snudbg.md

Lab Assignment: SNU Debugger (snuDbg)

Systems Programming, 2022 Fall

- **Due date: Sun.,Dec. 4, 11:59PM**
- TA: Jaewon Hur (hurjaewon@snu.ac.kr)

Introduction

In this lab assignment, you will be developing your own debugger, `snuDbg`. `snuDbg` is a simple Linux-based debugger, which relies on `ptrace` syscalls. In the following, we will first describe how the `ptrace` syscalls work, and how you will complete the assignment using the `ptrace` syscalls.

`ptrace` syscall

The ptrace() system call provides a means by which one process (the "tracer") may observe and control the execution of another process (the "tracee"), and examine and change the tracee's memory and registers. It is primarily used to implement breakpoint debugging and system call tracing.

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

The `ptrace` syscall takes four arguments: `request` specifies the action to be performed; `pid` denotes the target ID (i.e., thread or process ID) to perform the action; `addr` and `data` provide the extra parameter values for a certain `request`.

While this assignment would be sufficient to understand the following `request` values, you are strongly encouraged to read through the man page of `ptrace` to get more information.

- `PTRACE_TRACEME`: Indicate that this process is to be traced by its parent.
- `PTRACE_PEEKTEXT, PTRACE_PEEKDATA`: Read a word at the address `addr` in the tracee's memory, returning the word as the result of the ptrace() call.
- `PTRACE_POKETEXT, PTRACE_POKEDATA`: Copy the word `data` to the address `addr` in the tracee's memory.
- `PTRACE_GETREGS`: Copy the tracee's general-purpose or floating-point registers, respectively, to the address `data` in the tracer.
- `PTRACE_SETREGS`: Modify the tracee's general-purpose or floating-point registers, respectively, from the address `data` in the tracer.
- `PTRACE_SINGLESTEP`: Restart the stopped tracee as for PTRACE_CONT, but arrange for the tracee to be stopped at the next entry to or exit from a system call, or after execution of a single instruction, respectively.
- `PTRACE_CONT`: Restart the stopped tracee process.

Let's take a look at the simple code example of `ptrace`, which shows the tracee runs `/bin/ls` and the tracer captures the current register values of the tracee (the example is taken from https://gist.github.com/willb/14488/80deaf4363ed408a562c53ab0e56d8833a34a8aa).

```
pid_t child = fork();
if (child == 0) {
  ptrace(PTRACE_TRACEME, 0, NULL, NULL);
  execl("/bin/ls", "ls", NULL);
} else {
  int status;

  while(waitpid(child, &status, 0) && ! WIFEXITED(status)) {
    struct user_regs_struct regs;
    ptrace(PTRACE_GETREGS, child, NULL, &regs);
  }
}
```

In this example, `fork()` creates two execution contexts, where the parent process becomes the tracer and the child process becomes the tracee.

In the case of the child process, it invokes `ptrace()` with `PTRACE_TRACEME`, notifying the kernel that it would allow to be traced by other processes. Then the child goes on invoking the syscall `execl()` so that it replaces itself with the process executing `/bin/ls`.

In the case of the parent process, it first waits until the child is ready. Once ready, it invokes `ptrace()` with `PTRACE_GETREGS`, which obtains the all register values of the child process.

snuDbg

`snuDbg` is a simple Linux-based debugger. It takes the target program through the terminal parameter (i.e., `USAGE: ./snudbg <cmd>`), and provides various features to debug the target program.

In order to build `snuDbg`, run the command `make` in the `src` directory, which generates the executable file, `snudbg`.

```
$ cd src
$ make
gcc -g -o snudbg snudbg.c procmaps.c -Wall -Wextra -Werror -Wshadow -pedantic
$ ls -l snudbg
-rwxr-xr-x   1 blee           blee        40336 2022-10-06 14:31 snudbg
```

The following shows the running example, which debugs the program `rand` (the source code of `rand` is given in `test/sample-rand/rand.c`). Note that this shows the running result of the reference implementation, which completed all the tasks.

```
$ ./snudbg ../test/prebuilt/rand

[*] Tracer with pid=15549
[*] Tracee with pid=15550
[*] Loading the executable [../test/prebuilt/rand]
[*] [step 1] rip=7ffff7fd0103 child_status=1407
>>> help
[*] Available commands:
[*]      regs | get [REG] | set [REG] [value]
[*]      read [addr] [size] | write [addr] [value] [size]
[*]      step | continue | break [addr]
[*]      help
>>> regs
[*] HANDLE CMD: regs
        rax=0x0 rbx=0x0 rcx=0x0 rdx=0x0
        rbp=0x0 rsp=0x7fffffffde80 rsi=0x0 rdi=0x7fffffffde80
        r8=0x0 r9=0x0 r10=0x0 r11=0x0
        r12=0x0 r13=0x0 r14=0x0 r15=0x0
        rip=0x7ffff7fd0103 eflags=0x202
>>> get rsp
[*] HANDLE CMD: get [rsp]
        rsp=0x7fffffffde80
>>> step
[*] HANDLE CMD: step
[*] [step 2] rip=7ffff7fd0df0 child_status=1407
>>> read 0x4010 4
[*] HANDLE CMD: read [4010][555555558010] [4]
         555555558010 34 12 00 00
>>> break 0x1255
[*] HANDLE CMD: break [1255][555555555255]
>>> continue
[*] HANDLE CMD: continue
[*] [step 3] rip=555555555256 child_status=1407
[*]      FOUND MATCH BP: [0] [555555555255][e8]
>>> continue
[*] HANDLE CMD: continue
Wrong answer. Your rand_value was 1acfafb6
[*] Exited in 4 steps with status=0
```

