Debugging
System Programming Lab

Maksym Planeta
Björn Döbel

23.09.2021
Table of Contents

Introduction

Hands-on
  Tracing made easy
  Dynamic intervention
  Compiler-based helpers
  The GNU Debugger
  DIY debugging

The end
Names

Figure: Admiral Grace Hopper

Figure: 1940s: First actual case of bug being found

Definitions

**Bug**

is a flaw in computer system that results in an unexpected behavior.

**Debugging**

is a process of searching and fixing deviations from an expected behavior.
Debugging is not only finding living creatures in an electronic device:

- Program crash;
- Wrong result;
- Slow execution.
How the course goes

Source: See slide 50
How the course goes: Example with digging

Source: See slide 50
How the course goes: Example with digging

Source: See slide 50
How the course goes: Example with digging

Source: See slide 50
How the course goes: Example with digging

Source: See slide 50
Goals

- Teach tools used for debugging
- How the tools work “under the hood”
- The same applies for tomorrow
Introduction to debugging tools:

- strace
- ltrace
- gdb
- valgrind
- perf
- ptrace
- and even more...
Brendan D. Gregg’s diagram

http://www.brendangregg.com/linuxperf.html
Table of Contents

Introduction

Hands-on
  Tracing made easy
  Dynamic intervention
  Compiler-based helpers
  The GNU Debugger
  DIY debugging

The end
Tracing System Calls – strace

Inspect system calls performed by a program

- Filtering: `strace -e`
- Timing: `strace -t[tt] / strace -T`
- Statistics: `strace -c`
Assignment №1

1. Which system calls are performed when you run /bin/ls
2. How many calls are performed?
3. Why so many?
Tracing library calls – ltrace

Inspect all calls to *shared* libraries

- **Filtering:** `ltrace -e`
- **Timing:** `ltrace -t[tt]` / `ltrace -T`
- **Statistics:** `ltrace -c`
Assignment №2

$ wget http://svn.inf.tu-dresden.de/repos/advsysprog/debugging/01-strace/binary

Make it print “SUCCESS”!

Hints: file, strace, ltrace, objdump.
Dynamic linker resolves symbols according to LD_LIBRARY_PATH

Tell linker to load a library independent whether an application wants it or not by LD_PRELOAD

Details: man ld.so
Create a shared object containing empty implementations of:

```c
void *malloc(size_t size);
void free(void *ptr);
```
TARGET = mallocWrap.so
OBJ = wrap.o

CFLAGS = -Wall -g -fpic -D_GNU_SOURCE=1

$(TARGET): $(OBJ) Makefile
  $(CC) -shared -o $(TARGET) $(OBJ) -ldl

.PHONY: clean

clean:
  $(RM) $(OBJ) $(TARGET)
Create a C program with a memory leak:

```c
#include <stdlib.h>

int main(void)
{
    char *m = malloc(1024);
    (void)m;
    return 0;
}

Link with -rdynamic
```
Detecting Memory Leaks

- Use LD_PRELOAD to let the leaky program call your implementation of malloc/free.
- Track malloc/free information to report memory leaks at program termination.
- Use the “real” malloc/free to perform the actual work.
Interfacing the dynamic linker

`void *dlopen(const char *filename, int flag);`
`char *dlerror(void);`
`void *dlsym(void *handle, const char *symbol);`
`int dlclose(void *handle);`
C/C++ Function pointers

void * (*real_malloc) (size_t)
C/C++ Function pointers

```
void * (*real_malloc) (size_t)
```

- Function return value
C/C++ Function pointers

```c
void * (*real_malloc) (size_t)
```

- Function return value
- Variable name
C/C++ Function pointers

```c
void * (*real_malloc) (size_t)
```

- Function return value
- Variable name
- Function parameter types
C/C++ Function pointers

void * (*real_malloc) (size_t)

- Function return value
- Variable name
- Function parameter types
Finding real malloc

```c
#include <dlfcn.h>

// ...

void *(*real_malloc)(size_t) = NULL;
// ...

real_malloc = dlsym(RTLD_NEXT, "malloc")
```
Finding real malloc

```c
#include <dlfcn.h>

void *(*real_malloc)(size_t) = NULL;

real_malloc = dlsym(RTLD_NEXT, "malloc")

And please link with -d1!
```
Assignment №3, step 3

