# On the Linux Compatibility Layer in OpenBSD 5.0

Paul Irofti
pirofti@openbsd.org

Slackathon, 2011

# Outline

# What Can compat_linux(8) Do?

It can make Linux binaries seem native to OpenBSD.

## Example

- IDA Pro
- Skype
- Opera
- It could even allow us to run MatLab!

# Different Perspectives

Userland:

**Definition**

compat_linux(8) is a Linux binary emulation layer for OpenBSD.

Kernel:

**Definition**

compat_linux(8) is a Linux translator for OpenBSD.

# Because. . .

## Good Stuff

- No dual-booting
- Quickly test something w/o having to install Linux
- Lets very useful propetary software run on OpenBSD

# But Most Importantly...

It's the last fighting point in /sys/compat against tedu's crusade!

# Static Executable

## Setup

The process is very straight forward:

- sysctl kern.emul.linux=1
- run your application as you would any other
- the kernel takes care of everything

## Possible Problems

- calling an unimplemented syscall
- special (linux-only) device/driver requirements

# Dynamically Linked Executable

This gets a lot more complicated:

- sysctl kern.emul.linux=1
- ldd(1) the designated executable
- gather the required Linux shared librarires
- fetch the proper Linux loader for them
- make sure the executable knows where to look for them
- pray
- run your application as you would any other
- the kernel takes care of everything else

# Dynamically Linked Executable (2)

Yes, this is crazy!

# DIY

## Setup

If you have Linux installed and handy:

- sysctl kern.emul.linux=1
- fetch the dynamic libraries listed by ldd(1)
- throw them under /emul/linux
- run the executable
- no need to set any paths

The rest will be handled behind the scenes by OpenBSD.

## Possible problems

- the loader will screw with you
- you'll end up in a maze of shared libraries and dependencies

# The Linux Distro Package

## Setup

The easiest way is to:

- sysctl kern.emul.linux=1
- pkg_add fedora_base
- run your application as you would any other
- let the kernel take care of rest

## Possible Problems

- missing package in fedora_base
- Solution: fetch the rpm and untar it under /emul/linux

# Special Needs

## Supported Devices

- CD/DVD-ROM
- Sound via /sys/compat/ossaudio
- And probably other devices that you can just symlink to

# Just in Time!

## At Runtime

- a process starts execution
- the executable type is detected
- the proper compat layer is chosen
- each system call is redirected for /sys/compat to resolve
- afterwards, control is handled back to userland

# Per-process data

## Each Process...

- is emulated separately
- holds its own emulation data in `struct proc`
- can fork and do threading transparently

# Start-up Flow

## For Each New Process...

- probe from `exec_makecmds()`
- `linux_elf_probe()`
- check for OS note — GNU
- check for brand — Linux
- `emul_find()` → /emul/linux/<path>
- switch from native to emul_linux_elf
- return to `exec_makecmds()`

# struct emul

## Contents

The most important members are:

- name — native/linux
- errno array
- signaling function
- system call array
- `copyargs()`, `setregs()`, `coredump()`
- `proc_{exec,fork,exit}()`

# Why Is It Only Available On i386?

## `linux_machdep.c`

- signaling — `sendsig()` and `sigreturn()`
- I/O — permissions, trapframe, control
- LDT fiddling
- threads — `[g|s]et_thread_area()`

## Solution

Write these functions for other architectures.

# The Meat in compat/linux

Most of the work in the kernel is done by the syscalls implementation.

## Theory

All the system calls provided by the Linux kernel should be reimplemented in the OpenBSD kernel.

## Practice

The system call array maps most of the Linux syscalls to the ones in OpenBSD with minor translations.

# System Call Categories

The syscalls are split into multiple files:

- file — `creat`, `open`, `lseek`, `fstat`...
- mount — `mount`, `umount`
- sched — `clone`, `sched_[g|s]etparam`...
- exec — `execve`, `uselib`
- signal — `sigaction`, `signal`, `kill`, `pause`...
- socket — `socket`, `bind`, `connect`, `listen`...
- time — `clock_getres`, `gettime`
- blkio, cdrom, fdio — I/O control for the given devices

# The Prototype

## Definition

```
linux_sys_foobar(struct proc *p, void *v,
    register_t *retval);
```

## Parameters

- struct proc — the calling thread
- args — the syscall's arguments
- retval — the return value

# Where the Wild Syscalls Grow

## `syscalls.master`

- contains the name/number syscall pairs
- generates the syscall declarations
- generates the corresponding arguments structs
- prototype fields: number type [type-dependent]

## Types

- STD — always included
- UNIMPL — unimplemented, not included in the system
- NOARGS — included, does not define the args structure

# Examples

## Types

- 13 STD { int linux_sys_time(linux_time_t *t); }
- 41 NOARGS { int sys_dup(u_int fd); }
- 240 UNIMPL linux_sys_futex

## Args

```
struct linux_sys_mknod_args {
    syscallarg(char *) path;
    syscallarg(int) mode;
    syscallarg(int) dev;
};
```

# Mostly Harmless

## Subsystem

- its pretty much isolated
- easy to extend
- easy to learn
- ugly to actually hack on

## TODOs and WIPs

- futex support
- full support for the 2.6 kernel series
- update the userland package
- ports to other architectures

# So Long, and Thanks for All the Fish

Questions?