An Introduction to ptrace and Its Applications

Advanced Programming in the UNIX Environment

Chun-Ying Huang <chuang@cs.nctu.edu.tw>

Outline

Overview

Features and commands

Applications

- Instruction counter
- Execution trace dump
- System call tracer
- Change program control flow
- Automated Debugger

Note: Applications introduced here may be done by alternative approaches. Since this is a ptrace tutorial, everything is ptraced! ☺

Overview

ptrace is a system call

It allows one process (***tracer***) to observe and control the execution of another process (***tracee***)

- Memory reading and writing
- Register reading and writing
- Monitor runtime state changes

Applications

- System call tracer
- Debugger
- …

Procedures

The tracer has to ***attach*** to a tracee first

- Attachment and commands are applied to per-thread
- Every thread can be attached individually

The general form for ptrace system call

```
long ptrace(enum __ptrace_request request, pid_t pid,
            void *addr, void *data);
```

Common ptrace Commands

Command	Description
PTRACE_ATTACH	Attach to a process
PTRACE_DETACH	Detach from a tracee, and restart the process
PTRACE_TRACEME	Indicate that this process is to be traced by its parent
PTRACE_CONT	Restart the stopped tracee process
PTRACE_SYSCALL	PTRACE_CONT, but stop at the next entry to or exit from a syscall
PTRACE_SINGLESTEP	PTRACE_CONT, but stop after execution of a single instruction
PTRACE_PEEKDATA	Read a word from tracee's memory
PTRACE_PEEKUSER	Read a word from tracee's user data (usually registers)
PTRACE_GETREGS	Copy the tracee's general-purpose registers
PTRACE_POKEDATA	Write a word to tracee's memory
PTRACE_POKEUSER	Write a word to tracee's user data (usually registers)
PTRACE_SETREGS	Modify the tracee's general-purpose registers
PTRACE_SETOPTIONS	Set ptrace options

Basic Control

PTRACE_ATTACH

Usually a process can trace its child processes

- Parent: calls ptrace(PTRACE_ATTACH), or
- Child: calls ptrace(PTRACE_TRACEME)

But ptrace can actually attach to ***any*** process in the system – See restrictions in the next page

A SIGSTOP signal is sent to the tracee

- Use waitpid to wait for the child process
- Determine the status of the child by using WIFSTOPPED()

```
Long ptrace(PTRACE_ATTACH, child, 0, 0);
Returns 0 on success, or -1 on error
```

PTRACE_ATTACH: Restrictions

Restrict the ability to trace a process

/proc/sys/kernel/yama/ptrace_scope

- 0 – classic ptrace permission: no restriction
- 1 – restricted ptrace:
  allow a parent and process having CAP_SYS_PTRACE ability
- 2 – admin-only:
  allow only process having the CAP_SYS_PTRACE ability
- 3 – no attach

PTRACE_TRACEME

Turn a process to a tracee

- It should be traced by its parent

```
Long ptrace(PTRACE_TRACEME, 0, 0, 0);
Returns 0 on success, or -1 on error
```

A typical implementation

- A parent forks a child
- The child calls ptrace(PTRACE_TRACEME)
- The child then
  - raise(SIGSTOP) to stop itself, or
  - exec another program – SIGCHLD is sent to its parent w/ term signal = SIGTRAP
    - PTRACE_O_TRACEEXEC option should be disabled (default)
- The parent waits for the child, and then control the execution of its child

PTRACE_CONT

A tracee is stopped if

- It is attached by a tracer using PTRACE_ATTACH
- It is a child process and calls PTRACE_TRACEME and exec

We can ***restart*** (continue) the stopped process by sending PTRACE_CONT command from the tracer

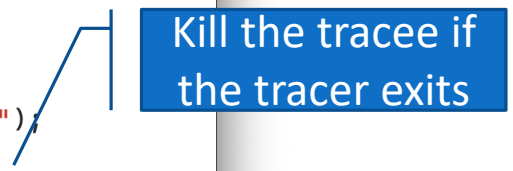
- ***sig*** is usually set to zero – no signal is deliver to the tracee

```
long ptrace(PTRACE_CONT, child, 0, sig);
Returns 0 on success, or -1 on error
```

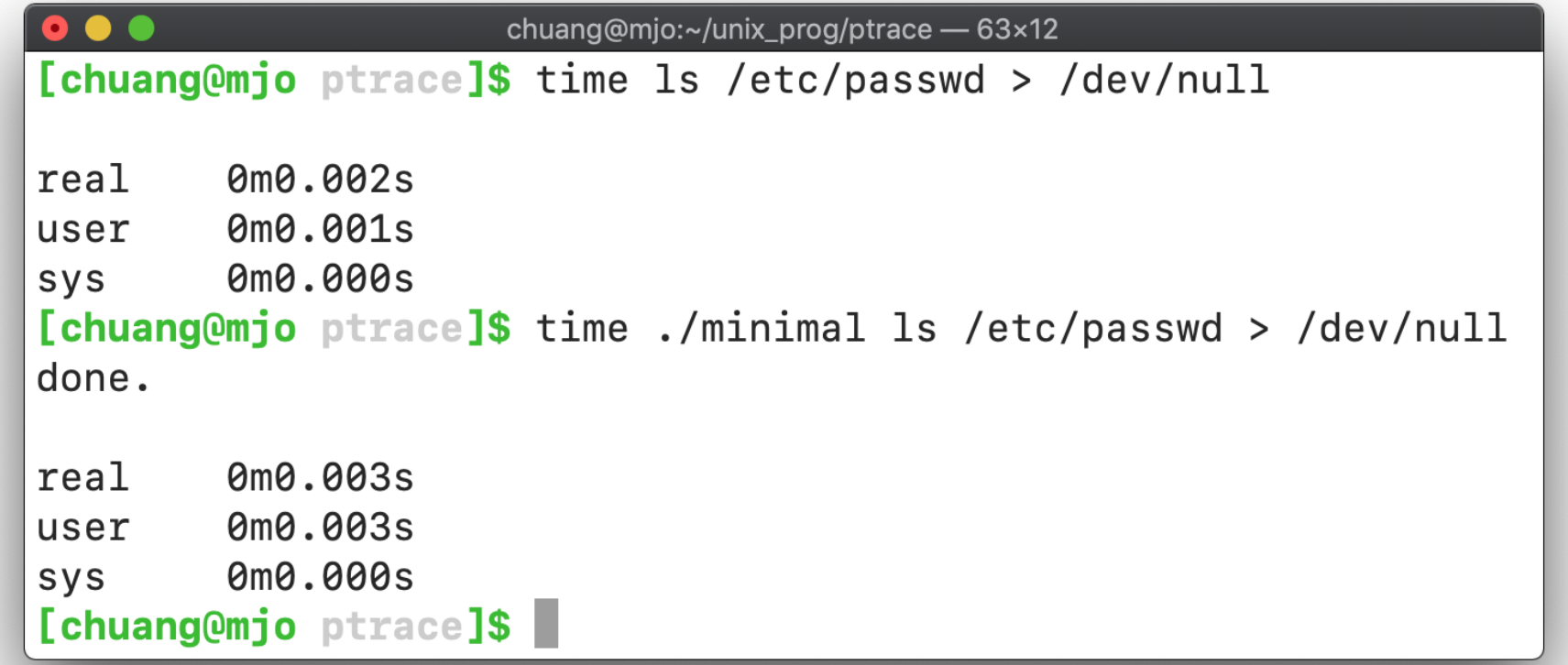
A Minimal Example

chuang@mjo:~/unix_prog/ptrace — 81×23

```
14 int main(int argc, char *argv[]) {
15         pid_t child;
16         if(argc < 2) {
17                 fprintf(stderr, "usage: %s program\n", argv[0]);
18                 return -1;
19         }
20         if((child = fork()) < 0) errquit("fork");
21         if(child == 0) {
22                 if (ptrace(PTRACE_TRACEME, 0, 0, 0) < 0) errquit("ptrace");
23                 execvp(argv[1], argv+1);
24                 errquit("execvp");
25         } else {
26                 int status;
27                 if(waitpid(child, &status, 0) < 0) errquit("wait");
28                 assert(WIFSTOPPED(status));
29                 ptrace(PTRACE_SETOPTIONS, child, 0, PTRACE_O_EXITKILL);
30                 ptrace(PTRACE_CONT, child, 0, 0);
31                 waitpid(child, &status, 0);
32                 perror("done");
33         }
34         return 0;
35 }
                                                    14,33         92%
```



A Minimal Example (Cont'd)



Practice

traceme

Finer Program Control Flow Granularity

In addition to PTRACE_CONT, you can have a ***finer*** program control flow granularity

- PTRACE_SINGLESTEP – stop the tracee before running an instruction
- PTRACE_SYSCALL – stop the tracee ***before*** the entry to and ***after*** the exit from a system call

When a tracee is stopped

- Memory content can be read or modified – PTRACE_PEEKTEXT (PTRACE_PEEKDATA), PTRACE_POKETEXT (PTRACE_POKEDATA)
- Register values can be read or modified – PTRACE_PEEKUSER, PTRACE_GETREGS, PTRACE_POKEUSER, PTRACE_SETREGS

PTRACE_SINGLESTEP

Restart the stopped tracee as for PTRACE_CONT

Stop the tracee ***after*** execution of a single instruction

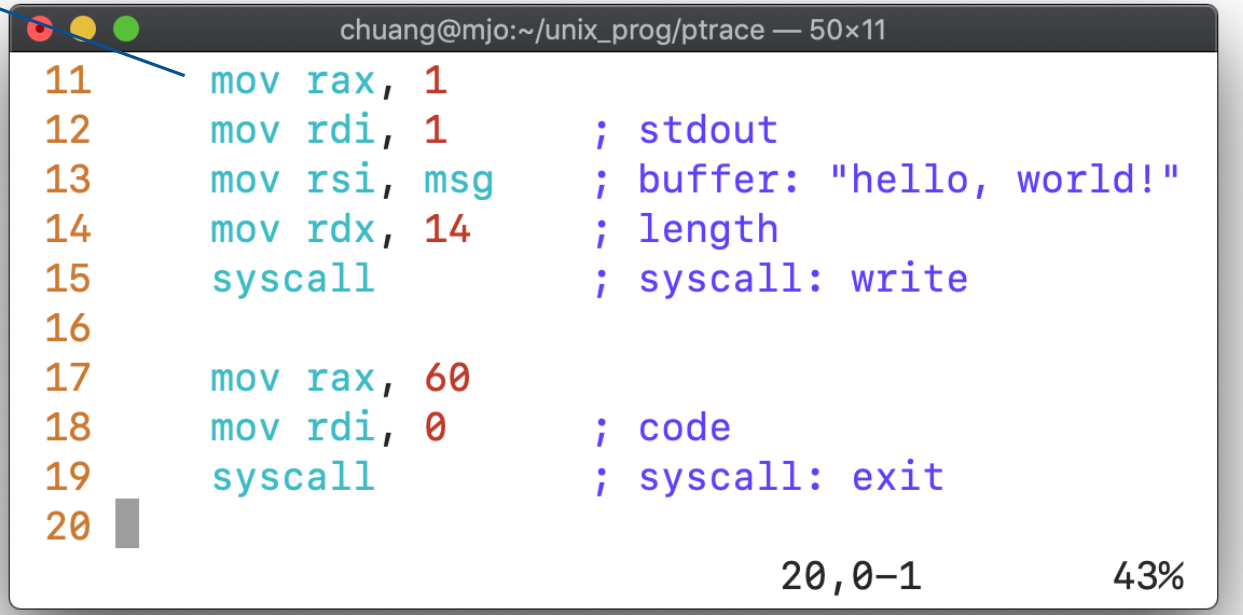
```
long ptrace(PTRACE_SINGLESTEP, child, 0, 0);
Returns 0 on success, or -1 on error
```

PTRACE_SINGLESTEP: Trace a "Hello, World" Example

Parent: fork() and then wait()

Child: TRACEME and then exec()

Stopped HERE!



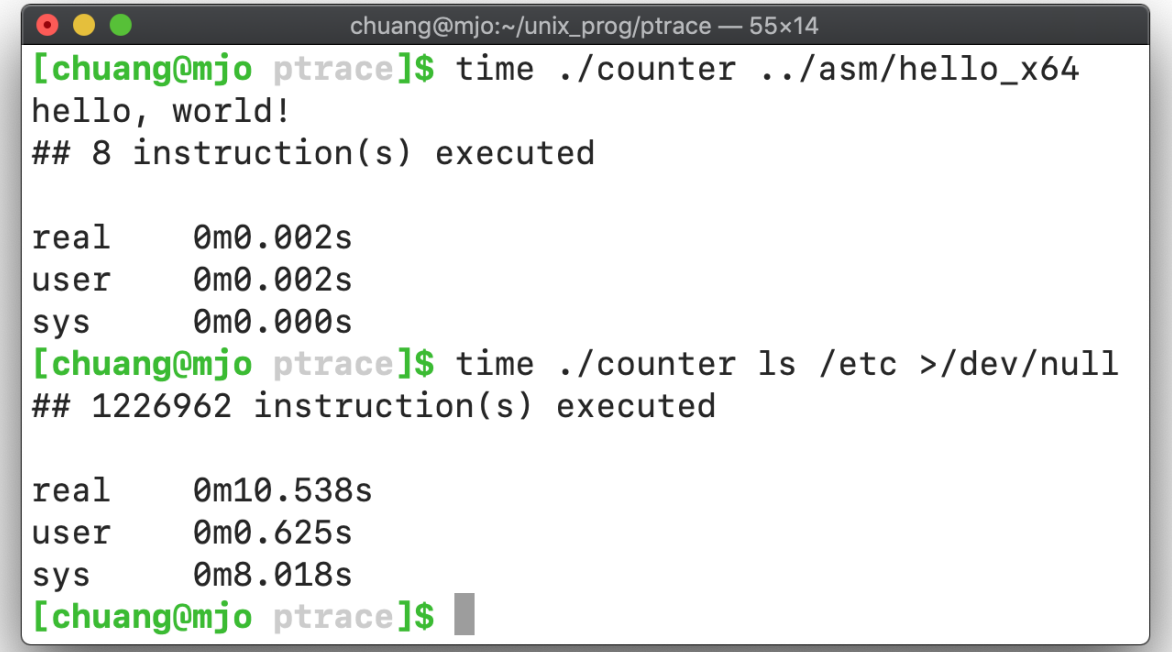
Application: Instruction Counter

Instruction Counter

chuang@mjo:~/unix_prog/ptrace — 90×27

```
14 int
15 main(int argc, char *argv[]) {
16     pid_t child;
17     if(argc < 2) {
18         fprintf(stderr, "usage: %s program [args ...]\n", argv[0]);
19         return -1;
20     }
21     if((child = fork()) < 0) errquit("fork");
22     if(child == 0) {
23         if (ptrace(PTRACE_TRACEME, 0, 0, 0) < 0) errquit("ptrace@child");
24         execvp(argv[1], argv+1);
25         errquit("execvp");
26     } else {
27         long long counter = 0LL;
28         int wait_status;
29         if(waitpid(child, &wait_status, 0) < 0) errquit("wait");
30         ptrace(PTRACE_SETOPTIONS, child, 0, PTRACE_O_EXITKILL);
31         while (WIFSTOPPED(wait_status)) {
32             counter++;
33             if (ptrace(PTRACE_SINGLESTEP, child, 0, 0) < 0) errquit("ptrace@parent");
34             if(waitpid(child, &wait_status, 0) < 0) errquit("wait");
35         }
36         fprintf(stderr, "## %lld instruction(s) executed\n", counter);
37     }
38     return 0;
39 }
                                                          39,2          92%
```