- In your malloc/free wrappers:
  - Track memory management info;
  - Redirect work to real malloc and free;

- Upon exit, print all pointers (and sizes) that were not free’d;

- You will need to be notified upon exit():
  - Wrap it;
  - Or use atexit();

- Optional & tricky.
  Check backtrace() to also store who malloc’d the leaked regions.
  www.sourceware.org/bugzilla/show_bug.cgi?id=956
An anecdote

1. Bug report on Strange sound on mp3 flash website
An anecdote

1. Bug report on Strange sound on mp3 flash website
2. Located in libflashplayer.so
An anecdote

1. Bug report on Strange sound on mp3 flash website
2. Located in libflashplayer.so
3. Reason: use of memcpy for overlapping regions
An anecdote

1. Bug report on Strange sound on mp3 flash website
2. Located in libflashplayer.so
3. Reason: use of memcpy for overlapping regions
4. Solution is to use memmove, but plugin is closed source
An anecdote

1. Bug report on Strange sound on mp3 flash website
2. Located in libflashplayer.so
3. Reason: use of memcpy for overlapping regions
4. Solution is to use memmove, but plugin is closed source
5. Linus’ workaround: use LD_PRELOAD to replace memcpy with memmove
Linus’ workaround

http://bugzilla.redhat.com/show_bug.cgi?id=638477#c38

- Write your own memcpy
- gcc -O2 -c mymemcpy.c
- ld -G mymemcpy.o -o mymemcpy.so
- LD_PRELOAD mymemcpy.so /opt/google/chrome/google-chrome &
Valgrind

Binary recompilation framework (Valgrind core)

Tools to perform program analysis:
  ▶ MemCheck
  ▶ Cachegrind
  ▶ Callgrind
  ▶ Helgrind
Assignment №4

Analyze some programs with Valgrind:
$> \text{svn co http://svn.inf.tu-dresden.de/repos/advsysprog/debugging/03-valgrind/}
$> \text{cd 03-valgrind}
$> \text{./build.sh}
Static checker

http://svn.inf.tu-dresden.de/repos/advsysprog/debugging/07-ccc/

scan-build

1. Install clang
2. Run scan-build make to analyze code
3. Run scan-view to see the report code

Link. List of static analyzers: http://spinroot.com/static/
Compiler sanitizers

Additional libraries which are able to detect: race conditions, memory bugs, undefined behavior, etc.

Address sanitizer

1. Install libasan;
2. Run make asan;
   This compiles programs with -fsanitize=address;
3. Run program;

See man gcc/-fsanitize.
The GNU Debugger

Interactive debugger (gdb):

- breakpoints, watchpoints;
- single-stepping, reverse-stepping;
- inspect/modify registers & memory;
- scripting.

Best used with binaries containing debug info
(compiled with -g option)
Breakpoints

▶ b[reak] {function | line | *address}
▶ w[atch] {variable | *address} {condition}
▶ c[ontinue]
Inspecting your program

- `list [±] [N]` – show program code
- `disasm` – disassemble current function
- `info registers` – show register content
- `print/FMT {variable | expression}` – evaluate and print expression or variable
- `x/FMT {address}` – examine memory
- `bt` – backtrace
Going forwards

- `s[tep]` – step to next source line;
- `s[tep]i` – step to next assembler instruction;
- `n[ext]` – step to next source line, proceeding through function calls;
- `n[ext]i` – step to next assembler instruction, proceeding through function calls;
- `f[i]nish` – run to return from current function
Going backwards

- `record full` – start full execution recording
- `record stop` – stop execution recording
- `rs[tep]` – step to previous source line;
- `rs[tep]i` – step to previous assembler instruction;
- `rn[ext]` – step to previous line, proceeding through function calls;
- `rn[ext]i` – step to previous assembler instruction, proceeding through function calls;
Remote debugging

