Everything you need to know to get started with strace

Introduction

According to its manual page (`strace (1)`), `strace` is a tool one can use to trace all the system calls made by a running process. Once attached to a running process, `strace` is able to intercept the system calls that are called by this process and retrieve all the information about them (parameters, return value, ...). `strace` is also able to intercept signals received by the process being traced.

How to intercept syscalls?

It is possible to trace a running process, using a single system call named `ptrace` (process trace).

Actually, `ptrace` is a very powerful interface, allowing one to control another process. `ptrace` allows one to interrupt a running process, resume it, get information about its stack, memory and registers, and also change them.

As you may have understood, `ptrace` is a very useful interface for debugging a program.

It is strongly advised to carefully read the manual page of `ptrace (2)` before starting the following exercises.

Learn by practicing!

If you did carefully read the manual page of `ptrace` , you now know that there are two ways of tracing a process:

- `PTRACE_ATTACH` -> Trace a running process, by specifying its PID.
- `PTRACE_TRACEME` -> Called by the tracee to indicate that the calling process is to be traced by its parent.

We are able to use `strace` two different ways: Either by giving it a command to execute and trace, or by giving it the PID of a process to trace.

Exercise 0

Write a simple program that will be used this way : `./ex_0 binary_to_trace` .

You will need to `fork` and do the following:

- In the child process: Call `ptrace` with the request `PTRACE_TRACEME` to trace the process, and use `execve` to execute the given command.
- In the parent: Wait for the child and trace it using `ptrace` in a loop with the request `PTRACE_SINGLESTEP` . Print `single step` at each step. Print the tracee exit status when the tracing is done.

```
alex@~/0x0b-strace/Concept$ gcc -Wall -Wextra -Werror -pedantic ex_0.c -o ex_0
alex@~/0x0b-strace/Concept$ ./ex_0 /bin/ls
single step
single step
single step
[...]
ex_0  ex_0.c
single step
single step
[...]
Exit status: 0
alex@~/0x0b-strace/Concept$ ./ex_0 /bin/ls test
single step
single step
single step
[...]
/bin/ls: cannot access test: No such file or directory
single step
single step
[...]
Exit status: 2
alex@~/0x0b-strace/Concept$
```

That's a loooot of printing here. The reason is that `PTRACE_SINGLESTEP` suspends the tracee every time the register `ip` changes. Remember, `ip` (Instruction Pointer) holds the address to the current bytecode to be executed.

Exercise 1

Now let's try to use the `PTRACE_SYSCALL` request. Again, read carefully the manual page of `ptrace` to understand this request. Remember you need to call it twice for every syscall made by the tracee.

Write a simple program that will be executed the same way as `ex_0` . But as you may have understood, this time, try to use the `PTRACE_SYSCALL` request, and print `syscall` when the syscall is made, and `return` when the syscall returns.

Example:

```
alex@~/0x0b-strace/Concept$ hbs ex_1.c -o ex_1
alex@~/0x0b-strace/Concept$ ./ex_1 /bin/ls
syscall return
syscall return
[...]
syscall return
ex_0  ex_0.c  ex_1  ex_1.c
syscall return
[...]
syscall return
syscallExit status: 0
alex@~/0x0b-strace/Concept$
```

Now that's way less printing than our first program that used `PTRACE_SINGLESTEP`. The reason is simple: `PTRACE_SYSCALL` suspends the tracee every time a syscall is made, and not at every step.

Exercise 2

`ptrace` allows you to retrieve the tracee's registers values using the request `PTRACE_GETREGS`. This will be useful to retrieve the parameters and return value when a syscall is made, in order to print them like `strace` does.

Write a program that prints the system call number when a syscall is made.

Example:

```
alex@~/0x0b-strace/Concept$ hbs ex_2.c -o ex_2
alex@~/0x0b-strace/Concept$ ./ex_2 /bin/ls
62
59
12
21
[...]
5
9
1
ex_2  ex_0.c  ex_1.c  ex_2.c
3
11
3
231
Exit status: 0
alex@~/0x0b-strace/Concept$
```

Going further

After completing the previous exercises, you should be able to get started with the `strace` project. At least, for the mandatory part ... ;)

If you're wondering how to get the name of a syscall from its number, take a look at the file `/usr/include/asm/unistd_64.h` (Parsing is awesome!)

Don't hesitate to read the concept page about `x86 Assembly` again, especially the part about the registers.