Debugging APIs

How debuggers work in Linux and Windows

Giovanni Lagorio

giovanni.lagorio@unige.it
https://csec.it/people/giovanni_lagorio
Twitter & GitHub: zxgio

DIBRIS - Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi
University of Genova, Italy

www.zenhack.it

Outline

1. Linux
   - Starting a debug session
   - Signal handling
   - Inspecting and controlling
2. Windows
   - Starting a debug session
   - Inspecting and controlling
   - Other APIs
3. Further information

Outline

1. Linux
   - Starting a debug session
   - Signal handling
   - Inspecting and controlling
2. Windows
   - Starting a debug session
   - Inspecting and controlling
   - Other APIs
3. Further information

Introduction

- A single syscall, `ptrace(2)`, provides a means by which a tracer process may observe *and control* the execution of another, the tracee
- Debuggers and `{l,s}trace` are based on it
- Interface:

```
#include <sys/ptrace.h>
long ptrace(enum __ptrace_request request,
            pid_t pid, void *addr, void *data);
```

  returns 0 on success, otherwise -1 and sets `errno` appropriately (except for `PTRACE_PEEK*` requests: they return the requested value)
- A tracer first needs to attach the tracee

Security and `ptrace_scope`

Once upon a time (and currently under WSL, since YAMA seems not enabled), users could examine the runtime state, i.e. memory/registers, of *all* their processes

- unless a process used `prctl(2)` to reset its `PR_SET_DUMPABLE` flag
- a compromised process could steal sensitive data from others

A more secure approach is allowing `ptrace` only from parents to a child

`/proc/sys/kernel/yama/ptrace_scope` may contain:

0. classic permissions
1. restricted permissions: only descendants, or opt-out via `prctl(2)`
   - default on Ubuntu, that's why sometime you need `sudo gdb -p ...`
2. admin-only attach
3. no-attach

Attaching

A trace can be initiated by

- calling `fork(2)` and having the resulting child do a `PTRACE_TRACEME`
  - arguments `pid`, `addr`, and `data` are ignored
  - this is the only request used by the tracee

  typically followed by
  - `raise(SIGSTOP);`
  - an invocation of `execve(2)`
    - unless `PTRACE_O_TRACEEXEC` option is in effect, all successful calls to execve by the traced process will cause it to be sent a `SIGTRAP`
- using `PTRACE_ATTACH`; in this case a `SIGSTOP` is sent to the tracee
  - note: it will not necessarily have stopped by the completion of this call
- ...(see the man page for more)

So, in typical scenarios, a signal is sent to the tracee...

Signal handling

While traced, the tracee will stop (freeze) each time a signal is delivered

- Even if the signal is being ignored (except for `SIGKILL`)
- The tracer will be notified at its next call to `wait/waitpid(2)`
- While the tracee is stopped, the tracer can use various requests to inspect and modify tracee's memory and registers

Signal handling in Gdb

Indeed, in gdb *Ctrl-C* does not terminate the target:

- all signals are first notified to the tracer (Gdb), and
- Gdb does not deliver *all* signals

```
info handle →
Signal        Stop      Print   Pass to program Description
SIGHUP        Yes       Yes     Yes             Hangup
SIGINT        Yes       Yes     No              Interrupt
SIGQUIT       Yes       Yes     Yes             Quit
SIGILL        Yes       Yes     Yes             Illegal instruction
SIGTRAP       Yes       Yes     No              Trace/breakpoint trap
SIGABRT       Yes       Yes     Yes             Aborted
SIGEMT        Yes       Yes     Yes             Emulation trap
SIGFPE        Yes       Yes     Yes             Arithmetic exception
SIGKILL       Yes       Yes     Yes             Killed
SIGBUS        Yes       Yes     Yes             Bus error
SIGSEGV       Yes       Yes     Yes             Segmentation fault
...
```

Use `help handle`, in gdb, to get more information

Refresher: `wait*`

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *wstatus);
pid_t waitpid(pid_t pid, int *wstatus, int options);
```

> *. . . these system calls are used to wait for state changes . . . , and obtain information . . . A state change is considered to be:*
>
> - *the child terminated;*
> - *the child was stopped by a signal; or*
> - *the child was resumed by a signal.*
>
> *In the case of a terminated child, . . . if a wait is not performed, then the terminated child remains in a zombie state.*
> *If a child has already changed state, then these calls return immediately. Otherwise, they block . . .*

Attaching and waiting the child

