CS451 – Software Analysis

Lecture 3

Tracing Binaries

Elias Athanasopoulos
athanasopoulos.elias@ucy.ac.cy

Tracing

- Binary code can be executed as a process on an operating system
- While a process executes, it interacts with the operating system
  - Processes use system calls
- We can learn a lot about binary code, without touching it
  - By just observing the process interaction with the system

`strace` – system call tracer

- `strace` is a tool for tracing a process and inspecting all system calls
- Each time the process issues a system call, the tool stops, collects information and resumes the *traced* process
- All collected information is printed in the screen

Example output

```
$ strace /bin/ls
execve("/bin/ls", ["/bin/ls"], 0x7fffdf0bbc20 /* 28 vars */) = 0
brk(NULL)                                = 0x5620df341000
arch_prctl(0x3001 /* ARCH_??? */, 0x7ffed192f380) = -1 EINVAL (Invalid argument)
access("/etc/ld.so.preload", R_OK)       = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=16859, ...}) = 0
mmap(NULL, 16859, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f097b2c0000
close(3)                                 = 0
openat(AT_FDCWD, "/lib64/libselinux.so.1", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\200z\0\0\0\0\0\0"...,
832) = 832
lseek(3, 157168, SEEK_SET)               = 157168
read(3, "\4\0\0\0 \0\0\0\5\0\0\0GNU\0\1\0\0\300\4\0\0\0\30\0\0\0\0\0\0\0"..., 48)
= 48
fstat(3, {st_mode=S_IFREG|0755, st_size=168536, ...}) = 0
…
```

What is in the output?

- `strace` prints only the system calls issued by the process
  - It will not print library function calls (e.g., malloc())
- For every system call, there is an attempt to print the call's arguments
  - Arguments may be binary and of arbitrary length
  - The tool will output a few bytes
  - Some arguments are *beautified* (e.g., -1 is printed as EINVAL)
- For every system call the result is also printed
  - syscall(arguments) = result

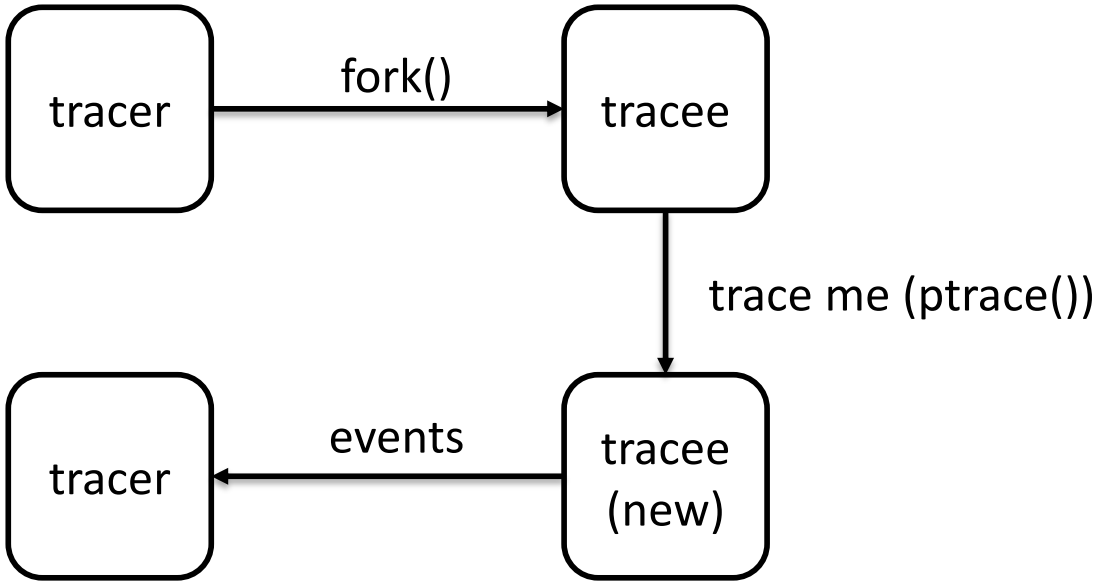
How `strace` works?

- Operating systems offer special system calls for tracing processes
  - This is how debuggers are developed
- Tracing a process means
  - Run the process and pause it under specific events
  - When paused, collect information related to the event that just happened, and resume the process until the next event
- Each time there are *two* processes involved
  - The *tracer* (the process that performs the analysis)
  - The *tracee* (the process that is analyzed)

`ptrace`

- Linux offers the system call `ptrace` for tracing processes
- This is a powerful system call, with many different options and usage
- Debuggers export a rich set of functions, but essentially, they are just wrappers around the `ptrace` system call

High-level construction of analysis



Steps

- The tracer needs to start the new process that is going to be analyzed
  - A new process can be started by cloning the existing one (fork()) and then replacing it with a new process image (execvp())
- Before the replacement, the new process needs to use ptrace() for allowing to be traced
  - Programs are not designed to be traced and they don't use internally ptrace()
  - We need to call ptrace() before execvp() so that the new executing process can be traced
- Once tracing has started, each system call event pauses the tracee
  - System call events are (a) entering a system call and (b) exiting a system call

How to handle events

- The tracer needs to call waitpid() for blocking until a new event has occurred
  - This happens when a system call is about to be called or a system call is just finished
- Each time a new event has occurred, the tracer can use again ptrace() to collect all the state of the tracee
  - This means all the current values of the CPU's registers
  - Recall: system calls are set using the CPU's registers
- The state is collected in a data structure called `user_regs_stats`

ptrace arguments

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

- `request` is one of the many options that ptrace() supports
- `pid` is the affected process id
- `addr` and `data` are used to transfer information
  - Their actual usage differs per ptrace() request

