arm

Linux Plumbers 2019

Moving the Linux ABI to Userspace

Lisbon, Portugal

Dave Martin <Dave.Martin@arm.com>

11 Sep, 2019

Disclaimer

a.k.a. Weaseling out of responsibility

This talk is about:

- An interesting bunch of hacks
- Some interesting things we could do.

This talk is not about:

- Why we should do any of this
- Whether we should do it at all.

Enjoy!

End goal

The Linux syscall interface can be fiddly to maintain and accumulates cruft.
So, is there a way to:

- move away from bare assembly syscalls?
- provide a single, consistent library interface to the kernel?
- Recent generic vDSO work suggests other ways of interfacing…

This talk explores some ideas.

The ABI

What's the ABI?

- Application Binary Interface
- $\rightarrow$ the machine-level interface between userspace and the kernel.

So...

What’s the kernel?

- A project?
- A binary blob?
- An interface?

The Linux kernel

The Linux kernel is mostly:

- a project developed by a specific community, and
- has access to privileged CPU features.

Linux userspace is mostly:

- code and data not developed by any single community, and
- has no direct access to privileged CPU features.

The Linux kernel ABI is the interface between the two.

So what?

Development by the Linux kernel community focuses on the Linux kernel project itself.
Alongside useful code, the kernel tends to accumulate cruft over time:

- boring glue code
- legacy junk
- useful, but superseded interfaces
- backwards compatibility shims
- things that seemed like a good idea at the time.

These are often exposed as ABI and so run in privileged mode … forever.
$\longrightarrow$ hard to change or remove without breaking compatibility.

Some random examples

int personality(unsigned long persona)
long set_robust_list(int pid, struct robust_list_head **head_ptr, size_t *len_ptr)
int rt_sigprocmask(int how, sigset_t *mask, sigset_t *old, size_t sigsetsize)
Exposure of these (and many others) as machine-level syscalls commits us to a particular interface:

- These are “syscalls”, so they trap into the kernel every time.
- Basically just access a variable on behalf of userspace.
- Require privilege because … er, why exactly?
- Can the variable live in user memory instead?
- Atomicity may be a concern, less so on newer CPU architectures.

What can we do to change this?

Alternatives

Force calls to the kernel to go through a userspace library. But:

- Who maintains it?
- Userspace C library projects? e.g., glibc, Bionic, musl libc
  $\longrightarrow$ fragmentation?
- Separate project?
  $\longrightarrow$ can't compel people to use it … fragmentation again.
- Ideally maintain as part of the kernel project instead, but unprivileged.
- Ideally tightly coupled to a kernel version so that we can refactor/reimplement as needed.
- $\longrightarrow$ we already have something like that: vDSO.

Backward compatibility

Up-to-date userspace would call the kernel through vDSO functions.

Old user binaries will still make “old-style” direct syscalls though!

- Keep backwards compatibility glue in the kernel?
- Could do, but makes the exercise a bit pointless…
- Bounce old-style syscalls back to userspace somehow?

Hack #1

Hack #1: here's one we made earlier

Do we already have infrastructure for this?

SECCOMP already allows us to bounce syscalls:

- Extend `struct seccomp_data` with vDSO bounds information
- C library installs a suitable filter and `SIGSYS` handler on process startup.

See [1] for code.

Hack #1: SECCOMP data

```
struct seccomp_data {
        __u32 arch;
        __u64 instruction_pointer;
        __u64 args[6];
        __u64 ip_bounds[2];
};
```

Hack #1: SECCOMP filter

Comparing the instruction pointer against two bounds isn't too hard…

```
/* Would be better in eBPF... */
static const struct sock_filter insns[] = {
        LD(arch, 0),
        JNE(AUDIT_ARCH_AARCH64, 19 /*trap*/),

        LD(instruction_pointer, 4),
        TAX,
        LD(ip_bounds[0], 4),
        JGTX(15 /*trap*/),
        JLTX(4 /*pass0*/),

        LD(instruction_pointer, 0),
        TAX,
        LD(ip_bounds[0], 0),
        JGTX(10 /*trap*/),
```

```
/*pass0*/
        LD(instruction_pointer, 4),
        TAX,
        LD(ip_bounds[1], 4),
        JLTX(6 /*trap*/),
        JGTX(4 /*pass1*/),

        LD(instruction_pointer, 0),
        TAX,
        LD(ip_bounds[1], 0),
        JLEX(1 /*trap*/),

/*pass1*/
        RET(SECCOMP_RET_ALLOW),

/*trap*/
        RET(SECCOMP_RET_TRAP),
};
```