- GDB can connect to remote GDB servers
  - Via TCP or serial line
  - `set target remote {address:port}`
- Heavily used in OS/embedded development
  Qemu, Bochs/x86, Valgrind, etc. contain their own GDB servers.
Scripting

- Write GDB commands into a text file
- Run `gdb -x {file}`

**Self-learning:** GDB/Python API

**Link:** UI for GDB [https://github.com/cyrus-and/gdb-dashboard](https://github.com/cyrus-and/gdb-dashboard)
Assignment №5

$> $ svn co
http://svn.inf.tu-dresden.de/repos/advsysprog/debugging/04-gdb/

Example addFunc seems to add $10 + 20$, but prints $20$. Why?
04-gdb/erathosthenes contains 4 versions of the Sieve of Eratosthenes.

But only one works properly.

What’s wrong with the rest?
Under the hood

System call `ptrace()`

- Child allows parent to intercept child interactions by
  
  ```c
  ptrace(PTRACE_TRACEME, 0, 0, 0);
  ```

- Parent/Debugger inspect and modifies child state by `ptrace` requests:
  - `PEEK/POKE`
  - `SETREGS/GETREGS`
  - `CONT/SYSCALL/SINGLESTEP`
Assignment №7

Tiny strace

Write a program that:

1. Gets another program on the command line;
2. Runs this program;
3. Uses ptrace() to inspect all system calls made by the traced program;
4. Printing system call numbers and return values is enough;
5. optional*. You can print function names as well.
Syscall in a nutshell

`ssize_t write(int fd, const void *buf, size_t count);`

⇓

```c
ret = write(1, "Hello World", 12);
```

⇓

```
movq $1, %rax; use the write syscall
movq $1, %rdi; write to stdout
movq $msg, %rsi; use string "Hello World"
movq $12, %rdx; write 12 characters
syscall; make syscall
```
Syscall in a nutshell

```c
ssize_t write(int fd, const void *buf, size_t count);

ret = write(1, "Hello World", 12);
```
Syscall in a nutshell

```c
ssize_t write(int fd, const void *buf, size_t count);
```

```c
down
ret = write(1, "Hello World", 12);
```

```asm
down
movq $1, %rax  ; use the write syscall
movq $1, %rdi  ; write to stdout
movq $msg, %rsi  ; use string "Hello World"
movq $12, %rdx  ; write 12 characters
syscall  ; make syscall
```
How to implement your own tracer

1. Fork a tracer

   2. Child actions:
      2.1 Prepare child to be traced;
      ptrace(PTRACE_TRACEME);
      2.2 Wait until tracing starts;
      raise(SIGSTOP);
      2.3 Start a tracee.
      execv(tracee, NULL);

   3. Parent actions:
      3.1 Wait child to stop
      3.2 Configure distinguishable tracing:
      ptrace(PTRACE_SETOPTIONS, child, 0, PTRACE_O_TRACEFORK);
      3.3 Trace until tracee exits (see next slide)
How to implement your own tracer

1. Fork a tracer
2. Child actions:
   - Prepare child to be traced; `ptrace(PTRACE_TRACEME);`
   - Wait until tracing starts; `raise(SIGSTOP);`
   - Start a tracee; `execv(tracee, NULL);`
3. Parent actions:
   - Wait child to stop
   - Configure distinguishable tracing: `ptrace(PTRACE_SETOPTIONS, child, 0, PTRACE_O_TRACEG);`
   - Trace until tracee exits (see next slide)
How to implement your own tracer

1. Fork a tracer
2. Child actions:
   2.1 Prepare child to be traced;
      `ptrace(PTRACE_TRACEME);`

3. Parent actions:
   3.1 Wait child to stop
   3.2 Configure distinguishable tracing:
      `ptrace(PTRACE_SETOPTIONS, child, 0, PTRACE_O_TRACESYSGOOD);`
   3.3 Trace until tracee exits (see next slide)
How to implement your own tracer

1. Fork a tracer
2. Child actions:
   2.1 Prepare child to be traced;
      `ptrace(PTRACE_TRACEME);`
   2.2 Wait until tracing starts;
      `raise(SIGSTOP);`
3. Parent actions:
   3.1 Wait child to stop
   3.2 Configure distinguishable tracing:
      `ptrace(PTRACE_SETOPTIONS, child, 0, PTRACE_O_TRACEDGOOD);`
   3.3 Trace until tracee exits (see next slide)
How to implement you own tracer

