

□ 1. Last time

□ 2. How debuggers work

□ Intro

□ Attaching to, and controlling, a process

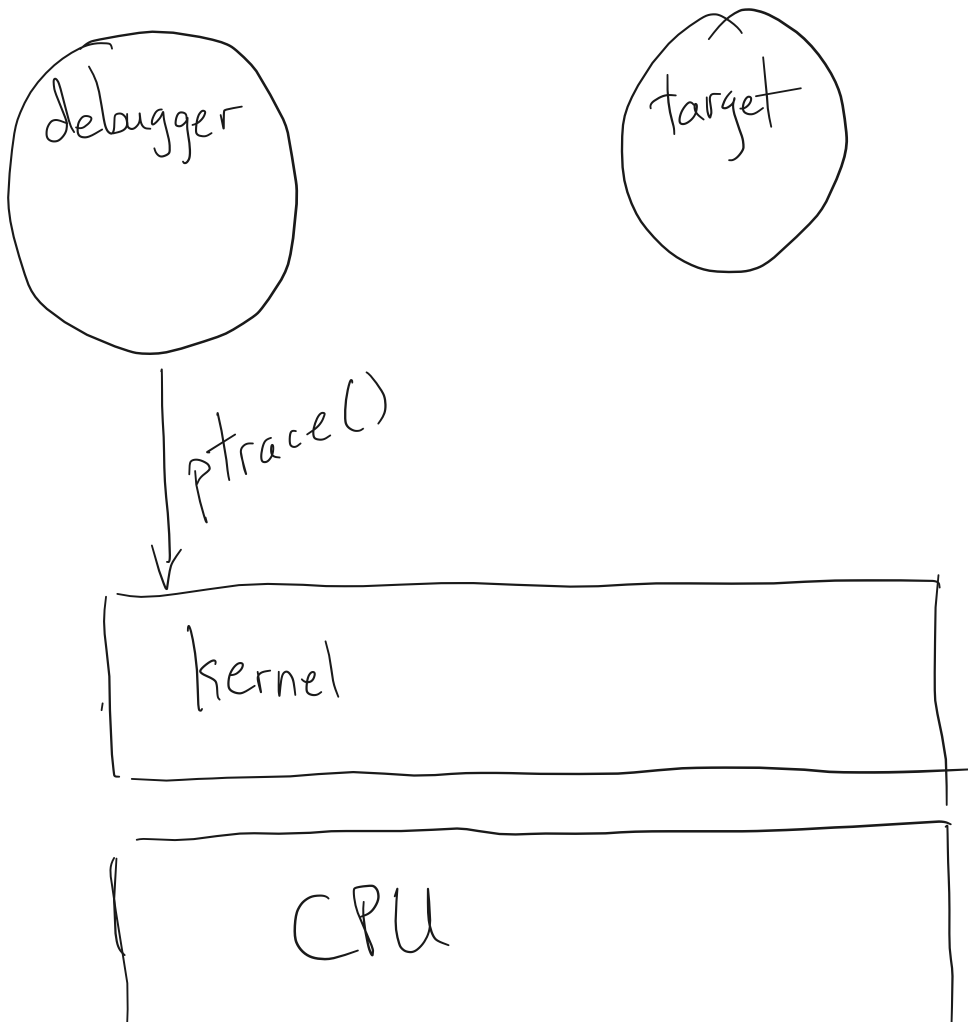
□ Reading and writing a program's memory and registers

□ Resolving addresses to program names

□ Single-stepping

□ Breakpoints

□ Watchpoints



CS202 Handout 14

1. Attaching to a Debugger

1.1 Launch a process that is attached

```
1 void launch_attached(const char* path,
2     char* const argv[]) {
3     int pid = fork();
4     if (pid == 0) {
5         ptrace(PTRACE_TRACEME, 0, NULL, NULL);
6         execv(path, argv);
7     }
8     return pid;
9 }
```

1.2 Attach to a running process

```
1 void attach_to_process(pid_t pid) {
2     ptrace(PTRACE_ATTACH, pid, NULL, NULL);
3 }
```

