Debug Watchdog for Linux

martin.uy

Open by default.

Agenda

- Motivation
- Introduction
- Background
- Implementation
- Solution architecture
- Demo

Motivation

- Framework for testing
  - Multiple layers
  - Multiple Java Virtual Machines (JVMs)
    - 1 process per test
    - Short living tests
- Need of debugging the JVM that executes each test

Introduction

- How can we debug in Linux?
  - Attach to an executing process:
    - *gdb -p <PID>*
  - Launch an executable binary from the debugger:
    - *gdb /usr/bin/ls*

GDB
The GNU Project
Debugger

Introduction

- Linux debugging API: **ptrace**
  - PTRACE_ATTACH
    - Attach to an executing process
  - PTRACE_TRACEME
    - Launch an executable binary to be debugged from the first instruction

Introduction

- PTRACE_TRACEME
  - How is a process launched in Linux?
    - sys_fork
    - sys_execve
  - In between these syscalls, sys_ptrace(PTRACE_TRACEME) is executed
  - sys_ptrace immediately returns but when next calling sys_execve, the process stops and its parent becomes debugger

Introduction

How can these APIs be used in this case?

- Executable binary is known but, who launches it? when? with which parameters? how long is the process going to live?
- Should the script interpreter be attached and its forks followed? (gdb *set follow-fork-mode*)

Introduction

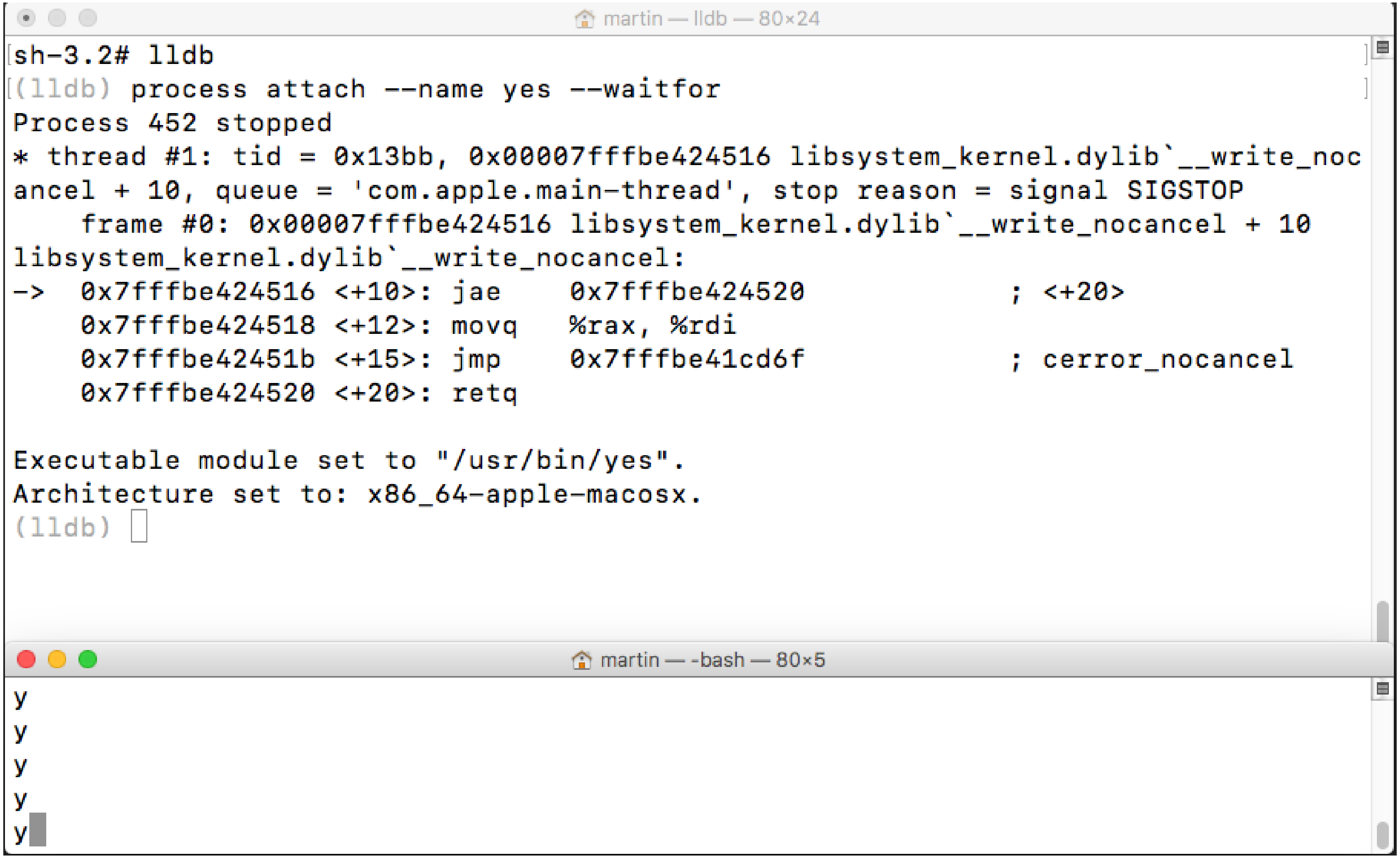
- Polling?

```
#!/bin/sh
progstr=$1
progpid=`pgrep -o $progstr`
while [ "$progpid" = "" ]; do
  progpid=`pgrep -o $progstr`
done
gdb -ex continue -p $progpid
```

https://stackoverflow.com/questions/4382348/is-there-any-way-to-tell-gdb-to-wait-for-a-process-to-start-and-attach-to-it

Introduction

- In macOS:

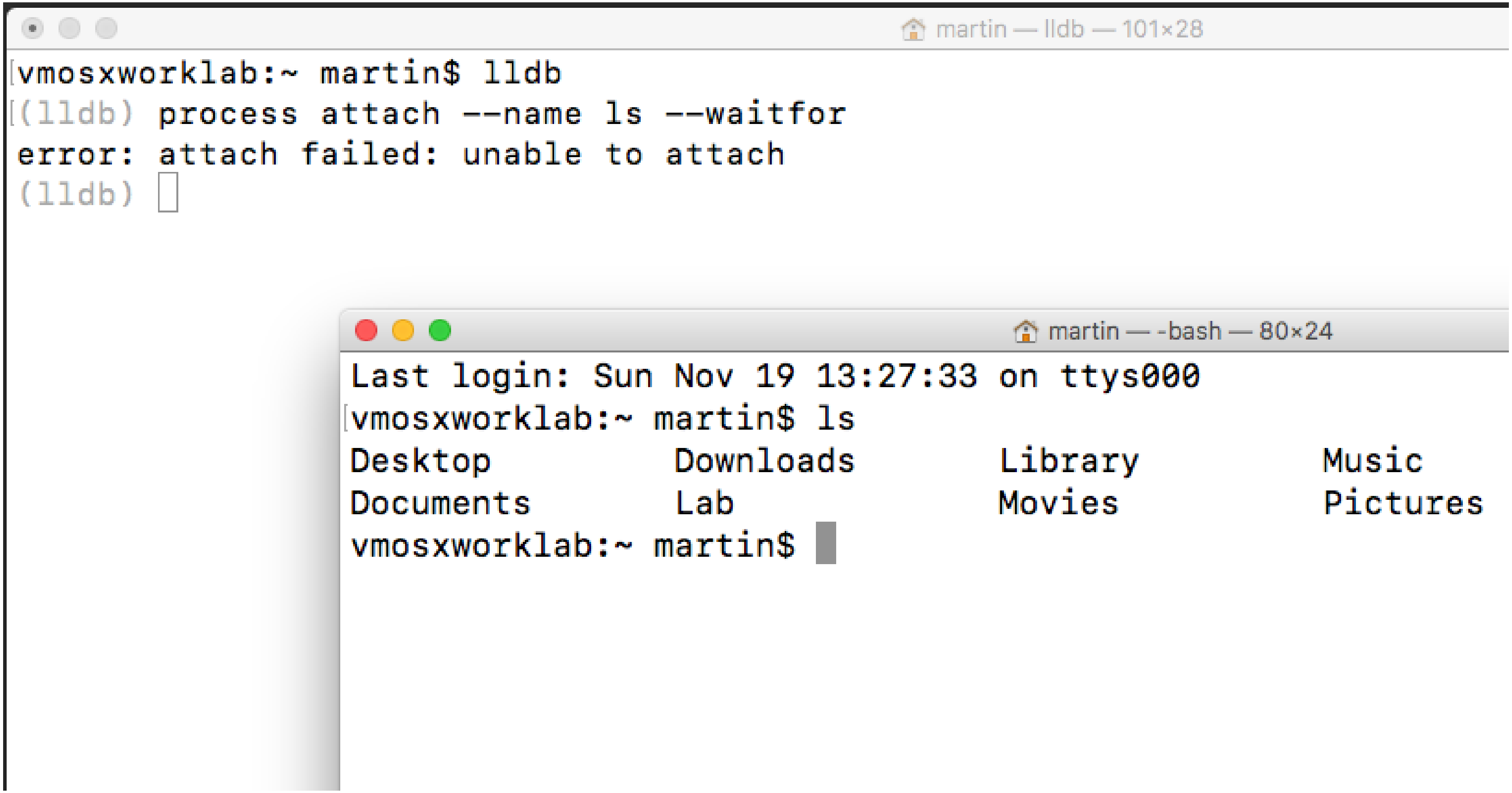


```
sh-3.2# lldb
(lldb) process attach --name yes --waitfor
Process 452 stopped
* thread #1: tid = 0x13bb, 0x00007fffbe424516 libsystem_kernel.dylib`__write_noc
ancel + 10, queue = 'com.apple.main-thread', stop reason = signal SIGSTOP
    frame #0: 0x00007fffbe424516 libsystem_kernel.dylib`__write_nocancel + 10
libsystem_kernel.dylib`__write_nocancel:
->  0x7fffbe424516 <+10>: jae    0x7fffbe424520            ; <+20>
    0x7fffbe424518 <+12>: movq   %rax, %rdi
    0x7fffbe42451b <+15>: jmp    0x7fffbe41cd6f            ; cerror_nocancel
    0x7fffbe424520 <+20>: retq

Executable module set to "/usr/bin/yes".
Architecture set to: x86_64-apple-macosx.
(lldb) 
```

```
y
y
y
y
y
```

