CS202 Handout 14

1. Attaching to a Debugger

1.1 Launch a process that is attached

```
void launch_attached(const char* path,
        char* const argv[]) {
  int pid = fork();
  if (pid == 0) {
    ptrace(PTRACE_TRACEME, 0, NULL, NULL);
    execv(path, argv);
  }
  return pid;
}
```

1.2 Attach to a running process

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

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

```
void wait_for_target(pid_t pid) {
  int status;
  while (1) {
    waitpid(pid, &status, 0);
    if (WIFSTOPPED(status)) {
      // The reason for the change was that pid stopped.

      // We should have stopped because of
      // either SIGTRAP or SIGSTOP.
      assert(WSTOPSIG(status) == SIGTRAP
        || WSTOPSIG(status) == SIGSTOP);

      break;

    } else if (WIFEXITED(status)) {
      // The process exited before we could attach.
      printf("Process exited\n");
      break;
    }
  }

  if (WIFSTOPPED(status)) {
      // In here, the debugger manipulates the target.
      // As a simple example, if the debugger wants to
      // "continue" the target, it executes the following:
      ptrace(PTRACE_CONT, pid, NULL, NULL);

  }
}
```

2. Interrupting the running target

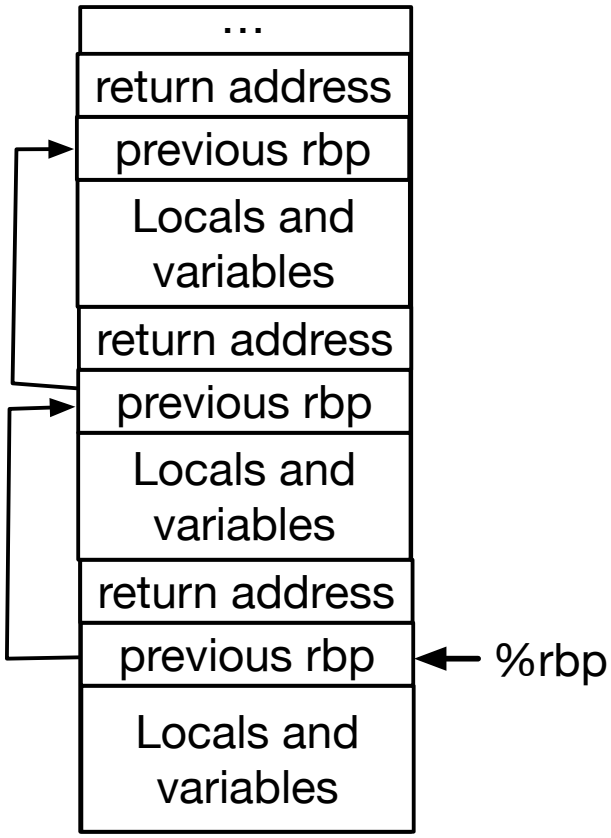
```c
void interrupt_target(pid_t pid) {
    // kill() is a system call that sends OS signals
    kill(pid, SIGSTOP);
    // Must use waitpid in order to
    // wait for the signal to be
    // delivered.
}
```

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.

```c
// Execute a single instruction in the process.
ptrace(PTRACE_SINGLESTEP, pid, NULL, NULL);

// Get non-floating point registers.
// This includes rsp, rip, rbp, etc.
struct user_regs_struct regs;
ptrace(PTRACE_GETREGS, pid, NULL, &regs);

// Get floating point registers.
struct user_fpregs_struct fpregs;
ptrace(PTRACE_GETFPREGS, pid, NULL, &fpregs);

// Set registers. This can be used to update
// register values.
ptrace(PTRACE_SETREGS, pid, NULL, &regs);

// Note: PTRACE_PEEKUSER and PTRACE_POKEUSER
// provide a more efficient way to read or
// write a single register.

// Read a word (8 bytes) from address `addr`
// in target process memory. Note, despite being
// called PTRACE_PEEKDATA, on Linux this can
// read any part of memory, including the
// text segment.
uint64_t val;
val = ptrace(PTRACE_PEEKDATA, pid, addr, NULL);

// Write a word (8 byte) to address `addr` in
// target procss memory.
ptrace(PTRACE_POKEDATA, pid, addr, val);

// Get information on the signal that caused
// the target procss to stop.
siginfo_t sinfo;
ptrace(PTRACE_GETSIGINFO, pid, &sinfo, NULL);
```