1.3 Wait for a target after issuing a ptrace() to it

```
1 void wait_for_target(pid_t pid) {
2     int status;
3     while (1) {
4         waitpid(pid, &status, 0);
5         if (WIFSTOPPED(status)) {
6             // The reason for the change was that pid stopped.
7
8             // We should have stopped because of
9             // either SIGTRAP or SIGSTOP.
10            assert(WSTOPSIG(status) == SIGTRAP
11                || WSTOPSIG(status) == SIGSTOP);
12
13            break;
14
15        } else if (WIFEXITED(status)) {
16            // The process exited before we could attach.
17            printf("Process exited\n");
18            break;
19        }
20    }
21
22    if (WIFSTOPPED(status)) {
23        // In here, the debugger manipulates the target.
24        // As a simple example, if the debugger wants to
25        // "continue" the target, it executes the following:
26        ptrace(PTRACE_CONT, pid, NULL, NULL);
27
28    }
29 }
```

2. Interrupting the running target

```
1 void interrupt_target(pid_t pid) {
2     // kill() is a system call that sends OS signals
3     kill(pid, SIGSTOP);
4     // Must use waitpid in order to
5     // wait for the signal to be
6     // delivered.
7 }
```

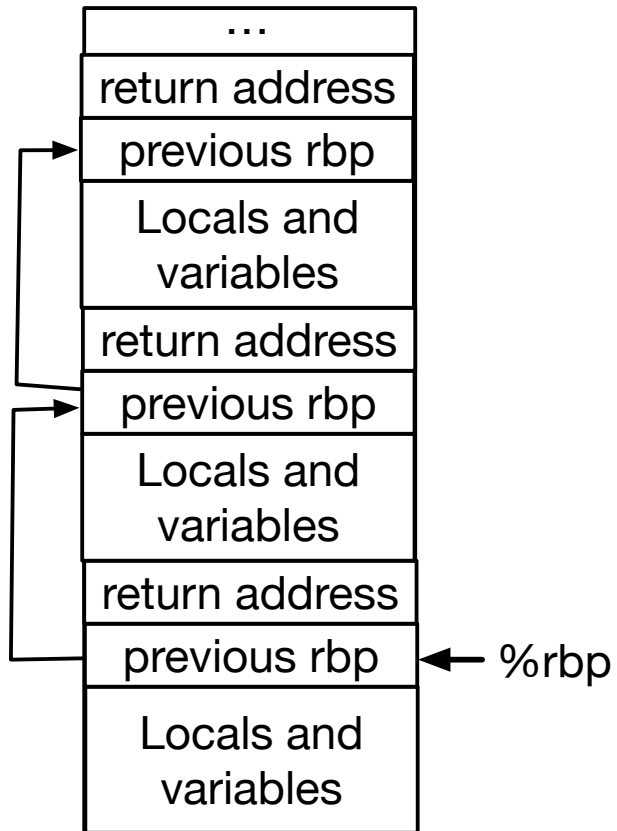
3. Other ptrace commands

All of these only make sense when the target process is stopped, for instance due to the use of `interrupt_target` from above.

```
1 // Execute a single instruction in the process.
2 ptrace(PTRACE_SINGLESTEP, pid, NULL, NULL);
3
4 // Get non-floating point registers.
5 // This includes rsp, rip, rbp, etc.
6 struct user_regs_struct regs;
7 ptrace(PTRACE_GETREGS, pid, NULL, &regs);
8
9 // Get floating point registers.
10 struct user_fpregs_struct fpregs;
11 ptrace(PTRACE_GETFPREGS, pid, NULL, &fpregs);
12
13 // Set registers. This can be used to update
14 // register values.
15 ptrace(PTRACE_SETREGS, pid, NULL, &regs);
16
```

```
17 // Note: PTRACE_PEEKUSER and PTRACE_POKEUSER
18 // provide a more efficient way to read or
19 // write a single register.
20
21 // Read a word (8 bytes) from address `addr`
22 // in target process memory. Note, despite being
23 // called PTRACE_PEEKDATA, on Linux this can
24 // read any part of memory, including the
25 // text segment.
26 uint64_t val;
27 val = ptrace(PTRACE_PEEKDATA, pid, addr, NULL);
28
29 // Write a word (8 byte) to address `addr` in
30 // target procss memory.
31 ptrace(PTRACE_POKEDATA, pid, addr, val);
32
33 // Get information on the signal that caused
34 // the target procss to stop.
35 siginfo_t sinfo;
36 ptrace(PTRACE_GETSIGINFO, pid, &sinfo, NULL);
```

