Debug Watchdog for Linux

martin.uy

Open by default.

Agenda

- Motivación
- Introducción
- Background
- Implementación
- Arquitectura de la solución
- Demo

Motivación

- Framework de testing
  - Múltiples capas
  - Múltiples Java Virtual Machines (JVMs)
    - 1 proceso por cada test
    - Tests de corta duración
- Necesidad de debuggear la JVM que ejecuta cada test

Introducción

- ¿Cómo se debuggea en Linux?
  - Attacharse a un proceso en ejecución:
    - *gdb -p <PID>*
  - Lanzar un binario ejecutable desde el debugger:
    - *gdb /usr/bin/ls*

Introducción

- API de debugging en Linux: **ptrace**
  - PTRACE_ATTACH
    - Attacharse a un proceso en ejecución
  - PTRACE_TRACEME
    - Lanzar un binario ejecutable para ser debuggeado desde la primera instrucción

Introducción

- PTRACE_TRACEME
  - ¿Cómo se lanza un proceso en Linux?
    - sys_fork
    - sys_execve
  - Entre medio de esas syscalls se ejecuta sys_ptrace(PTRACE_TRACEME)
  - sys_ptrace retorna inmediatamente pero en la próxima llamada a sys_execve, el proceso se detiene y su padre pasa a ser el debugger

Introducción

¿Cómo aplicar estas APIs en este caso?

- Se conoce el binario ejecutable pero, ¿quién lo lanza? ¿cuándo? ¿con qué parámetros? ¿cuánto tiempo vive el proceso?
- ¿Attacharse al intérprete de un script y seguir sus forks? (gdb *set follow-fork-mode*)