Introduction

- In macOS:

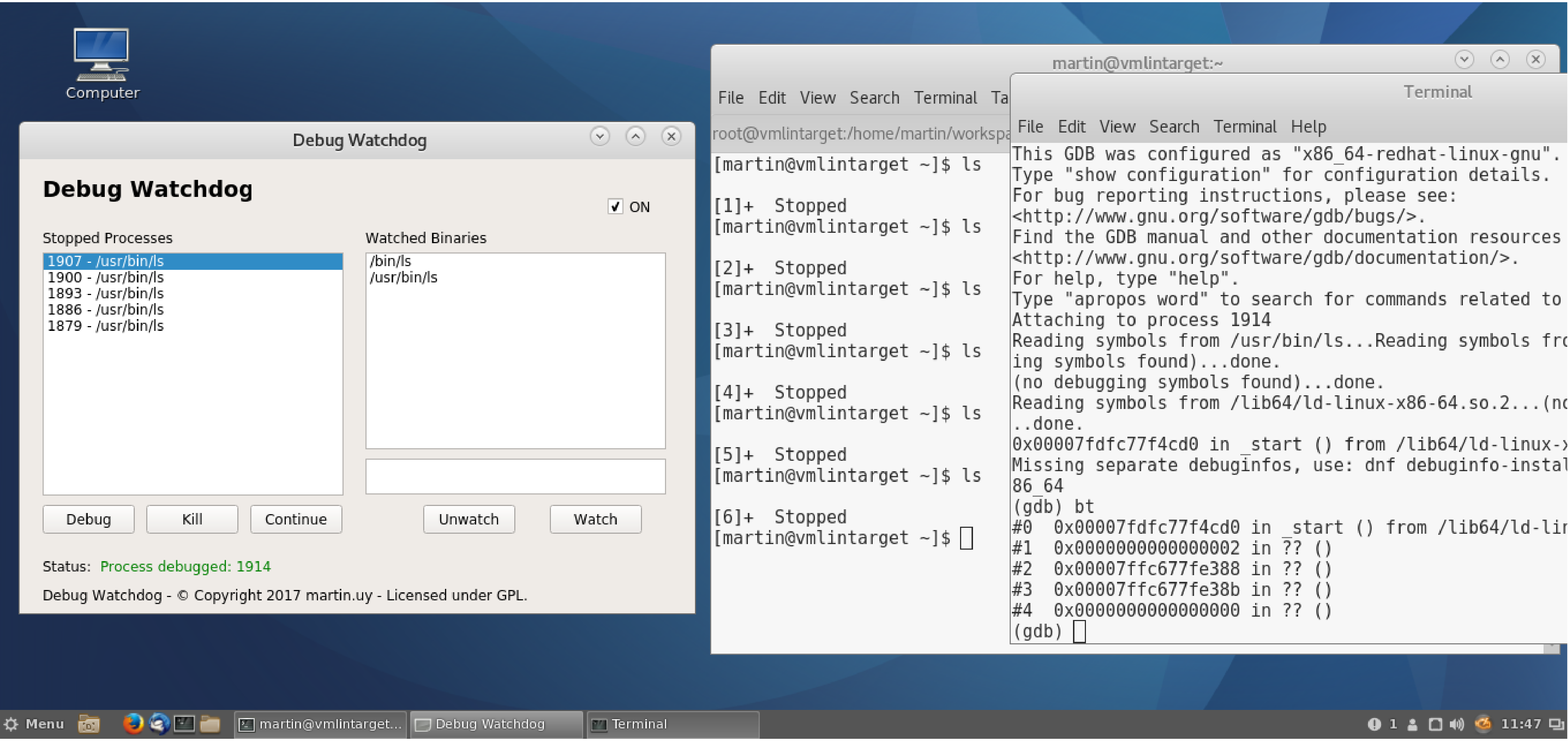


```
vmosxworklab:~ martin$ lldb
(lldb) process attach --name ls --waitfor
error: attach failed: unable to attach
(lldb) 
```

```
Last login: Sun Nov 19 13:27:33 on ttys000
vmosxworklab:~ martin$ ls
Desktop         Downloads       Library         Music
Documents       Lab             Movies          Pictures
vmosxworklab:~ martin$ 
```

Introduction

- Debug Watchdog for Linux (v1.0)



Debug Watchdog

☑ ON

Stopped Processes

1907 - /usr/bin/ls
1900 - /usr/bin/ls
1893 - /usr/bin/ls
1886 - /usr/bin/ls
1879 - /usr/bin/ls

Watched Binaries

/bin/ls
/usr/bin/ls

Status: Process debugged: 1914

Debug Watchdog - © Copyright 2017 martin.uy - Licensed under GPL.

```
root@vmlintarget:/home/martin/workspa
[martin@vmlintarget ~]$ ls

[1]+  Stopped
[martin@vmlintarget ~]$ ls

[2]+  Stopped
[martin@vmlintarget ~]$ ls

[3]+  Stopped
[martin@vmlintarget ~]$ ls

[4]+  Stopped
[martin@vmlintarget ~]$ ls

[5]+  Stopped
[martin@vmlintarget ~]$ ls

[6]+  Stopped
[martin@vmlintarget ~]$ 
```

```
This GDB was configured as "x86_64-redhat-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to
Attaching to process 1914
Reading symbols from /usr/bin/ls...Reading symbols fro
ing symbols found)...done.
(no debugging symbols found)...done.
Reading symbols from /lib64/ld-linux-x86-64.so.2...(no
..done.
0x00007fdfc77f4cd0 in _start () from /lib64/ld-linux-x
Missing separate debuginfos, use: dnf debuginfo-instal
86_64
(gdb) bt
#0  0x00007fdfc77f4cd0 in _start () from /lib64/ld-lin
#1  0x0000000000000002 in ?? ()
#2  0x00007ffc677fe388 in ?? ()
#3  0x00007ffc677fe38b in ?? ()
#4  0x0000000000000000 in ?? ()
(gdb) 
```

Introduction

- Debug Watchdog for Linux (v1.0)
  - Linux x86_64
    - Tested in Fedora
  - GPL license
    - Contributions welcomed :-)
  - GitHub
    - https://github.com/martinuy/debugwatchdog

Background

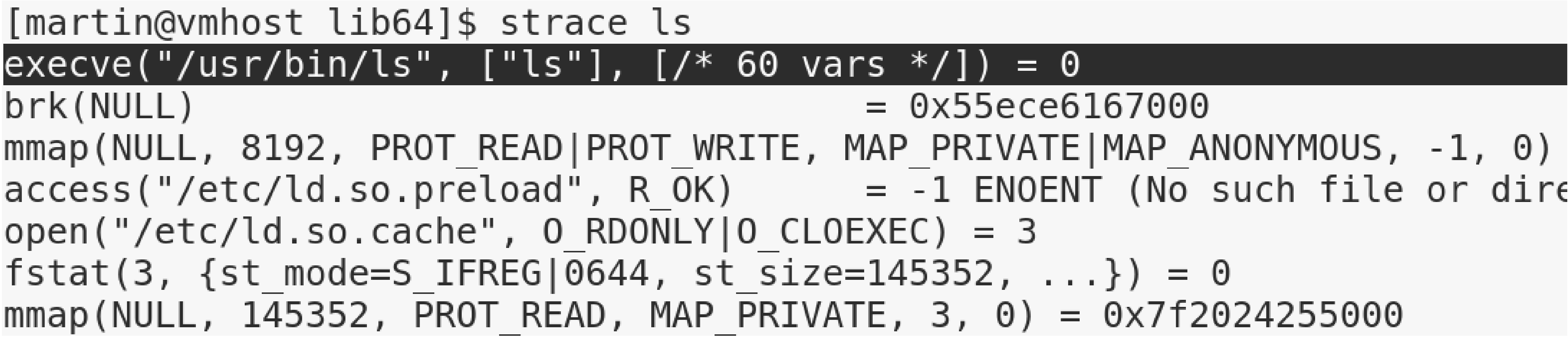
- How to detect when a process is launched?
    - Graphical User Interface?
    - Daemon?
    - Command-line?
    - Script?
    - Libc?