4. Stack Frames and Unwinding



```c
// debugger.c: Simple "debugging" program intended to show
//    how one process (this one) can manipulate another
//    one. This program is hard-coded to work with a
//    target process called "target". It is also hard-coded
//    to set a breakpoint before the target's second printf,
//    and at that point, to modify the variable 'x' in the target
//    to be equal to 202.

#include <unistd.h>
#include <sys/types.h>
#include <sys/ptrace.h>
#include <sys/wait.h>
#include <sys/user.h>
#include <stdio.h>
#include <assert.h>
#include <stdint.h>

void wait_for_target(int pid);
void continue_on(int pid);
void single_step(int pid);
uint64_t set_breakpoint(int pid, uint64_t addr);
void preserve_brkpoint_and_continue(int pid, uint64_t addr, uint64_t orig_inst);
void rewind_rip(int pid);
void set_x_in_target(int pid, uint32_t val);
void show_target_breakpoint(int pid, uint64_t addr);

// We inspect the target binary to learn at what address the value of 'x' (in
// the target) is read from the stack prior to the second printf.
// That is the address below. That's a hack. A real debugger would infer
// the address by examining debugging information (symbol tables and so on).
#define STACK_LOAD 0x401887

int main(int argc, char** argv)
{
    int pid = fork();
    if (pid == 0) {
        ptrace(PTRACE_TRACEME, 0, NULL, NULL);
        if (execl("target", "target", NULL) < 0)
            perror("exec()");
    }

    if (pid < 0)
       perror("failed to fork\n");

    wait_for_target(pid);

    // When setting a breakpoint, we have to keep around the
    // original contents of the target's code at the memory location.
    uint64_t original = set_breakpoint(pid, STACK_LOAD);
    show_target_breakpoint(pid, STACK_LOAD);
    continue_on(pid);

    set_x_in_target(pid, 202);

    preserve_brkpoint_and_continue(pid, STACK_LOAD, original);

    return 0;

}

void wait_for_target(int pid)
{
    int status;

    while (1) {

        waitpid(pid, &status, 0);

        if (WIFSTOPPED(status)) {
            if (WSTOPSIG(status) != SIGTRAP &&
                WSTOPSIG(status) != SIGSTOP)  {
                    printf("target stopped due to signal: %d\n", WSTOPSIG(status));
                }
```



```c
            break;
        } else if (WIFEXITED(status)) {
            printf("target exited\n");
            break;
        }
    }
}

void continue_on(int pid)
{
    if (ptrace(PTRACE_CONT, pid, NULL, NULL) < 0)
        perror("ptrace continue");

    wait_for_target(pid);
}


void single_step(int pid)
{
   if (ptrace(PTRACE_SINGLESTEP, pid, NULL, NULL) < 0)
        perror("ptrace singlestep");

   wait_for_target(pid);
}

uint64_t set_breakpoint(int pid, uint64_t addr)
{
    uint64_t orig_instruction = ptrace(PTRACE_PEEKDATA, pid, addr, NULL);

    // The next lines insert an instruction in the target that raises an
    // exception. Specifically, on x86, 0xcc is a special instruction that
    // causes the CPU to raise the "breakpoint exception".
    uint64_t orig_upper_bytes = orig_instruction & ~(uint64_t)0xff;

    if (ptrace(PTRACE_POKEDATA, pid, addr, orig_upper_bytes | 0xcc) < 0)
        perror("ptrace pokedata");

   return orig_instruction;
}

void preserve_brkpoint_and_continue(int pid, uint64_t addr, uint64_t orig_inst)
{
    // Write the original instruction back so it can execute
    if (ptrace(PTRACE_POKEDATA, pid, addr, orig_inst) < 0)
        perror("ptrace pokedata");

    // Right here, %rip is one past the instruction we wish to re-execute.
    // The function below puts %rip where it should be.
    rewind_rip(pid);

    // Execute the restored instruction
    single_step(pid);

    // At this point, the target is past the breakpoint, so
    // set the breakpoint again, and continue. If this were a real debugger
    // we would have to capture the return value of set_breakpoint, to be able
    // to (again) restore the original instruction, in case it's
    // different versus when we first captured it.
    set_breakpoint(pid, addr);
    continue_on(pid);
}

// Read %rip (with all other registers), decrement it in the local
// data structure, and then set all of the registers, with the updated %rip.
void rewind_rip(int pid)
{
    struct user_regs_struct regs;
    if (ptrace(PTRACE_GETREGS, pid, NULL, &regs) < 0)
        perror("ptrace getregs");

    printf("%%rip in target is 0x%llx but we want it to be 0x%x\n", regs.rip, STACK_LOAD);

    regs.rip--;
```

```
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
```



```
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 }
```