Instruction Counter (Cont'd)



Practice

countme

Reading from a Tracee

PEEKTEXT, PEEKUSER, and GETREGS

We can further read memory content and register values from a tracee

- Clear ***errno*** before calling PEEK* functions!
- Get a word from a memory address of the tracee

```
long ptrace(PTRACE_PEEKTEXT, child, addr, 0);
Returns a word, check errno for errors
```

- Get a register word from a tracee's user data area

```
long ptrace(PTRACE_PEEKUSER, child, offset, 0);
Returns a word, check errno for errors
```

- Get general-purpose register values from a tracee

```
long ptrace(PTRACE_GETREGS, child, 0, data);
Returns 0 on success, or -1 on error
```

PEEKTEXT and PEEKUSER: Address, Offset, and "Words"

PEEKTEXT: addr alignment is hardware dependent

- On Intel, it is not necessary to be aligned
- You can pass any address you like

PEEKUSER: offset usually has to be ***word***-aligned

PEEK* functions returns a "word" from the tracee

The size of the term "word" used in ptrace documents is architecture-dependent

- On 32-bit machine, it is a 4-byte data
- On 64-bit machine, it is a 8-byte data

PEEKUSER and GETREGS: Data Structure

The two commands work with the data structure defined in <sys/user.h>

GETREGS

- Get all general-purpose register values in one-shot
- Store in **user_regs_struct** data structure

PEEKUSER

- Get only-one register value in one command

chuang@mjo:~/unix_prog/ptrace — 54×36

```
42 struct user_regs_struct
43 {
44   __extension__ unsigned long long int r15;
45   __extension__ unsigned long long int r14;
46   __extension__ unsigned long long int r13;
47   __extension__ unsigned long long int r12;
48   __extension__ unsigned long long int rbp;
49   __extension__ unsigned long long int rbx;
50   __extension__ unsigned long long int r11;
51   __extension__ unsigned long long int r10;
52   __extension__ unsigned long long int r9;
53   __extension__ unsigned long long int r8;
54   __extension__ unsigned long long int rax;
55   __extension__ unsigned long long int rcx;
56   __extension__ unsigned long long int rdx;
57   __extension__ unsigned long long int rsi;
58   __extension__ unsigned long long int rdi;
59   __extension__ unsigned long long int orig_rax;
60   __extension__ unsigned long long int rip;
61   __extension__ unsigned long long int cs;
62   __extension__ unsigned long long int eflags;
63   __extension__ unsigned long long int rsp;
64   __extension__ unsigned long long int ss;
65   __extension__ unsigned long long int fs_base;
66   __extension__ unsigned long long int gs_base;
67   __extension__ unsigned long long int ds;
68   __extension__ unsigned long long int es;
69   __extension__ unsigned long long int fs;
70   __extension__ unsigned long long int gs;
71 };
72
73 struct user
74 {
75   struct user_regs_struct       regs;
76   ...
                                   76,6          28%
```

Application: Execution Trace Dump

Execution Trace Dump (1/3)

Revise the 'instruction counter' example

chuang@mjo:~/unix_prog/ptrace — 128×22

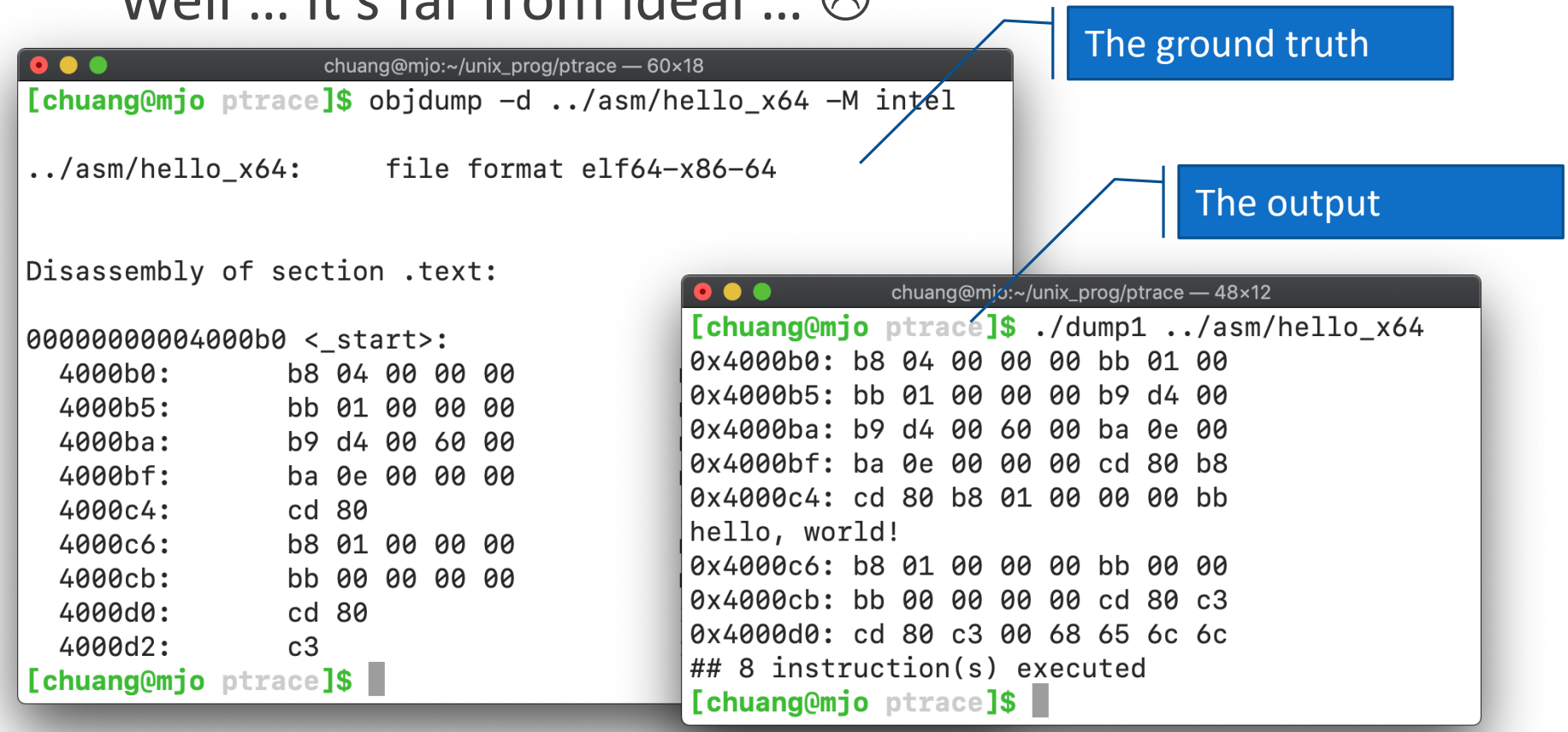
```
34          while (WIFSTOPPED(wait_status)) {
35              long ret;
36              unsigned long long rip;
37              struct user_regs_struct regs;
38              unsigned char *ptr = (unsigned char *) &ret;
39              counter++;
40  #if USE_PEEKUSER
41              if((rip = ptrace(PTRACE_PEEKUSER, child, ((unsigned char *) &regs.rip) - ((unsigned char *) &regs), 0)) != 0) {
42  #else
43              if(ptrace(PTRACE_GETREGS, child, 0, &regs) == 0) {
44                  rip = regs.rip;
45  #endif
46                  ret = ptrace(PTRACE_PEEKTEXT, child, rip, 0);
47                  fprintf(stderr, "0x%llx: %2.2x %2.2x %2.2x %2.2x %2.2x %2.2x
48                      rip,
49                      ptr[0], ptr[1], ptr[2], ptr[3], ptr[4], ptr[5], ptr[6], ptr[7]);
50              }
51              if(ptrace(PTRACE_SINGLESTEP, child, 0, 0) < 0) errquit("ptrace@parent");
52              if(waitpid(child, &wait_status, 0) < 0) errquit("waitpid");
53          }
54          fprintf(stderr, "## %lld instruction(s) executed\n", counter);
                                                                  54,2-8        89%
```

Calculate the offset

Get everything in one-shot

Execution Trace Dump (2/3)

Well … it's far from ideal … ☹



Execution Trace Dump (3/3)

Testing it with a "larger" program

```
chuang@mjo:~/unix_prog/ptrace — 82×6
[chuang@mjo ptrace]$ time ./dump1 ls /etc/passwd > /dev/null 2>/tmp/dump.log

real    0m7.283s
user    0m0.699s
sys     0m4.322s
[chuang@mjo ptrace]$
```

```
chuang@mjo:~/unix_prog/ptrace — 82×10
     1 0x7f53d3aec000: 48 89 e7 e8 f8 0d 00 00
     2 0x7f53d3aec003: e8 f8 0d 00 00 49 89 c4
     3 0x7f53d3aece00: f3 0f 1e fa 55 48 89 e5
     4 0x7f53d3aece04: 55 48 89 e5 41 57 41 56
     5 0x7f53d3aece05: 48 89 e5 41 57 41 56 41
     6 0x7f53d3aece08: 41 57 41 56 41 55 41 54
     7 0x7f53d3aece0a: 41 56 41 55 41 54 49 89
     8 0x7f53d3aece0c: 41 55 41 54 49 89 fc 53
     9 0x7f53d3aece0e: 41 54 49 89 fc 53 48 83
"/tmp/dump.log" 372376 lines --0%--                    1,1           Top
```

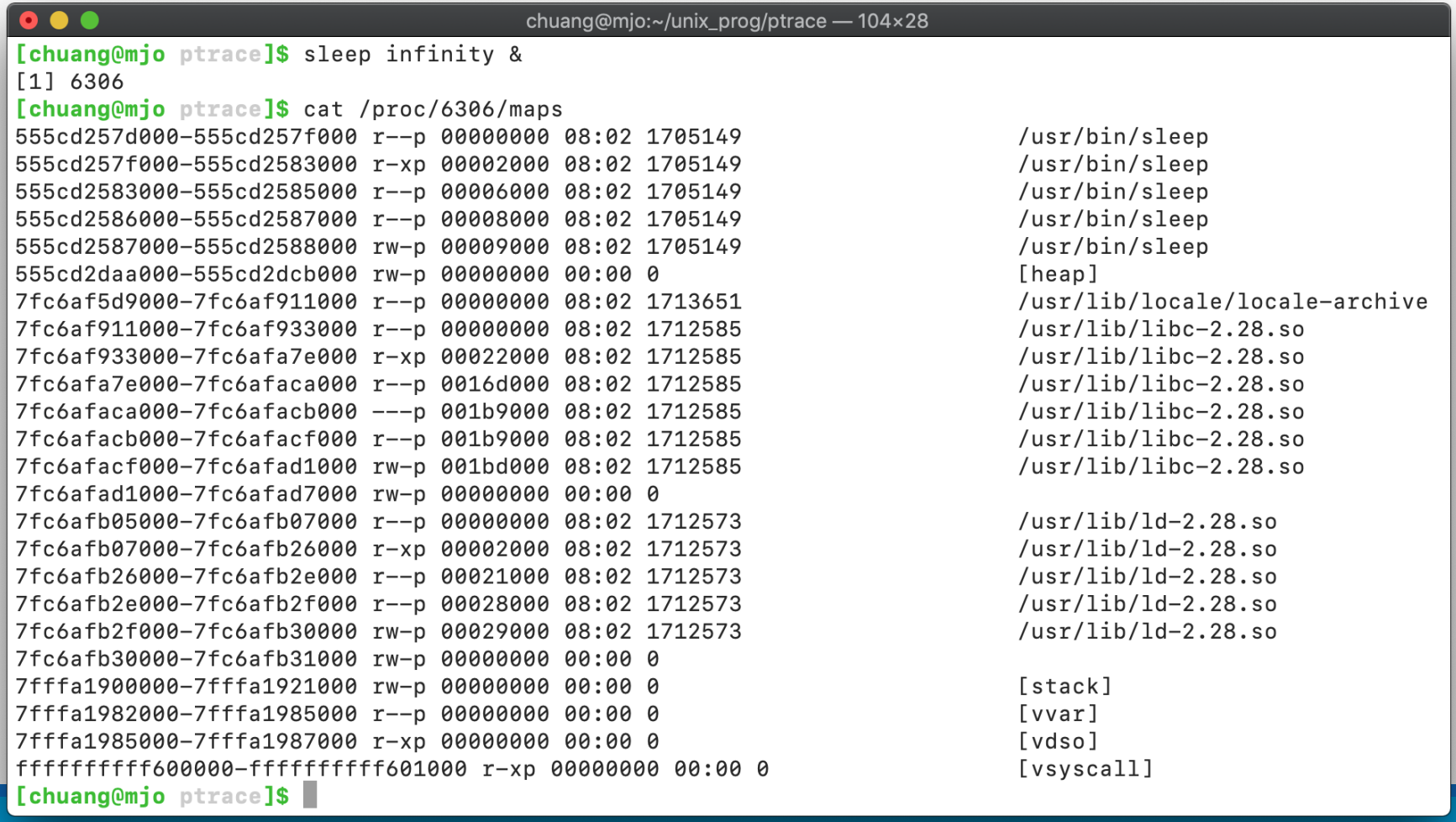
It's really unfriendly

How Could We Revise "dump1"?