Background

- Hook *execve* in libc, but:
  - Not every executable binary would be caught
    - I.e. libc statically linked, libc in containers, process launched without libc, etc.
  - Libc in file system has to be overwritten, instead of patching in run time only
    - Binary rewriting
    - Undo changes
  - A “global” LD_PRELOAD would be desired

Background

- How to detect when a process is launched?



```
[martin@vmhost lib64]$ strace ls
execve("/usr/bin/ls", ["ls"], [/* 60 vars */]) = 0
brk(NULL)                               = 0x55ece6167000
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or dire
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=145352, ...}) = 0
mmap(NULL, 145352, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7f2024255000
```

- There are multiple launchers but only one syscall: *sys_execve* (kernel)

Hook *sys_execve*!

Background

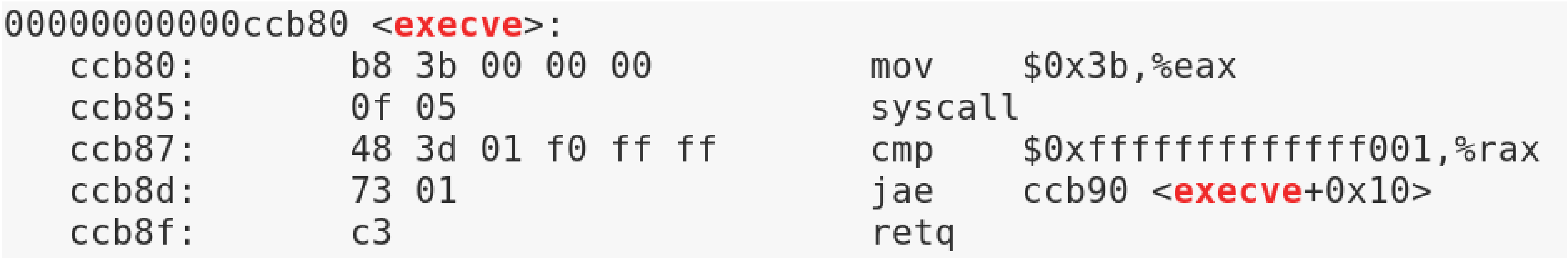
- What is a syscall?
  - Call to a kernel service, through a special architecture instruction
  - Processor executes the service in privileged mode
  - Thread that performs the syscall is transformed, temporarily, into a kernel thread
    - Each thread has 2 stacks: one in user and other in kernel

Background

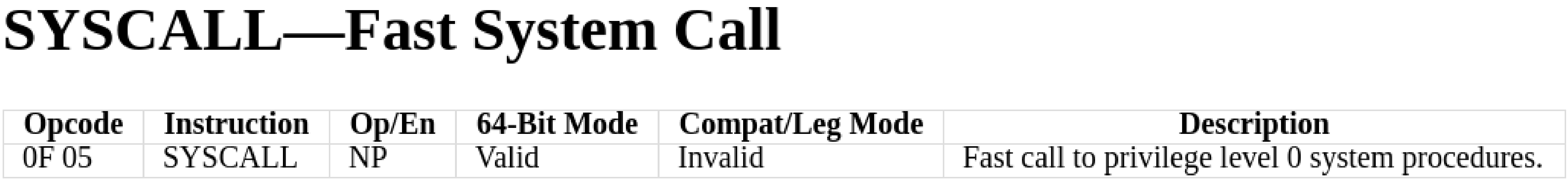
- What is a syscall?
  - Applications API in Linux is *libc*: syscalls are not executed directly
  - It can be done if:
    - binary interface (ABI) for the architecture is followed; or,
    - through *syscall* (*libc*) function

Background

- Syscall from user point of view (*libc*)



```
0000000000ccb80 <execve>:
   ccb80:       b8 3b 00 00 00          mov    $0x3b,%eax
   ccb85:       0f 05                   syscall
   ccb87:       48 3d 01 f0 ff ff       cmp    $0xfffffffffffff001,%rax
   ccb8d:       73 01                   jae    ccb90 <execve+0x10>
   ccb8f:       c3                      retq
```



SYSCALL—Fast System Call

Opcode	Instruction	Op/En	64-Bit Mode	Compat/Leg Mode	Description
0F 05	SYSCALL	NP	Valid	Invalid	Fast call to privilege level 0 system procedures.

* http://www.felixcloutier.com/x86/SYSCALL.html

Background

- SYSCALL instruction (x86_64)
  - Processor switches to privileged mode
  - RIP (user) → RCX
  - IA32_LSTAR MSR (syscalls entry point address in kernel: entry_SYSCALL_64) → RIP
  - RFLAGS → R11
  - RSP is not saved: saving is user or kernel responsibility
  - Etc.

Background

- Syscall from kernel point of view: entry_64.S

```
…
SYSCALL does not save anything on the stack
 * and does not change rsp.
 *
 * Registers on entry:
 * rax  system call number
 * rcx  return address
 * r11  saved rflags (note: r11 is callee-clobbered register
in C ABI)
 * rdi  arg0
 * rsi  arg1
 * rdx  arg2
 * r10  arg3 (needs to be moved to rcx to conform to C ABI)
 * r8   arg4
 * r9   arg5
 * (note: r12-r15, rbp, rbx are callee-preserved in C ABI)
… */
```

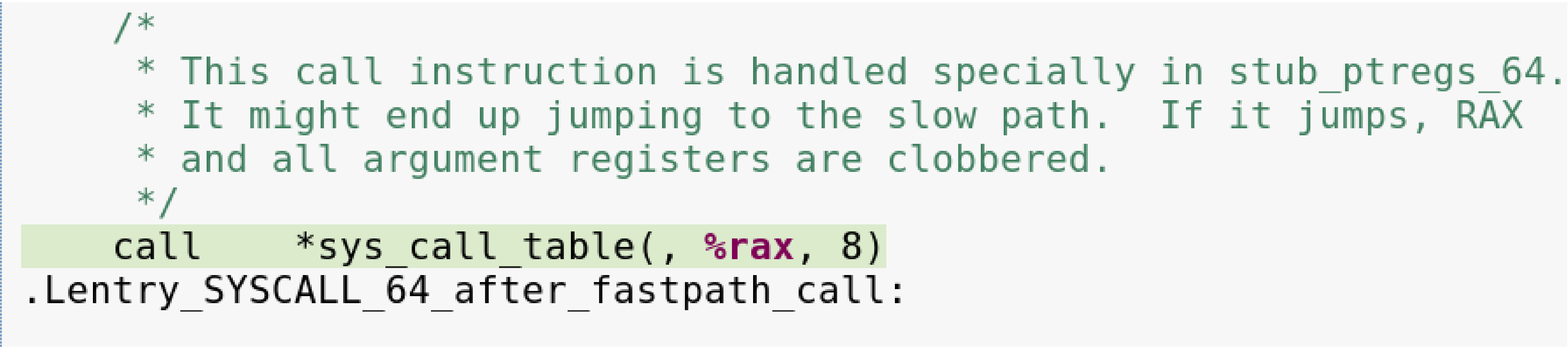
Background

- Syscall from kernel point of view: entry_64.S

```
ENTRY(entry_SYSCALL_64)
…
/* Construct struct pt_regs on stack */
    pushq   $__USER_DS          /* pt_regs->ss */
    pushq   PER_CPU_VAR(rsp_scratch) /* pt_regs->sp */
    pushq   %r11           /* pt_regs->flags */
    pushq   $__USER_CS          /* pt_regs->cs */
    pushq   %rcx           /* pt_regs->ip */
    pushq   %rax           /* pt_regs->orig_ax */
    pushq   %rdi           /* pt_regs->di */
    pushq   %rsi           /* pt_regs->si */
    pushq   %rdx           /* pt_regs->dx */
...
```

