Lowlevel Userspace Programming

Petr Baudis
pasky@suse.cz
SUSE Labs

Novell

What'll Be Going On

Commented tour through random bits of system code

Assembly-level userspace programming

- Low-level kernel ABI
- The ELF-land, behind the mirror and dynamic linking

Magic syscalls

- Write your own dosemu: ucontext, vm86 (`loadtycoon`)
- The ptrace() interface (`retty` – steal process' tty)

`http://pasky.or.cz/plz-lowlevel/`

What's Expected from You

Basic C programming knowledge

Basic Linux environment familiarity

Basic Linux programming experience

To ask questions

Why's It Useful for You

Security hacking prerequisite (buffer overflows and stuff)

Tinkering with closed-source programs

Low-level debugging

Possibly kernel hacking

It's fun to understand how things work

Assembly Crash Course

Assembly in 30 seconds

Processor-specific programming at the instruction level

We'll be talking about **x86 (i386)** from now on:

```
pusha
push %eax
mov $0x80, %eax
int $0x10
popa
```

AT&T syntax (not Intel syntax!)

Novell

Registers and Stack

Registers: `%eax, %ebx, %ecx, %edx, ...`

(fast work with small chunks of data)

Stack: `[%esp, %ebp]`

(storage of larger chunks of local data,

and passing parameters to subroutines)

Stack grows *downwards*!

Low-level Kernel ABI

Low-level Kernel ABI - Syscalls

ABI = Application Binary Interface (c.f. API)

System calls (syscalls) – call some function inside the kernel from userspace

Several syscall gate methods:

- `int 0x80`
- `sysenter` (Intel)
- `syscall` (AMD)

Combined by syscall vsyscall ;-) (see next slides)

Low-level Kernel ABI – int 0x80 gate

The slowest but simplest

`int` instruction triggers a software interrupt

- (e.g. BIOS and DOS provide some interrupts as a system interface)
- Linux provides just one – `0x80`
- `%eax` contains syscall number (<asm/unistd.h>)
- other registers contain syscall arguments (usually)

An `int 0x80` call switches to kernel mode and dispatches control to an in-kernel sys_whatever() function

Low-level Kernel ABI - vsyscalls

Virtual syscalls

Switching to kernel mode is rather slow (relatively speaking)

2.6 kernels map a read-only page to userspace that contains some code and data

Instead of switching to kernel mode, applications call code from the vsyscall page in userspace

Example: “syscall()”, time()

Low-level Kernel ABI - others

Other kinds of ABI:

- sysctl
- /proc
- /sys
- netlink
- ...

ELF and Mr. ld.so

Executable and Linkable Format

The usually used binary format for code in current UNIX

Used for object files (.o), shared libraries (.so) and executables

Divided to sections (matter for linking) and segments (matter for executing)

Sections: .text, .data, .interp, ...

readelf, objdump

How Programs Get Executed

- Program caller does the fork() and execve() syscalls
- Kernel loads the executable, determines its format
- If ELF, interpreter (dynamic linker - /lib/ld-linux.so) is loaded by kernel
- The interpreter loads the actual program according to ELF headers and performs dynamic linking on demand, .init section text is executed
- The interpreter jumps at program's entry point (as specified in ELF header)

Dynamic Linking

Library routines aren't part of the executable but are in a (system-wide) library – shared object, .so

Symbols (functions or variables) are referenced in the executable by name and looked up in libraries the executable is dynamically linked to (also by specifying their names)

For each symbol, list of its references in executable is kept

Dynamic linker updates the references at runtime to refer to memory image of loaded library

Dynamic Linking - GOT

Text section (==code) should be shared between several instances of the program

Relocating inside of .text compromises that

Global Offset Table is thus allocated in private data section; each symbol has an entry with the address there

Text section merely references symbol's GOT entry (its address is same in all instances)

Dynamic Linking – PLT

Relocating all symbols at execution time can incur unnecessary overhead

Function symbol references can be referenced lazily using the **Procedure Linkage Table**

When calling function, code jumps not directly at the function address, but into PLT; each function has own entry there containing several instructions

First instruction is a `jmp` referencing the GOT entry, which is initialized by address of second instruction in PLT entry

The rest of PLT entry calls dynamic linker to resolve the symbol reference

Dynamic linker saves symbol address to GOT and calls it

The Magic Begins

mmap()

Not so magic one, and commonly known too

Map given file (or anonymous memory) to process' address space (possibly at fixed address)

Novell

Your Very Own dosemu

vm86(): Switch CPU to real-mode emulation mode

In fact not very useful unless you actually *are* making exactly dosemu (or having very special needs, like calling VESA BIOS)

If you are running “foreign” but protected-mode code, alternatively just:

- Run it as-is
- Possibly permit direct I/O using `ioperm()`
- Use custom SIGSEGV handler to emulate sw interrupts etc. sigaction(2) can pass signal handler context (especially registers) information: *ucontext_t (see <sys/ucontext.h>)

ptrace()

Linux provides a method for processes to manipulate other living processes – ptrace()

Appropriate permissions required

Types of manipulation:

- Peek/poke process code and data
- Peek/poke process registers
- Peek/poke process signal information
- Trace program execution
  - Stop at next instruction
  - Stop at next syscall

Commented code tour

retty: “Steal” tty of running process and redirect it to the current tty temporarily (“mini-`screen`”)

General idea – inject code to the running process that will reopen stdin/stdout/stderr as the new tty; when detaching, restore stdin/stdout/stderr to the original tty

Questions?

Novell

Thank You

Dan's turn!

http://pasky.or.cz/plz-lowlevel/

pasky@suse.cz

General Disclaimer

This document is not to be construed as a promise by any participating company to develop, deliver, or market a product. Novell, Inc., makes no representations or warranties with respect to the contents of this document, and specifically disclaims any express or implied warranties of merchantability or fitness for any particular purpose. Further, Novell, Inc., reserves the right to revise this document and to make changes to its content, at any time, without obligation to notify any person or entity of such revisions or changes. All Novell marks referenced in this presentation are trademarks or registered trademarks of Novell, Inc. in the United States and other countries. All third-party trademarks are the property of their respective owners.

SUSE
A NOVELL COMPANY

Novell