Graham Dungan
April 15, 2025
UIN: 332001764

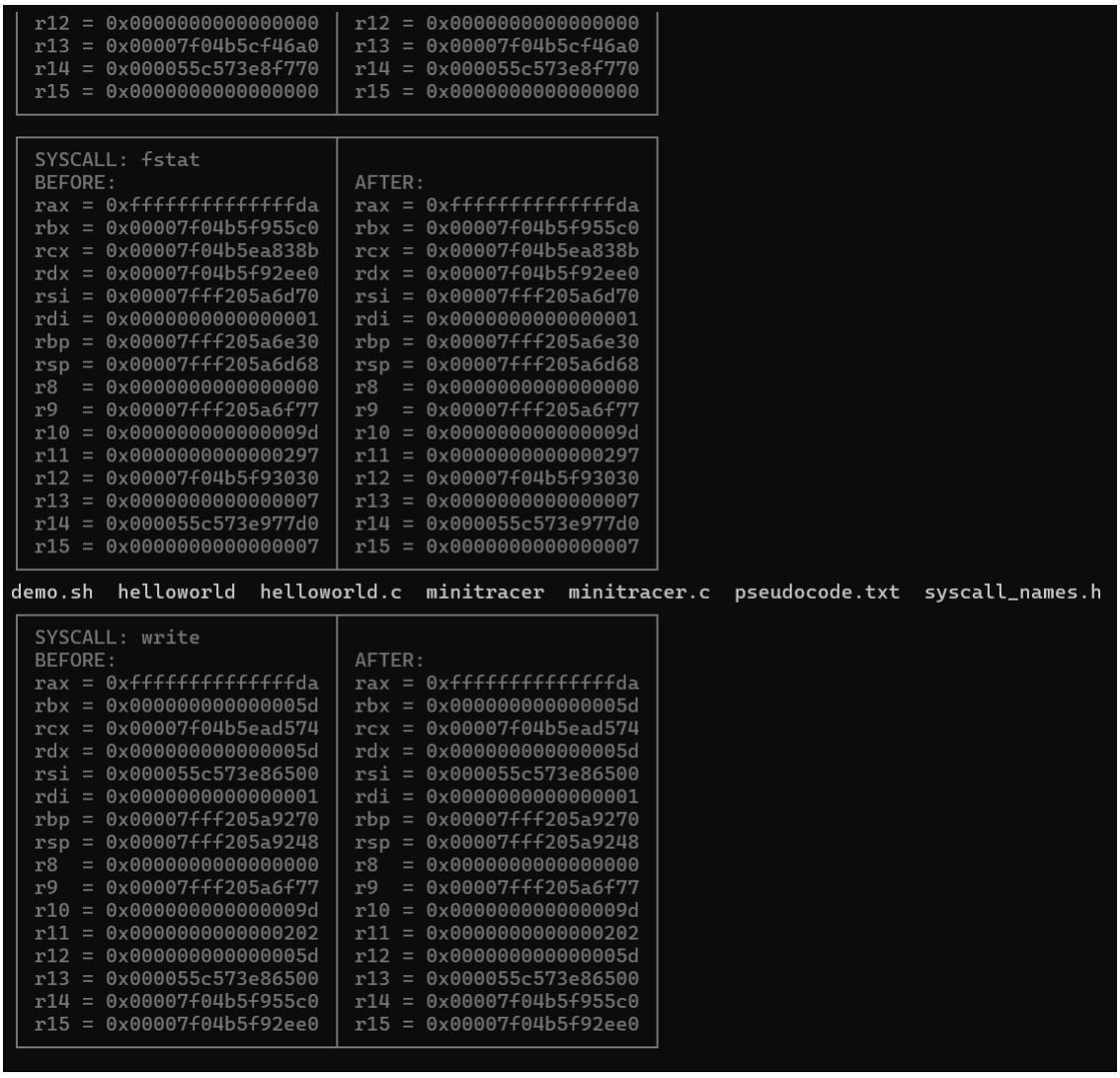
CSCE 413: Software Security
Class 34: PTRACE

Demonstration Quickstart

A demonstration script `demo.sh` has been included. This demo script runs `/bin/ls` with `minitracer`. To run this demo,

1. Enter the `Class35` directory.
   `cd Class35`
2. *(Optional)* If the script does not have execution permission, add such permissions.
   `chmod +x ./demo.sh`
3. Run the `demo.sh` script.
   `./demo.sh`

What follows is a screenshot of the output of `demo.sh`.



Creating `minitracer`

To create a minitracer with `ptrace`, I created `minitracer.c`. What follows is a truncated version of the program to discuss functionality.

```c
int main(int argc, char *argv[]) {
    pid_t child_pid = fork();
    if (child_pid) {
        return run_trace(child_pid);
    } else {
        return run_child(argc - 1, argv + 1);
    }
    return 0;
}

int run_child(int argc, char *argv[]) {
    char *args[argc + 1];
    memcpy(args, argv, argc * sizeof(char *));
    args[argc] = NULL;

    ptrace(PTRACE_TRACEME, 0, NULL, NULL);
    kill(getpid(), SIGSTOP);
    execvp(argv[0], args);

    fprintf(stderr, "Failed to execvp %s!\n", argv[0]);
    return 1;
}

int run_trace(pid_t child_pid) {
    int status;
    struct user_regs_struct before_regs;
    struct user_regs_struct after_regs;

    waitpid(child_pid, &status, 0)
    ptrace(PTRACE_SETOPTIONS, child_pid, 0, PTRACE_O_TRACESYSGOOD);

    while (1) {
        ptrace(PTRACE_GETREGS, child_pid, NULL, &before_regs);
        if (wait_for_syscall(child_pid) != 0) break;  // If the child exited, exit
        ptrace(PTRACE_GETREGS, child_pid, NULL, &after_regs);

        long sc = after_regs.orig_rax;
        const char *syscall_name = NULL;
        if (sc >= 0 && sc < sizeof(syscall_names) / sizeof(syscall_names[0])) syscall_name =
            syscall_names[sc];
        if (!syscall_name) syscall_name = "unknown";

        print_regs(&before_regs, &before_regs, syscall_name);
        if (wait_for_syscall(child_pid) != 0) break;
    }
    return 0;
}

int wait_for_syscall(pid_t child_pid) {
    int status;
    ptrace(PTRACE_SYSCALL, child_pid, NULL, NULL);
    waitpid(child_pid, &status, 0);
    if (WIFEXITED(status) || WIFSIGNALED(status)) {
        return 1;
    }
    return 0;
}
```