```c
int attach(pid_t pid)
{
	if (ptrace(PTRACE_ATTACH, pid, 0, 0))
		perror("PTRACE_ATTACH");
	else {
		int status;
		if (waitpid(pid, &status, 0) == -1)
			perror("waitpid");
		else if (WIFSTOPPED(status) && WSTOPSIG(status)==SIGSTOP) {
			printf("Successfully attached to %jd\n", (intmax_t)pid);
			return 0;
		}
	}
	return -1;
}
```

Detaching

When the tracer is finished tracing, it can

- cause the tracee to continue executing `PTRACE_DETACH`
- exit; then the tracees are automatically detached

Poor's man anti-debugging technique

Since each thread can have at most one tracer. . .

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ptrace.h>

int main()
{
        if (ptrace(PTRACE_TRACEME, 0, 0, 0) == -1) {
                printf("Hello world!\n");
                return EXIT_SUCCESS;
        }
        /* evil behaviour */
        printf("I'm evil!!!\n");
}
```

More info: `https://seblau.github.io/posts/linux-anti-debugging`

Continuing tracee execution

- `PTRACE_CONT` restarts the stopped tracee process
  - If `data` is nonzero, it is interpreted as the number of a signal to be delivered to the tracee; otherwise, no signal is delivered
- `PTRACE_SYSCALL` and `PTRACE_SINGLESTEP` restart *but* arrange for the tracee to be stopped at
  - the next entry-to/exit-from a system call, or
  - after execution of a single instruction

  From the tracer's perspective, the tracee will appear to have been stopped by receipt of a `SIGTRAP`
  i.e., `WIFSTOPPED(status) && WSTOPSIG(status)==SIGTRAP`

Reading/writing memory

- `PTRACE_PEEKTEXT` and `PTRACE_PEEKDATA` read a word at `addr` in the tracee's memory
  - Linux has no separate text/data address spaces, so these are equivalent
  - Remember to reset `errno` before, and check it afterwards
  - *word* means a 32/64-bit integer, depending on the OS variant
  - See the man page for other details and corresponding `*POKE*` requests

E.g.,

```
int read_lives(pid_t pid)
{
        errno = 0;
        int lives = (int)ptrace(PTRACE_PEEKDATA, pid, LIVES_ADDX, 0);
        if (errno) {
                perror("PTRACE_PEEKDATA");
                return -1;
        }
        return lives;
}
```

Reading/writing registers

PTRACE_GETREGS and PTRACE_GETFPREGS copy the tracee's general-purpose or floating-point registers, respectively, to the address data in the tracer; see <sys/user.h>

E.g.,

```
struct user_regs_struct regs;
if (ptrace(PTRACE_GETREGS, pid, 0, &regs)) {
        perror("ptrace PTRACE_GETREGS");
        return -1;
}
printf("RAX = 0x%llx\n", regs.rax);
```

→ c-examples/ptrace/pac_lives.c

Starting a debug sessions

You can either

- create a new process, by using `CreateProcess` and specifying the flag `DEBUG_PROCESS`
  - unless `_NO_DEBUG_HEAP` is set to 1, the runtime system uses a "debug-friendly" heap [HP07]
- attach to an existing one, by using `DebugActiveProcess`

Then, you typically enter in the debug-loop and call `WaitForDebugEvents`
https://docs.microsoft.com/en-us/windows/win32/debug/writing-the-debugger-s-main-loop

Debug event

When a debugging event occurs, the system:

- suspends all threads in the process being debugged and
- notifies the debugger of the event

```
typedef struct _DEBUG_EVENT {
  DWORD dwDebugEventCode, dwProcessId, dwThreadId;
  union {
    EXCEPTION_DEBUG_INFO      Exception;
    CREATE_THREAD_DEBUG_INFO  CreateThread;
    CREATE_PROCESS_DEBUG_INFO CreateProcessInfo;
    EXIT_THREAD_DEBUG_INFO    ExitThread;
    EXIT_PROCESS_DEBUG_INFO   ExitProcess;
    LOAD_DLL_DEBUG_INFO       LoadDll;
    UNLOAD_DLL_DEBUG_INFO     UnloadDll;
    OUTPUT_DEBUG_STRING_INFO  DebugString;
    RIP_INFO                  RipInfo;
  } u;
} DEBUG_EVENT, *LPDEBUG_EVENT;
```

Exceptions raised by the target process (exactly like Unix signals) are first delivered to the debugger...