Directions

- Show module names – /proc/{pid}/maps
- Show assembly codes – Capstone

/proc/{pid}/maps



Capstone

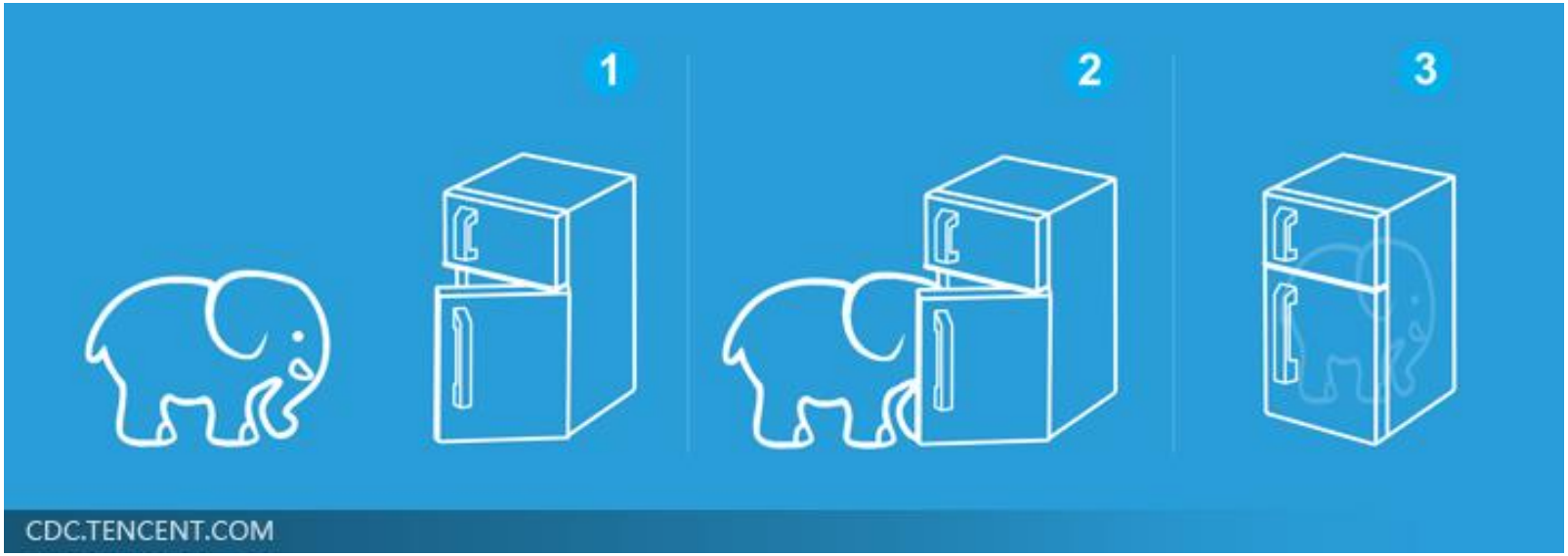
The ultimate disassembler

- http://www.capstone-engine.org/

Features

- Open source
- Multi-architecture support: ARM, Intel, PowerPC, MIPS, …, support 32-bit and 64-bit architecture
- Easy-to-use
- Thread-safe
- Provide many language bindings: C/C++, python, ruby, …
- …

Disassemble in Three Steps



Disassemble in Three Steps

Please read /usr/include/capstone/capstone.h

Initialize the engine

```
cs_err cs_open(cs_arch arch, cs_mode mode, csh *handle);
```

Disassemble

```
size_t cs_disasm(csh handle,
        const uint8_t *code, size_t code_size, uint64_t address,
        size_t count, cs_insn **insn);
```

Close the engine

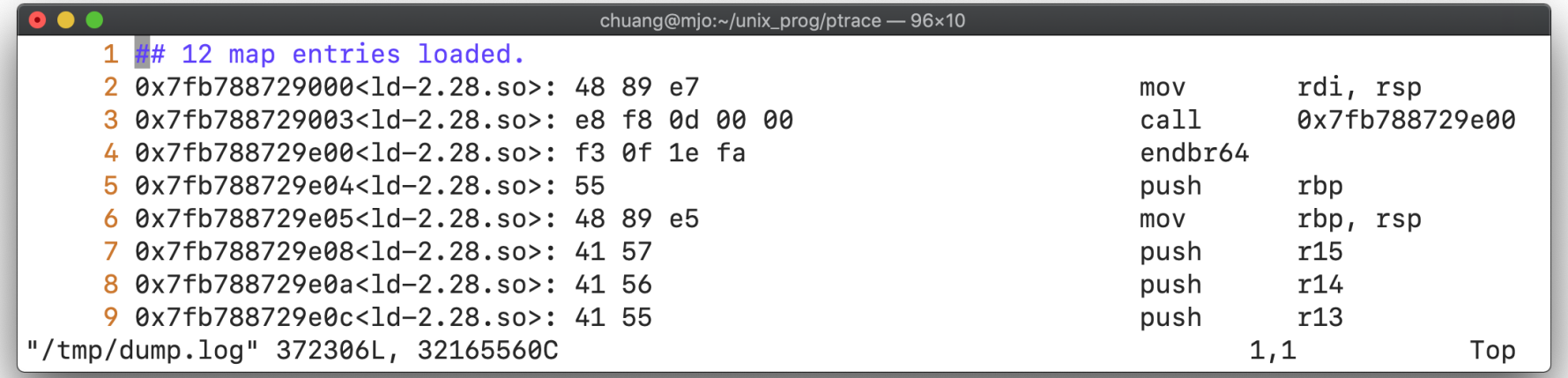
```
cs_err cs_close(csh *handle);
```

Disassemble in Three Steps

chuang@mjo:~/unix_prog/ptrace — 88×26

```
 1 #include <capstone/capstone.h>
 2
 3     static csh cshandle = 0;    // globally accessible
 4     if(cs_open(CS_ARCH_X86, CS_MODE_64, &cshandle) != CS_ERR_OK)
 5             return -1;
 6
 7     ...
 8
 9     cs_insn *insn;
10     if((count = cs_disasm(cshandle, (uint8_t*) buf, bufsz, rip, 0, &insn)) > 0) {
11         int i;
12         for(i = 0; i < count; i++) {
13             instruction in;
14             in.size = insn[i].size;
15             in.opr  = insn[i].mnemonic;
16             in.opnd = insn[i].op_str;
17             memcpy(in.bytes, insn[i].bytes, insn[i].size);
18         }
19         cs_free(insn, count);
20     }
21
22     ...
23
24     cs_close(&cshandle);
25
                                                    10,54-57      Top
```

Application: Revised Execution Trace Monitor



Practice

capstone

PTRACE_SYSCALL, and System Call Tracer

PTRACE_SYSCALL

Restart the stopped tracee as for PTRACE_CONT

Stop the tracee ***before*** the entry to and ***after*** the exit from execution of a system call

```
long ptrace(PTRACE_SYSCALL, child, 0, 0);
Returns 0 on success, or -1 on error
```

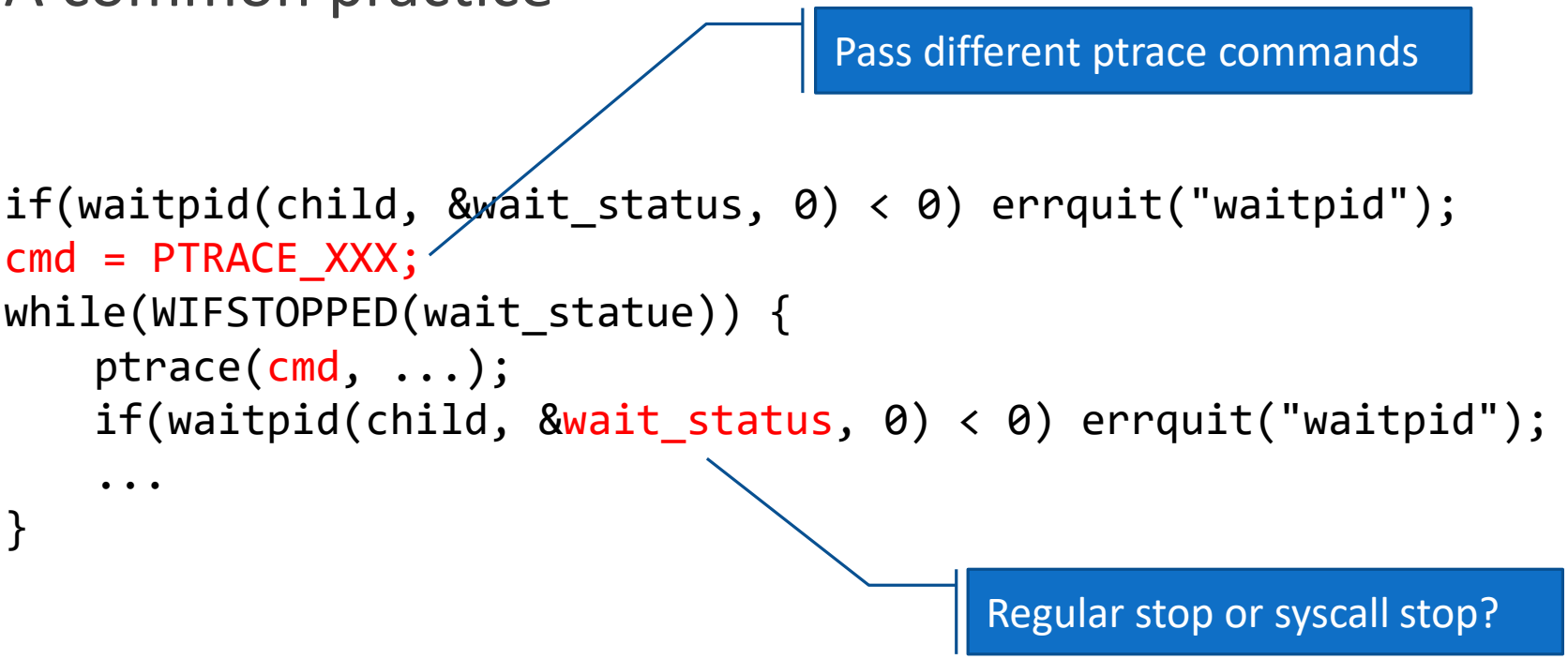
Key issues

- How to differentiate ***regular stop*** and ***syscall stop***?
- How to differentiate ***entry*** point (before entering a syscall) and ***exit*** point (after returned from a syscall)?

Regular Stop vs. Syscall Stop

The tracer always receive stop status from a tracee

A common practice



Regular Stop vs. Syscall Stop (Cont'd)

The PTRACE_O_TRACESYSGOOD option

- Mark an additional 0x80 flag for syscall stop

```
ptrace(PTRACE_SETOPTIONS, child, 0,
       ...|PTRACE_O_TRACESYSGOOD);

...

if(WSTOPSIG(wait_status) & 0x80) {
   // syscall stop
}
```

Entry vs. Exit Point

You have to differentiate entry and exit points by yourself

- Setup a counter for syscall stops
- The first syscall stop is for an entry point
- The second syscall stop is for an exit point
- … and so on

```
int enter = 0x01;
...
if(WSTOPSIG(wait_status) & 0x80) { // syscall stop
    if(enter) { ... } else { ... }
    enter ^= 0x01;
}
```

Read Syscall Parameters

Linux Intel x86_64 syscall calling convention

- RAX = system call id
- Parameters: RDI, RSI, RDX, R10, R8, R9

Register values can be read by using GETREGS

For entry point

- System call id is stored in **orig_rax** of **user_regs_struct** structure

For exit point

- System call return value is stored in **rax** of **user_regs_struct** structure

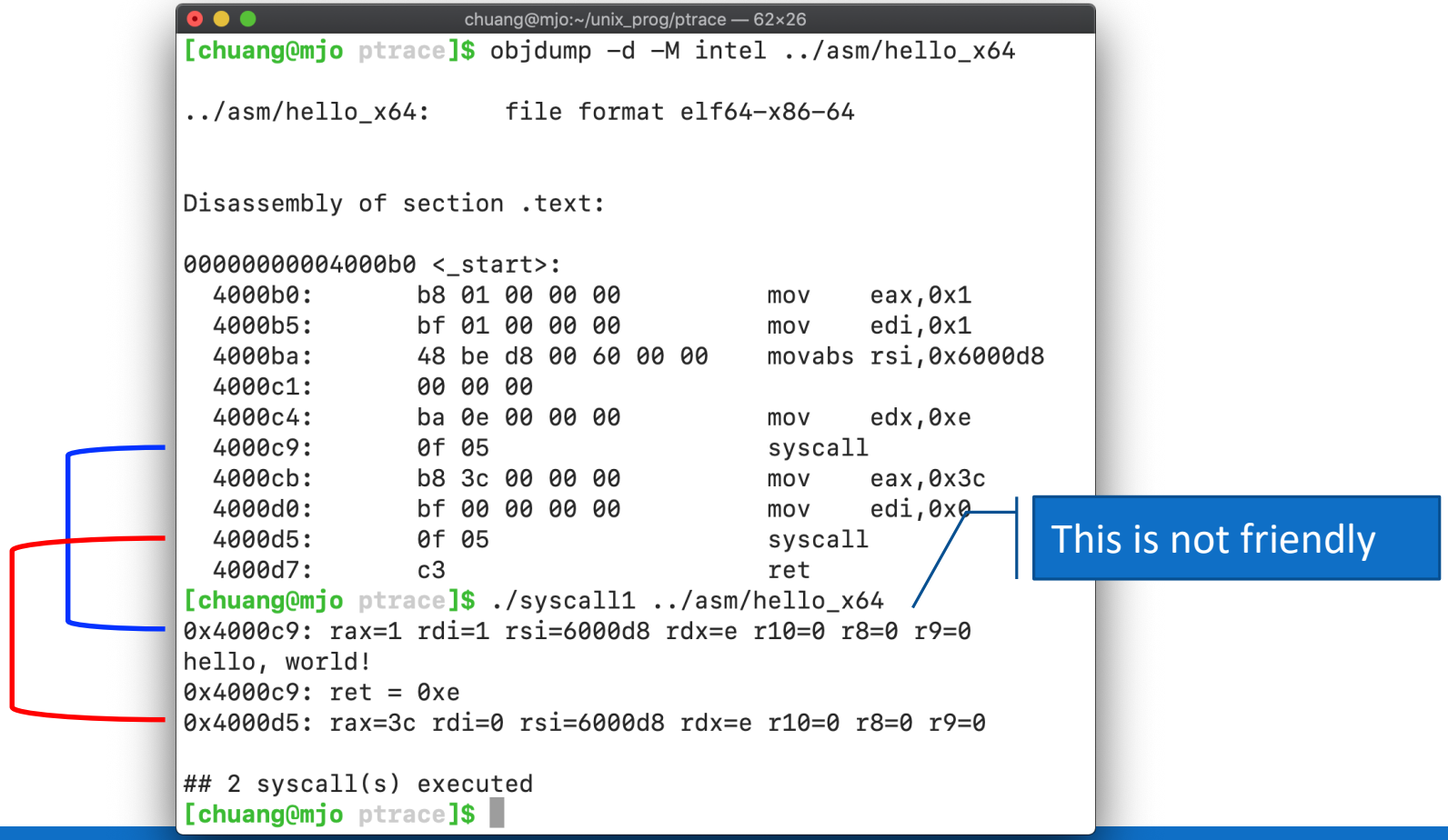
The rest parameters simply follow the calling convention

Application: System Call Tracer

chuang@mjo:~/unix_prog/ptrace — 109×22

```
32        if(waitpid(child, &wait_status, 0) < 0) errquit("waitpid");
33        ptrace(PTRACE_SETOPTIONS, child, 0, PTRACE_O_EXITKILL|PTRACE_O_TRACESYSGOOD);
34        while (WIFSTOPPED(wait_status)) {
35            struct user_regs_struct regs;
36            if(ptrace(PTRACE_SYSCALL, child, 0, 0) != 0) errquit("ptrace@parent");
37            if(waitpid(child, &wait_status, 0) < 0) errquit("waitpid");
38            if(!WIFSTOPPED(wait_status) || !(WSTOPSIG(wait_status) & 0x80)) continue;
39            if(ptrace(PTRACE_GETREGS, child, 0, &regs) != 0) errquit("ptrace@parent");
40            if(enter) { // syscall enter
41                /* rip has to subtract 2 because machine code of 'syscall' = 0x0f 05 */
42                fprintf(stderr, "0x%llx: rax=%llx rdi=%llx rsi=%llx rdx=%llx r10=%llx r8=%llx r9=%llx\n",
43                    regs.rip-2, regs.orig_rax, regs.rdi, regs.rsi, regs.rdx, regs.r10, regs.r8, regs.r9);
44                if(regs.orig_rax == 0x3c || regs.orig_rax == 0xe7)
45                    fprintf(stderr, "\n"); /* exit || exit_group */
46                counter++;
47            } else {    // syscall exit
48                fprintf(stderr, "0x%llx: ret = 0x%llx\n", regs.rip-2, regs.rax);
49            }
50            enter ^= 0x01;
51        }
52        fprintf(stderr, "## %lld syscall(s) executed\n", counter);
                                                                    51,3-9        88%
```