What follows is a brief explanation of `minitracer.c`'s design. Lines 1-9 are the `main` function. The actual program has argument handling, but has been excluded for brevity. The `main` function immediately forks. The child process runs `run_child`, and the parent process runs `run_trace`.

`run_child` prepares the child to be traced and then `execvp`'s the desired program. Line 16 prepares the child for this with `PTRACE_TRACEME`, and then runs the supplied program in the following `execvp`.

`run_trace` allows the parent to trace the child. It first waits for the child to send a signal indicating that it is ready to be traced. Once ready, line 30 calls `ptrace` to set the options for what it traces for the child. In this scenario, the parent process will use `PTRACE_O_TRACESYSGOOD` to indicate that it wants to trace the child process's system calls.

`run_trace` then enters a continuous loop on line 32 that only exits once the child has terminated. In this loop, it uses `ptrace` to get the registers before any system calls are made and stores these in `before_regs`. It then calls the helper function `wait_for_syscall`. `wait_for_syscall` waits for a system call to be made by the child. Once the child sends a signal, it is checked to see if the child has terminated or if it is a signal indicating that a system call was made. The `wait_for_syscall` function will return a boolean indicating the status of the child. This boolean will indicate that the child has either terminated (by which the parent will then exit the infinite loop) or if the child made a system call.

After the child has made the system call registers are saved to `after_regs`. Using `after_regs`, the system call used is translated to a human-readable name via an externally defined `syscall_names` array. All registers, before and after, are then printed with `print_regs`, which has been excluded for the sake of brevity. Another call `wait_for_syscall` is made to collect `before_regs` for the next iteration.

This loop will continue running to watch for all system calls made by the child process. The loop will only exit once the child has terminated.

Tracing a Program

`minitracer` was originally tested with a simple program called `helloworld`,

```
#include <stdio.h>
int main() {
    printf("Hello World!\n");
    return 0;
}
```

Since this program is small and makes fewer system calls, we can easily compile `minitracer.c` and run it on `helloworld` using;

```
gcc minitracer.c -o minitracer ./minitracer ./helloworld
```

What follows is a portion of the output;

```
SYSCALL: brk
BEFORE:                          AFTER:
rax = 0xffffffffffffffda         rax = 0xffffffffffffffda
rbx = 0x000056160b55c000         rbx = 0x000056160b55c000
rcx = 0x00007f623995f77b         rcx = 0x00007f623995f77b
rdx = 0x00007f6239a4cd70         rdx = 0x00007f6239a4cd70
rsi = 0x00007f6239a45b20         rsi = 0x00007f6239a45b20
rdi = 0x000056160b55c000         rdi = 0x000056160b55c000
rbp = 0x00007ffebad75020         rbp = 0x00007ffebad75020
rsp = 0x00007ffebad74ff8         rsp = 0x00007ffebad74ff8
r8  = 0x0000000000021000         r8  = 0x0000000000021000
r9  = 0x0000000000000000         r9  = 0x0000000000000000
r10 = 0x0000000000000001         r10 = 0x0000000000000001
r11 = 0x0000000000000206         r11 = 0x0000000000000206
r12 = 0x000056160b53b000         r12 = 0x000056160b53b000
r13 = 0x00007f6239a4cd70         r13 = 0x00007f6239a4cd70
r14 = 0x00007f6239a45b20         r14 = 0x00007f6239a45b20
r15 = 0x00007f6239a45ac0         r15 = 0x00007f6239a45ac0
Hello World!
SYSCALL: write
BEFORE:                          AFTER:
rax = 0xffffffffffffffda         rax = 0xffffffffffffffda
rbx = 0x000000000000000d         rbx = 0x000000000000000d
rcx = 0x00007f623995e574         rcx = 0x00007f623995e574
rdx = 0x000000000000000d         rdx = 0x000000000000000d
rsi = 0x000056160b53b2a0         rsi = 0x000056160b53b2a0
rdi = 0x0000000000000001         rdi = 0x0000000000000001
rbp = 0x00007ffebad75240         rbp = 0x00007ffebad75240
rsp = 0x00007ffebad75218         rsp = 0x00007ffebad75218
r8  = 0x00007f6239a45b20         r8  = 0x00007f6239a45b20
r9  = 0x0000000000000410         r9  = 0x0000000000000410
r10 = 0x0000000000000001         r10 = 0x0000000000000001
r11 = 0x0000000000000202         r11 = 0x0000000000000202
r12 = 0x000000000000000d         r12 = 0x000000000000000d
r13 = 0x000056160b53b2a0         r13 = 0x000056160b53b2a0
r14 = 0x00007f6239a465c0         r14 = 0x00007f6239a465c0
r15 = 0x00007f6239a43ee0         r15 = 0x00007f6239a43ee0
```

As seen here, this program successfully traces a supplied program with `ptrace` and prints the values of registers before and after a system call is made.