Introducción

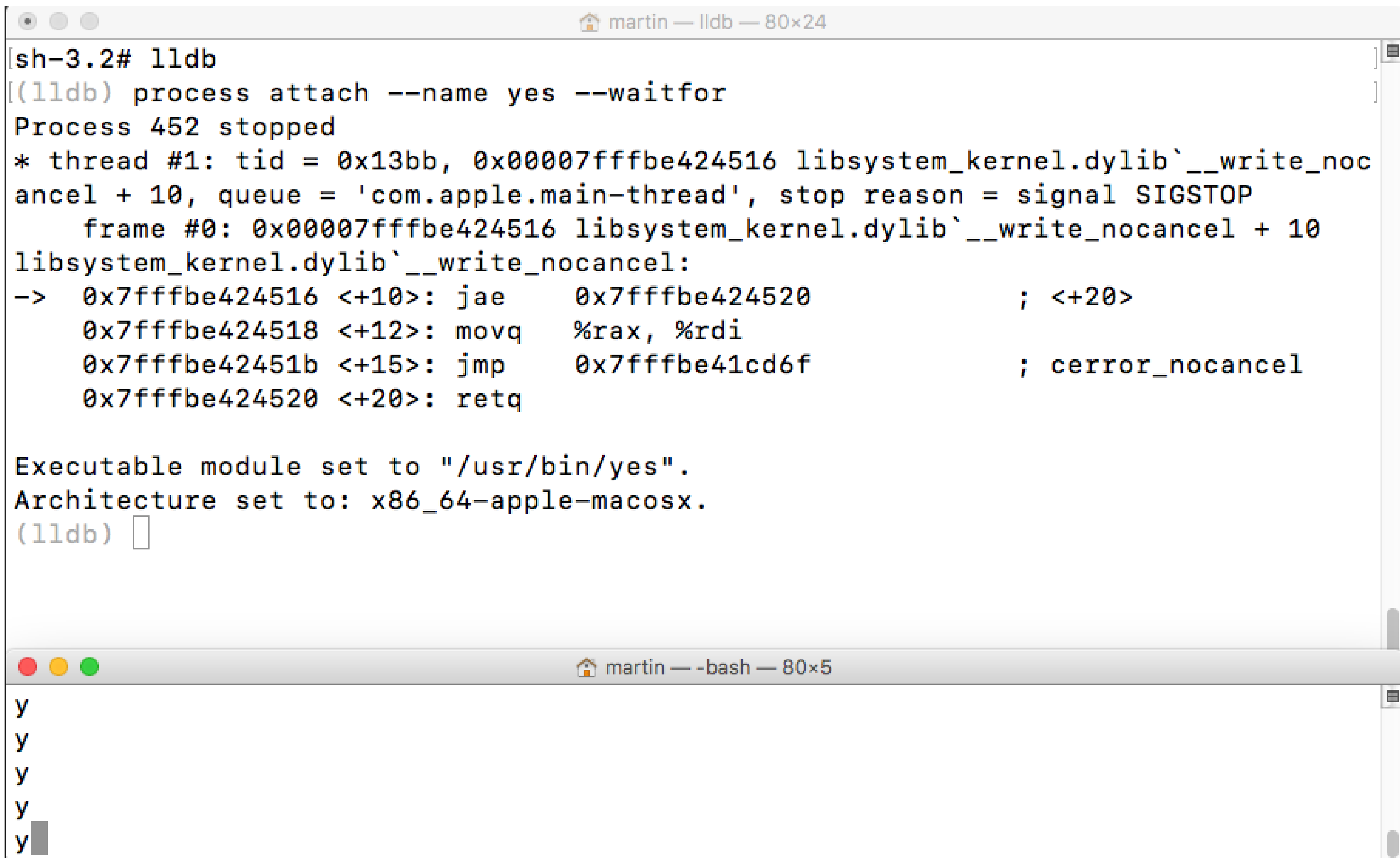
- ¿Polling?

```
#!/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

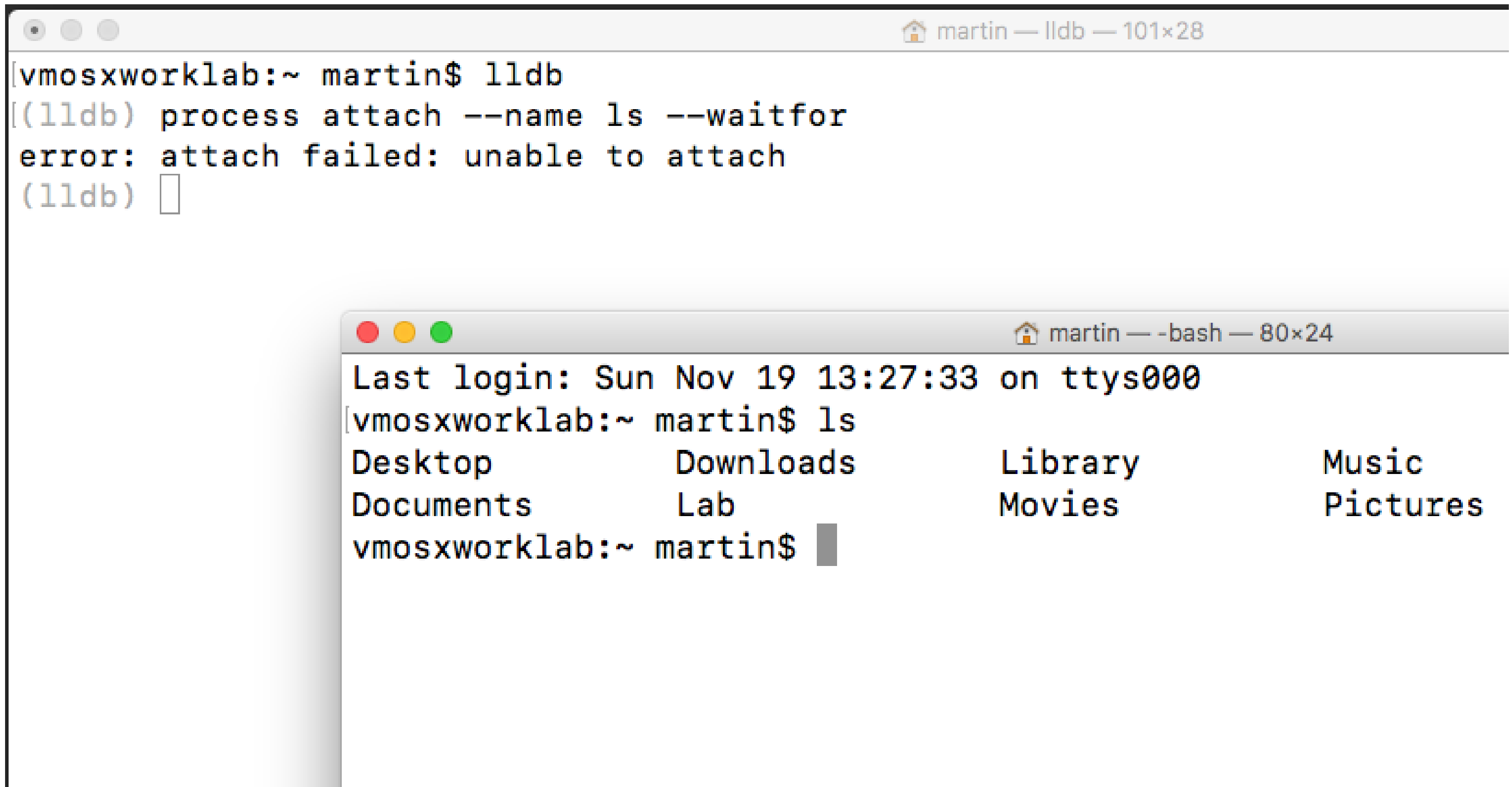
Introducción

- En macOS:



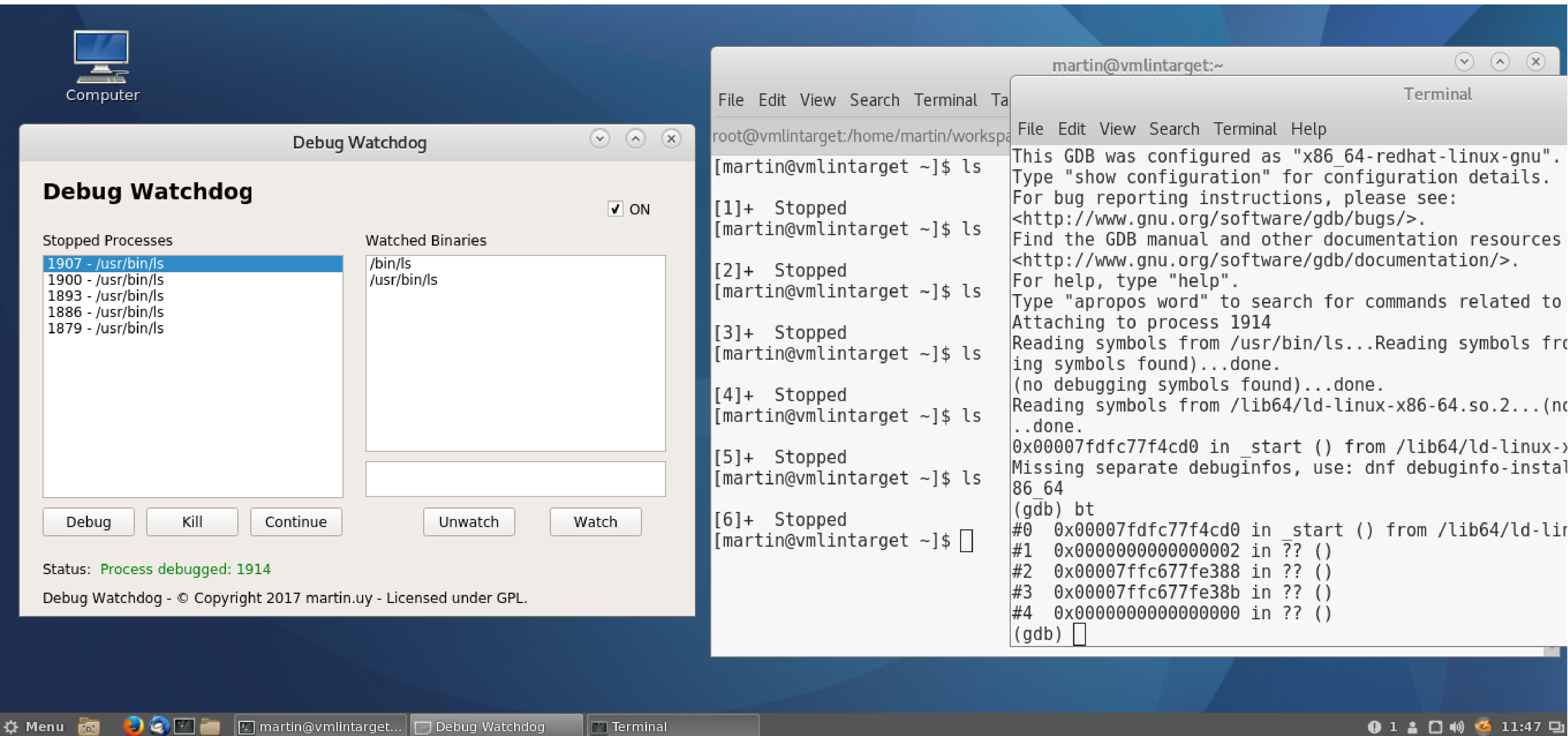
Introducción

- En macOS:



Introducción

- Debug Watchdog for Linux (v1.0)

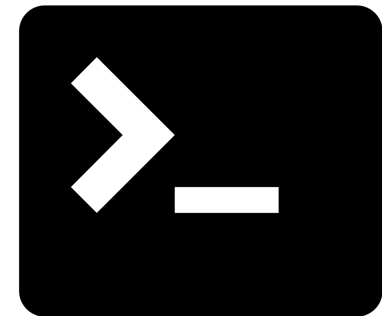


Introducción

- Debug Watchdog for Linux (v1.0)
  - Linux x86_64
    - Probado en Fedora
  - Licencia GPL
    - Contribuciones bienvenidas :-)
  - GitHub
    - https://github.com/martinuy/debugwatchdog

Background

- ¿Cómo detectar que un proceso es lanzado?
  - ¿Graphical User Interface?
  - ¿Daemon?
  - ¿Command-line?
  - ¿Script?
  - ¿Libc?



Background

- Hookear *execve* en libc, pero:
  - No todos los binarios ejecutables serían capturados
    - Ej. libc estáticamente linkeada, libc en contenedores, proceso lanzado sin libc, etc.
  - Habría que sobreescribir libc en disco, en lugar de patchear en run time únicamente
    - Reescribir el binario
    - Deshacer cambios
  - Quisiéramos un LD_PRELOAD "global"

Background

- ¿Cómo detectar que un proceso es lanzado?

