Lowlevel Userspace Programming

Petr Baudis
pasky@ucw.cz

What Will 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

- mmap() and how to use it to control hardware
- Write your own dosemu: ucontext, vm86 (`loadtycoon`)
- The ptrace() interface (`retty` - steal process' tty)

Anything else you want!

http://pasky.or.cz/pres/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

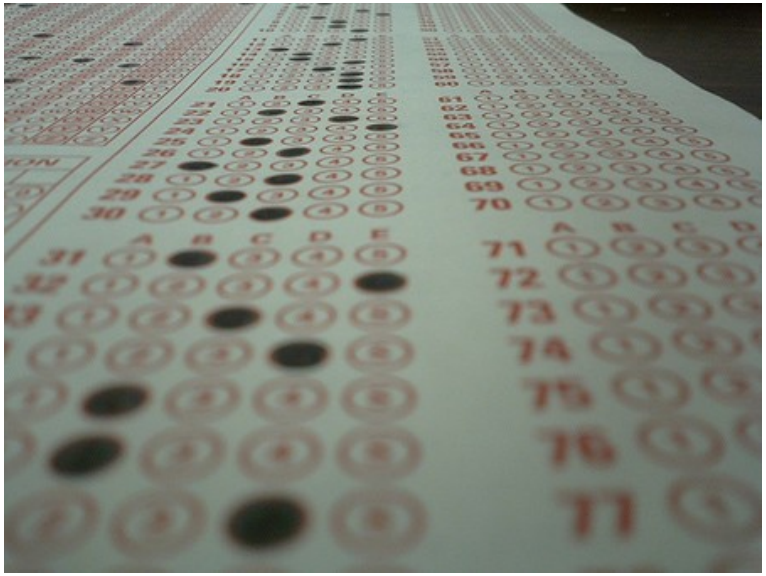
Debugging problems!

Security hacking prerequisite

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

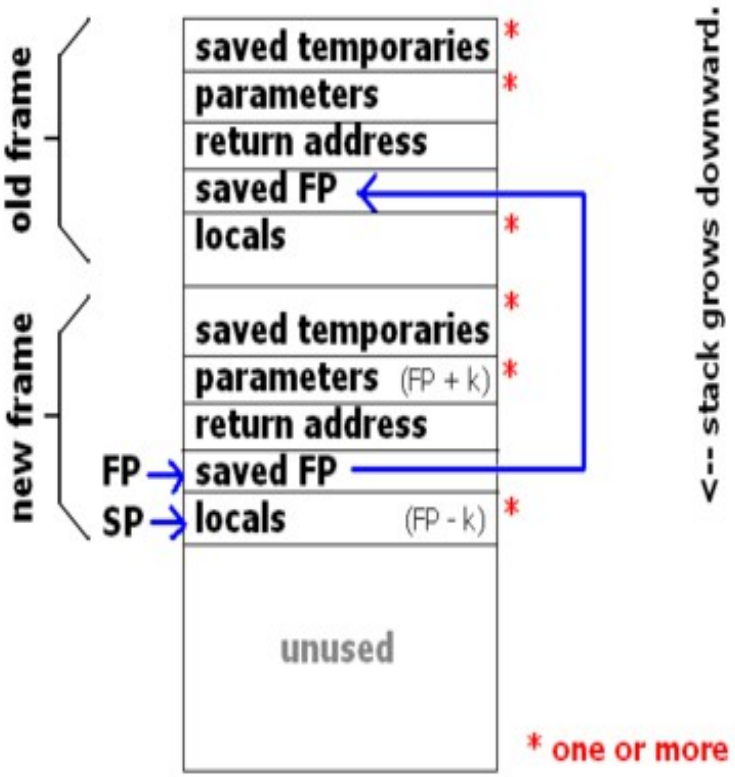
We'll be talking about **x86 (i386)** – AT&T syntax:

```
pusha
mov $0x1, %eax
xor %ebx, %ebx
int $0x80
mov %eax, (%esp)
popa
```

gcc -S -fverbose-asm file.c, objdump --source -d

[gdb] disass, info reg

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, passing parameters to subroutines)

Frames - backtrace()

Masinka

- Sometimes useful: Run raw x86 assembly within Linux process context
- Compile as a symbol, call from C program
- ...or just make it a main!

