Status and Desiderata for Syscall Tracing and Virtualization Support

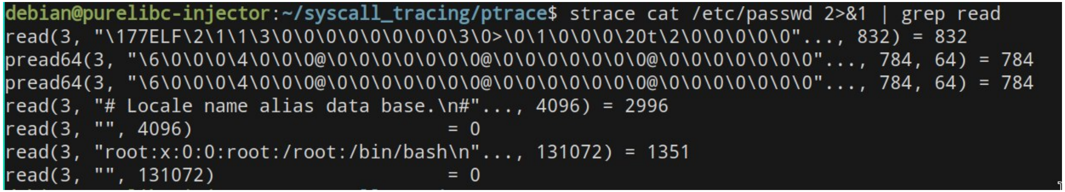
Renzo Davoli, Davide Berardi

FOSDEM 2025: February 2nd, 2025

This work was partially supported by the project SERICS (PE00000014) under the NRRP MUR program funded by the European Union - NextGenerationEU.

System call tracing

We will focus on Linux system in this presentation.
Mainly for debug features system calls can be hijacked to print information on which systemcall has been executed:
An example of this feature is `strace`, a program that prints every systemcall executed by another program.



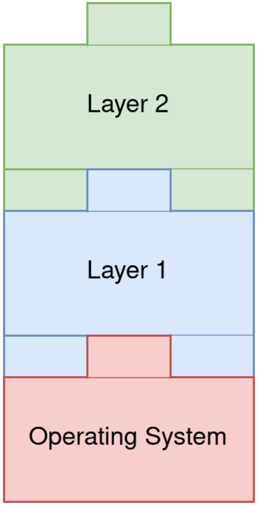
```
debian@purelibc-injector:~/syscall_tracing/ptrace$ strace cat /etc/passwd 2>&1 | grep read
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\20t\2\0\0\0\0\0"..., 832) = 832
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
pread64(3, "\6\0\0\0\4\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0@\0\0\0\0\0\0\0"..., 784, 64) = 784
read(3, "# Locale name alias data base.\n#"..., 4096) = 2996
read(3, "", 4096)                       = 0
read(3, "root:x:0:0:root:/root:/bin/bash\n"..., 131072) = 1351
read(3, "", 131072)                     = 0
```

https://github.com/virtualsquare/syscall_tracing

System call based virtualization

System calls are an interface to virtualize the system creating a virtualization similar to os-level one (i.e. namespaces).

With this feature is possible to create the concept of MLOS: Multiple Layer Operating System. Every layer has the same interface of the other (systemcalls, e.g. POSIX?)



https://github.com/virtualsquare/syscall_tracing

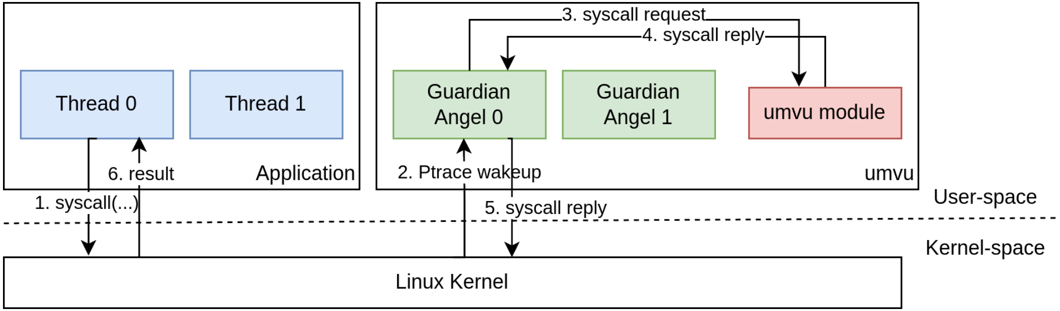
How to hijack systemcalls

Systemcalls can be hijacked by instructing the kernel to tell to an userspace program (hypervisor) which systemcall has been invoked and with which parameters.

We will refer to this technique as “hypervisor mode”.
On the other side programs can declare to be traced by themselves (e.g. by changing the function called when executing a specific systemcall-wrapper, we refer to this as “self-virtualization”.