arm

Hack #1: SECCOMP SIGSYS handler

user handler (arm64):

```
static void sys_handler(int n _unused,
                        siginfo_t *si _unused,
                        void *uc_)
{
        ucontext_t *uc = uc_;
        int nr = (int)uc->uc_mcontext.regs[8];

        /* Handle interesting syscalls: */

        switch (nr) {
        case __NR_personality:
                return do_personality(uc);
        case __NR_umask:
                return do_umask(uc);
        /* ... */
        }
```

- User SIGSYS handler can intercept and emulate syscalls as desired
- Extracting the syscall number and arguments requires arch-specific code
- Other aspects may be more generic.

arm

Hack #1: SECCOMP SIGSYS handler (2)

```
        /* Otherwise, fall back to the real syscall: */

        uc->uc_mcontext.regs[0] = __kernel_syscall(nr,
                uc->uc_mcontext.regs[0],
                uc->uc_mcontext.regs[1],
                uc->uc_mcontext.regs[2],
                uc->uc_mcontext.regs[3],
                uc->uc_mcontext.regs[4],
                uc->uc_mcontext.regs[5],
                uc->uc_mcontext.regs[6]);
}
```

- For other random syscalls, call a fallback wrapper in the vDSO.

See [2] for code.

arm

Hack #1: Any good?

Well, it's:

- Invasive
  - signals are great…
- Broken
  - stack overflows… and random SIGSEGVs
  - interacts badly with existing SECCOMP users…
- Slow!
  - adds significant overhead to every syscall…
    even with the BPF JIT.

So, fun (sort of), but maybe not the right approach.

Hack #2

Hack #2: Low-level bouncer

What's the best we could theoretically do?

- Intercept syscalls in the per-arch syscall entry code.
- Bounce syscalls via a custom mini-signal to a handler in the vDSO.

Such mechanisms can be controversial [3].
However, this mechanism would not be ABI:

- in theory…
- private, probably per-arch interface between the kernel and vDSO
- Privileged mode kernel still must not trust the vDSO (consider ROP attacks, etc.)

See [4] for code.

Hack #2: Low-level bouncer (2)

Code in the vDSO's handler can now do what it likes:

- Make one or more real syscalls
- Do something purely in userspace
- Or, a mixture.

Hack #2: Syscall frame

When bouncing a syscall, save the bare minimum on the user stack before jumping to the bounce handler in userspace.

e.g., for arm64:

Register	Purpose
x8	syscall number
sp	initial stack pointer
pc	instruction pointer at the original syscall
pstate	condition flags etc.

Hack #2: Syscall frame restore

On some architectures, it's impossible to restore execution state from the syscall frame directly in userspace…

- Need to jump and restore registers with a single instruction.
- x86: `ret` or `iret` works
- arm: `pop {pc}` works
- arm64: hmmm, can't do it directly: must use a syscall.
- $\longrightarrow$ add a flag to the system call number to request this.
  (not illustrated on the next slide, but straightforward).

Hack #2: Syscall entry

arch/arm64/kernel/syscall.c:

```
asmlinkage void el0_svc_handler(struct pt_regs *regs)
{
        if (regs->pc < (unsigned long)current->mm->context.vdso ||
            regs->pc >= (unsigned long)current->mm->context.vdso +
                                (vdso_pages << PAGE_SHIFT)) {
                if (push_syscall_frame(regs))
                        arm64_notify_segfault(regs->sp);
                        return;
                }

                regs->pc = (unsigned long)VDSO_SYMBOL(
                                current->mm->context.vdso, __kernel_syscall);
        } else {
                /* syscall from vDSO */
                el0_svc_common(regs, nr, __NR_syscalls, sys_call_table);
        }
}
```

Hack #2: vDSO

The simple fallback code for a bounced syscall becomes (arm64):

arch/arm64/kernel/vdso/syscall.S:

```
ENTRY(__kernel_syscall)
        .cfi_startproc
        .cfi_def_cfa    sp, 0
        .cfi_offset     x8, 0
        .cfi_offset     sp, 8
        .cfi_offset     x30, 16

        orr         w8, w8, #__REDIRECTED_SYSCALL
        svc         #0
        .cfi_endproc
ENDPROC(__kernel_syscall)
```

Unwind directives allow backtracing through this custom frame, without userspace needing built-in knowledge of the layout. (The example above is not quite right, though.)