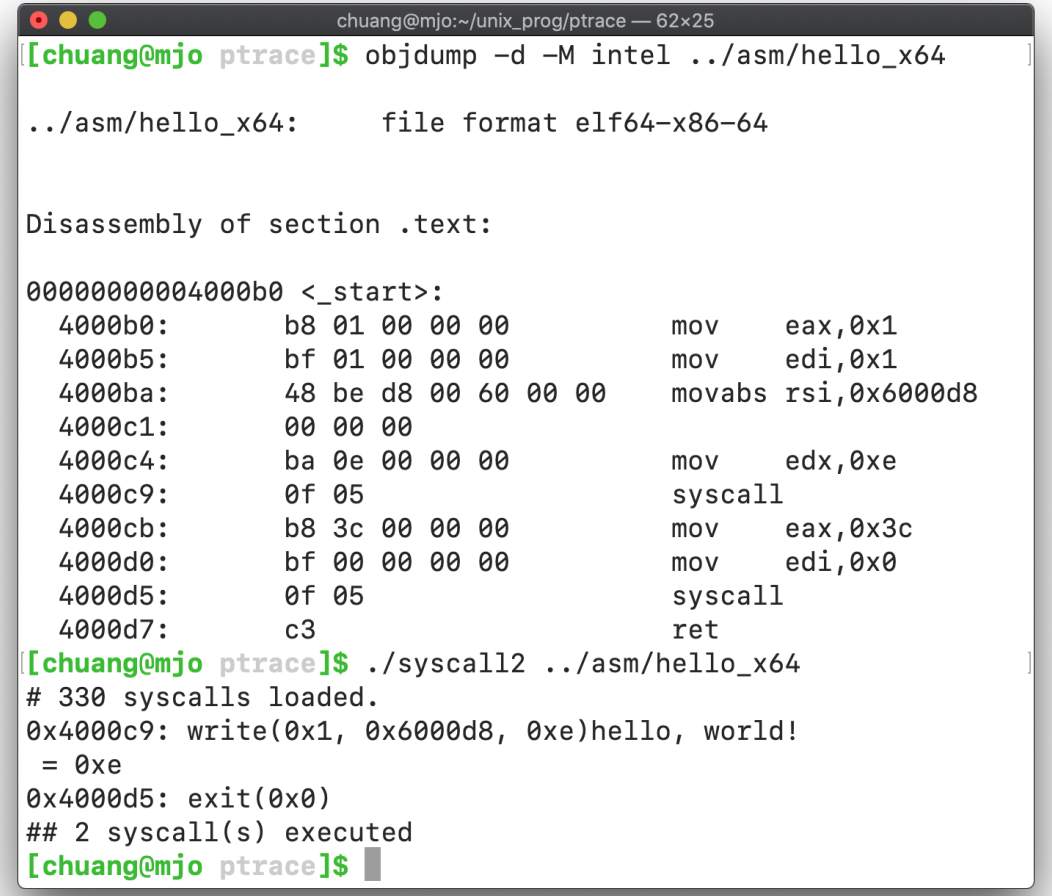
Application: System Call Tracer



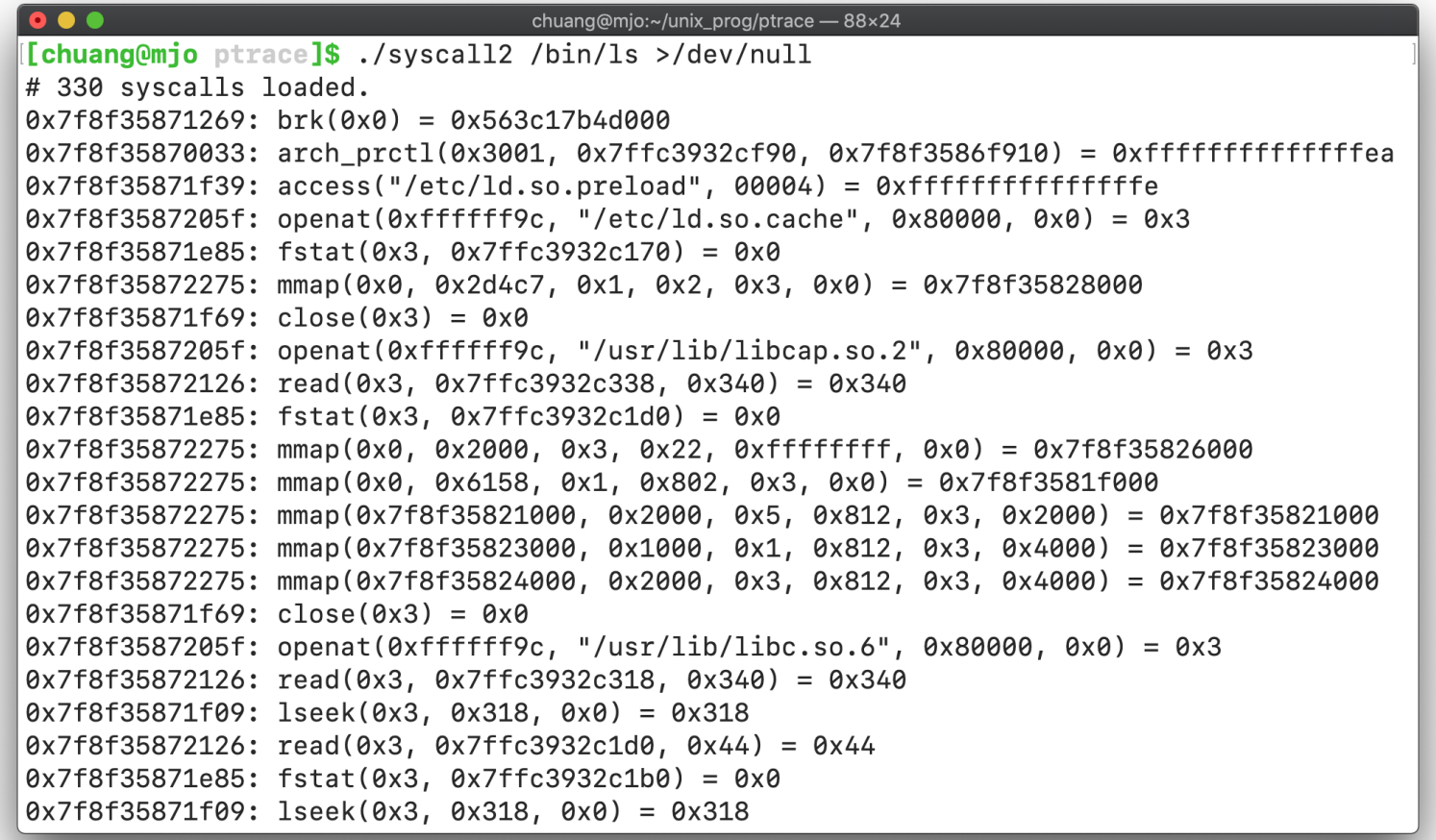
Interpreting System Calls

It can be completely done by yourself

- Create a mapping table from a system call id to its name
- Show raw values of parameters
- Alternatively, peek data from the tracee, e.g., strings.



Interpreting System Calls – Another Example



Writing to a Tracee

POKETEXT, POKEUSER, and SETREGS

We can also write memory content and register values to a tracee

- Write a word to a memory address of the tracee

```
long ptrace(PTRACE_POKETEXT, child, addr, word);
Returns 0 on success, or -1 on error
```

- Write a register word to a tracee's user data area

```
long ptrace(PTRACE_POKEUSER, child, offset, word);
Returns 0 on success, or -1 on error
```

- Write general-purpose register values to a tracee

```
long ptrace(PTRACE_SETREGS, child, 0, data);
Returns 0 on success, or -1 on error
```

POKETEXT and POKEUSER: Address, Offset, and "Words"

Similar to PEEK* commands

POKETEXT: addr alignment is hardware dependent

- On Intel, it is not necessary to be aligned
- You can pass any address you like

POKEUSER: offset usually has to be ***word***-aligned

The size of the term "word" used in ptrace documents is architecture-dependent

- On 32-bit machine, it is a 4-byte data
- On 64-bit machine, it is a 8-byte data

POKEUSER and SETREGS: Data Structure

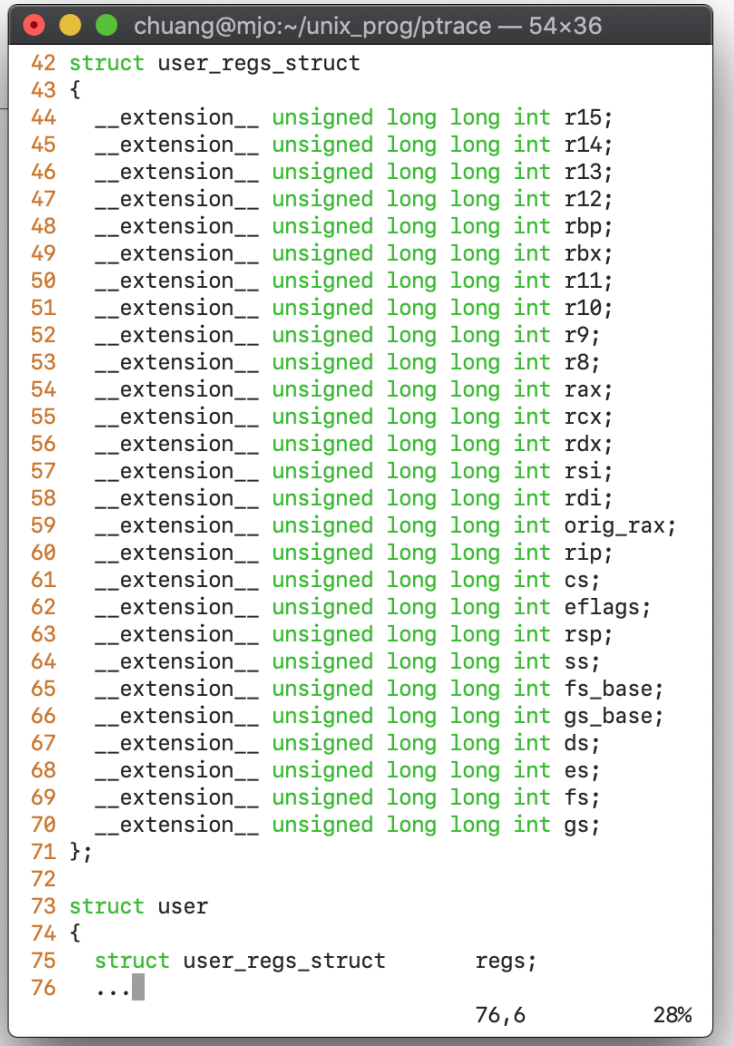
The two commands work with the data structure defined in <sys/user.h>

SETREGS

- Write all general-purpose register values in one-shot
- Store in **user_regs_struct** data structure

POKEUSER

- Write only one register value in one command



Application: Change Program Control Flow

Change Program Control Flow

It is pretty straightforward to use **JMP** instruction

Common JMP OP codes in x86_64

Assembly	OP code	Description
JMP rel8	EB + offset	Jump to RIP + 8-bit sign-extended address
JMP rel32	E9 + offset	Jump to RIP + 32-bit sign-extended address
JMP r/m64	FF + r/m64	Jump to the target address

Official OP code Reference: https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf

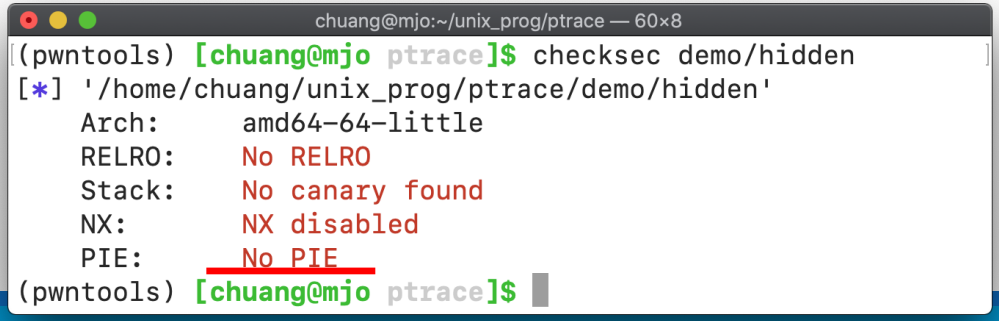
Change Program Control Flow (Cont'd)

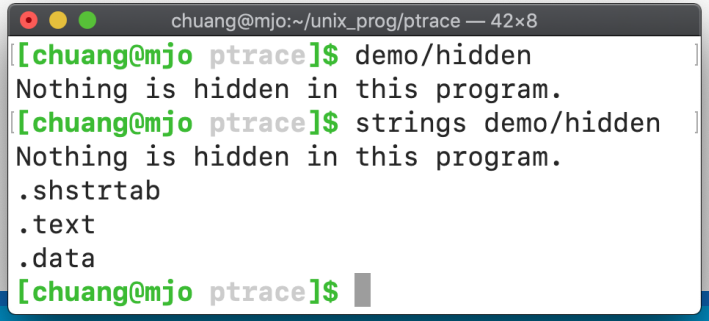
You have to know …

- Address to modify the codes
- Address to jump

It is sometimes complicated to get the addresses

- But here we use a pretty simple program to demonstrate this feature
- The demo/hidden program