`https://github.com/virtualsquare/syscall_tracing`

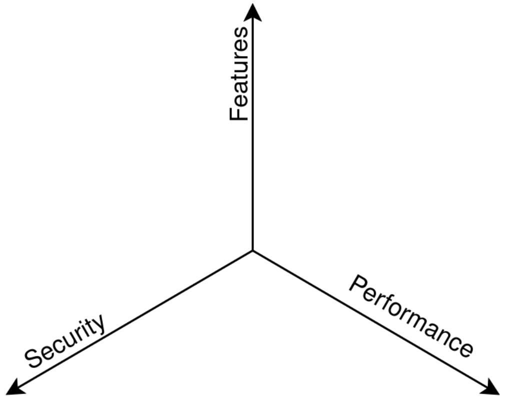
How to hijack systemcalls



`https://github.com/virtualsquare/syscall_tracing`

Main users

There are many use cases for these approaches, with **no silver bullet**. For instance, we could classify every approach using the following plot:



https://github.com/virtualsquare/syscall_tracing

Main users

Some users of this feature (hypervisors) are:

fakeroot	a program to “trick” root-only programs to be used by unprivileged users.
wine	a compatibility layer to run Windows programs under Linux.
gVisor	a total rewrite of a kernel in go language. Made by Google, its main features are security and performance (due to this fact it employs real virtualization, KVM, when available).

`https://github.com/virtualsquare/syscall_tracing`

Main users

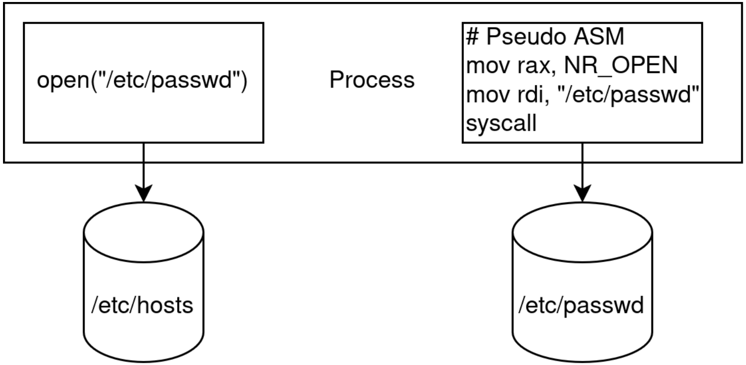
Some users of this feature (hypervisors) are:

vuos and mlos Combining various approaches, we created a multi layered operating system, which can be used to focus on security (e.g. blocking system calls or higjacking files), features (e.g. adding support for new network protocols or file systems), and performance (selecting the best emulation method).

`https://github.com/virtualsquare/syscall_tracing`

Problem: Security

Let us suppose to trace a program for security. We want to confine it in a sub-filesystem.
If we trace it with a not-so-strong method (e.g. purelibc), it can **escape the sandbox!!!**

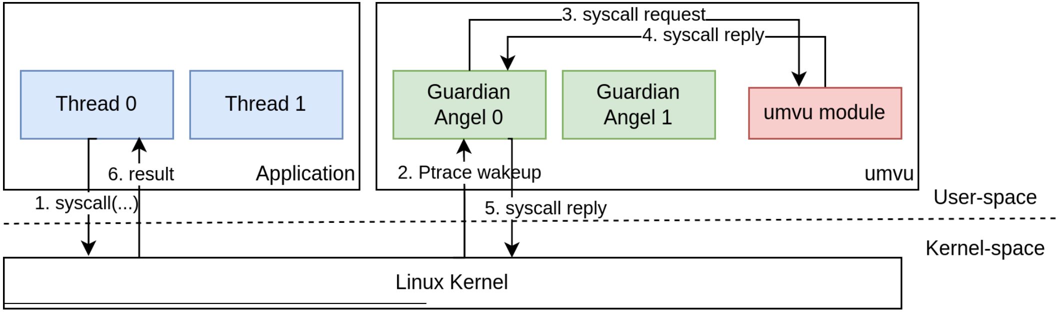


https://github.com/virtualsquare/syscall_tracing

Problem: Speed

Multi threading is not easy to achieve in all the implementation, we will get to this for every case. For instance, how to create a multi threaded hypervisor to handle multiple events (e.g. select / poll?).