Background

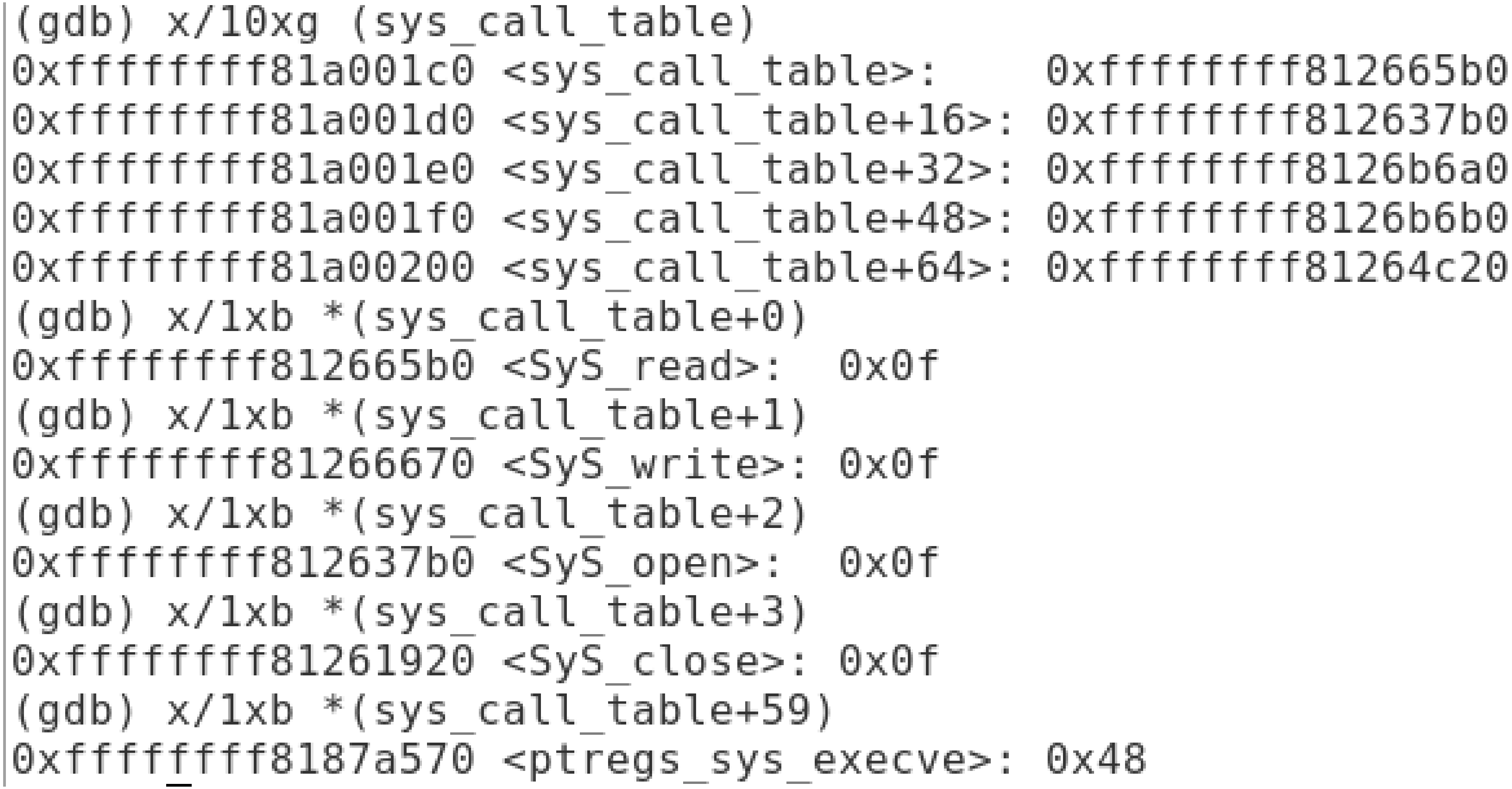
- Syscall from kernel point of view: entry_64.S



```
	/*
	 * This call instruction is handled specially in stub_ptregs_64.
	 * It might end up jumping to the slow path.  If it jumps, RAX
	 * and all argument registers are clobbered.
	 */
	call	*sys_call_table(, %rax, 8)
.Lentry_SYSCALL_64_after_fastpath_call:
```

Background

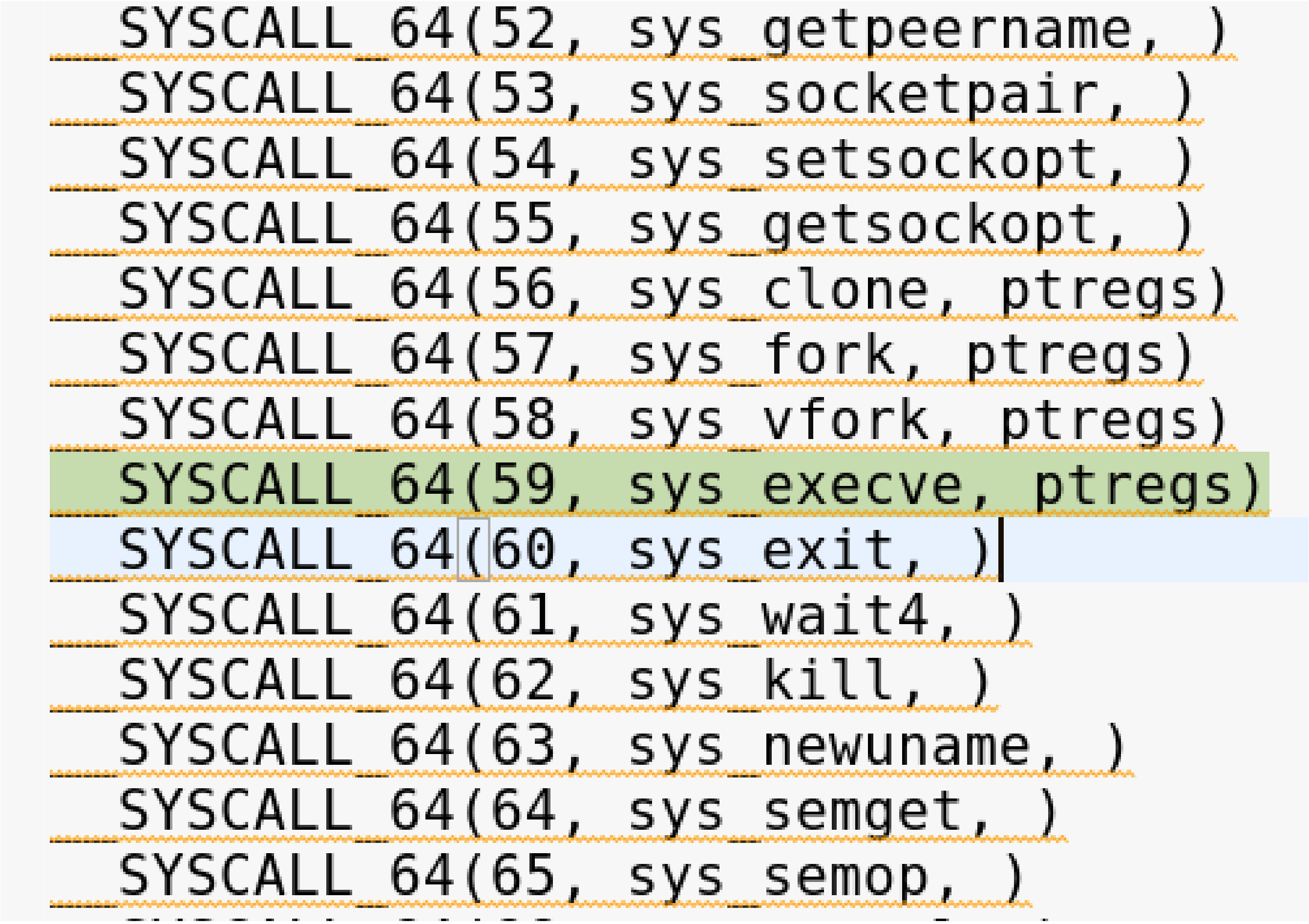
- Syscalls table



```
(gdb) x/10xg (sys_call_table)
0xffffffff81a001c0 <sys_call_table>:    0xffffffff812665b0
0xffffffff81a001d0 <sys_call_table+16>: 0xffffffff812637b0
0xffffffff81a001e0 <sys_call_table+32>: 0xffffffff8126b6a0
0xffffffff81a001f0 <sys_call_table+48>: 0xffffffff8126b6b0
0xffffffff81a00200 <sys_call_table+64>: 0xffffffff81264c20
(gdb) x/1xb *(sys_call_table+0)
0xffffffff812665b0 <SyS_read>:  0x0f
(gdb) x/1xb *(sys_call_table+1)
0xffffffff81266670 <SyS_write>: 0x0f
(gdb) x/1xb *(sys_call_table+2)
0xffffffff812637b0 <SyS_open>:  0x0f
(gdb) x/1xb *(sys_call_table+3)
0xffffffff81261920 <SyS_close>: 0x0f
(gdb) x/1xb *(sys_call_table+59)
0xffffffff8187a570 <ptregs_sys_execve>: 0x48
```