Once running, `snuDbg` shows the user prompt `>>>`, which waits for the command from the user. `snuDbg` supports various commands, which can be listed using the command `help`:

- **regs**: Shows values of all registers.
- **get [REG]**: Show the value of the register `REG`.
- **set [REG] [value]**: Set the value of the register `REG` with `value`.
- **read [addr] [size]**: Show the memory data values at the address `addr` with the size `size`.
- **write [addr] [value] [size]**: Set the memory data values at the address `addr` using the data `value` and the size `size`.
- **step**: Execute a single instruction.
- **continue**: Continue executing a program (until it reaches a breakpoint).
- **break [addr]**: Install the breakpoint at the address `addr`.

Task: Complete the implementation of snuDbg

Your task is to complete the implementation of `snuDbg`. All your code changes should be done in `snudbg.c`, and you should not modify any other files.

In order to correctly grade your implementation, you should strictly follow all the instructions in `snudbg.c`. You should need to implement the function with the annotation `TODO`, where you will need to drop `TODO_UNUSED()` macros (this is used to avoid compilation errors). If the function has the annotation `INSTRUCTION: YOU SHOULD NOT CHANGE THIS FUNCTION`, that means you should not modify the implementation of that function as it is critical for the grading.

The followings are hints or tips for this assignment.

- **Address translation**: You should be careful about the representation of the address. All addresses provided through the user prompts are the addresses embedded within the program binary (i.e., the address right after linking). Thus, it is not the virtual address after being loaded into the memory. This will need you to translate the user-provided address into the virtual address (HINT: look at the code `get_iamge_baseaddr()` and `construct_procmaps()`).

- **Error handling**: If possible, our grading script won't be testing corner cases to see if you have well handled all bizzare corner cases. So please focus on implementing all the debugging features described in this document.

- **General output format**: You do not need to strictly follow the output format of the reference implementation (which is shown above). As long as you don't modify the functions as instructed in `snudbg.c`, your implementation would receive full marks. The exception is when you implement the command `read`, which we elaborate next.

- **Output format of read command**: When implementing the `read` command, you should be using the function `dump_addr_in_hex()` to print the data address. This is because the testing (i.e., `test/run_test.py`) relies on the output format of this function.

Manual Testing

To help your **debugging process** of `snuDbg`, we provide two sample programs, `rand` (see `test/sample-rand/rand.c`) and `array` (see `test/sample-array/array.c`). Note that we provide the pre-built binaries for these two programs (check `test/prebuilt/rand` and `test/prebuilt/array`). This is because 1) addresses of these binaries may subject to change depending on compiler versions and platforms and 2) the address has to be fixed for easy input/ouput testing.

To help you to get the better sence of how the reference implementation of `snuDbg` works, we provide input/output pairs. Note that your `snuDbg` does not need to strictly follow the output format of the reference implementation. As long as your `snuDbg` passes all the tests of `test/run_test.py`, you should be fine.

Automated Testing

In the end, you can test your implementation using the testing script `test/run_test.py`. This runs the following three tests.

1. `test_rand_mem_write`: This tests the pre-built binary `rand` with the input `rand.mem-write.input`. The key of this testing is to setup the breakpoint and once the breakpoint is fired, it overwrites the global variable holding the random value.

2. `test_rand_set_reg`: This tests the binary `rand` with the input `rand.set-reg.input`. The key of this testing is to assign the `ZF` flag at the breakpoint such that the run always passes the random value check.

3. `test_array_check`: This tests the binary `array` with the input `array.check.input`. The key of this testing is to check `snuDbg`'s feature in dumping the memory.

Once your implementation is near to complete, you should be able to get the following result.

```
$ ./run_test.py
[+] PASS: test_rand_mem_write
[+] PASS: test_rand_set_reg
[+] PASS: test_array_check
```

Since the grading would be similar to how `run_test.py` checks your `snuDbg`, you are encouraged to have a look at how the testing is done to avoid potential missing points.

Logistics

This is an individual project, so you should work alone. You may discuss with your classmates, but such a discussion should be well noted in your submission and you should write your own code.

Submission

Prepare your submission with following commands:

```
$ ./prepare-submit.sh
[*] Remove tar file...
[*] Compress files...
src/snudbg.c
[*] Successfully Compressed!
[*] Done! You are ready to submit: assign6.tar.gz
```

Upload `assign6.tar.gz` to the submission server. The URL of the submission server will be provided later.

References

- ptrace(2) - Linux manual page: https://man7.org/linux/man-pages/man2/ptrace.2.html
- How do debuggers (really) work?: https://events.static.linuxfound.org/sites/events/files/slides/slides_16.pdf
- GDB Internals Manual: https://sourceware.org/gdb/wiki/Internals