The "Hidden" Part

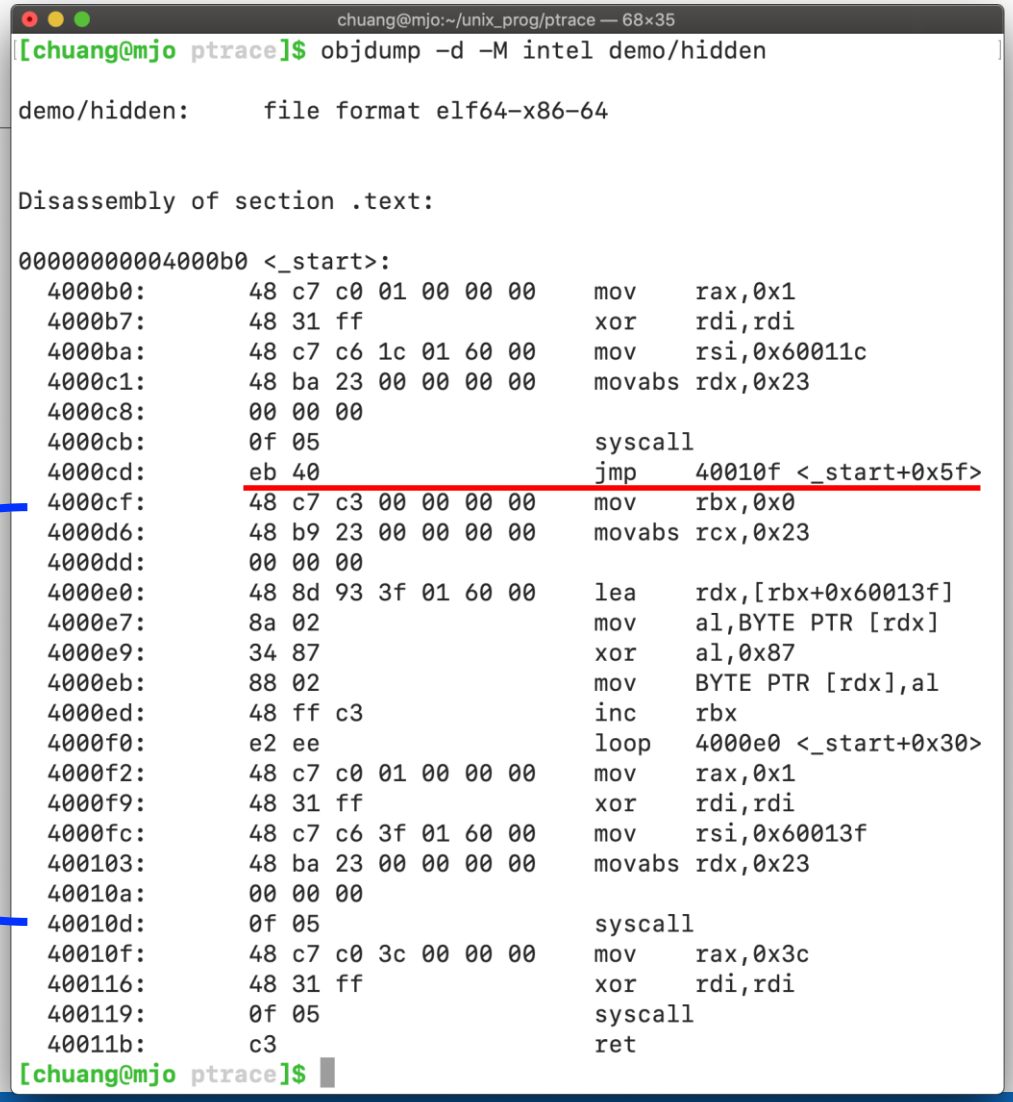
No PIE

- Addresses are fixed!

The hidden part

We want to see the hidden part!

- Patch on-the-fly
- Jump to 0x4000cf from beginning
- Alternatively, we can *remove* "0xeb 0x40"



Jump to 0x4000cf

The starting address

- 0x4000b0 – This is the program entry point
- ptrace stops at here, and we can then patch the codes!

The target address

- 0x4000cf – The entry point of the hidden part

We want to patch the code @ 0x4000b0

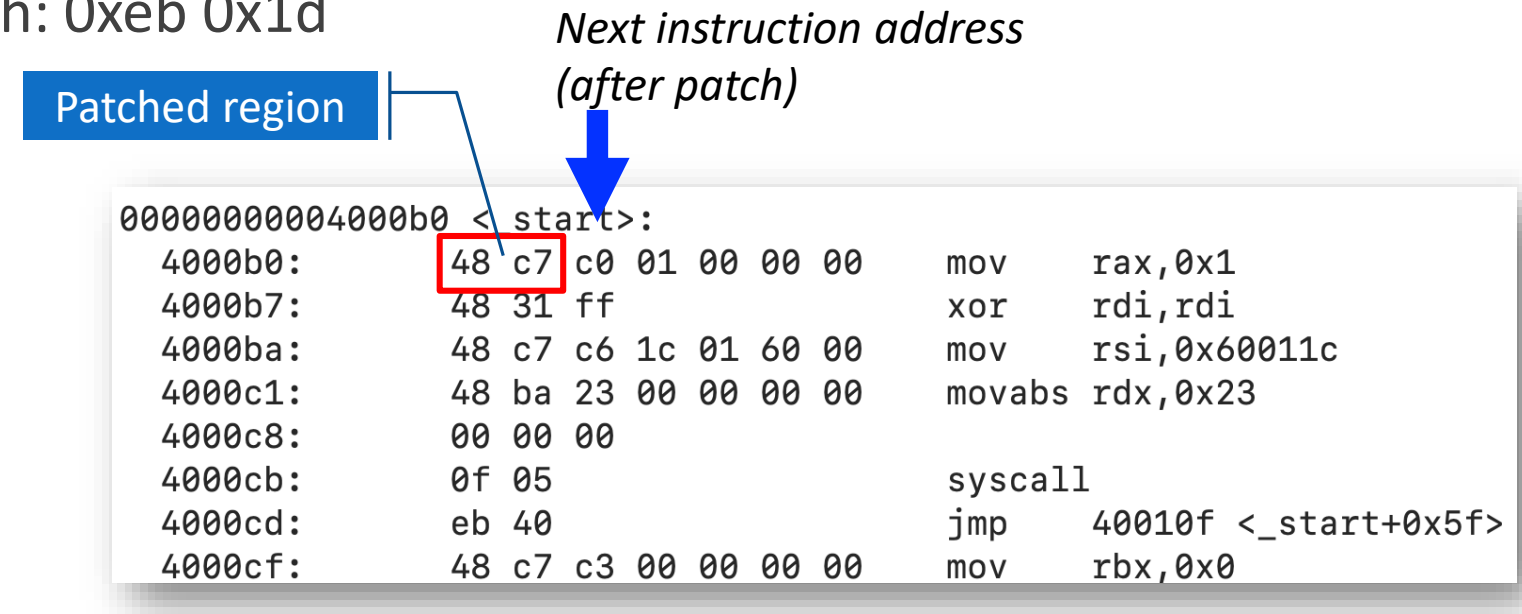
- "JMP 0x4000cf", by using relative jumping
- The corresponding machine code: 0xeb **0x??** (two-byte)
- What is the value for **0x??**

Calculate the Address

Simple arithmetic!

Offset = target_addr – patched_addr – patched_size

- 0x4000cf – 0x4000b0 – 2 = 0x1d
- Patch: 0xeb 0x1d



The ptrace Code

chuang@mjo:~/unix_prog/ptrace — 92×23

```
14 int main(int argc, char *argv[]) {
15     pid_t child;
16     if((child = fork()) < 0) errquit("fork");
17     if(child == 0) {
18         if(ptrace(PTRACE_TRACEME, 0, 0, 0) < 0) errquit("ptrace");
19         execlp("./demo/hidden", "./demo/hidden", NULL);
20         errqui
21     } else {
22         int status;
23         unsigned char code[] = { 0xeb, 0x1d, 0x90, 0x90, 0x90, 0x90, 0x90, 0x90 };
24         unsigned long *lcode = (unsigned long*) code;
25         if(waitpid(child, &status, 0) < 0) errquit("waitpid");
26         assert(WIFSTOPPED(status));
27         ptrace(PTRACE_SETOPTIONS, child, 0, PTRACE_O_EXITKILL);
28         if(ptrace(PTRACE_POKETEXT, child, 0x4000b0L, *lcode) != 0) errquit("poketext");
29         ptrace(PTRACE_CONT, child, 0, 0);
30         waitpid(child, &status, 0);
31         perror("done");
32     }
33     return 0;
34 }
35
                                                        35,0-1        Bot
```

Patched code

Fill NOP's

Patched address

Alternatively, Remove "eb 40"

chuang@mjo:~/unix_prog/ptrace — 92×24

```
14 int main(int argc, char *argv[]) {
15     pid_t child;
16     if((child = fork()) < 0) errquit("fork");
17     if(child == 0) {
18         if(ptrace(PTRACE_TRACEME, 0, 0, 0) < 0) errquit("ptrace");
19         execlp("./demo/hidden", "./demo/hidden", NULL);
20         errquit("execlp");
21     } else {
22         int status;
23         long code;
24         if(waitpid(child, &status, 0) < 0) errquit("waitpid");
25         assert(WIFSTOPPED(status));
26         ptrace(PTRACE_SETOPTIONS, child, 0, PTRACE_O_EXITKILL);
27         code = ptrace(PTRACE_PEEKTEXT, child, 0x4000cd, 0);
28         if(ptrace(PTRACE_POKETEXT, child, 0x4000cdL,
29               (code & 0xffffffffffff0000) | 0x9090) != 0) errquit("poketext");
30         ptrace(PTRACE_CONT, child, 0, 0);
31         waitpid(child, &status, 0);
32         perror("done");
33     }
34     return 0;
35 }
36
                                                    36,0-1        Bot
```

Get the code

Replace with 0x9090

Application: Automated Debugger

The Scenario

With ptrace, now we can …

- Inspecting memory and registers, and change the content
- Patch codes on-the-fly

The "guess" program example

- A "secret" is randomly generated
- The "secret" is then compared against user inputs
- We want to compromise the program and ensure that ***any*** guess from the user will be success

The "guess" Program

chuang@mjo:~/unix_prog/ptrace — 50×16

```
10 int main() {
11         char buf[16];
12         srand(time(0) ^ getpid());
           secret = rand();
           setvbuf(stdin, NULL, _IONBF, 0);
           printf("Show me the key: ");
           fgets(buf, sizeof(buf), stdin);
           if(strtol(buf, NULL, 0) == secret) {
                   printf("Bingo!\n");
                   // do something secret ...
           } else {
                   printf("No no no ...\n");
           }
           return 0;
                                          24,2          90%
```

chuang@mjo:~/unix_prog/ptrace — 73×21

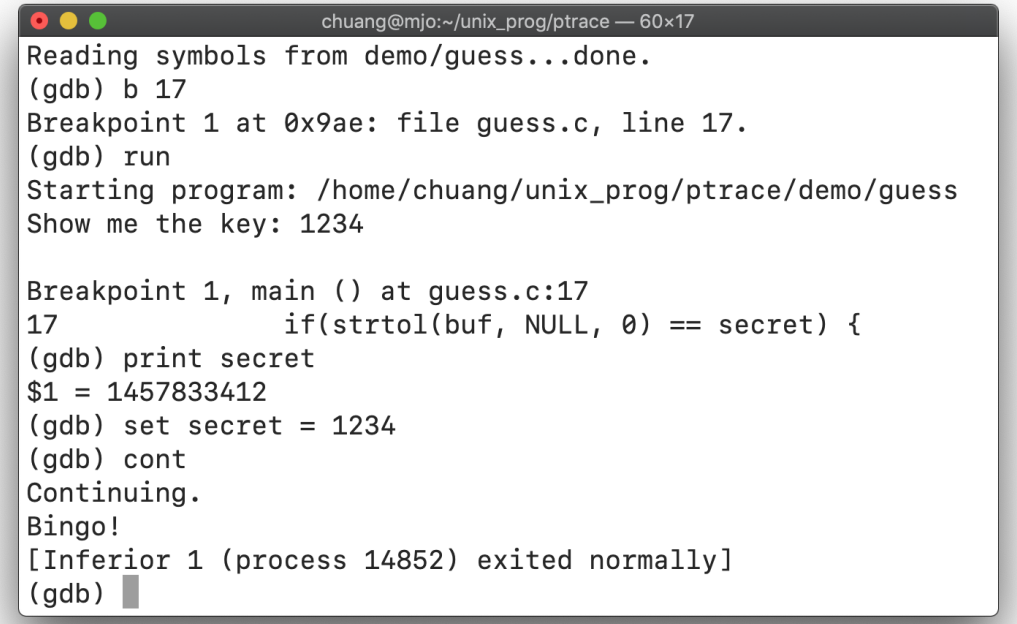
```
165 000000000000092a <main>:
166  92a:   55                      push   rbp
167  92b:   48 89 e5                mov    rbp,rsp
168  92e:   53                      push   rbx
169         ...
170  996:   48 8b 15 73 06 20 00    mov    rdx,QWORD PTR [rip+0x200673]
171  99d:   48 8d 45 d0             lea    rax,[rbp-0x30]
172  9a1:   be 10 00 00 00          mov    esi,0x10
173  9a6:   48 89 c7                mov    rdi,rax
174  9a9:   e8 12 fe ff ff          call   7c0 <fgets@plt>
175  9ae:   48 8d 45 d0             lea    rax,[rbp-0x30]
176  9b2:   ba 00 00 00 00          mov    edx,0x0
177  9b7:   be 00 00 00 00          mov    esi,0x0
178  9bc:   48 89 c7                mov    rdi,rax
179  9bf:   e8 0c fe ff ff          call   7d0 <strtol@plt>
180  9c4:   8b 15 52 06 20 00       mov    edx,DWORD PTR [rip+0x200652]
181  9ca:   89 d2                   mov    edx,edx
182  9cc:   48 39 d0                cmp    rax,rdx
183  9cf:   75 0e                   jne    9df <main+0xb5>
184         ...
                                                169,1         71%
```

The GDB Way (High Level)

Setup a breakpoint right before the check

Change the value of the secret (or the input)

Bingo!

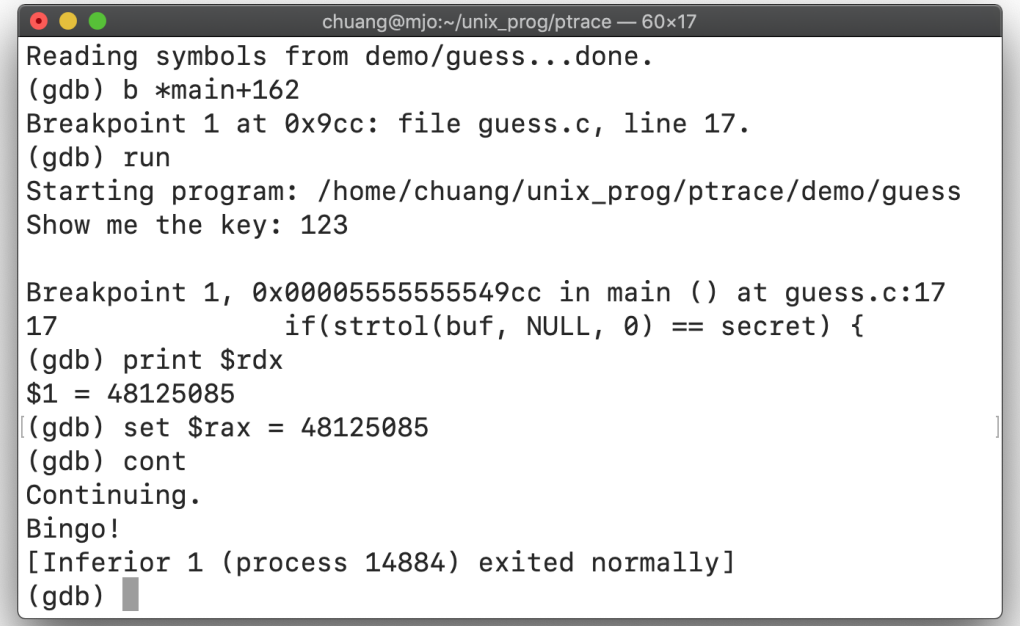


The GDB Way (Low Level)

Setup a breakpoint right before the check

Change the value of the secret (or the input)

Bingo!



