Lec 8: Debugger

CS492E: Introduction to Software Security

Sang Kil Cha
Why Use Debugger?

- Help developers run other programs in a controlled environment
- Help examine execution context for every program point
How?

Debugger can access the registers/memory of another process, but how?

With the help of the OS!
Debugging APIs

- Linux/macOS: ptrace
- Windows: functions in evntrace.h, debugapi.h

Debuggers are just a program that uses those APIs
Why Learn Debugging APIs?

Because you have *full control* over a program execution:

- Dynamically analyze program behaviors
- Programatically control program executions (debugging, cracking, etc.)
- …
Debugging Internals

- **Tracee**: a process to be traced
- **Tracer**: a process to control (trace) the tracee
- OS provides an interface between the two via *interrupts*
Two Main Methods

• Create and run a new tracee from scratch
  – (GDB) run

• Attach to an existing process
  – (GDB) attach
ptrace

#include <sys/ptrace.h>

long ptrace(enum __ptrace_request request, pid_t pid, void *addr, void *data);
Creating a Tracee

```c
pid_t child_pid;

child_pid = fork();
if (child_pid == 0) {
    ptrace(PTRACE_TRACEME, 0, 0, 0); // become a tracee
    exec(...); // execl, execve, etc.
} else if (child_pid > 0) { // I am the tracer
    int wait_status;
    wait(&wait_status); // this will return when the tracee is ready
    // ...
} else { /* fatal error here */ }
```
Reading/Writing Registers/Memory

- Read a word from memory
  \[ \text{ptrace}(PTRACE_PEEKTEXT, \text{child_pid}, \text{addr}, 0); \]

- Write a word \((v)\) to memory
  \[ \text{ptrace}(PTRACE_POKETEXT, \text{child_pid}, \text{addr}, v); \]

- Read registers
  \[ \text{struct user_regs_struct } \text{regs}; \]
  \[ \text{ptrace}(PTRACE_GETREGS, \text{child_pid}, 0, \&\text{regs}); \]

- Write to registers
  \[ \text{regs.eax} = 1; \]
  \[ \text{ptrace}(PTRACE_SETREGS, \text{child_pid}, 0, \&\text{regs}); \]
Running Tracee

• Single-stepping: stop at every instruction

```c
while (1) {
    ptrace(PTRACE_SINGLESTEP, child_pid, 0, 0);
    // peek/poke the child process
}
```

• Run until an interrupt occurs

```c
ptrace(PTRACE_CONT, child_pid, 0, 0);
```

• Run until a syscall is invoked from the tracee

```c
ptrace(PTRACE_SYSCALL, child_pid, 0, 0);
```
Breakpoints?

- Tracer waits for an interrupt (with PTRACE_CONT)
- Tracee issues an interrupt at a breakpoint, but how?
SIGTRAP = INT3

- **INT3** instruction is a one-byte (0xcc) instruction in Intel that is dedicated for setting up a software breakpoint.

- When a user inserts a breakpoint at an address A, the debugger will replace the byte at A with 0xcc, and will remember the original value.

- Once a breakpoint is hit by the tracee, then the tracer will restore the original byte value, modify the EIP to A, so that the modified instruction can be executed normally.
Example

8049120:    b8 2c c0 04 08          mov    eax,0x804c02c
8049125:    2d 2c c0 04 08          sub    eax,0x804c02c // breakpoint
804912a:    c1 f8 02

8049120:    b8 2c c0 04 08          mov    eax,0x804c02c
8049125:    cc                      int3
8049126:    2c c0                   sub    al, 0xc0
8049128:    04 08                   add    al, 0x8
804912a:    c1 f8 02                sar    eax,0x2
Software vs. Hardware Breakpoints

• Software breakpoints require modifying the code

• Intel CPU provides special registers for configuring H/W breakpoints
  - No need to change the code (thus, more reliable)
  - Can break on memory access, too
  - But the number of settable breakpoints is largely limited
  - (GDB) rwatch, awatch, etc.
Useful Tools Implemented with ptrace

- **strace**: Syscall tracing tool
- **ltrace**: library call tracing tool
- **GDB**: debugger
- ...
Exercise

Important notes:
- Read the manual for ptrace system call
- Read the manual for wait system call
Conclusion

• Understanding the debugging internals is essential for writing your own dynamic analysis tools

• On *nix world, ptrace is used to implement a debugger
Questions?