4. Stack Frames and Unwinding



Apr 16, 25 2:52

debugger.c

Page 1/3

```

1 // debugger.c: Simple "debugging" program intended to show
2 // how one process (this one) can manipulate another
3 // one. This program is hard-coded to work with a
4 // target process called "target". It is also hard-coded
5 // to set a breakpoint before the target's second printf,
6 // and at that point, to modify the variable 'x' in the target
7 // to be equal to 202.
8
9 #include <unistd.h>
10 #include <sys/types.h>
11 #include <sys/ptrace.h>
12 #include <sys/wait.h>
13 #include <sys/user.h>
14 #include <stdio.h>
15 #include <assert.h>
16 #include <stdint.h>
17
18 void wait_for_target(int pid);
19 void continue_on(int pid);
20 void single_step(int pid);
21 uint64_t set_breakpoint(int pid, uint64_t addr);
22 void preserve_brkpoint_and_continue(int pid, uint64_t addr, uint64_t orig_inst);
23 void rewind_rip(int pid);
24 void set_x_in_target(int pid, uint32_t val);
25 void show_target_breakpoint(int pid, uint64_t addr);
26
27 // We inspect the target binary to learn at what address the value of 'x' (in
28 // the target) is read from the stack prior to the second printf.
29 // That is the address below. That's a hack. A real debugger would infer
30 // the address by examining debugging information (symbol tables and so on).
31 #define STACK_LOAD 0x401887
32
33 int main(int argc, char** argv)
34 {
35     int pid = fork();
36     if (pid == 0) {
37         ptrace(PTRACE_TRACEME, 0, NULL, NULL);
38         if (execl("target", "target", NULL) < 0)
39             perror("exec()");
40     }
41
42     if (pid < 0)
43         perror("failed to fork\n");
44
45     wait_for_target(pid);
46
47     // When setting a breakpoint, we have to keep around the
48     // original contents of the target's code at the memory location.
49     uint64_t original = set_breakpoint(pid, STACK_LOAD);
50     show_target_breakpoint(pid, STACK_LOAD);
51     continue_on(pid);
52
53     set_x_in_target(pid, 202);
54
55     preserve_brkpoint_and_continue(pid, STACK_LOAD, original);
56
57     return 0;
58 }
59
60 void wait_for_target(int pid)
61 {
62     int status;
63
64     while (1) {
65         waitpid(pid, &status, 0);
66
67         if (WIFSTOPPED(status)) {
68             if (WSTOPSIG(status) != SIGTRAP &&
69                 WSTOPSIG(status) != SIGSTOP) {
70                 printf("target stopped due to signal: %d\n", WSTOPSIG(status));
71             }
72         }
73     }