How it Works?

Let's do it with ptrace

- Alternatively, you can try to work with "LD_PRELOAD"
- Replace either the **rand** or the **strtol** function
- But … it changes the behavior of ***all*** rand/strtol functions invoked in this program – not good for large programs

With ptrace, we can have a fine grained control of the program flow!

- We don't want single-step tracing – It's too slow!
- Run and stop the program at any point you want

Recall the Steps

Identify the location we want to stop the program

Setup a breakpoint

Run the program

Program stopped

Change the value

Continue the execution

BINGO

With ptrace, The Steps Becomes ...

Run and ***suspend*** the program

Identify the location we want to stop the program

Setup a breakpoint

Continue the execution

Program stopped

Change the value, ***and restore the breakpoint***

Continue the execution

BINGO

Identify the Location

Reverse Engineering for Low Level Control

We have already reverse engineered it!

- Its main + 162 (0xA2), or 0x9cc in the text segment
- But where is it in the ***runtime*** process memory?
- PIE is enabled!

We have to know "How .text is loaded"

```
chuang@mjo:~/unix_prog/ptrace — 52×8
(pwntools) [chuang@mjo ptrace]$ checksec demo/guess
[*] '/home/chuang/unix_prog/ptrace/demo/guess'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
(pwntools) [chuang@mjo ptrace]$
```

```
chuang@mjo:~/unix_prog/ptrace — 73×21
165 000000000000092a <main>:
166  92a:   55                      push   rbp
167  92b:   48 89 e5                mov    rbp,rsp
168  92e:   53                      push   rbx
169           ...
170  996:   48 8b 15 73 06 20 00    mov    rdx,QWORD PTR [rip+0x200673]
171  99d:   48 8d 45 d0             lea    rax,[rbp-0x30]
172  9a1:   be 10 00 00 00          mov    esi,0x10
173  9a6:   48 89 c7                mov    rdi,rax
174  9a9:   e8 12 fe ff ff          call   7c0 <fgets@plt>
175  9ae:   48 8d 45 d0             lea    rax,[rbp-0x30]
176  9b2:   ba 00 00 00 00          mov    edx,0x0
177  9b7:   be 00 00 00 00          mov    esi,0x0
178  9bc:   48 89 c7                mov    rdi,rax
179  9bf:   e8 0c fe ff ff          call   7d0 <strtol@plt>
180  9c4:   8b 15 52 06 20 00       mov    edx,DWORD PTR [rip+0x200652]
181  9ca:   89 d2                   mov    edx,edx
182  9cc:   48 39 d0                cmp    rax,rdx
183  9cf:   75 0e                   jne    9df <main+0xb5>
184           ...
                                                    169,1         71%
```

We need to know the ELF format

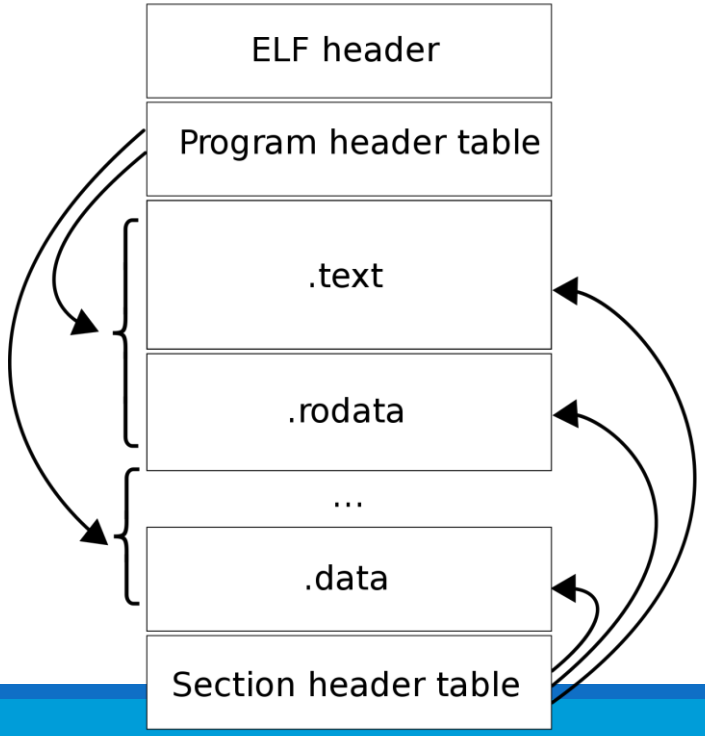
ELF Format

ELF: Executable and Linkable Format

Commonly used in UNIX-like operating systems

An ELF file contains

- One ELF header
- Program header (zero or more)
- Section header (zero or more)
- Data, referred to by program and section headers



ELF (File) Header – Overview

64-Bit (64 bytes, 0x40)

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
7F	'E'	'L'	'F'	Cla	End	Ver	ABI	Av	Unused						
Type		Machine		Version				Entry Point (8-byte)							
Program Header Offset (0x40)								Section Header Offset							
Flags				Size (64)		PH Size		# of PH		SH Size		# of SH		SH Idx	

32-bit (52 bytes, 0x34)

0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
7F	'E'	'L'	'F'	Cla	End	Ver	ABI	Av	Unused						
Type		Machine		Version				Entry Point (4-byte)				PH Offset (0x34)			
SH Offset				Flags		Size (52)		PH Size		# of PH		SH Size		# of SH	
SH Idx															

Cla (Class):
1 – 32-bit
2 – 64-bit

End (Endianness):
1 – Little
2 – Big

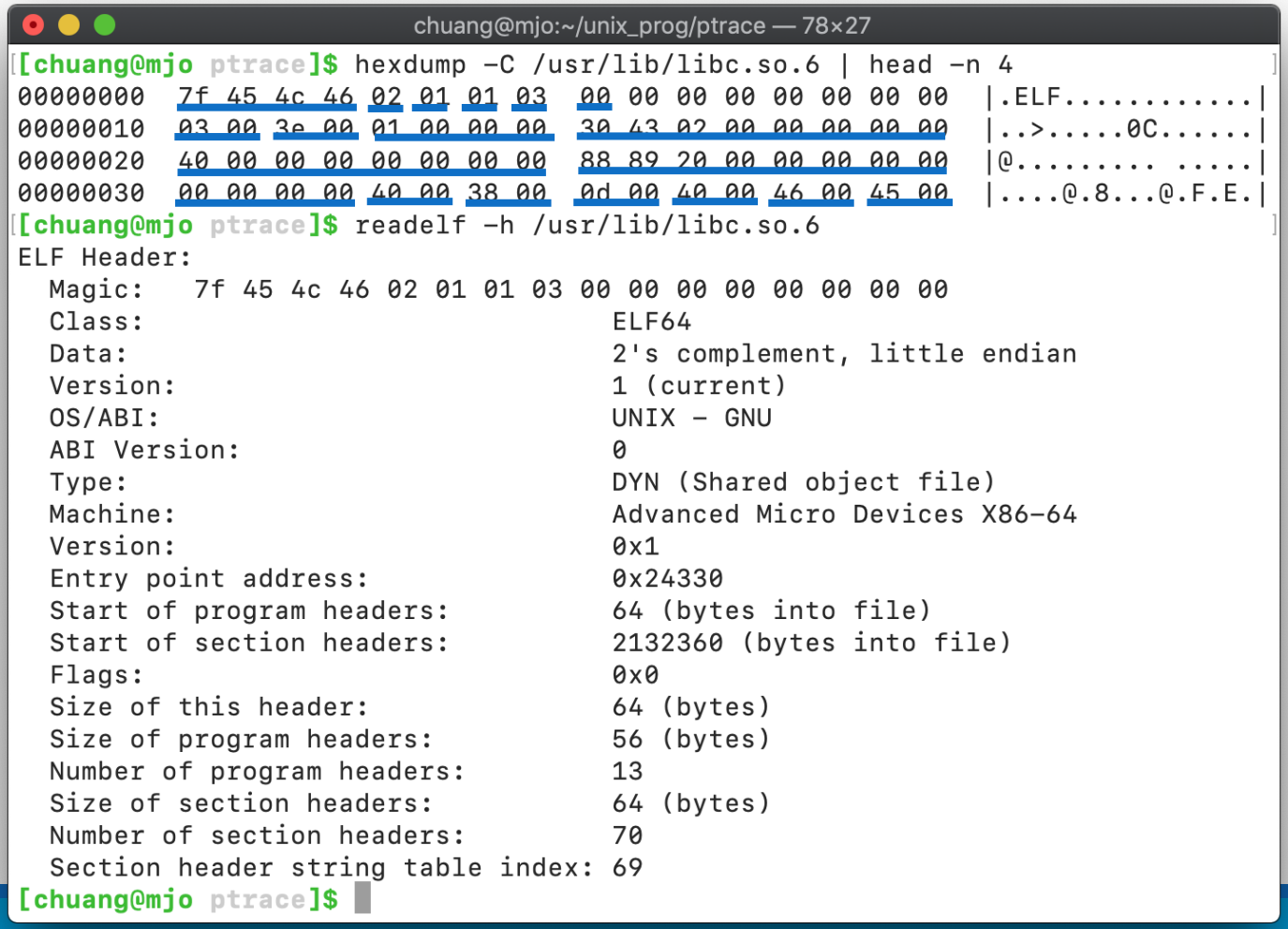
ABI: 03 – Linux

Av (ABI version)

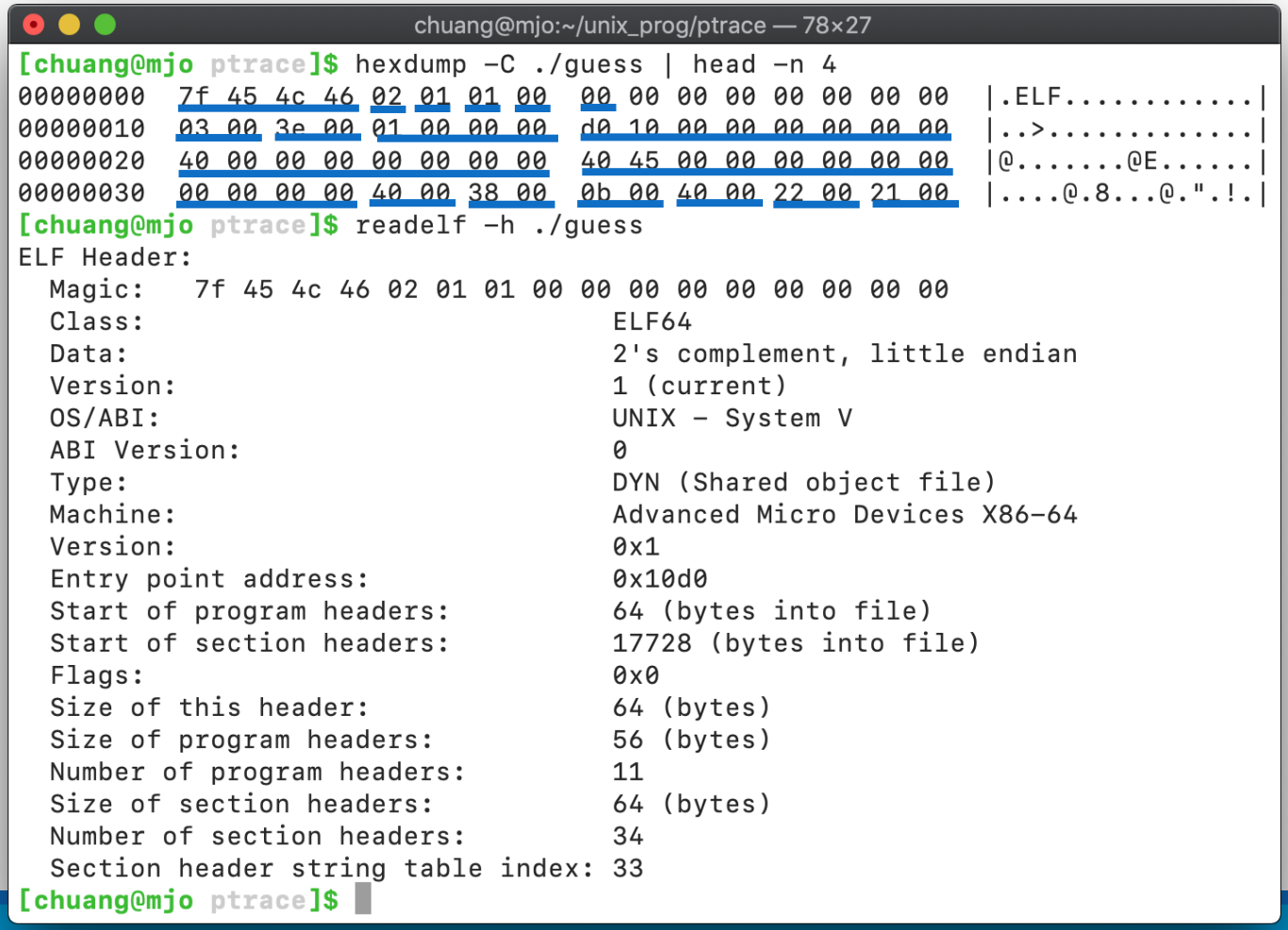
Type:
02 – Executable
03 – Dynamic

Machine:
03 – x86

ELF (File) Header – Shared Object



ELF (File) Header – Executable



Load of The Text Segment

The text segment is stored in the executable

- The **ELF** executable file format
- The **PT_LOAD** program header (**phdr**) – Describe what parts in the ELF file will be loaded
- The segment header (**shdr**) – Describe what segments are available in the ELF file, and their offsets in the file, e.g., **.text**, **.symtab**, **.strtab**, …