1. Fork a tracer
2. Child actions:
   2.1 Prepare child to be traced;
      `ptrace(PTRACE_TRACEME);`
   2.2 Wait until tracing starts;
      `raise(SIGSTOP);`
   2.3 Start a tracee.
      `execv(tracee, NULL);`
3. Parent actions:
   3.1 Wait child to stop
   3.2 Configure distinguishable tracing:
      `ptrace(PTRACE_SETOPTIONS, child, 0, PTRACE_O_TRACEF);`
   3.3 Trace until tracee exits (see next slide)
How to implement you own tracer

1. Fork a tracer
2. Child actions:
   2.1 Prepare child to be traced;
      `ptrace(PTRACE_TRACEME);`
   2.2 Wait until tracing starts;
      `raise(SIGSTOP);`
   2.3 Start a tracee.
      `execv(tracee, NULL);`
3. Parent actions:
How to implement you own tracer

1. Fork a tracer
2. Child actions:
   2.1 Prepare child to be traced;
      ptrace(PTRACE_TRACEME);
   2.2 Wait until tracing starts;
      raise (SIGSTOP);
   2.3 Start a tracee.
      execv(tracee, NULL);
3. Parent actions:
   3.1 Wait child to stop
How to implement your own tracer

1. Fork a tracer
2. Child actions:
   2.1 Prepare child to be traced;
      `ptrace(PTRACE_TRACEME);`
   2.2 Wait until tracing starts;
      `raise(SIGSTOP);`
   2.3 Start a tracee.
      `execv(tracee, NULL);`
3. Parent actions:
   3.1 Wait child to stop
   3.2 Configure distinguishable tracing:
      `ptrace(PTRACE_SETOPTIONS, child, 0, PTRACE_O_TRACESYSGOOD);`
How to implement your own tracer

1. Fork a tracer
2. Child actions:
   2.1 Prepare child to be traced;
       `ptrace(PTRACE_TRACEME);`
   2.2 Wait until tracing starts;
       `raise(SIGSTOP);`
   2.3 Start a tracee.
       `execv(tracee, NULL);`
3. Parent actions:
   3.1 Wait child to stop
   3.2 Configure distinguishable tracing:
       `ptrace(PTRACE_SETOPTIONS, child, 0, PTRACE_O_TRACESYSGOOD);`
   3.3 Trace until tracee exits (see next slide)
Setting up tracer

Tracer loop:
1. Wait for a syscall start
2. Fetch syscall number (in RAX or EAX)
   \[\text{syscall} = \text{ptrace}(\text{PTRACE PEEKUSER}, \text{child}, \text{sizeof}(\text{long}) \times \text{ORIG RAX});\]
3. Wait for a syscall end
4. Fetch return code
   \[\text{retval} = \text{ptrace}(\text{PTRACE PEEKUSER}, \text{child}, \text{sizeof}(\text{long}) \times \text{RAX});\]
Catching a syscall

1. Continue tracee until next syscall:
   
   \[ \text{ptrace(PTRACE\_SYSCALL, child, 0, 0);} \]
Catching a syscall

1. Continue tracee until next syscall:
   ```c
   ptrace(PTRACE_SYSCALL, child, 0, 0);
   ```

2. Wait until until tracee is stopped:
   ```c
   waitpid(child, &status, 0);
   ```
Catching a syscall

1. Continue tracee until next syscall:
   ```c
   ptrace(PTRACE_SYSCALL, child, 0, 0);
   ```

2. Wait until tracee is stopped:
   ```c
   waitpid(child, &status, 0);
   ```

3. Check if stopped because of syscall:
   ```c
   WIFSTOPPED(status) &&
   WSTOPSIG(status) & 0x80
   ```
Catching a syscall