```
[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
```

- Hay múltiples lanzadores pero una única syscall: *sys_execve* (kernel)

¡Hookear *sys_execve*!

Background

- ¿Qué es una syscall?
  - Llamada a un servicio de kernel, mediante una instrucción especial de la arquitectura
  - El procesador ejecuta el servicio en modo privilegiado
  - El thread que hace la syscall se transforma, temporalmente, en un thread de kernel
    - Cada thread tiene 2 stacks: uno en user y otro en kernel

Background

- ¿Qué es una syscall?
  - La API para aplicaciones en Linux es *libc*: no se ejecutan syscalls de forma directa
  - Se puede hacer si:
    - se sigue la interfaz binaria (ABI) especificada para la arquitectura; o,
    - a través de la función *syscall* (*libc*)

Background

- Syscall vista desde user (*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

- Instrucción SYSCALL (x86_64)
  - Procesador pasa a modo privilegiado
  - RIP (user) → RCX
  - IA32_LSTAR MSR (dirección del punto de entrada en kernel para syscalls: entry_SYSCALL_64) → RIP
  - RFLAGS → R11
  - RSP no es salvado: salvarlo es responsabilidad de user o kernel
  - Etc.

Background

- Syscall vista desde kernel: 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 vista desde kernel: 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 vista desde kernel: 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

- Tabla de syscalls

