574954cad1b054efee58f4fab)

AddressSanitizer can not provide additional info.
SUMMARY: AddressSanitizer: SEGV (/lib64/libc.so.6+0x66470) (BuildId: 37e4ac6a7fb96950b0e6bf72d73d94f3296c77eb)
==302530==ABORTING
```

# GDB: The GNU Debugger

## Horrifically powerful

- ▶ Can step through a program at an assembly level
- ▶ Can watch registers and the stack change line by line
- ▶ Can program scripts and to react when certain things happen
- ▶ Can debug systems remotely
- ▶ Ported to every sort of computer you could wish for

## Horrifically unintuitive

- ▶ It has a built in GUI but you'll wish it didn't (`layout regs`)
- ▶ Lots of unintuitive single letter commands
- ▶ Cryptic output
- ▶ But its the standard debugger!
  - ▶ Others (and GUIs) exist, but still worth having a basic knowledge of the CLI tool

## Okay, lets try and debug

- ▶ run runs your program (with arguments if passed)
- ▶ bt gives you a backtrace by reading out the stack

```
$ 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)
722      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  0x000007ffff7e2360a in __fprintf (stream=<optimized out>,
    format=<optimized out>) at fprintf.c:32
#2  0x000000000040125f in main ()
```

## Lets make it a little easier

- ▶ -g adds debugging informations.
- ▶ -Og optimizes for debuggability

```
$ 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
333      movl    %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___getline (
    lineptr=lineptr@entry=0x7fffffffdf0, n=n@entry=0x7fffffffdf0, fe8,
    delimiter=delimiter@entry=10, fp=0x7ffff7fa5aa0 <_IO_2_1_stdin_>)
    at iogetline.c:111
#2  0x00007ffff7e237d1 in __getline (lineptr=lineptr@entry=0x7fffffffdf0,
    n=n@entry=0x7fffffffdf0, 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...

Breakpoints let us stop when we reach a line of the program or an address.

- ▶ Create them with b address
  - ▶ Delete them with d
  - ▶ Run on to the next one with 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      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\f\000"
(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\u00b7your\u00b7log:\u00b7");
    getline(&message, &message_len, stdin);

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

    fprintf(file, "%s:\u00b7%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 <<<"hello"

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

<<`input` runs your program with input

`b` set breakpoints

`c` continue after hitting a breakpoint

`bt` get a backtrace

`info` get information about registers or variables or anything else

`x` examine a variable/pointer

`disas` see the assembly code you're running

`help` get help!

## 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!

```
make journal2  
strace ./journal2 <<<'Hello' 2>&1
```

## Too much output!

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

- ▶ ...or you could just use grep...

```
make journal2
strace -e '/open.*' ./journal2 <<<hello 2>&1
```

```
cc      journal2.c -o journal2
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/usr/lib/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) +++
```

## Or if you prefer OpenBSD

```
ktrace ./journal2 <<<hello  
kdump | grep -A 2 -B 2 open
```

```
--  
63085 journal2 CALL close(3)  
63085 journal2 RET close 0  
63085 journal2 CALL open(0x34d8c02f033,0x10000<0_RDONLY|0_CLOEXEC>)  
63085 journal2 NAMI "/usr/Lib/libc.so.100.3"  
63085 journal2 RET open 3  
63085 journal2 CALL fstat(3,0x70bf220f2088)  
63085 journal2 STRU struct stat { dev=1077, ino=3784328, mode=-r--r--r--, nlink=1, uid=0<"root">, gid=7<"bin"> }  
--  
63085 journal2 CALL mprotect(0x34debfa5000,0x1000,0x1<PROT_READ>)  
63085 journal2 RET mprotect 0  
63085 journal2 CALL open(0,0x20a<0_RDWR|0_APPEND|0_CREAT>,0666<S_IRUSR|S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH>)  
63085 journal2 RET open -1 errno 14 Bad address  
63085 journal2 CALL kbind(0x70bf220f2268,24,0x5e8161d98e625637)  
63085 journal2 RET kbind 0  
--
```

## 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+");

    printf("Type\u2014your\u2014log:\u2014");
    getline(&message, &message_len, stdin);

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

    fprintf(file, "%s:\u2014%s\n", timestamp, message);
    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+");

    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;
}
```

## And if you can't spot the difference