Hack #2: Any good?

- Not invasive
  - almost the same as a bare syscall
  - minimal extra stack overhead
  - syscalls from the vDSO don't bounce, so maybe store the syscall frame data in a per-task page $\longrightarrow$ zero user stack overhead.
  - tracers/debuggers may see the transient call into the vDSO, but this isn't clearly "wrong".
- Not broken
  - systemd boots
  - haven't seen anything fail so far…
- Less slow
  - Base syscall overhead increases by 2–4$\times$.

Hack #2: Does speed matter?

Yes! But:

- Bounced syscalls would only happen in legacy binaries:
  syscall → bounce → vDSO handler → real syscall from vDSO.
- To avoid bounces, just port things that make direct syscalls:
  - compiler support libraries and C libraries
  - language runtimes / JITs (You know who you are.)
- “New-style” syscalls via direct vDSO calls only incur the extra overhead of calling a shared library:
  call vDSO → real syscall from vDSO.
- Unported software still works, just a bit slower if syscall-intensive.

Hack #2: Possible pitfalls

Interesting questions that I don't answer yet:

- How effectively can we hide this from regular userspace? Does it matter?
- What if userspace unmaps or moves the vDSO?
- Can static binaries use the vDSO? (technically possible, but would need explicit libc support).
- Interaction with ptrace:
  Currently, only syscalls from the vDSO count as "real" for ptrace: bounced syscalls don't count.
- Does it break ABIs that expose syscall internals?
  Maybe, but software using these has to cope with an evolving syscall interface anyway:
  - SECCOMP
  - ftrace
  - ptrace…

Where can we go next?

Hack #3…

Things that mostly just access variables for userspace might be moved entirely or partly to the vDSO:

- Using a per-task user page to store the data:
  personality(), set_robust_list(), get_robust_list(), possibly some prctl() subcommands;
- Using a per `task->fs` user page:
  umask(), maybe other stuff;
- Glue for superseded syscalls, e.g.:
  wait(), waitpid(), pause(), open(), unlink(), chmod(), rename(), access(), stat(), nanosleep(), sigprocmask(), sigsuspend(), pause(), pselect() …
- Maybe, move the signal mask to userspace, along with aspects of
  rt_sigprocmask(), rt_sigsuspend(), pselect6(), ppoll(), epoll_pwait(), and parts of the signal infrastructure.

Much of this is non-trivial (or impossible?), but there are some interesting hacking opportunities…

… and beyond

Eventually, we could:

- Define the kernel interface as a set of vDSO functions
- Freely redesign communication across the privileged/unprivileged boundary from one kernel version to the next
- Use symbol versioning to manage forward/backward compatibility
  $\longrightarrow$ for ELF, at least
  $\longrightarrow$ can reduce need to keep adding new function names.

Big project, but we wouldn't have to do it all in one go…

Discussion

References

1. SECCOMP-based syscall bouncer (kernel):
   http://linux-arm.org/git?p=linux-dm.git;a=shortlog;h=refs/heads/seccomp/vdso/plumbers/head
   git://linux-arm.org/linux-dm.git seccomp/vdso/plumbers/head
2. SECCOMP-based syscall bouncer (userspace):
   http://linux-arm.org/git?p=bouncer.git;a=shortlog;h=refs/heads/master
   git://linux-arm.org/bouncer.git master
3. Re: RFC: userspace exception fixups:
   https://lore.kernel.org/lkml/CAHk-=wjJhdr3JCnGrMKqL-prxYd__kkAspKVYBO3BYYmq2hu4A@mail.gmail.com/
4. Native syscall bouncer for arm64:
   http://linux-arm.org/git?p=linux-dm.git;a=shortlog;h=refs/heads/arm64/vdso-syscall/plumbers/head
   git://linux-arm.org/linux-dm.git arm64/vdso-syscall/plumbers/head

arm

Thank you!