Background

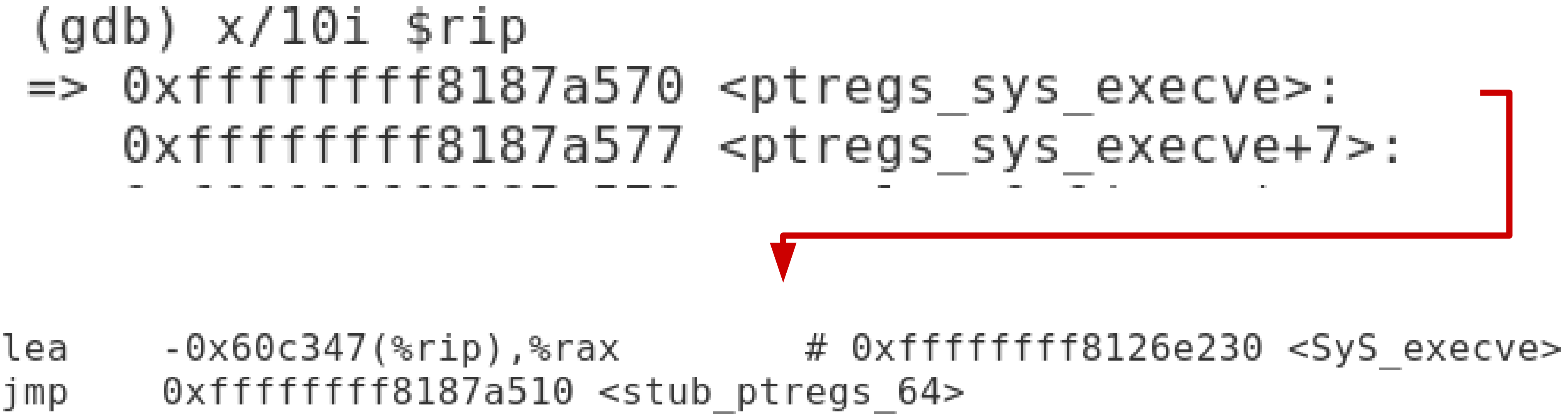
- Syscall from kernel point of view: syscalls_64.h



```
__SYSCALL_64(52, sys_getpeername, )
__SYSCALL_64(53, sys_socketpair, )
__SYSCALL_64(54, sys_setsockopt, )
__SYSCALL_64(55, sys_getsockopt, )
__SYSCALL_64(56, sys_clone, ptregs)
__SYSCALL_64(57, sys_fork, ptregs)
__SYSCALL_64(58, sys_vfork, ptregs)
__SYSCALL_64(59, sys_execve, ptregs)
__SYSCALL_64(60, sys_exit, )
__SYSCALL_64(61, sys_wait4, )
__SYSCALL_64(62, sys_kill, )
__SYSCALL_64(63, sys_newuname, )
__SYSCALL_64(64, sys_semget, )
__SYSCALL_64(65, sys_semop, )
```

Background

- Some syscalls in the table point directly to implementation and others to a previous *stub*:



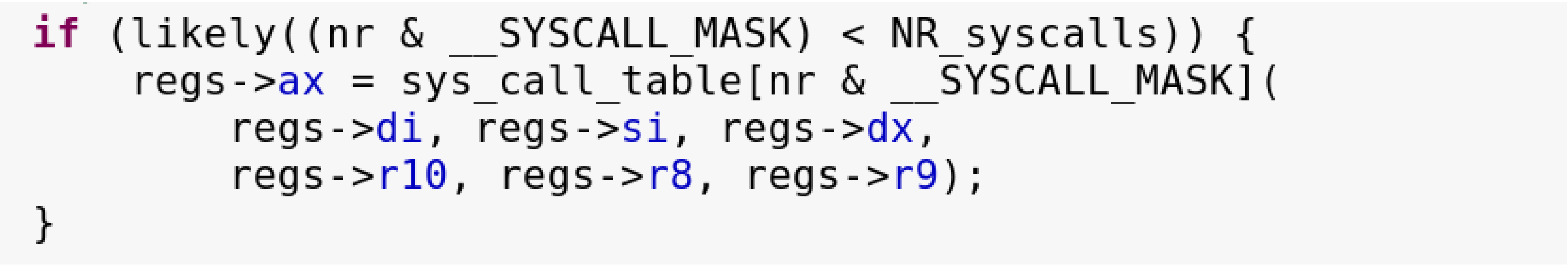
```
(gdb) x/10i $rip
=> 0xffffffff8187a570 <ptregs_sys_execve>:
   0xffffffff8187a577 <ptregs_sys_execve+7>:
```

```
lea    -0x60c347(%rip),%rax        # 0xffffffff8126e230 <SyS_execve>
jmp    0xffffffff8187a510 <stub_ptregs_64>
```

- stub_ptregs_64
  - jump to “slow path” first (entry_SYSCALL64_slow_path)

Background

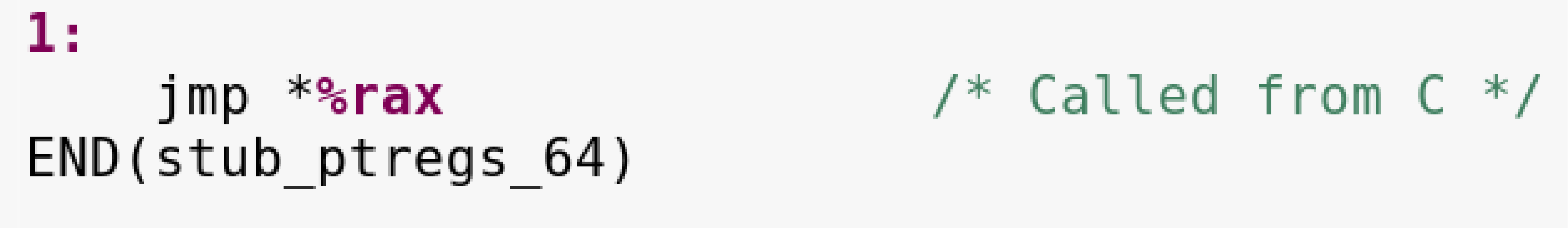
- entry_SYSCALL64_slow_path
  - Save extra registers (rbx, rbp, r12-r15) in the pt_regs structure previously pushed to the stack
  - Call do_syscall_64, with pt_regs structure as parameter
- do_syscall_64 (struct pt_regs *regs):



```
if (likely((nr & __SYSCALL_MASK) < NR_syscalls)) {
    regs->ax = sys_call_table[nr & __SYSCALL_MASK](
        regs->di, regs->si, regs->dx,
        regs->r10, regs->r8, regs->r9);
}
```

Background

- do_syscall_64
  - Even though ptregs_sys_execve and stub_ptregs_64 are called again, stub_ptregs_64 flow goes straight to the syscall this time:



```
1:
	jmp	*%rax				/* Called from C */
END(stub_ptregs_64)
```

Background

- Why is this done?
  - C-ABI requires some registers to be saved by the callee (rbx, rbp, r12-r15)
  - However, kernel does not do it -for performance- unless the syscall explicitly requires it
  - pt_regs structure (previously saved in the stack) allows original registers value to be restored

Implementation

- How to hook sys_execve?
  - Patch
    - syscalls table
    - sys_execve implementation
  - Jump to a trampoline (in a kernel module previously loaded) before sys_execve returns
  - What would be less invasive?