```
(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

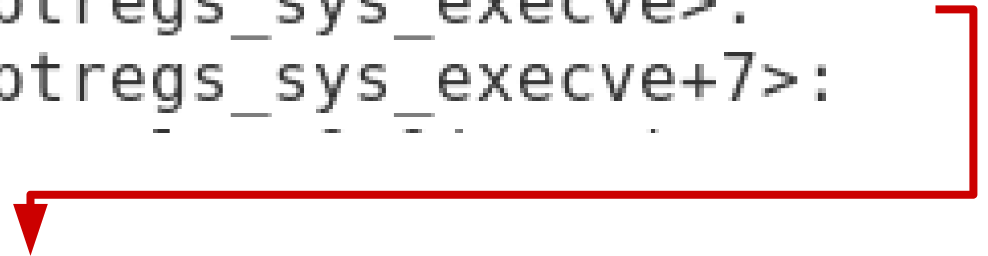
- Syscall vista desde kernel: syscalls_64.h

```
__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

- Algunas syscalls en la tabla van directo a su implementación y otras a un *stub* previo:

```
(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
  - salto al “slow path” primero (entry_SYSCALL64_slow_path)

Background

- entry_SYSCALL64_slow_path
  - Guarda registros extra (rbx, rbp, r12-r15) dentro de la estructura pt_regs previamente pusheada al stack
  - Llama a do_syscall_64, con la estructura pt_regs como parámetro
- 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
  - Si bien se vuelve a llamar a ptregs_sys_execve y stub_ptregs_64, el flujo de stub_ptregs_64 esta vez va directo a la syscall:

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

Background

- ¿Por qué se hace esto?
  - La C-ABI requiere que ciertos registros los preserve el llamado (rbx, rbp, r12-r15)
  - Sin embargo, el kernel no lo hace -por performance- a no ser que la syscall lo requiera explícitamente
  - La estructura pt_regs (guardada previamente en el stack) sirve para restaurar los valores originales de estos registros

Implementación

- ¿Cómo hookear sys_execve?
  - Patchear
    - tabla de syscalls
    - implementación de sys_execve
  - Ir a un trampolín (en un módulo de kernel previamente cargado) antes de que sys_execve retorne
  - ¿Qué es menos invasivo?

  Minimizar los parches en kernel afuera del módulo; reducir el riesgo

Implementación

- Si se patchea la tabla de syscalls, no se puede ir directo a sys_execve:
  - ¿qué sucede con los stubs previos y la estructura pt_regs?
- Por lo tanto, hook para la tabla de syscalls:

```
  .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
```

Implementación

```
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;
    }
```

- Implementación en el Módulo; solo se modifica la tabla de syscalls afuera del Módulo
- Agregar código antes o después de llamar a sys_execve

Implementación

- Resolver símbolos: ¿dónde está sys_execve_stub_ptregs_64_hook? ¿dónde está el verdadero SyS_execve?
  - Las direcciones virtuales se randomizan en cada booteo (KASLR)
  - kallsyms (/proc/kallsysms y kernel API)
  - kallsyms_lookup_name("sys_execve_stub_ptregs_64_hook")

Implementación

```
[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
```

Implementación

```
[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
```

Implementación

- Otros desafíos
  - Escribir una dirección de memoria con protección solo-lectura
    - Habilitar y deshabilitar escritura en memoria de solo-lectura mediante el registro *cr0*:
    - write_cr0 (read_cr0 () & (~ 0x10000))
    - write_cr0 (read_cr0 () | 0x10000)

Implementación

- Wrap-up

Syscalls table

sys_execve

Kernel image

common.c

entry_64.S

exec.c

sys_execve

sys_execve_stub_ptregs

sys_execve_hook

Module (Debug Watchdog)