Hypervisor mode is slow, it pass in kernel space two times per syscall, slowing down the programs.



https://github.com/virtualsquare/syscall_tracing

Problem: Speed

```
timing (no syscall capture, for comparison):
time for i in {1..1000}; do cat /etc/hostname > /dev/null; done
real     0m0.988s
user     0m0.655s
sys      0m0.372s
```

https://github.com/virtualsquare/syscall_tracing

Technique 1: ptrace

Ptrace can be used to hijack system calls. A tracer process can be designed to wake up when the child execute a system call. The systemcall gets to the tracer which can change the systemcall number and make the child continue.

Event notification: wait(2)

+ Tracks every systemcall
+ Hypervisor Mode
+ Easy to implement
+ Extremely Tested
- Somewhat architecture dependant
- Difficulty in Multithreading
- Speed
- Security

```
time for i in {1..1000}; do ./ptrace_virt /etc/passwd > /dev/null; done
real     0m5.985s
user     0m2.347s
sys      0m2.615s
```

https://github.com/virtualsquare/syscall_tracing

PTRACE_GET_SYSCALL_INFO

Getting the systemcall information in ptrace is difficult and could be problematic, for instance registers are get, one by one, by using a memory area which is architecture dependant (PTRACE_PEEK_USER).

PTRACE_GETREGSET and PTRACE_SETREGSET have been developed to get registers in block, but they're still architecture dependent.

To overcome this problem of incompatibility between architectures, PTRACE_GET_SYSCALL_INFO has been developed.

`https://github.com/virtualsquare/syscall_tracing`

Desiderata: PTRACE_SET_SYSCALL_INFO

We got PTRACE_GET_SYSCALL_INFO, but how can we modify the systemcall that gets executed or its parameters? We still need to rely on *PEEK*/*POKE* architecture dependant interfaces!!!

PTRACE_SET_SYSCALL_INFO is still missing :(

Fortunately currently discussed in the kernel mailing list thanks to Dmitry V. Levin :)

Technique 2: ptrace + seccomp

Seccomp can be used in conjunction with ptrace to make the process a little bit more secure. It avoids a second event (ptrace gets an event for every syscall in and out).

Event notification: wait(2)

+ Tracks every systemcall

+ Hypervisor Mode

+ Easy to implement

+ Well tested

- Somewhat architecture dependant

- Difficulty in Multithreading

- Speed!!!

```
time for i in {1..1000}; do ./ptrace_seccomp_virt cat /etc/passwd > /dev/
    null; done
real     0m4.342s
user     0m2.182s
sys      0m1.631s
```

https://github.com/virtualsquare/syscall_tracing

Technique 3: seccomp_unotify

Seccomp can also be used without ptrace. It notifies a tracer of a systemcall and then **the tracer executes the modified system call and return the result**.

It requires tricks to send the file descriptor from the tracee to the tracer and back (pidfd_getfd over shared memory and SECCOMP_IOCTL_NOTIF_ADDFD).

Event notification: fd+ioctl

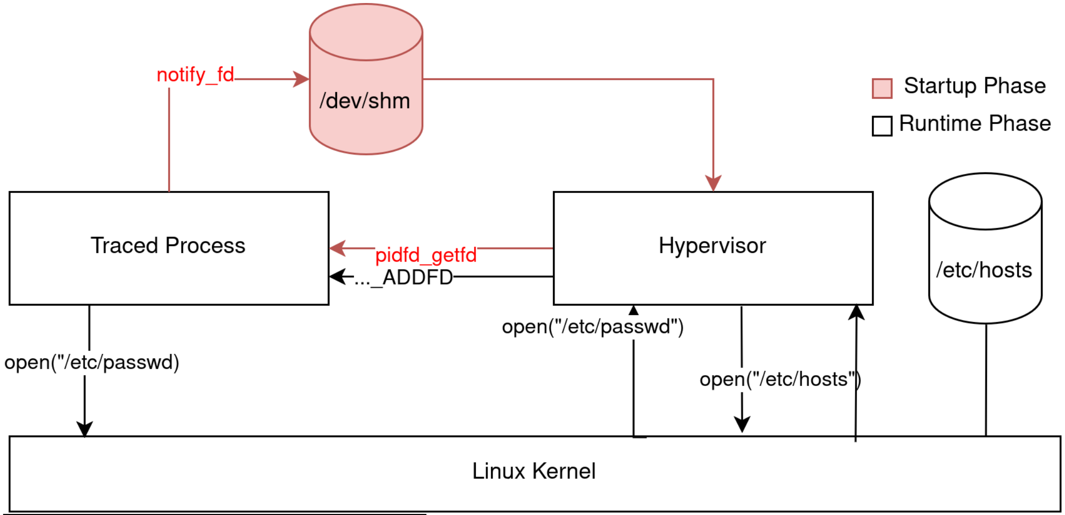
+ Tracks every systemcall
+ Hypervisor Mode
+ Security

