System Call Intermediation

December 27, 2025

1 Introduction

This document explores implementation options for syscall intermediation. In general, existing interfaces provide excellent hooks for extending functionality without requiring reworking of the dependent components, and syscalls are an excellent example.

In particular, the ability to proxy system calls allows for transparent post-hoc modification of:

security fine grained access control including parameters like network destination

distribution transparent distributed service proxy, process distribution, and fault tolerance

parameterization synthesis of configuration files, peer network identity

packaging explicit per-executable control of dependncies

with existing source/binaries. Each of these topics requires its own treatment to outline the possible utility, but this document explores possible points of intermediation, that is where can we intercept the systems calls, modify them in radical ways, but still preserve the default path for cases where its desirable.

Rather than focus on the value of specific features enabled by syscall intermediation, lets consider todays cloud environment. It shouldn't be much of a perceptual shift to imagine that all the various VM and container management APIs, storage systems, SDN features, etc. collectively constitute a *distributed operating system*. Sadly, that handful of APIs isn't a very cohesive design. Starting a process or performing i/o should really be the most common and trivial of things and not require service-specific adaptation libraries and different security and token infrastructure.

We can maybe carry forward the lessons from Unix a little bit. If paramteriziation is required then it can be in the construction of an object (open), leaving the access to standard and universal access (read/write).

The following table breaks down intermeidatation approaches by security, portability and development effort and overhead.

	security	compatibility	effort	overhead
ld.so	poor-ish	high	low	none-ish
LD_PRELOAD	none	high	low	none-ish
binary rewriting	fair	high	medium	none
ptrace	fair	medium	low	substantial
seccomp	good	high	fair	3x syscall overhead plus
kernel	high	high	high	none
asymmetric multiprocessing	high	high	medium- high	plus an ipi
hypervisor	high	high	medium	vmexit plus other
wasm	?	med	medium	minimal

2 ld.so

We start by making the assumption that users are vectoring syscalls through some standard library, like libc, and that replacing these functions is sufficient to perform our intermediation. This might be useful in a voluntary setting, but it clearly establishes no security boundary around the process. It also trivially fails to account for languages like Go which use the syscall

2.1 LD_PRELOAD

For demos or strictly opt-in uses, we can use the LD_PRELOAD flag to preferentially use symbols from our replacement libc.so before the system provided one.

2.2 Binary Rewriting

We can tighten the security boundary by taking the compiled executable, disassembling it, changing the syscall entry points, and reassemblying it. This generally works ok

Ultimately though, we cant really guard against someone synthesizing their own syscall instruction after the fact, including the use of legacy entry points like `int $80` on x86.

3 ptrace

Ptrace is a debugging interface from SysV. It allows a debugging process to control a target process. It can stop programs or threads, examine registers, call functions, and importantly trap the syscalls from the target and allow arbitrary modification from the debugging process.

Since ptrace was never intended to be used as a sandbox, and the code on the kernel side is a bit crufty, its probably not safe to assume that a determined process wouldn't be able to either subvert the controls or escape entirely.

The primary problem with the ptrace is performance. Not only does the target syscall wake up the debugger process, and require additional syscalls to examine the state of the target and resume it, but the only data interface is a syscall for a single-word transfer.

4 seccomp

Seccomp is Linux facility which is used (usually in conjunction with namespaces) to construct sandboxes for containers and other kinds of processes. It works by installing a BPF (classical, not EBPF) filter that traps system calls. Facilities for rewriting system calls are limited, but it is possible to conditionally forward system calls to a monitor process.

`SECCOMP_IOCTL_NOTIF_ADDFD` BPF filters can optionally send the system call to a second process using `SECCOMP_RET_USER_NOTIF`.

4.1 security

The man page for `seccomp_unotify` goes to some length to argue that forwarded syscalls are unsuitable for enforcement of security properties due to a scheduling window between the monitor process and the orignal process, which might attempt to bypass the restriction

5 kernel

Kernel implementation of intermediation involves redirecting the syscall hander. Some operating systems (FreeBSD) have facilities for multiplexing the system call table based on executable type to support emulating other operating systems, but Linux does not. For security reasons modifying the system call table from a loadable module is obfuscated by hiding the relevant symbols. It would be possible in Linux It might be possible to overwrite the LSTAR register on x86_64, but this register is per-cpu, and its not clear who else might be writing it under what conditions. Other architectures like ARM present system calls as interrupts, so this wouldn't be an easy option there.

FreeBSD would be a reasonable target for in-kernel support, but its not clear how complete the linux-emulation is, or to what degree it will continue to evolve with linux.

There also are a few linux-workalike mini kernels, for example moss in rust and nanos in C. These are great targets for doing kernel research, but are likely never going to be complete.

6 Asymmetric Multiprocessing

To avoid intrusive changes and whole-kernel compilation, we might consider an arrangement where some processors run intermediated executables using a custom kernel stub, and the remaining processors are left running the commodity linux kernel to provide services/drivers.

This means that the default 'local' service of system calls would need to be plumbed using a message queue and ipi style approach. The service side of this in linux could likely be a loadable module. If the stub included a small network stack this partitioning could be extended separate whole 'application' vs 'service' machines.

This approach avoids many of the portability problems involved in using other kernels in that the service partition is running the appropriate version with the correct behaviour. We just have to be very careful in the proxy not to make unnecessary assumptions.

Sadly, there is no real tooling to support multiple kernels running in the same SMP system (except the recent Parker), so the system would likely have to boot into the redirect kernel, which would then start an included linux on some subset of the nodes.

The two kernels do need to share resources. Devices aren't really a problem since all the drivers would be located on the linux side, but moving memory resources across the boundary represents a problem that must be solved.

7 Hypervisor

The hypervisor gives us very general source of indirection to intercept system calls, at least on x86_64. If we proxy accesses to the LSTAR register we can record the entry for the kernel, but leave the underlying architectural register pointing at an invalid guest physical address. This will force a VMEXIT on every system call

This comes with a pretty substantial cost, and is unlikely to be convenient unless you happen to already have your hands in a hypervisor. It does also introduce a kind of unpleasant assymmetry of function, and would likely require kernel and version detection in order to use guest facilties and do passthorugh.

A hypervisor might also be a good implementation host for asymmetric multiprocessing. Instead of trapping system calls between the target and the guest operating system, boot the guest using the proxy stub. System calls can be forwarded to the host using hypercalls.

8 wasm

WASI

9 Conclusion

Syscall intermeidation seems like a potentialy useful facility, particularily if we use it to augment system semantics in a cloud environment. While there are several implementation methods, none of them is simultaneouly secure as a sandbox and non-invasive towards the Linux kernel.

Ideally there could be an abstract implementation of things like a policy engine and a remote syscall handing facility, but the various implementation environments, in particular the runtime dependencies in the various kernel environments vs userspace make that a difficult proposition.