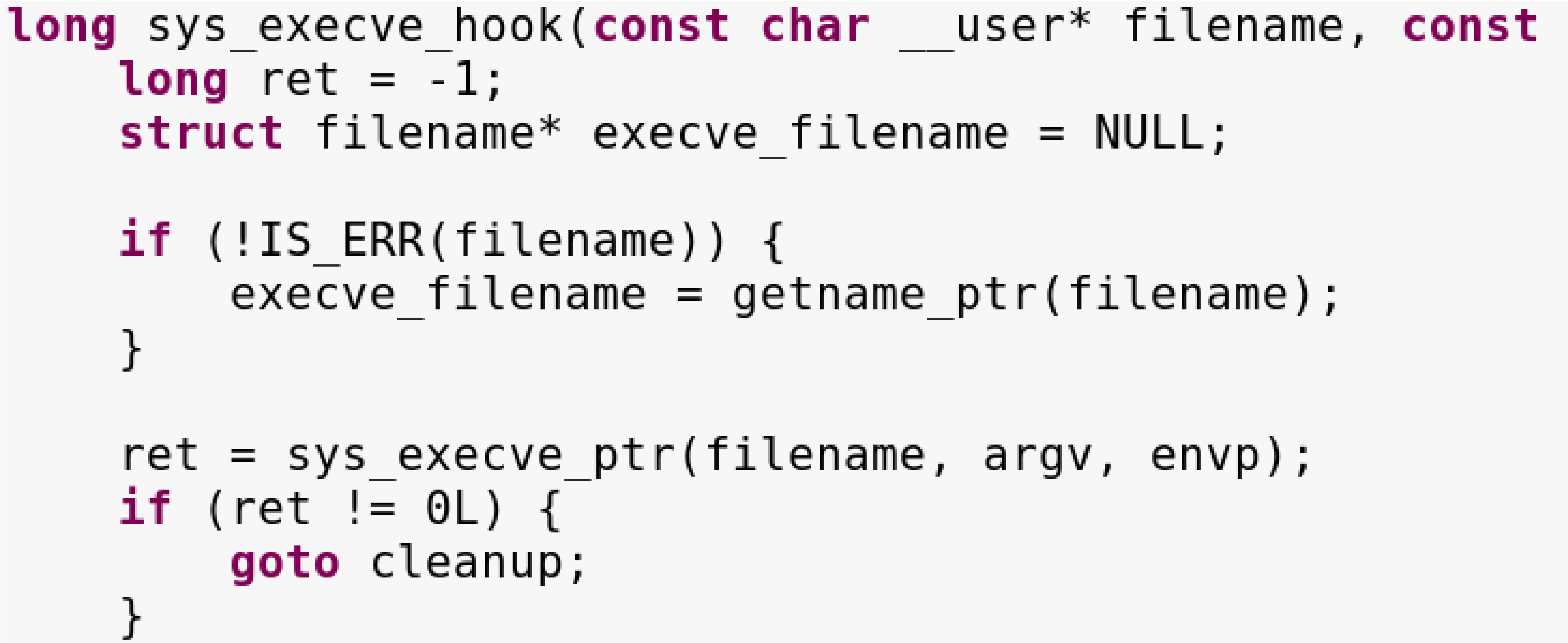
  Minimize patches out of the kernel Module; lower risk

Implementation

- If syscalls table were patched, it’s not possible to jump directly to sys_execve:
  - What would happen with previous stubs and pt_regs structure?
- Thus, hook for syscalls table:

```
  .text
  .align 8
  .globl sys_execve_stub_ptregs_64_hook
  .type sys_execve_stub_ptregs_64_hook, @function
sys_execve_stub_ptregs_64_hook:
  movq sys_execve_hook_ptr, %rax
  jmp    *stub_ptregs_64_ptr
```

Implementation



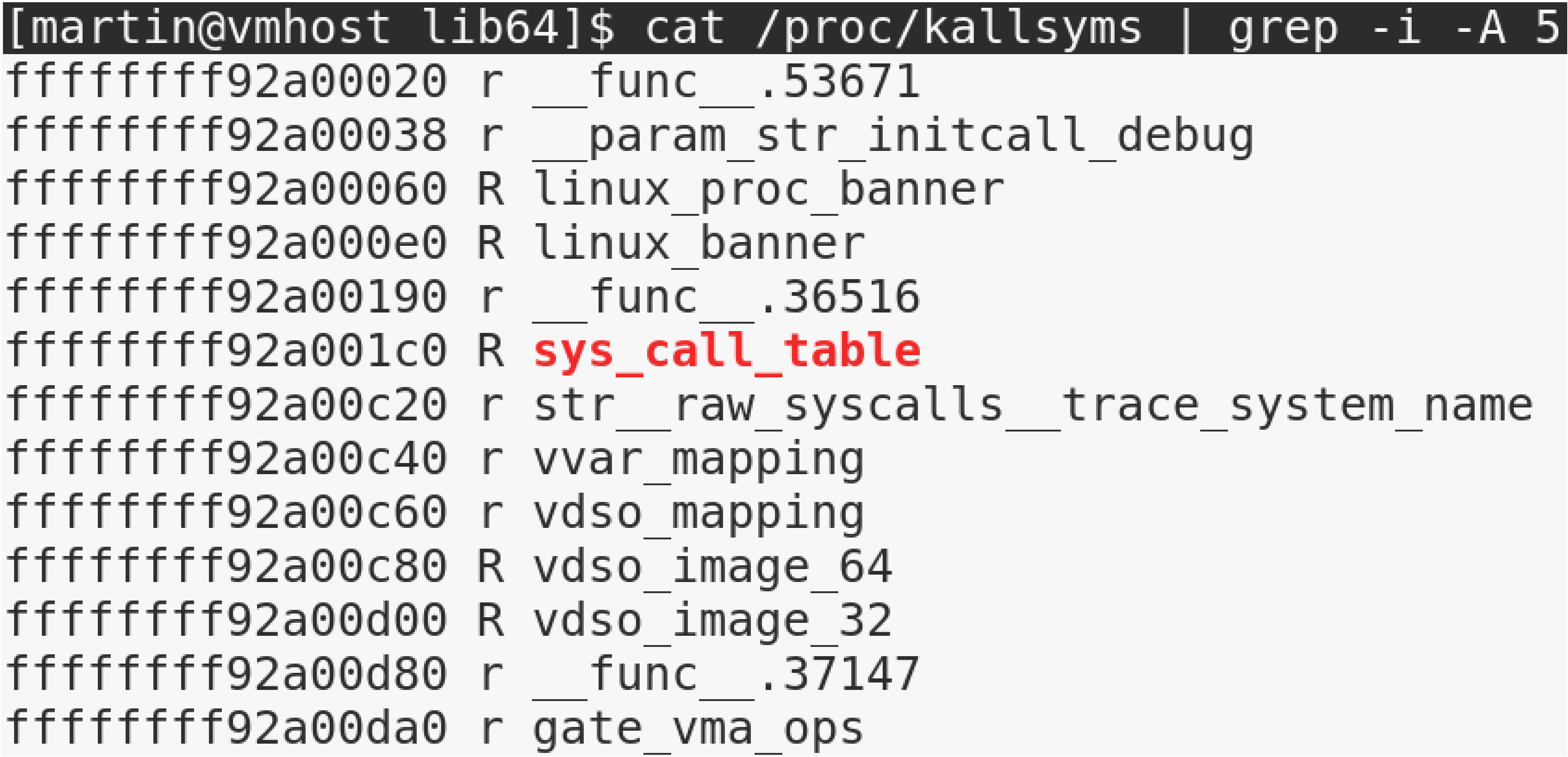
```
long sys_execve_hook(const char __user* filename, const
    long ret = -1;
    struct filename* execve_filename = NULL;

    if (!IS_ERR(filename)) {
        execve_filename = getname_ptr(filename);
    }

    ret = sys_execve_ptr(filename, argv, envp);
    if (ret != 0L) {
        goto cleanup;
    }
```

- Implementation in the Module; only syscalls table is modified out of the Module
- Add code before or after calling sys_execve