1 2 3 4 5 6 7 8 9 10 11

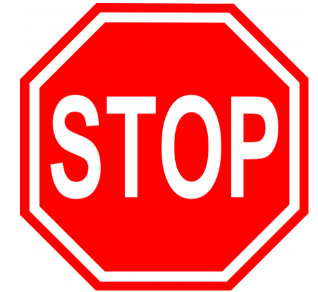
Implementación

- Wrap-up
  - Tabla de syscalls patcheada
  - Se llama a un stub en un módulo cargado previamente (sys_execve_stub_ptregs_64_hook)
  - Se devuelve el control al “flujo normal” pero con el registro RAX apuntando a la función sys_execve_hook (también ubicada en el Módulo)

Implementación

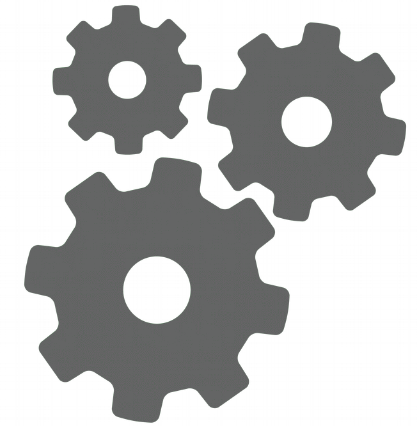
- Wrap-up
  - El “flujo normal” llama a la función sys_execve_hook (con los parámetros originales de sys_execve)
  - Se llama al verdadero sys_execve (reenviando los parámetros)
  - Se pone a debuggear el proceso (si fuera el binario ejecutable requerido)
  - Se retorna normalmente (stubs de salida del “flujo normal”)

Implementación



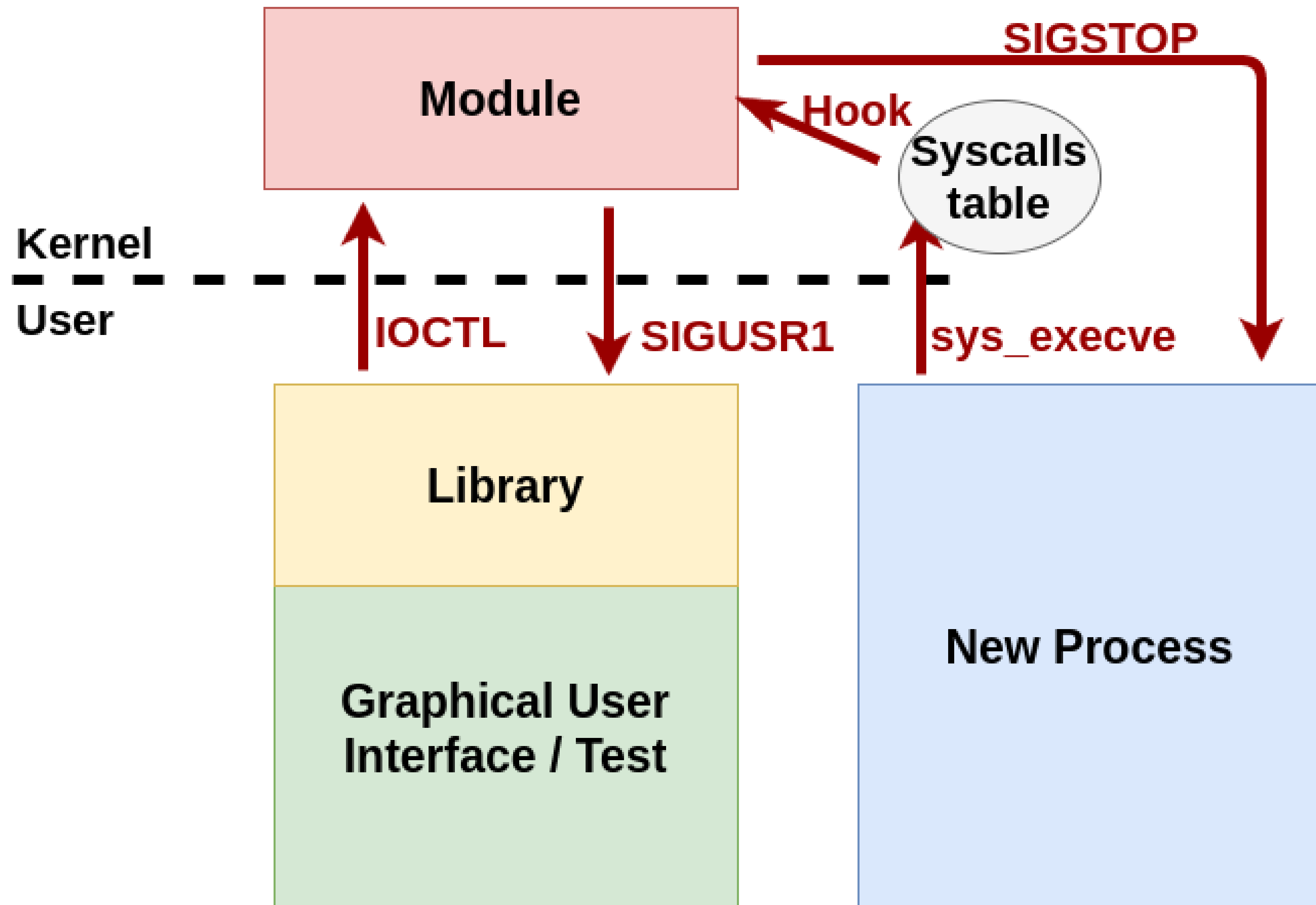
- ¿Cómo se pone a debuggear el proceso?
  - ¿PTRACE_TRACEME?
    - Funciona pero el debugger será el padre del proceso: gdb no se puede attachar porque no puede haber más de un debugger a la vez
  - Detener el proceso enviándole SIGSTOP desde kernel
    - el proceso no llega a ejecutar ninguna instrucción
    - gdb puede attacharse a un proceso detenido