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: “linux-gate.so”

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 (uevent), signalfd, ...
- ...

Memory Mapping Hacks

mmap()

Not so magic one, and commonly known too

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

Memory Mapping for GPIO

The processor is just a circuit with bunch of pins!
And in your programs, you are wiggling with these pins...

Every system programmer should have basic experience with programming microcontrollers.

On most archs, GPIO is just mapped on memory region with fixed address and special semantics.

Memory is a wonderful place!

```
int mem_fd = open("/dev/mem",O_RDWR|O_SYNC);
volatile uint32_t *gpio_map = mmap(NULL,
    GPIO_BLOCK_SIZE, PROT_READ|PROT_WRITE,
    MAP_SHARED|MAP_FIXED, mem_fd, GPIO_BASE);
```

Mapping PCI Device Interfaces

PCI cards and such share the same address space as main memory – they are again mapped to various memory regions.

- Each card advertises its regions in a standard way.
- Even for unrecognized cards, kernel publishes their regions in `/sys/bus/pci/devices/.../resource*`
- Special file that is mapped to the memory. mmap() it and you are set!

Always be careful about MAP_SHARED vs. MAP_PRIVATE!

https://github.com/billfarrow/pcimem

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), executables and core dumps

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

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

readelf | eu-readelf, objdump

How Kernel Executes Programs

- Program caller does the fork() and execve() syscalls
  - ...or glibc interface: system(), posix_spawn(), ..
  - .
- Kernel determines the format (is it shebang "#!" or "\7fELF"?), loads the executable
  - fs/exec.c, fs/binfmt-elf.c
  - security checks, aux vector, envp, personality, …
- Executable and interpreter (dynamic linker: /lib/ld-linux.so) is loaded by kernel

How Programs Are Loaded

- ELF interpreter is usually the “dynamic linker”
  - Linker can be executed directly
  - Behavior can be affected significantly (ld.so(8))
- The interpreter performs relocation, sets up TLS, links in libraries, .init section text is executed, ...
  - elf/rtld.c
- 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); immediately or on-demand (lazy) - LD_BIND

Interfaces: dlopen(), dl_iterate_phdr(), LD_AUDIT: latrace

Commented code tour: screenenv

Dynamic Linking - Relocations

For each symbol, list of its references is kept

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

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)

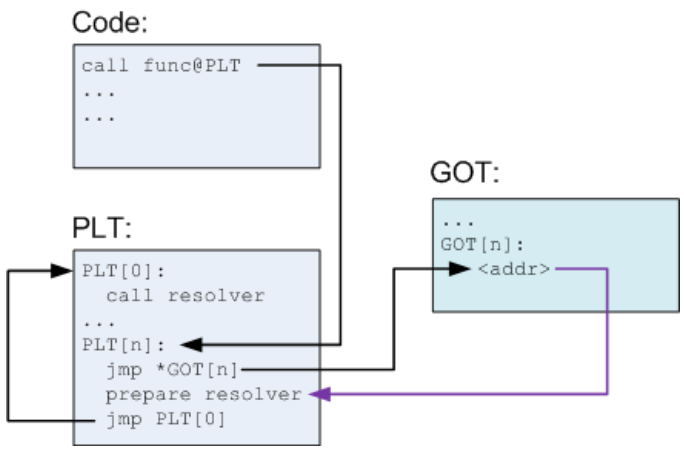
Dynamic Linking - PLT

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, that calls dynamic linker to resolve the symbol reference



Dynamic Linking - Goodies

GNU_IFUNC

Visibility, weak symbols

Versioned symbols

Various Magic

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 have 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>)

Commented code tour: loadtycoon

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`”)

Commented code tour

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

First, set up and manage a pty (like screen, script, GNU expect)

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?

Thank You

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

[Cpress] Lukas Jelinek – Jádro systému Linux
ELF Specification (+ platform supplements)
U. Drepper: How to write dynamic libraries
Algoritmy a jejich implementace (MFF UK)
POSIX2008