- Tricky!
- Speed

```
time for i in {1..1000}; do ./seccomp_unotify_virt cat /etc/passwd > /dev
    /null; done
real    0m4.585s
user    0m1.132s
sys     0m0.776s
```

https://github.com/virtualsquare/syscall_tracing

Technique 3: seccomp_unotify



https://github.com/virtualsquare/syscall_tracing

Desiderata: eBPF maps in seccomp

Currently seccomp only support BPF, not eBPF, with eBPF it is possibile to use maps (with no decreases of security!) and we could accelerate virtualization of file descriptors.
Let us assume the following problem:

1. We could have mixed real and virtualized file descriptors (for instance we virtualized `open` but not `socket`).
2. Then a systemcalls which refers to a real file descriptor (for instance `read` on the socket) is requested to the system.
3. The system wakes up the hypervisor (tracer)
4. The tracer understand that the file descriptor is not a virtual one, returning the control to the virtualized process.
5. The kernel returns the control to the traced process.

`https://github.com/virtualsquare/syscall_tracing`

Desiderata: eBPF maps in seccomp

Currently seccomp only support BPF, not eBPF, with eBPF it is possibile to use maps (with no decreases of security!) and we could accelerate virtualization of file descriptors.
Let us assume the following problem:

1. We could have mixed real and virtualized file descriptors (for instance we virtualized `open` but not `socket`.
2. Then a systemcalls which refers to a real file descriptor (for instance `read` on the socket) is requested to the system.
3. The system wakes up the hypervisor (tracer)
4. The tracer understand that the file descriptor is not a virtual one, returning the control to the virtualized process.
5. The kernel returns the control to the traced process.

Pure overhead!!!

`https://github.com/virtualsquare/syscall_tracing`

Technique 4: prctl PR_SET_SYSCALL_USER_DISPATCH

prctl has a specific value that can be used to get systemcall in userspace. It is tricky!!! It is designed for foreign os emulation (i.e. Wine or Limbo).

Self-Virtualization solution

Event notification: SIGSYS+ucontext

+ Speed

+ Thread safe

- EXTREMELY arch dependent

- Tricky!!!

```
time for i in {1..1000}; do ./prctl_purelibc_virt /etc/passwd > /dev/null
    ; done
real    0m0.913s
user    0m0.602s
sys     0m0.346s
```

https://github.com/virtualsquare/syscall_tracing

Technique 5: purelibc

Purelibc is an "overlay" library. It aims to convert glibc from a libc+system interfacing library to a libc-only library. A process can trace the system call generated by itself by purelibc.

```
static sfun _native_syscall;
...
_native_syscall=_pure_start(mysc,PUREFLAG_STDALL);
```

+ Fast!!!
+ Multithread safe
+ Can be preloaded for libraries or executables
+ Almost arch indipendent
- Incomplete interface!!
- Security

```
time for i in {1..1000}; do ./puretest_virt /etc/passwd > /dev/null; done
real    0m0.968s
user    0m0.618s
sys     0m0.384s
```

https://github.com/virtualsquare/syscall_tracing

Desiderata: self virtualization

We could make the overall utilization cleaner by creating a simple modification to glibc:

1. Create a “real” systemcall interface, which executes the real system call.
2. Hijack the systemcall function executing a callback in our “tracer”.

No current support in glibc for this approach.

`https://github.com/virtualsquare/syscall_tracing`

Conclusions

- We saw different techniques that can be used to hijack system calls, currently implemented in Linux kernel and userspace (loader).
- No silver bullet.
- Some desiderata, still lacking adopted support.
- A performant layer to virtualize systemcalls will create an extremely flexible system! (more than using containers).

`https://github.com/virtualsquare/syscall_tracing`

Thank you for your attention!

`renzo@cs.unibo.it`

`dave@ihateyour.cloud` | `berardi.dav@gmail.com`