Minimal strace

- We are going to implement a simplified version of strace
- The minimal strace tool will be able to trace any process and output system call information
- Compared to the original strace tool, it will not *beautify* the output a lot

Initialization

- Starting a new process for analysis using fork()/execvp()

```
    /* fork() for executing the program that is analyzed.  */
    pid_t pid = fork();
    switch (pid) {
        case -1: /* error */
            die("%s", strerror(errno));
        case 0:  /* Code that is run by the child. */
            /* Start tracing.  */
            ptrace(PTRACE_TRACEME, 0, 0, 0);
            /* execvp() is a system call, the child will block and
               the parent must do waitpid().
               The waitpid() of the parent is in the label
               waitpid_for_execvp.
             */
            execvp(argv[1], argv + 1);
            die("%s", strerror(errno));
    }
```

Main loop

- The main loop calls ptrace() with PTRACE_SYSCALL on the pid of the analyzed process
  - The blocks until a new event is happening

```
while (1) {
        /* Enter next system call.
           It can be the entrance or the exit of the system call.
         */
        if (ptrace(PTRACE_SYSCALL, pid, 0, 0) == -1)
            die("%s", strerror(errno));
        /* Block until process state change (i.e., next event). */
        if (waitpid(pid, 0, 0) == -1)
            die("%s", strerror(errno));

...
}
```

Collect information

- We call ptrace() with PTRACE_GETREGS to receive all information related to the event
  - Collected information will be copied to regs
  - If ptrace() fails (-1) then we check errno to see if the analyzed process has just finished

```
/* Collect information about the system call.  */
struct user_regs_struct regs;
if (ptrace(PTRACE_GETREGS, pid, 0, &regs) == -1) {
    if (errno == ESRCH) {
        /* System call was exit; so we need to end.  */
        fprintf(stderr, "\n");
        exit(regs.rdi);
    }
    die("%s", strerror(errno));
}
```

Output the information

- `regs` has all the process' state
  - E.g., `regs.rax` holds the value of the RAX register
  - The number of the system call is in the original RAX (before the call)

```
if (regs.rax == -ENOSYS) {
        /* We are in the system call's entrance. */
        long syscall = regs.orig_rax;

        /* Output the system call. */
        fprintf(stderr, "%ld(%ld, %ld, %ld, %ld, %ld, %ld)",
            syscall,
            (long)regs.rdi, (long)regs.rsi, (long)regs.rdx,
            (long)regs.r10, (long)regs.r8,  (long)regs.r9);
} else
        /* We are in the system call's exit. */
        fprintf(stderr, " = %ld\n", (long)regs.rax);
}
```

Example program

- Minimal strace can run with any program
- We can start with a simplified program

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main(int argc, char *argv[]) {

    fprintf(stderr, "My pid is: %d.\n", getpid());

    return 1;
}
```

Compile and run

```
$ gcc –Wall min_strace.c –o min_strace
$ gcc –Wall example.c –o example
$ ./min_strace ./example
12(0, 140347349429278, 79, 0, 29, 1) = 20934656
158(12289, 140729364022816, 140347349389248, 1, 140347351558776, 0) = -22
21(140347349441712, 4, 0, 8, 32, 0) = -2
257(4294967196, 140347349428579, 524288, 0, 0, 4195201) = 3
5(3, 140729364019248, 140729364019248, 0, 0, 4195201) = 0
9(0, 16859, 1, 2, 3, 0) = 140347351535616
3(3, 16859, 1, 2, 3, 0) = 0
257(4294967196, 140347351567856, 524288, 0, 140729364019583, 0) = 3
0(3, 140729364019608, 832, 0, 140729364019583, 0) = 832
5(3, 140729364019248, 140729364019248, 0, 140347351567856, 140347351564752) = 0
9(0, 8192, 3, 34, 4294967295, 0) = 140347351527424
8(3, 808, 0, 0, 3, 2) = 808
0(3, 140729364018416, 32, 0, 3, 2) = 32
9(0, 3950400, 5, 2050, 3, 0) = 140347345326080
10(140347347144704, 2093056, 0, 2, 3, 0) = 0
9(140347349237760, 24576, 3, 2066, 3, 1814528) = 140347349237760
9(140347349262336, 14144, 3, 50, 4294967295, 0) = 140347349262336
3(3, 41, 140347349253296, 50, 140347345326080, 1879048226) = 0
158(4098, 140347351532800, 140347351535152, 144, 1, 64) = 0
10(140347349237760, 16384, 1, 140347345326080, 140729365922296, 140347345326080) = 0
10(6291456, 4096, 1, 0, 140347345376440, 140347349260224) = 0
10(140347351556096, 4096, 1, 140347349278720, 140347345347184, 140347349278720) = 0
11(140347351535616, 16859, -120, -179, 0, 13) = 0
39(1, 140729364023128, 140729364023144, 3, 140347349261600, 140347349261600) = 220740
1(2, 140729364012992, 19, 0, 140347351532800, 140729364012626)My pid is: 220740.
 = 19
231(1, 60, 1, 140729364022500, 231, -128)
```

Homework

- Beautify minimal strace to output the names of some popular system calls
  - You can find all information at https://chromium.googlesource.com/chromiumos/docs/+/master/constants/syscalls.md#x86_64-64_bit