Linux: The text segment is mmap'ed into the virtual memory space of the process

- It belongs to one of the PT_LOAD program headers

ELF: PHDR and SHDR (hello) – Simple Case w/o PIE

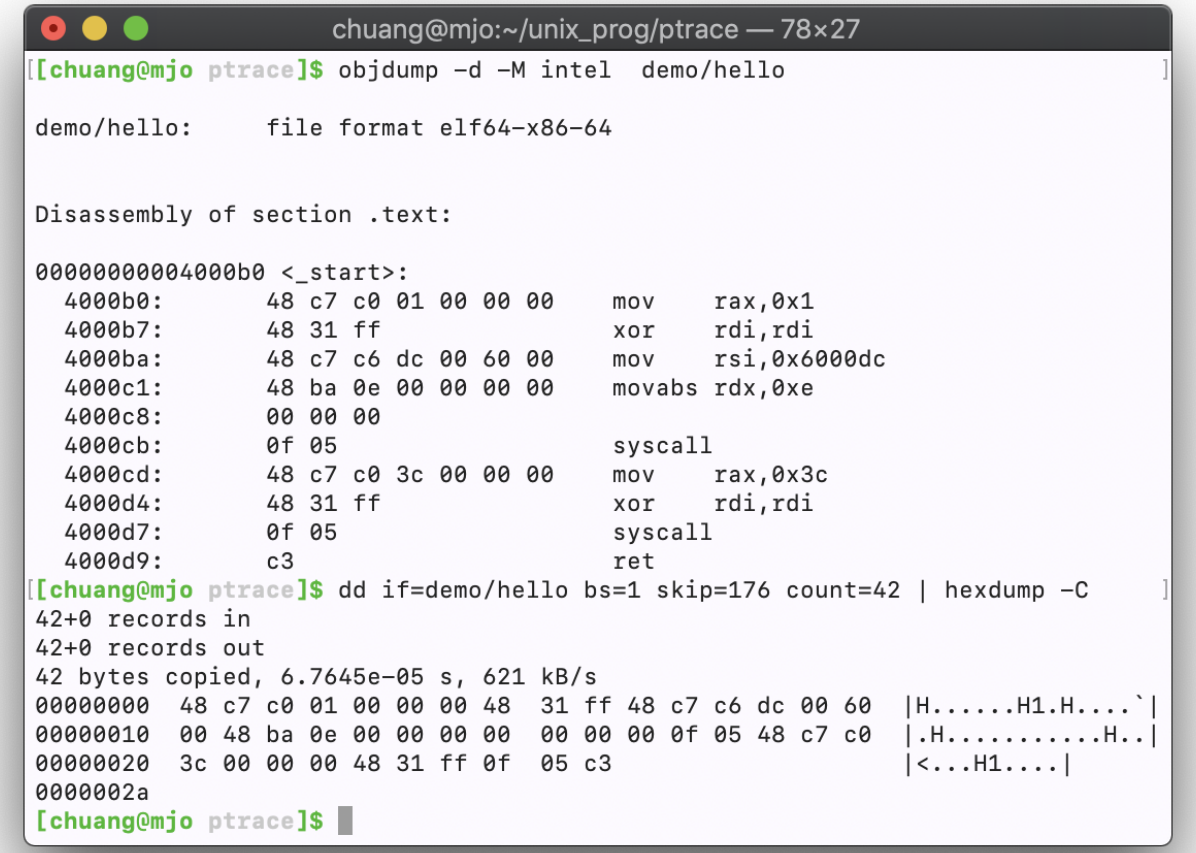
We can get the codes directly from an executable, as well as the process memory space

- Text segment: 0x00b0 (176), size = 0x2a (42)
- Loaded memory region: 0x0000 ~ 0x00da @ 0x400000

```
chuang@mjo:~/unix_prog/ptrace — 75×19
[chuang@mjo ptrace]$ readelf -l demo/hello

Elf file type is EXEC (Executable file)
Entry point 0x4000b0
There are 2 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x00000000000000da 0x00000000000000da  R E    0x200000
  LOAD           0x00000000000000dc 0x00000000006000dc 0x00000000006000dc
                 0x000000000000000e 0x000000000000000e  RW     0x200000

 Section to Segment mapping:
  Segment Sections...
   00     .text
   01     .data
[chuang@mjo ptrace]$
```

```
chuang@mjo:~/unix_prog/ptrace — 75×24
[chuang@mjo ptrace]$ readelf -S demo/hello
There are 6 section headers, starting at offset 0x218:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .text             PROGBITS         00000000004000b0  000000b0
       000000000000002a  0000000000000000  AX       0     0     16
  [ 2] .data             PROGBITS         00000000006000dc  000000dc
       000000000000000e  0000000000000000  WA       0     0     4
  [ 3] .symtab           SYMTAB           0000000000000000  000000f0
       00000000000000d8  0000000000000018           4     5     8
  [ 4] .strtab           STRTAB           0000000000000000  000001c8
       0000000000000023  0000000000000000           0     0     1
  [ 5] .shstrtab         STRTAB           0000000000000000  000001eb
       0000000000000027  0000000000000000           0     0     1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  l (large), p (processor specific)
[chuang@mjo ptrace]$
```

Hello: Codes from ELF



Hello: Codes from Process Memory

```
chuang@mjo:~/unix_prog/ptrace — 104×25
Reading symbols from demo/hello...(no debugging symbols found)...done.
(gdb) starti
Starting program: /home/chuang/unix_prog/ptrace/demo/hello

Program stopped.
0x00000000004000b0 in _start ()
(gdb) info proc mapping
process 1747
Mapped address spaces:

          Start Addr           End Addr       Size     Offset objfile
            0x400000           0x401000     0x1000        0x0 /home/chuang/unix_prog/ptrace/demo/hello
            0x600000           0x601000     0x1000        0x0 /home/chuang/unix_prog/ptrace/demo/hello
      0x7ffff7ffa000     0x7ffff7ffd000     0x3000        0x0 [vvar]
      0x7ffff7ffd000     0x7ffff7fff000     0x2000        0x0 [vdso]
      0x7ffffffde000     0x7fffffffe000    0x21000        0x0 [stack]
  0xffffffffff600000 0xffffffffff601000     0x1000        0x0 [vsyscall]
(gdb) x/42bx 0x4000b0
0x4000b0 <_start>:      0x48    0xc7    0xc0    0x01    0x00    0x00    0x00    0x48
0x4000b8 <_start+8>:    0x31    0xff    0x48    0xc7    0xc6    0xdc    0x00    0x60
0x4000c0 <_start+16>:   0x00    0x48    0xba    0x0e    0x00    0x00    0x00    0x00
0x4000c8 <_start+24>:   0x00    0x00    0x00    0x0f    0x05    0x48    0xc7    0xc0
0x4000d0 <_start+32>:   0x3c    0x00    0x00    0x00    0x48    0x31    0xff    0x0f
0x4000d8 <_start+40>:   0x05    0xc3
(gdb)
```

ELF: PHDR and SHDR (guess) – PIE Enabled

chuang@mjo:~/unix_prog/ptrace — 75×43

```
[chuang@mjo ptrace]$ readelf -l demo/guess

Elf file type is DYN (Shared object file)
Entry point 0x820
There are 9 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000000040 0x0000000000000040
                 0x00000000000001f8 0x00000000000001f8  R      0x8
  INTERP         0x0000000000000238 0x0000000000000238 0x0000000000000238
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000c08 0x0000000000000c08  R E    0x200000
  LOAD           0x0000000000000d70 0x0000000000200d70 0x0000000000200d70
                 0x00000000000002a0 0x00000000000002b0  RW     0x200000
  DYNAMIC        0x0000000000000d80 0x0000000000200d80 0x0000000000200d80
                 0x00000000000001f0 0x00000000000001f0  RW     0x8
  NOTE           0x0000000000000254 0x0000000000000254 0x0000000000000254
                 0x0000000000000044 0x0000000000000044  R      0x4
  GNU_EH_FRAME   0x0000000000000abc 0x0000000000000abc 0x0000000000000abc
                 0x000000000000003c 0x000000000000003c  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x0000000000000d70 0x0000000000200d70 0x0000000000200d70
                 0x0000000000000290 0x0000000000000290  R      0x1

 Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynst
r .gnu.version .gnu.version_r .rela.dyn .rela.plt .init .plt .plt.got .text
 .fini .rodata .eh_frame_hdr .eh_frame
   03     .init_array .fini_array .dynamic .got .data .bss
   04     .dynamic
   05     .note.ABI-tag .note.gnu.build-id
   06     .eh_frame_hdr
   07
   08     .init_array .fini_array .dynamic .got
[chuang@mjo ptrace]$
```

chuang@mjo:~/unix_prog/ptrace — 75×43

```
[chuang@mjo ptrace]$ readelf -S demo/guess
There are 34 section headers, starting at offset 0x24c0:

Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .interp           PROGBITS         0000000000000238  00000238
       000000000000001c  0000000000000000   A       0     0     1
  [ 2] .note.ABI-tag     NOTE             0000000000000254  00000254
       0000000000000020  0000000000000000   A       0     0     4
  [ 3] .note.gnu.build-i NOTE             0000000000000274  00000274
       0000000000000024  0000000000000000   A       0     0     4
  [ 4] .gnu.hash         GNU_HASH         0000000000000298  00000298
       0000000000000024  0000000000000000   A       5     0     8
  [ 5] .dynsym           DYNSYM           00000000000002c0  000002c0
       0000000000000198  0000000000000018   A       6     1     8
  [ 6] .dynstr           STRTAB           0000000000000458  00000458
       00000000000000d1  0000000000000000   A       0     0     1
  [ 7] .gnu.version      VERSYM           000000000000052a  0000052a
       0000000000000022  0000000000000002   A       5     0     2
  [ 8] .gnu.version_r    VERNEED          0000000000000550  00000550
       0000000000000030  0000000000000000   A       6     1     8
  [ 9] .rela.dyn         RELA             0000000000000580  00000580
       00000000000000d8  0000000000000018   A       5     0     8
  [10] .rela.plt         RELA             0000000000000658  00000658
       00000000000000f0  0000000000000018  AI       5    22     8
  [11] .init             PROGBITS         0000000000000748  00000748
       0000000000000017  0000000000000000  AX       0     0     4
  [12] .plt              PROGBITS         0000000000000760  00000760
       00000000000000b0  0000000000000010  AX       0     0     16
  [13] .plt.got          PROGBITS         0000000000000810  00000810
       0000000000000008  0000000000000008  AX       0     0     8
  [14] .text             PROGBITS         0000000000000820  00000820
       0000000000000262  0000000000000000  AX       0     0     16
  [15] .fini             PROGBITS         0000000000000a84  00000a84
       0000000000000009  0000000000000000  AX       0     0     4
  [16] .rodata           PROGBITS         0000000000000a90  00000a90
       000000000000002a  0000000000000000   A       0     0     4
  [17] .eh_frame_hdr     PROGBITS         0000000000000abc  00000abc
       000000000000003c  0000000000000000   A       0     0     4
  [18] .eh_frame         PROGBITS         0000000000000af8  00000af8
```

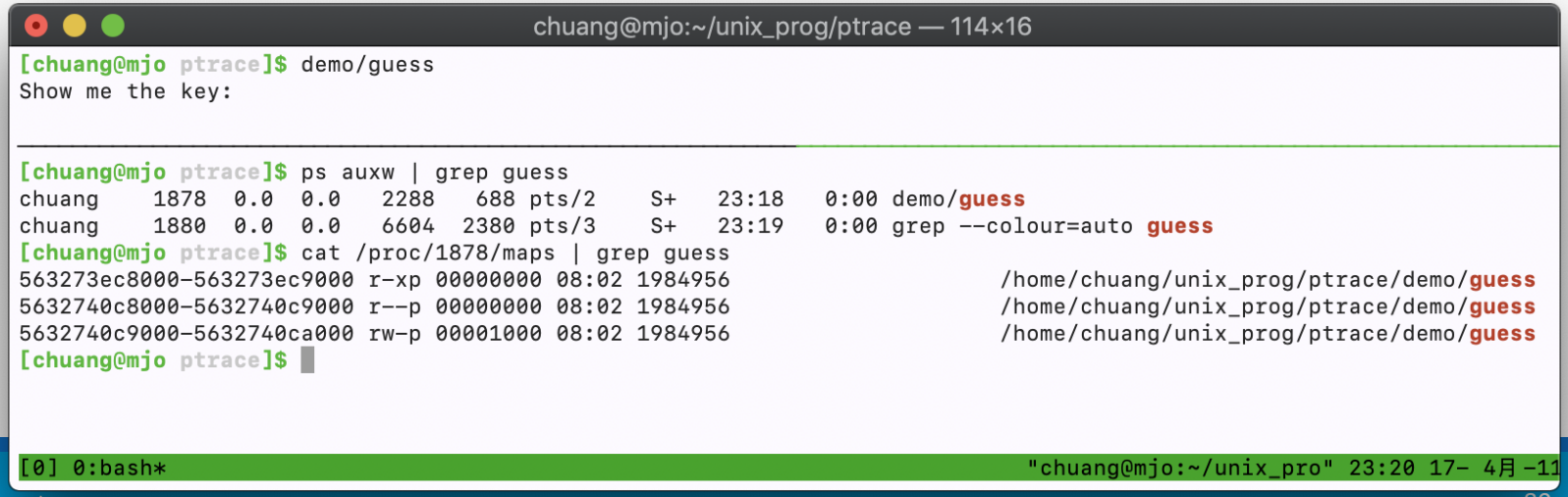
guess: Target Address

From SHDR

- Text segment offset: 0x820, size = 0x262 bytes

From PHDR

- Permission: (R)ead and (E)xecute
- Loaded file offset: 0x0000, size = 0xc08 bytes



Handle Breakpoints

The INT instruction

INT is an x86/x86_64 instruction to generate a software interrupt

- It takes a single byte integer parameter (0-255)
- INT **0x??**
- The corresponding machine code: 0xcd **0x??**

System component and software can register functions to handle software interrupts

Examples

- int 0x13 – For invoking x86 BIOS disk IO functions (real-mode)
- int 0x10 – For invoking x86 BIOS video functions (read-mode)
- int 0x21 – For invoking x86 DOS APIs (real-mode)
- int 0x80 – For invoking x86 Linux system call functions

The Magic "0xCC" Code

For Intel x86/x86_64 platform

A special instruction for implementing breakpoints in debuggers

Similar to INT 3, but use only a single byte

- Good for code patching! ☺
- Fit to any size of instructions!