```
diff -u journal{2,3}.c
```

```
--- journal2.c 2024-02-07 11:14:29.060025998 +0000
+++ journal3.c 2024-02-07 12:16:09.220079001 +0000
@@ -8,6 +8,8 @@
     char timestamp[128];
     time_t t;
     struct tm *tmp;
+
+    if (argc < 2) { printf("Usage %s %s\n", argv[0]); exit(1); }
     FILE *file = fopen(argv[1], "a+");
     printf("Type your log:");
```

## 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

```
make journal3
ltrace ./journal3 documents/log.txt <<<hello 2>&1
```

```
fopen("documents/log.txt", "a+")      = 0
printf("Type\u005cyour\u005clog:\u005c")      = 15
getline(0x7ffd196b0018, 0x7ffd196b0020, 0x76a22f8538e0, 0x7ffd196b0020) = 6
time(0)                                = 1707308599
localtime(0x7ffd196b0028)              = 0x76a22f85a320
strftime("20", 256, "%C", 0x76a22f85a320) = 2
fprintf(0, "%s:\u005cs\n", "20", "hello\n" <no return ...>
--- SIGSEGV (Segmentation fault) ---
+++ killed by SIGSEGV +++
```

```
strace -e openat ./journal3 documents/log.txt <<<hello 2>&1
```

```
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
openat(AT_FDCWD, "/usr/lib/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) ++
```

## 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+");
    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);
    return 0;
}
```

## What has changed again?

```
diff -u journal{3,4}.c
```

```
--- journal3.c 2024-02-07 12:31:13.196788801 +0000
+++ journal4.c 2024-02-07 12:31:13.293455473 +0000
@@ -1,6 +1,7 @@
 #include <stdio.h>
 #include <stdlib.h>
 #include <time.h>
+#include <errno.h>

 int main(int argc, char *argv[]) {
     char *message = NULL;
@@ -11,6 +12,10 @@
     if (argc < 2) { printf("Usage: %s <path/to/log>\n", argv[0]); exit(1); }
     FILE *file = fopen(argv[1], "a+");
+    if (file == NULL) {
+        perror("Failed to open log");
+        exit(2);
+    }

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

## 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
```

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\_T\_FMT as an argument for the %c conversion specification, and with ERA\_D\_T\_FMT 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
==36111== Rerun with --Leak-check=full to see details of leaked memory
==36111==
==36111== For lists of detected and suppressed errors, rerun with: -s
==36111== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

## Wrap up

In this lecture we've gone 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

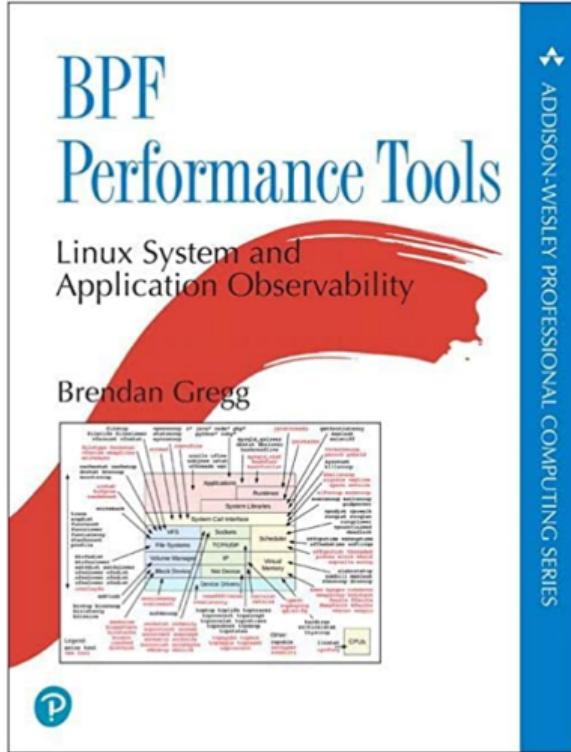
Python pylint, mypy

Shellscript shellcheck

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

ASan Address Sanitizer; checks for pointer shenanigans

UBSan Undefined Behaviour Sanitizer; checks for C gotchas



Linux has a (reasonably) new instrumentation framework called eBPF

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

## This weeks lab

Is `evil` fun!

- ▶ I'm gonna give you 5 crackmes
- ▶ They'll ask you for the password
  - ▶ You have to work out what it is

Practice using the debugging tools to work out what the program is expecting.

Hmmm...

We're early...

- ▶ It's almost like I planned for this...

Well we could show you another use for debugging tools

- ▶ Sometimes being able to see what a program is doing is useful for **other** things ; - )