Arquitectura



- Componentes del proyecto:
  - Módulo (C, kernel)
  - Librería (C)
  - Test (C)
  - UI (Qt/C++)

Arquitectura



Arquitectura

- Módulo
  - Se carga dinámicamente
    - Se requiere CAP_SYS_MODULE capability
  - Una única instancia, un único proceso de user se puede comunicar
    - Owner task id
    - Si este proceso muere, otro puede tomar ownership

Arquitectura

- Módulo
  - Múltiples threads ejecutando:
    - sys_execve_hook (cualquier task)
    - IOCTLs (owner task o ¿cualquier task?)
    - Locking de sincronización (mutex_lock/unlock)

Arquitectura

- Módulo
  - Comunicación Librería - Módulo
    - Device character
    - IOCTLs
      - Initialize / Finalize
      - Watch / Unwatch
      - Obtener lista de procesos detenidos

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

Arquitectura

- Módulo
  - Comunicación Módulo - Librería
    - SIGUSR1
      - Notificar a la Librería que hay al menos un nuevo proceso detenido

Arquitectura

- Módulo
  - ¿Cómo descargar el Módulo de forma segura?
    - Restaurar entrada original de sys_execve en tabla de syscalls
    - Descargar el Módulo
    - Pero, ¿qué sucede si hay un thread que leyó la tabla de syscalls justo antes de restaurarse y salta a ejecutar en memoria ahora desmapeada?

Arquitectura

- Librería
  - Inicializar
    - Registrar callback de notificación de procesos detenidos
    - Cargar el Módulo
  - Finalizar
    - Descargar el Módulo
  - Watch / Unwatch de binarios ejecutables
  - Registrar un callback para manejo de errores
  - Multi-threading

Arquitectura

- Librería
  - Requerimiento: desactivar manejo de SIGUSR1 en todos los threads del proceso
  - Thread de notificación de procesos detenidos
    - *sigwaitinfo* para recibir señales SIGUSR1 enviadas desde el Módulo
      - no hay handling asincrónico de señales
    - Llama al callback previamente registrado

Demo

Q & A

¡Gracias!

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