Exception dispatching logic

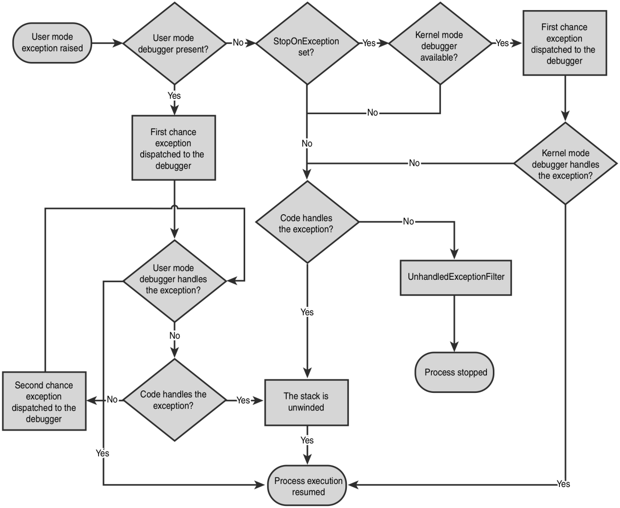

Figure 3.3 Exception dispatching logic

From [HP07]

Inspecting and controlling

The debugger can interact with the user, and read/write

- CPU registers, via `GetThreadContext`/`SetThreadContext`
- Target's memory, via `ReadProcessMemory`/`WriteProcessMemory`
    - when a debugger changes code in the target, it should also call `FlushInstructionCache`

After the debugger processes the event, `ContinueDebugEvent` allows the target process to resume its execution

Other APIs

- A debugger can detach using `DebugActiveProcessStop`
  - By default, the target is terminated, unless `DebugSetProcessKillOnExit` has been used
- A process can "set a breakpoint" by calling `DebugBreak`, which is simply an indirect way to execute `INT3`
  - `DebugBreakProcess` allows a process to raise an `EXCEPTION_BREAKPOINT` inside another process
- `OutputDebugString` allows processes to send debug output to a debugger
  - Or other applications (via `DBWIN_BUFFER` memory-mapped file, see `http://unixwiz.net/techtips/outputdebugstring.html`) like Sysinternals' DebugView `docs.microsoft.com/en-us/sysinternals/downloads/debugview`

Outline

Linux

The following are a refresher of what we saw, with some additional details

- How do debuggers (really) work - Pawel Moll
  `https://www.youtube.com/watch?v=xqrxg3hl10o`
- How C++ Debuggers work - Simon Brand - Meeting C++ 2017
  `https://www.youtube.com/watch?v=Q3Rm95Mk03c`
  See also his blog post series "Writing a Linux Debugger":
  `https://blog.tartanllama.xyz/writing-a-linux-debugger-setup/`
- Modern Linux C++ debugging tools - under the covers - Greg Law - CppCon 2019 `https://youtu.be/WoRmXjVxuFQ`
- More GDB wizardry and 8 other essential Linux application debugging tools - Greg Law - ACCU 2019
  `https://www.youtube.com/watch?v=Yq6g_kvyvPU`
- Cool New Stuff in GDB 9, 10 and 11 - Greg Law - ACCU 2022
  `https://youtu.be/KLXnNWYa5YA`

Windows

On MSDN:

- Creating a basic debugger
  `https://docs.microsoft.com/en-us/windows/win32/debug/creating-a-basic-debugger`

Books:

- Advanced Windows Debugging [HP07]
- Windows 10 System Programming, Part 2
  `https://leanpub.com/windows10systemprogrammingpart2`

Exception handling and stack unwinding

- C++ Exception Handling - The gory details of an implementation
  https://www.youtube.com/watch?v=XpRL7exdFL8
- C++ Exceptions and Stack Unwinding
  https://www.youtube.com/watch?v=_Ivd3qzgT7U
- Deep Wizardry: Stack Unwinding
  https://blog.reverberate.org/2013/05/deep-wizardry-stack-unwinding.html
- DWARF-based stack unwinding [BKZN19]
- Dwarf/ELF .eh_frame parsing for function identification
  https://bitlackeys.org/#eh_frame
  - For function identification, see also Nucleus (https://bitbucket.org/vusec/nucleus), based on [ASB17]

References

[ASB17] Dennis Andriesse, Asia Slowinska, and Herbert Bos.
Compiler-agnostic function detection in binaries.
In *Security and Privacy (EuroS&P), 2017 IEEE European Symposium on*, pages 177–189. IEEE, 2017.

[BKZN19] Théophile Bastian, Stephen Kell, and Francesco Zappa Nardelli.
Reliable and fast dwarf-based stack unwinding.
Proceedings of the ACM on Programming Languages, 3(OOPSLA):1–24, 2019.

[HP07] Mario Hewardt and Daniel Pravat.
Advanced Windows Debugging.
Pearson Education, 2007.