```

Apr 16, 25 2:52

debugger.c

Page 2/3

```

74         break;
75     } else if (WIFEXITED(status)) {
76         printf("target exited\n");
77         break;
78     }
79 }
80
81 void continue_on(int pid)
82 {
83     if (ptrace(PTRACE_CONT, pid, NULL, NULL) < 0)
84         perror("ptrace continue");
85
86     wait_for_target(pid);
87 }
88
89 void single_step(int pid)
90 {
91     if (ptrace(PTRACE_SINGLESTEP, pid, NULL, NULL) < 0)
92         perror("ptrace singlestep");
93
94     wait_for_target(pid);
95 }
96
97 uint64_t set_breakpoint(int pid, uint64_t addr)
98 {
99     uint64_t orig_instruction = ptrace(PTRACE_PEEKDATA, pid, addr, NULL);
100
101     // The next lines insert an instruction in the target that raises an
102     // exception. Specifically, on x86, 0xcc is a special instruction that
103     // causes the CPU to raise the "breakpoint exception".
104     uint64_t orig_upper_bytes = orig_instruction & ~(uint64_t)0xff;
105
106     if (ptrace(PTRACE_POKEDATA, pid, addr, orig_upper_bytes | 0xcc) < 0)
107         perror("ptrace pokedata");
108
109     return orig_instruction;
110 }
111
112 void preserve_brkpoint_and_continue(int pid, uint64_t addr, uint64_t orig_inst)
113 {
114     // Write the original instruction back so it can execute
115     if (ptrace(PTRACE_POKEDATA, pid, addr, orig_inst) < 0)
116         perror("ptrace pokedata");
117
118     // Right here, %rip is one past the instruction we wish to re-execute.
119     // The function below puts %rip where it should be.
120     rewind_rip(pid);
121
122     // Execute the restored instruction
123     single_step(pid);
124
125     // At this point, the target is past the breakpoint, so
126     // set the breakpoint again, and continue. If this were a real debugger
127     // we would have to capture the return value of set_breakpoint, to be able
128     // to (again) restore the original instruction, in case it's
129     // different versus when we first captured it.
130     set_breakpoint(pid, addr);
131     continue_on(pid);
132 }
133
134 // Read %rip (with all other registers), decrement it in the local
135 // data structure, and then set all of the registers, with the updated %rip.
136 void rewind_rip(int pid)
137 {
138     struct user_regs_struct regs;
139     if (ptrace(PTRACE_GETREGS, pid, NULL, &regs) < 0)
140         perror("ptrace getregs");
141
142     printf("%p in target is 0x%llx but we want it to be 0x%x\n", regs.rip, STACK_LOAD);
143
144     regs.rip--;
145 }

```

Apr 16, 25 2:52

debugger.c

Page 3/3

```

147
148     if (ptrace(PTRACE_SETREGS, pid, NULL, &regs) < 0)
149         perror("ptrace setregs");
150 }
151
152 // This function sets the variable 'x' in the target, which
153 // lives at the address 4 bytes below the frame pointer.
154 // Although what we are trying to do is conceptually
155 // straightforward, the code winds up being complicated
156 // by the fact that PEEK_DATA and POKE_DATA read only in 64-bit
157 // quantities. So the code has to take care to preserve the stuff "before"
158 // and "after" the relevant slot, which is only 32 bits.
159 void set_x_in_target(int pid, uint32_t newval)
160 {
161     struct user_regs_struct regs;
162     uint64_t x_in_stack;
163     if (ptrace(PTRACE_GETREGS, pid, NULL, &regs) < 0)
164         perror("ptrace getregs");
165
166     // Read all 64 bits from the relevant location in the target's stack frame
167     // and display the bottom 32 bits.
168     x_in_stack = ptrace(PTRACE_PEEKDATA, pid, regs.rbp - 4, NULL);
169     printf("Checking: *((%rbp-4):%ld\n", x_in_stack & 0xffffffff);
170
171     // Zero out the bottom four bytes and then set those bytes to be newval.
172     x_in_stack &= ~(uint64_t)0xffffffff;
173     x_in_stack |= newval;
174
175     // Write 64 bits into the target's stack frame
176     if (ptrace(PTRACE_POKEDATA, pid, (uint64_t)regs.rbp - 4, x_in_stack) < 0)
177         perror("ptrace pokedata");
178
179     x_in_stack = ptrace(PTRACE_PEEKDATA, pid, regs.rbp - 4, NULL);
180     printf("Checking: *((%rbp-4):%ld\n", x_in_stack & 0xffffffff);
181 }
182
183 void show_target_breakpoint(int pid, uint64_t addr)
184 {
185     uint64_t instruction = ptrace(PTRACE_PEEKDATA, pid, addr, NULL);
186     printf("instruction at addr 0x%lx is now: 0x%lx\n", addr, instruction);
187 }
188

```

Apr 16, 25 0:44

target.c

Page 1/1

```

1  #include <sys/types.h>
2  #include <unistd.h>
3  #include <signal.h>
4  #include <stdio.h>
5
6  // After building this program, you can use
7  //      $ objdump -d target > target.s
8  // to disassemble the binary. This tells us
9  // at what line of code is the second printf.
10 int main(int argc, char** argv)
11 {
12     int x = 10;
13     printf("s:x=%d\n", argv[0], x);
14     // When this program is run by the program "debugger", the value of
15     // x will be changed mid-execution, to be set to 202.
16     printf("s:x=%d\n", argv[0], x);
17     return 0;
18 }

```