Trigger INT3, and then send a SIGTRAP to the tracee

- A ptraced child process will be stopped ***right after*** the 0xcc instruction
- A non-ptraced child process will be terminated (core dumped)

To Setup a Breakpoint

Get the code @ target address (PEEKTEXT)

Replace the first byte with 0xCC, and then write it back (POKETEXT)

When 0xCC is executed by the CPU

- The tracee is stopped (SIGTRAP)
- The tracer is notified (SIGCHLD w/ term signal = SIGTRAP)

Once We Have Hit a Breakpoint …

Determine which breakpoint is hit

- RIP can be read by PEEKUSER or GETREGS
- The address for **0xcc** instruction should be (**RIP – 1**)

We can do anything we want to monitor or change the program states

Once you have done …

- Restore the code – 0xcc must be replaced with the original byte
- Restore the RIP – We have to launch the real instruction
- Continue program execution

If You Plan to Reuse the Breakpoint …

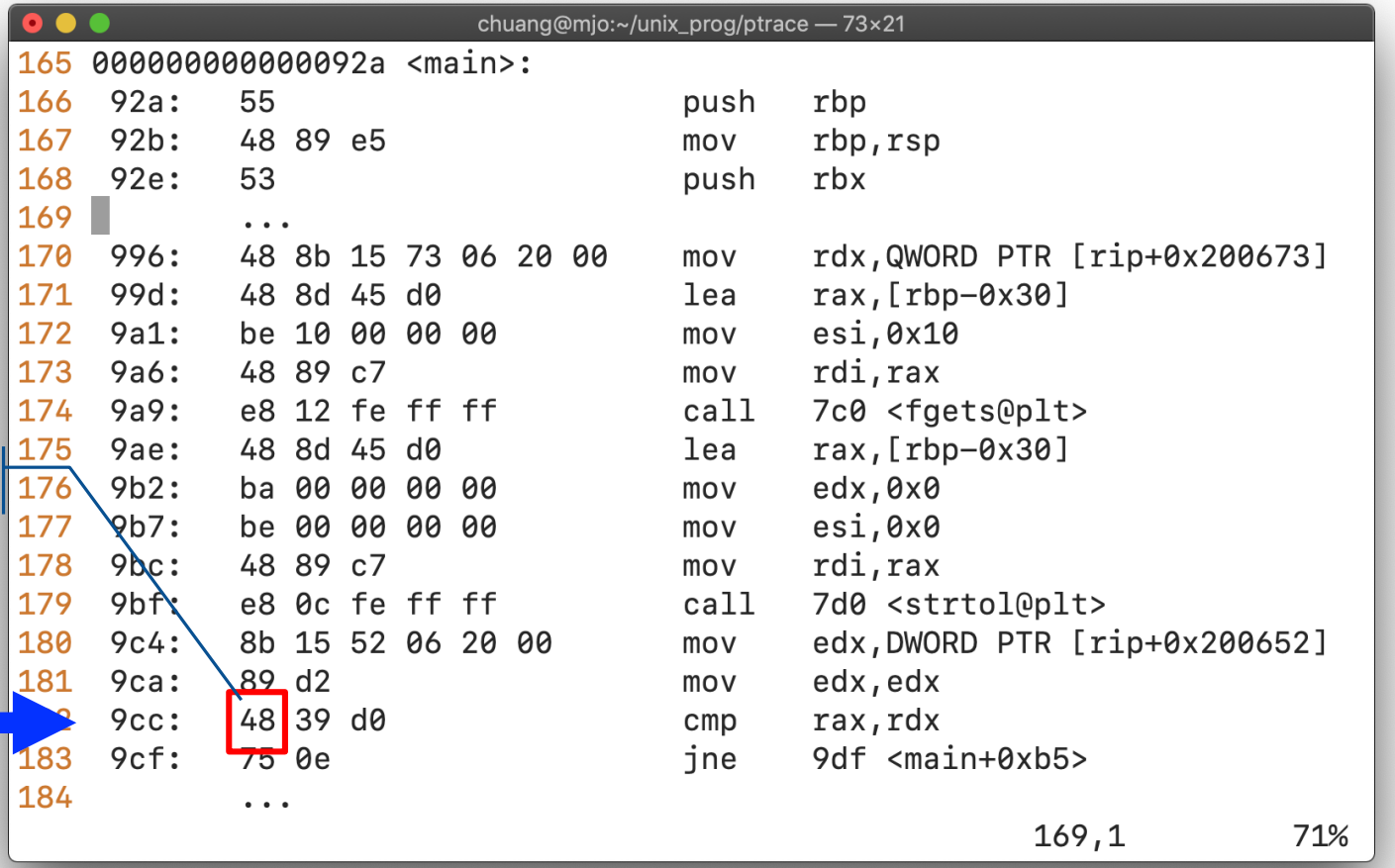
The breakpoint is eliminated when you restored the code

Next time you hit the same address, it does not stop

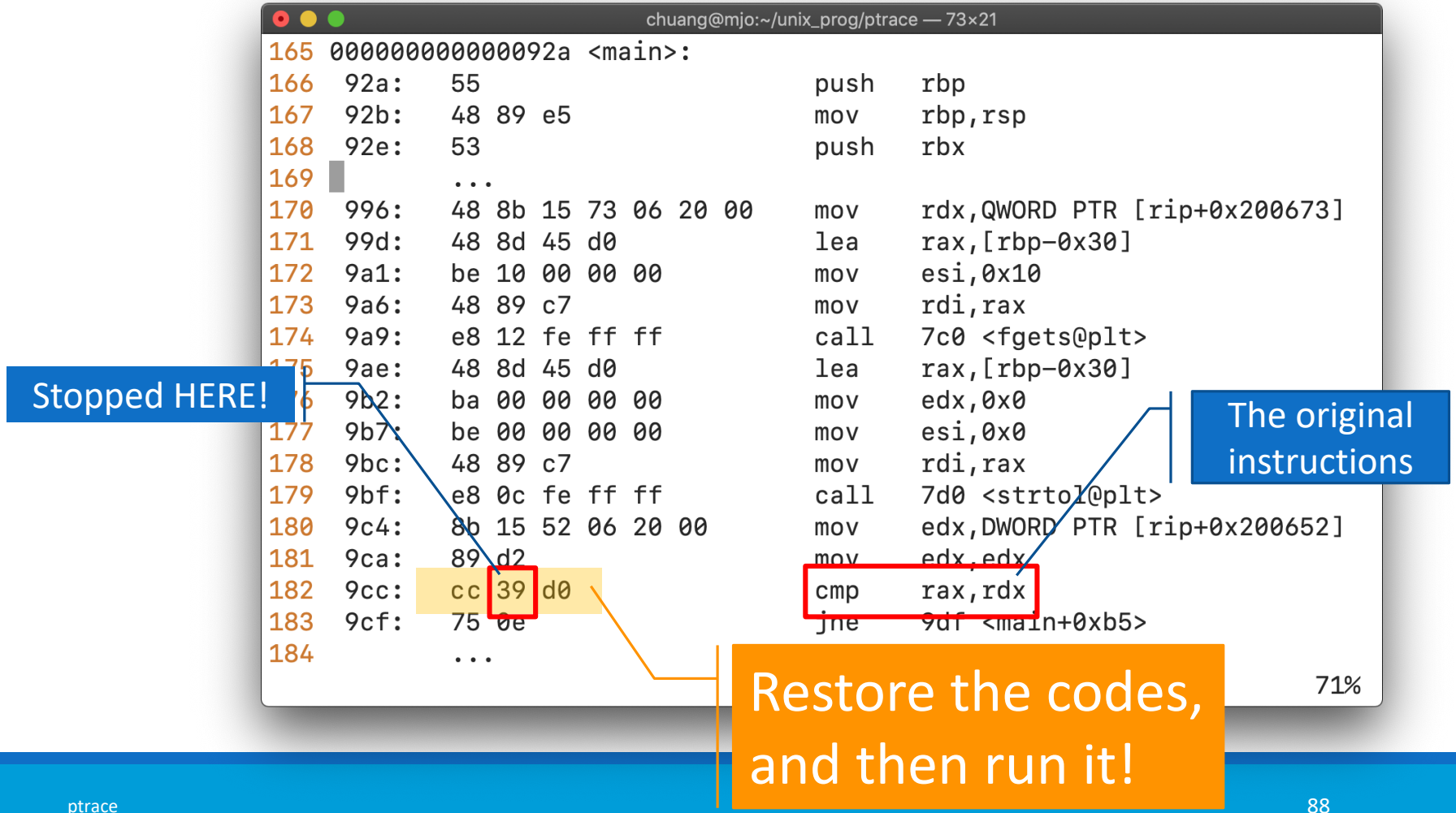
Solution

- You can use SINGLESTEP to run the restored instruction
- Afterward, you can put **0xcc** back to the address, to ensure that next time the tracee will be trapped

The "guess" Program – Before Patched



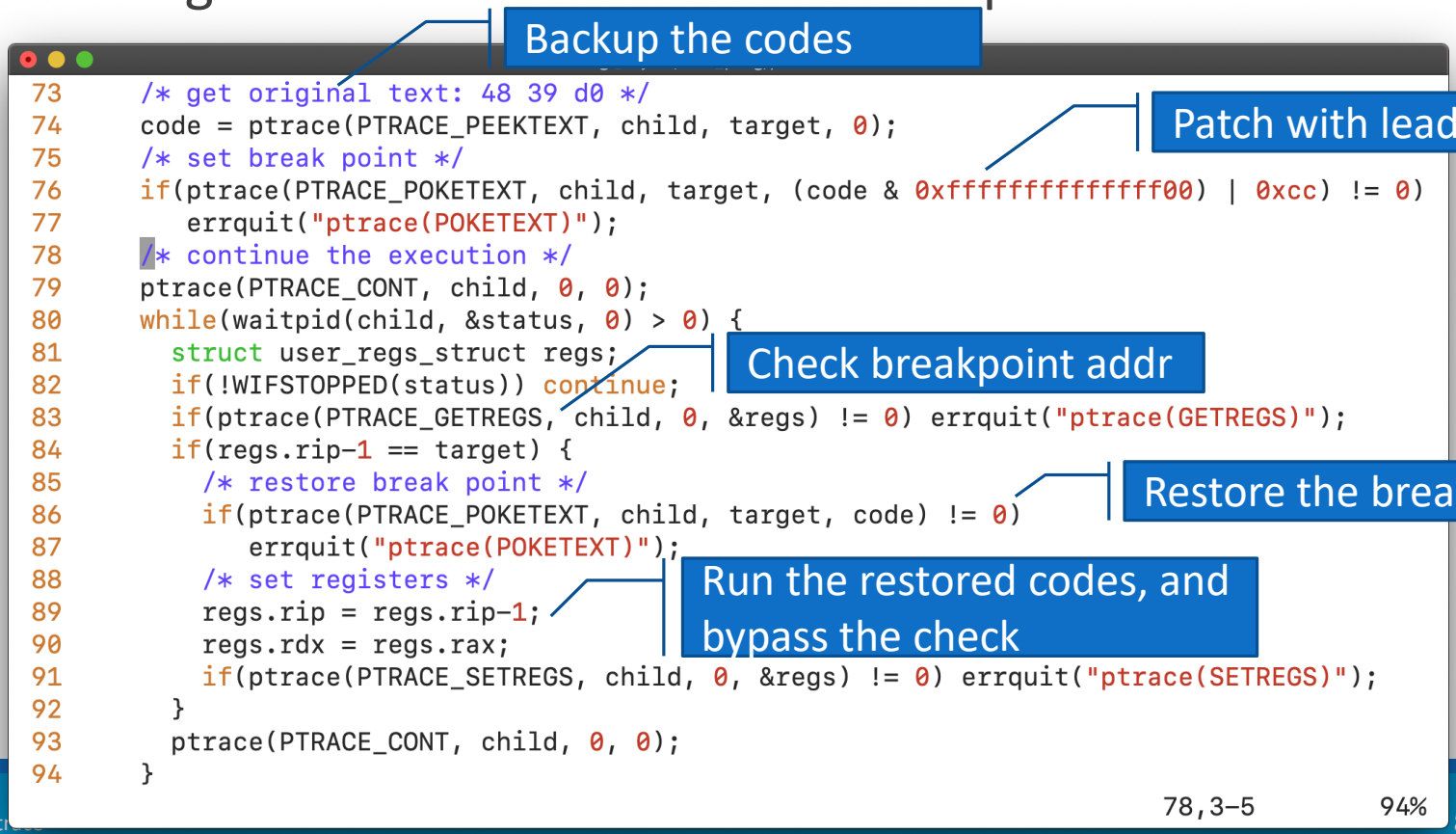
The "guess" Program – After Stopped



The ptrace Codes

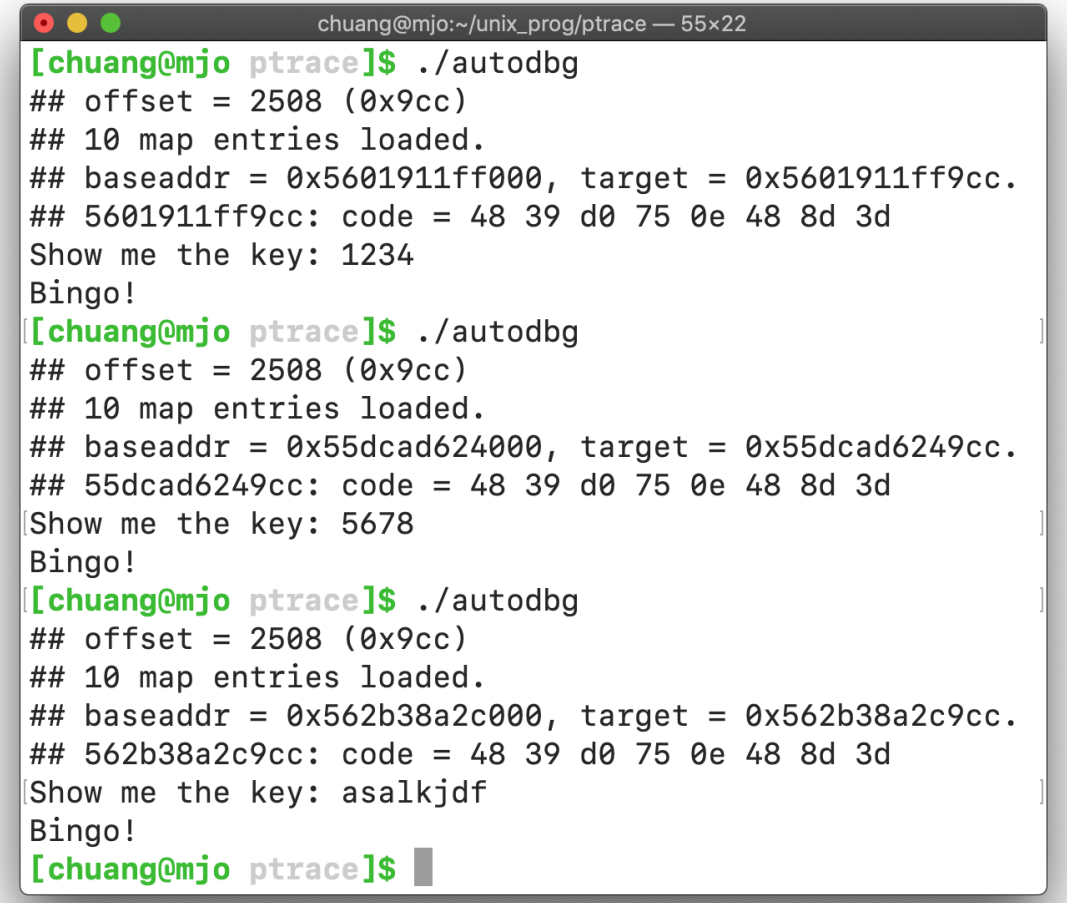
Suppose we have obtained the **target** address

- It's segment base address + the breakpoint offset




```
73    /* get original text: 48 39 d0 */
74    code = ptrace(PTRACE_PEEKTEXT, child, target, 0);
75    /* set break point */
76    if(ptrace(PTRACE_POKETEXT, child, target, (code & 0xffffffffffffff00) | 0xcc) != 0)
77       errquit("ptrace(POKETEXT)");
78    /* continue the execution */
79    ptrace(PTRACE_CONT, child, 0, 0);
80    while(waitpid(child, &status, 0) > 0) {
81      struct user_regs_struct regs;
82      if(!WIFSTOPPED(status)) continue;
83      if(ptrace(PTRACE_GETREGS, child, 0, &regs) != 0) errquit("ptrace(GETREGS)");
84      if(regs.rip-1 == target) {
85        /* restore break point */
86        if(ptrace(PTRACE_POKETEXT, child, target, code) != 0)
87           errquit("ptrace(POKETEXT)");
88        /* set registers */
89        regs.rip = regs.rip-1;
90        regs.rdx = regs.rax;
91        if(ptrace(PTRACE_SETREGS, child, 0, &regs) != 0) errquit("ptrace(SETREGS)");
92      }
93      ptrace(PTRACE_CONT, child, 0, 0);
94    }
                                                    78,3-5        94%
```

The ptrace Codes (Run)



Summary

Summary

ptrace is a powerful tool

It can be used to build many applications

- Instruction counter
- Execution trace dump
- System call tracer
- Change program control flow
- Automated Debugger

See ptrace(2) for more details

Q & A