Implementation

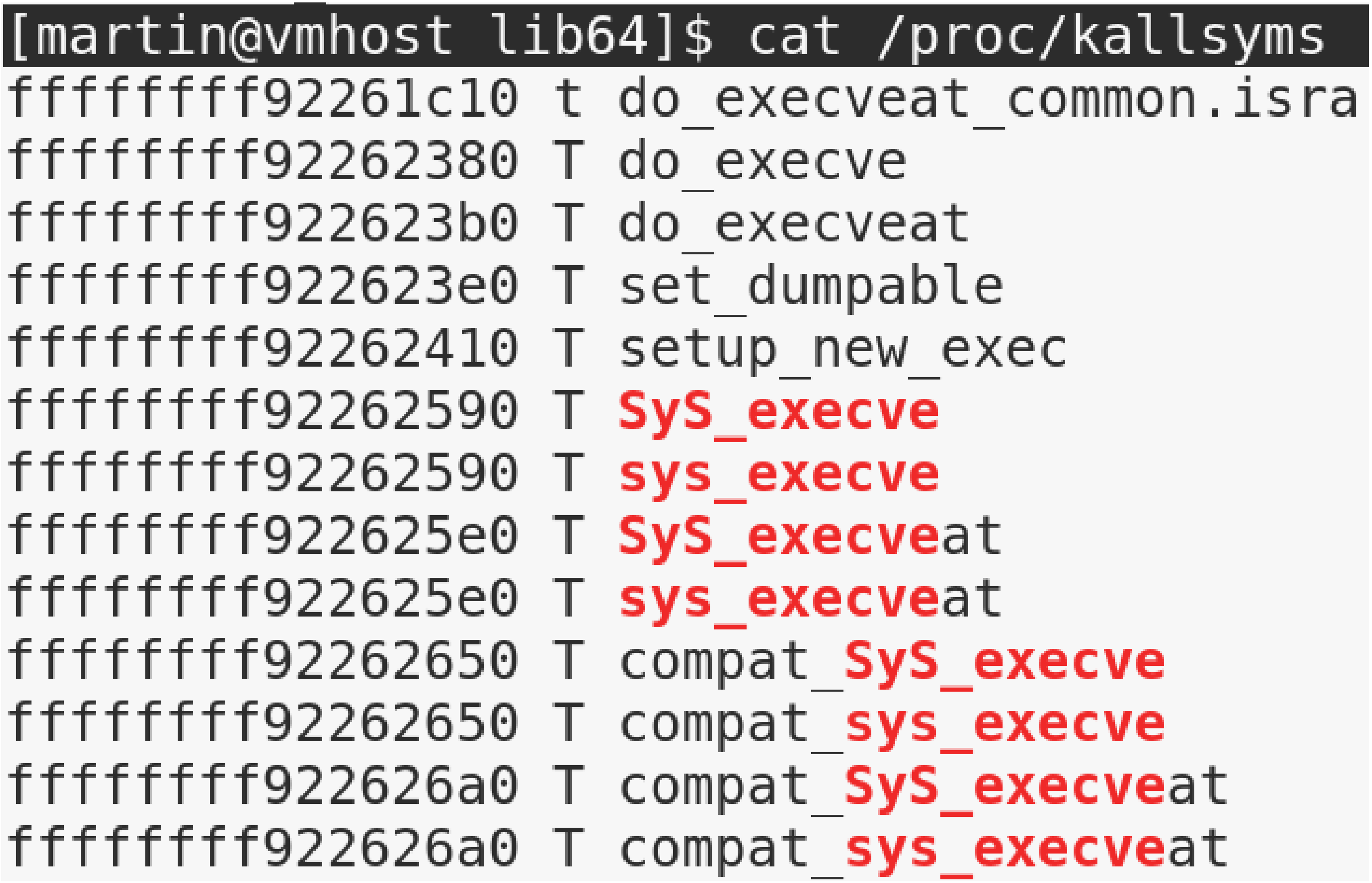
- Resolve symbols: where is sys_execve_stub_ptregs_64_hook located? where is original sys_execve located?
    - Virtual addresses are randomized in each boot (KASLR)
    - kallsyms (/proc/kallsysms and kernel API)
    - kallsyms_lookup_name("sys_execve_stub_ptregs_64_hook")

Implementation



```
[martin@vmhost lib64]$ cat /proc/kallsyms | grep -i -A 5
ffffffff92a00020 r __func__.53671
ffffffff92a00038 r __param_str_initcall_debug
ffffffff92a00060 R linux_proc_banner
ffffffff92a000e0 R linux_banner
ffffffff92a00190 r __func__.36516
ffffffff92a001c0 R sys_call_table
ffffffff92a00c20 r str__raw_syscalls__trace_system_name
ffffffff92a00c40 r vvar_mapping
ffffffff92a00c60 r vdso_mapping
ffffffff92a00c80 R vdso_image_64
ffffffff92a00d00 R vdso_image_32
ffffffff92a00d80 r __func__.37147
ffffffff92a00da0 r gate_vma_ops
```

Implementation



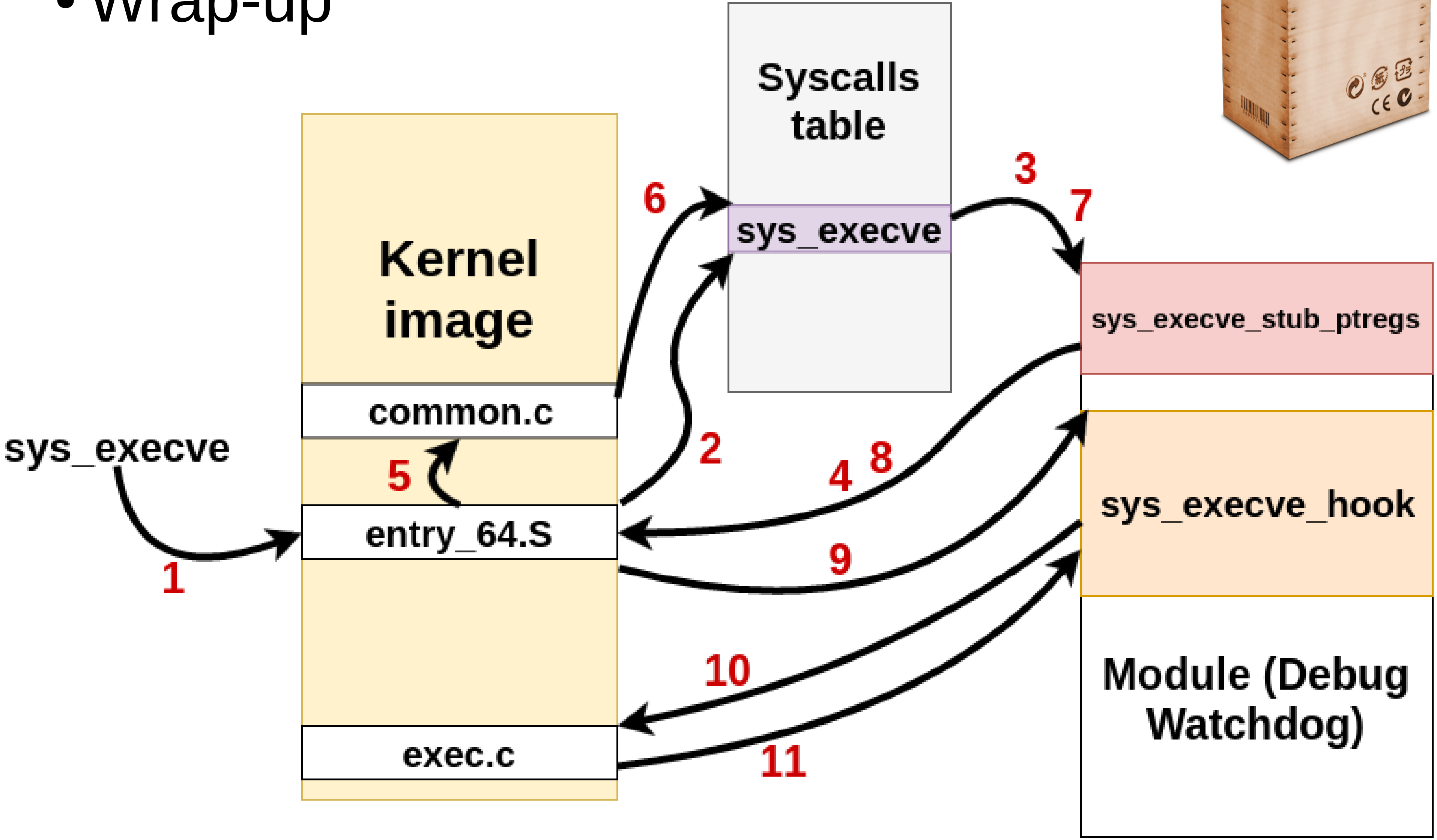
```
[martin@vmhost lib64]$ cat /proc/kallsyms
ffffffff92261c10 t do_execveat_common.isra
ffffffff92262380 T do_execve
ffffffff922623b0 T do_execveat
ffffffff922623e0 T set_dumpable
ffffffff92262410 T setup_new_exec
ffffffff92262590 T SyS_execve
ffffffff92262590 T sys_execve
ffffffff922625e0 T SyS_execveat
ffffffff922625e0 T sys_execveat
ffffffff92262650 T compat_SyS_execve
ffffffff92262650 T compat_sys_execve
ffffffff922626a0 T compat_SyS_execveat
ffffffff922626a0 T compat_sys_execveat
```

