CS451 – Software Analysis

Lecture 11

Software Breakpoints

Elias Athanasopoulos
athanasopoulos.elias@ucy.ac.cy

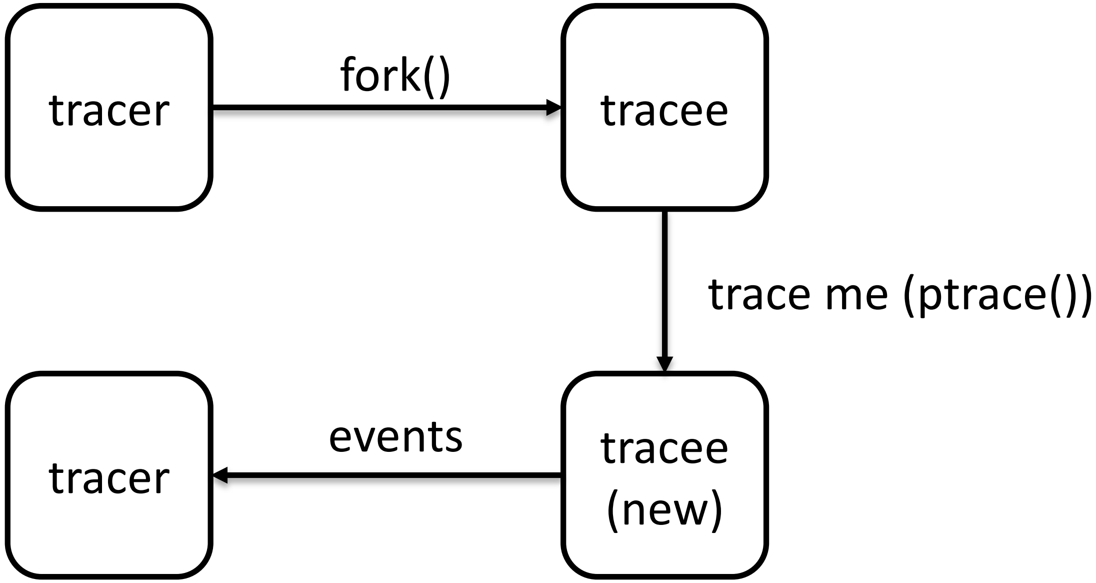
Static vs dynamic analysis

- Static disassembly, linear or recursive, inspects the program’s code at rest
- Often, we need to run the program and inspect the actual code as it is executing while processing specific inputs
- This can be often facilitated by debuggers

Debuggers

- Programs that use special services from the operating system to inspect a process, while executing
  - Linux supports ptrace()
  - The process must explicitly permit tracing
- We have seen in the past how to use ptrace() for inspecting system calls

Recall


tracer
fork()
tracee
trace me (ptrace())
tracer
events
tracee (new)

Events

- So far, we have used ptrace() passively
- We just inspect events, such as the entrance and the exit of a system call
- ptrace() supports a rich set of features that can make analysis more interactive

Software breakpoints

- One fundamental feature of a debugger is to stop the process before it is about to execute a very specific code address
  - This is called a breakpoint
- There are software and hardware breakpoints
  - Hardware breakpoints are called *watches*
  - They are facilitated by the hardware, and we cannot have many of them at a given point

Example of a breakpoint

```
$ gdb ./test
GNU gdb (GDB) Red Hat Enterprise Linux 8.2-16.el8
[...]
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./test...(no debugging symbols found)...done.
(gdb) b foo
Breakpoint 1 at 0x4005da
(gdb) r
Starting program: /home/elathan/epl451/src/week6/test
Missing separate debuginfos, use: yum debuginfo-install glibc-2.28-
164.el8.x86_64


Breakpoint 1, 0x00000000004005da in foo ()
(gdb) disas
Dump of assembler code for function foo:
   0x00000000004005d6 <+0>: push   %rbp
   0x00000000004005d7 <+1>: mov    %rsp,%rbp
=> 0x00000000004005da <+4>: mov    0x200a5f(%rip),%rax
```

Remarks

- Breakpoints are set on specific code addresses
  - We need to make the process stop when it is *about* to execute a very specific memory location
- Most of the times, we use symbolic names of functions, for instance *‘b foo’*
  - gdb does automatically the translation for us
  - gdb has loaded the symbol table and when a known symbol is supplied, the translation happens automatically

Unknown symbols

```
$ gdb ./test
GNU gdb (GDB) Red Hat Enterprise Linux 8.2-16.el8
Copyright (C) 2018 Free Software Foundation, Inc.
[…]
(gdb) b printf
Function "printf" not defined.
Make breakpoint pending on future shared library
load? (y or [n])
```

How can we stop a process on demand?

- Programs are not designed to stop on demand
  - They can be considered more like an instruction stream
- The operating system can be interrupted on specific events
  - When the process needs to execute a system call, the operating system is interrupted for serving the system call
  - This interruption can be used to collect information about the running process (recall the minimal strace tool)

Beyond system calls

- We need a mechanism for stopping the process on every instruction, not only when a system call is about to happen
- We need to introduce the interrupt, ourselves

`int3`

- Intel supports `int3`, a one-byte instruction that, if executed, generates a software interrupt
  - The opcode is `0xCC`
  - This interrupt in principle can stop the process when running, exactly as it happens with system calls
- But programs do not have this instruction

How to use `int3`?

- We can replace the code in the address, where we want the breakpoint, with `int3`
- If we run the program and if we reach the given location, the process will emit a software interrupt
- We can therefore stop the process, replace `int3` with the original instruction and perform the analysis

High-level idea

Load the program

address	**opcode**
a1	op1
a2	op2
a3	op3
a1	op4
a5	op5

Insert the breakpoint

address	**opcode**
a1	op1
a2	op2
a3	**int3**
a1	op4
a5	op5

Run the program

	address	**opcode**
	a1	op1
	a2	op2
>	**a3**	**int3**
	a1	op4
	a5	op5

Place back the original instruction

address	**opcode**
a1	op1
a2	op2
a3	**op3**
a1	op4
a5	op5

Mechanics

- We need to replace existing instructions of the program for turning on breakpoints
- We need to be able to *write* and *read* from the traced process
- ptrace() supports this

```
/* read and store to r */
r = ptrace(PTRACE_PEEKDATA, pid, addr, 0)
/* write addr with value */
ptrace(PTRACE_POKEDATA, pid, addr, value)
```

Replacing bytes in x64

- You need to be a bit careful when replacing the code in 64 bits
- `int3` is one byte, however we write 8 bytes

```
    /* Insert the breakpoint. */
    long trap = (previous_code & 0xFFFFFFFFFFFFFF00) | 0xCC;
    if (ptrace(PTRACE_POKEDATA, pid,
              (void *)BREAKPOINT_ADDR, (void *)trap) == -1)
        die("(pokedata) %s", strerror(errno));
```

Homework

- Assignment 2 is your homework!