1. Continue tracee until next syscall:
   \[ \text{ptrace}(	ext{PTRACE\_SYSCALL}, \text{child}, 0, 0); \]

2. Wait until tracee is stopped:
   \[ \text{waitpid}(\text{child}, &\text{status}, 0); \]

3. Check if stopped because of syscall:
   \[ \text{WIFSTOPPED}(&\text{status}) \&\& \]
   \[ \text{WSTOPSIG}(\text{status}) \& 0x80 \]

4. If not, check if tracee exited:
   \[ \text{WIFEXITED}(\text{status}) \]
Catching a syscall

1. Continue tracee until next syscall:
   ```c
   ptrace(PTRACE_SYSCALL, child, 0, 0);
   ```

2. Wait until tracee is stopped:
   ```c
   waitpid(child, &status, 0);
   ```

3. Check if stopped because of syscall:
   ```c
   WIFSTOPPED(status) &&
   WSTOPSIG(status) & 0x80
   ```

4. If not, check if tracee exited:
   ```c
   WIFEXITED(status)
   ```

5. if not go to step 1.
Deciphering syscalls*

1. You need to decode syscall number to an actual function;
2. Find out which parameters it uses;
3. Fetch parameters;
4. Special treatment to strings;

More info: https://github.com/nelhage/ministrace/tree/master
Table of Contents

Introduction

Hands-on
   Tracing made easy
   Dynamic intervention
   Compiler-based helpers
   The GNU Debugger
   DIY debugging

The end
$> \text{svn co} \\
\text{http://svn.inf.tu-dresden.de/repos/advsysprog/debugging/08-formula/}

Debugging for fun & profit.
The program calculates:

\[
f = 333.75b^6 + a^2(11a^2b^2 - b^6 - 121b^4 - 2) + 5.5b^8 + \frac{a}{2b}
\]

where, \( a = 77617, b = 33096 \)
$> \text{svn co}
\text{http://svn.inf.tu-dresden.de/repos/advsysprog/debugging/08-formula/}

Debugging for fun & profit.
The program calculates:

\[ f = 333.75b^6 + a^2(11a^2b^2 - b^6 - 121b^4 - 2) + 5.5b^8 + \frac{a}{2b} \]

where, \( a = 77617 \), \( b = 33096 \)
Prints: \( f = -1.1805916207174113e21 \)
Assignment №8 (optional)

$> \text{svn co}
\text{http://svn.inf.tu-dresden.de/repos/advsysprog/debugging/08-formula/}

Debugging for fun & profit.
The program calculates:

$$f = 333.75b^6 + a^2(11a^2b^2 - b^6 - 121b^4 - 2) + 5.5b^8 + \frac{a}{2b}$$

where, \(a = 77617\), \(b = 33096\)
Prints: \(f = -1.1805916207174113e21\)
Assignment №8 (optional)

$> \text{svn co}
\text{http://svn.inf.tu-dresden.de/repos/advsysprog/debugging/08-formula/}

Debugging for fun & profit.
The program calculates:

\[ f = 333.75b^6 + a^2(11a^2b^2 - b^6 - 121b^4 - 2) + 5.5b^8 + \frac{a}{2b} \]

where, \( a = 77617, \ b = 33096 \)
Prints: \( f = -1.1805916207174113e21 \)
The right answer: \( f = -0.8273960599468213681 \ldots \)
Assignment №8 (optional)

$> \text{svn co}

http://svn.inf.tu-dresden.de/repos/advsysprog/debugging/08-formula/

Debugging for fun & profit.
The program calculates:

\[
f = 333.75b^6 + a^2(11a^2b^2 - b^6 - 121b^4 - 2) + 5.5b^8 + \frac{a}{2b}
\]

where, \( a = 77617 \), \( b = 33096 \)
Prints: \( f = -1.1805916207174113e21 \)
The right answer: \( f = -0.8273960599468213681\ldots \)

Why? How to fix?
Conclusion

There are only two hard things in Computer Science: cache invalidation, naming things, and off-by-one errors.

— Phil Karlton
Sources for slide 7

1. Nenad Stojkovic, flickr: nenadstojkovic, CC-BY 2.0
6. Billy Brown, flickr, CC-BY 2.0
8. https://openstax.org/, CC-BY 4.0