Implementation

- Other challenges
  - Write a memory address with read-only protection
    - Enable and disable writing read-only memory through *cr0* register:
    - write_cr0 (read_cr0 () & (~ 0x10000))
    - write_cr0 (read_cr0 () | 0x10000)

Implementation

- Wrap-up



Implementation

- Wrap-up
  - Syscalls table patched
  - A stub in a previously loaded Module is called (sys_execve_stub_ptregs_64_hook)
  - Control returns to “normal flow” but with RAX register pointing to sys_execve_hook function (also located in the Module)

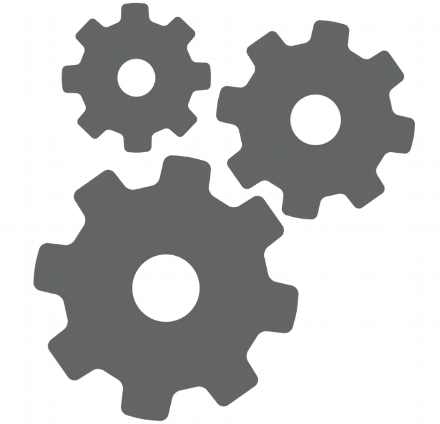
Implementation

- Wrap-up
  - “Normal flow” calls sys_execve_hook function (with original sys_execve parameters)
  - Original sys_execve is called (forwarding parameters)
  - Process is debugged (if it were the required executable binary)
  - Returns normally (“normal flow” exit stubs)

Implementation

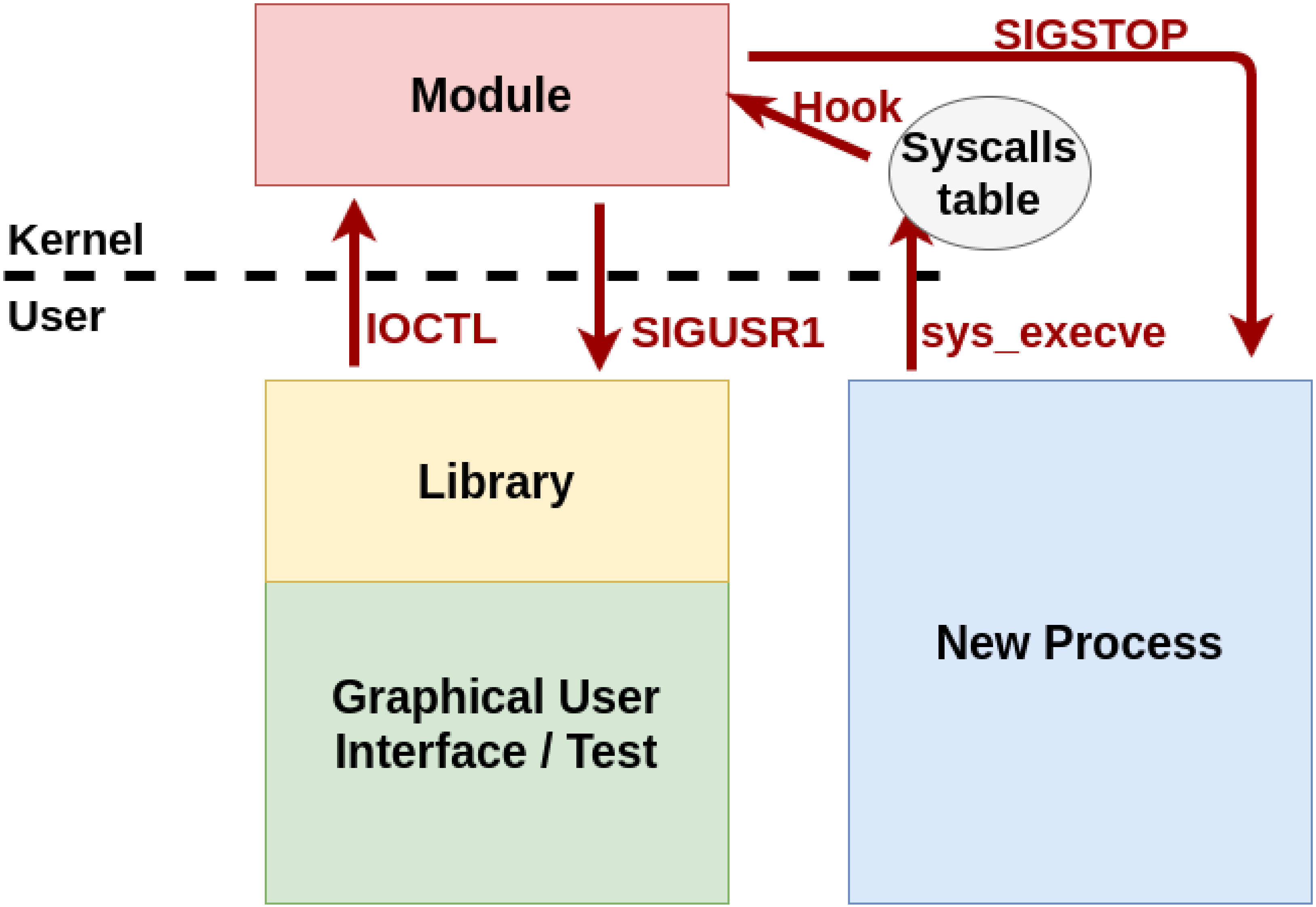
- How to debug a process?
  - PTRACE_TRACEME?
    - Works but the debugger will be the parent process: gdb cannot attach because there cannot be more than one debugger at the same time
  - Stop the process sending SIGSTOP from kernel
    - process does not execute any instruction
    - gdb can then attached to the stopped process

Architecture



- Project components:
  - Module (C, kernel)
  - Library (C)
  - Test (C)
  - UI (Qt/C++)

Architecture



Module

Hook

Syscalls table

SIGSTOP

Kernel

User

IOCTL

SIGUSR1

sys_execve

Library

Graphical User Interface / Test

New Process

Architecture

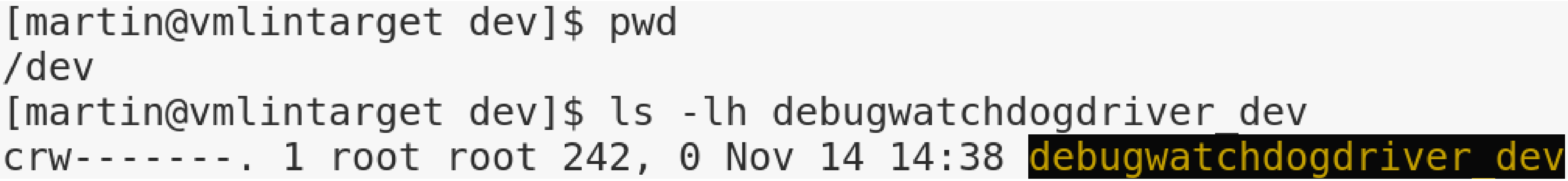
- Module
  - Dynamically loaded
    - CAP_SYS_MODULE capability is required
  - Only one instance, only one user process can communicate
    - Owner task id
    - If this process dies, another one can take ownership

Architecture

- Module
  - Multiple threads executing:
    - sys_execve_hook (any task)
    - IOCTLs (owner task or other task?)
    - Synchronization locking (mutex_lock/unlock)

Architecture

- Module
  - Library – Module communication
    - Device character
    - IOCTLs
      - Initialize / Finalize
      - Watch / Unwatch
      - Obtain a list of stopped processes



```
[martin@vmlintarget dev]$ pwd
/dev
[martin@vmlintarget dev]$ ls -lh debugwatchdogdriver_dev
crw-------. 1 root root 242, 0 Nov 14 14:38 debugwatchdogdriver_dev
```

Architecture

- Module
  - Library – Module communication
    - SIGUSR1
      - Notify the Library that there is at least one newly stopped process

Architecture

- Module
  - How to unload the Module in a safe way?
    - Restore original sys_execve entry in syscalls table
    - Unload the Module
    - But, what happens if a thread reads the syscalls table just before restoration and jumps to execute in now unmapped memory?

Architecture

- Library
  - Initialize
    - Register a callback for stopped processes notification
    - Load Module
  - Finalize
    - Unload Module
  - Watch / Unwatch executable binaries
  - Register a callback for error handling
  - Multi-threading

Architecture

- Library
  - Requirement: disable SIGUSR1 handling in every process thread
  - Stopped processes notification thread
    - *sigwaitinfo* to receive SIGUSR1 signals sent from the Module
      - no asynchronous signals handling
    - Calls previously registered callback

Demo

Q & A

Thanks!

http://martin.uy/blog/debug-watchdog-for-linux-v1-0/