aqua

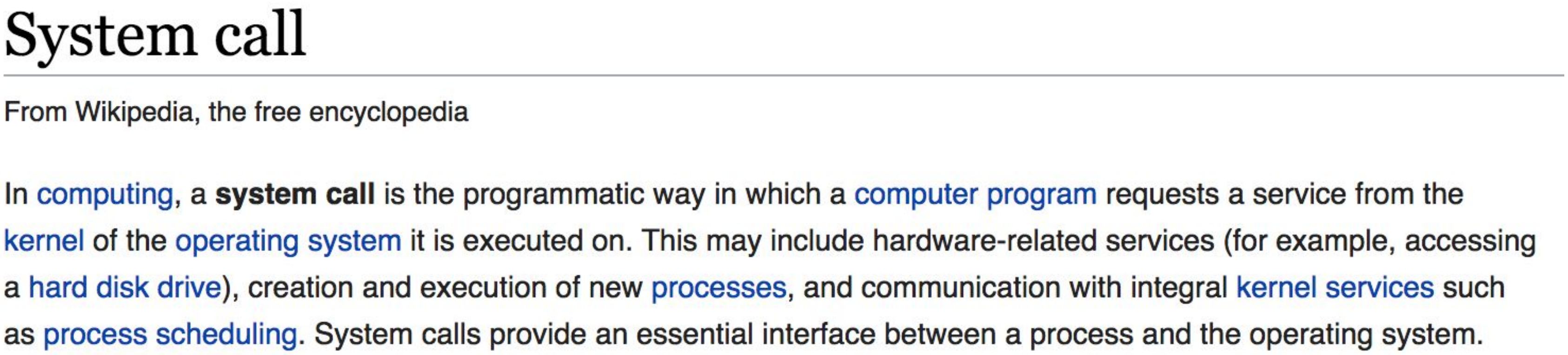
A Go programmer’s guide to syscalls

Liz Rice

@LizRice | @AquaSecTeam

Syscalls

- What are syscalls?
- How syscalls work
- Fun with ptrace
- Syscalls and security



System call

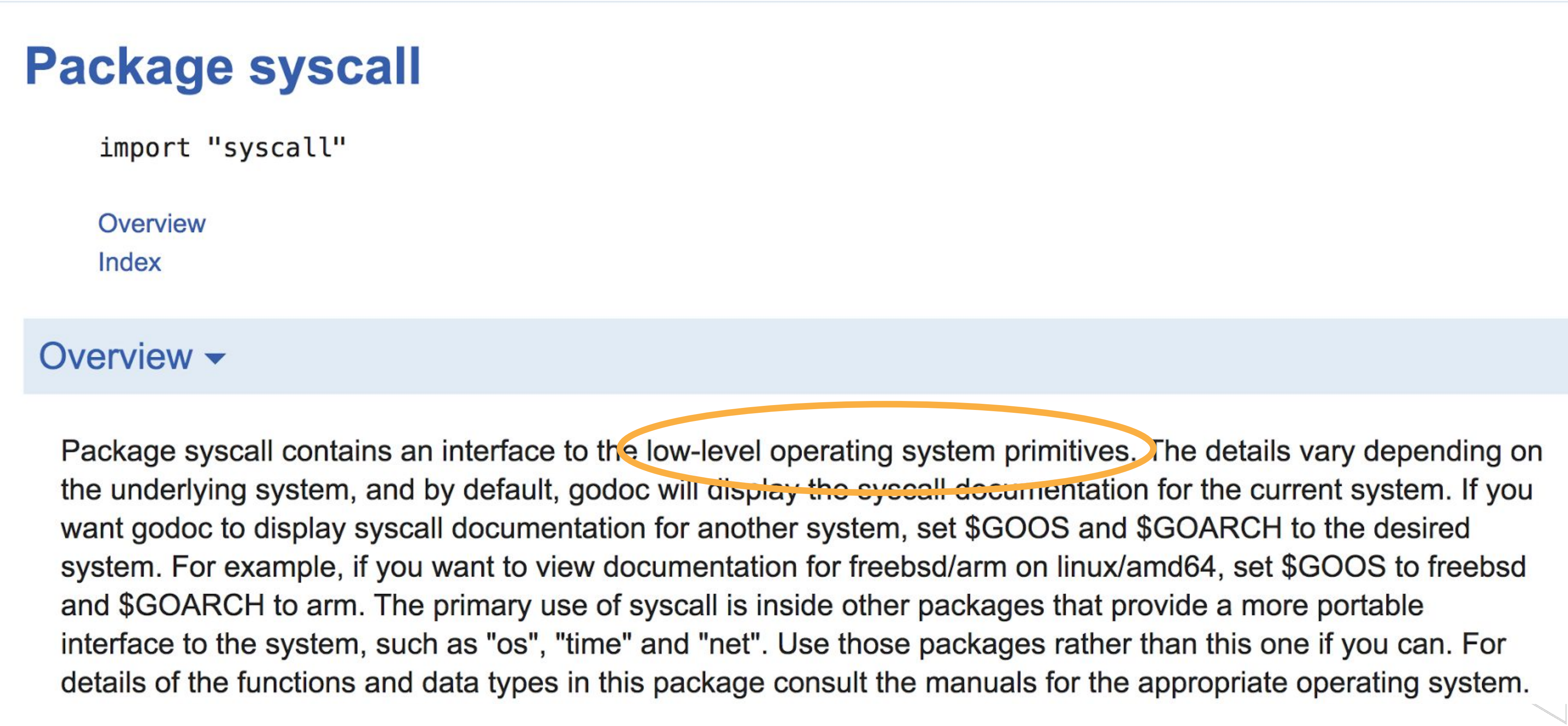
From Wikipedia, the free encyclopedia

In computing, a **system call** is the programmatic way in which a computer program requests a service from the kernel of the operating system it is executed on. This may include hardware-related services (for example, accessing a hard disk drive), creation and execution of new processes, and communication with integral kernel services such as process scheduling. System calls provide an essential interface between a process and the operating system.

What do you need syscalls for?

- Files
- Devices
- Processes
- Communications
- Time & date

See them with **strace**



Package syscall

```
import "syscall"
```

Overview

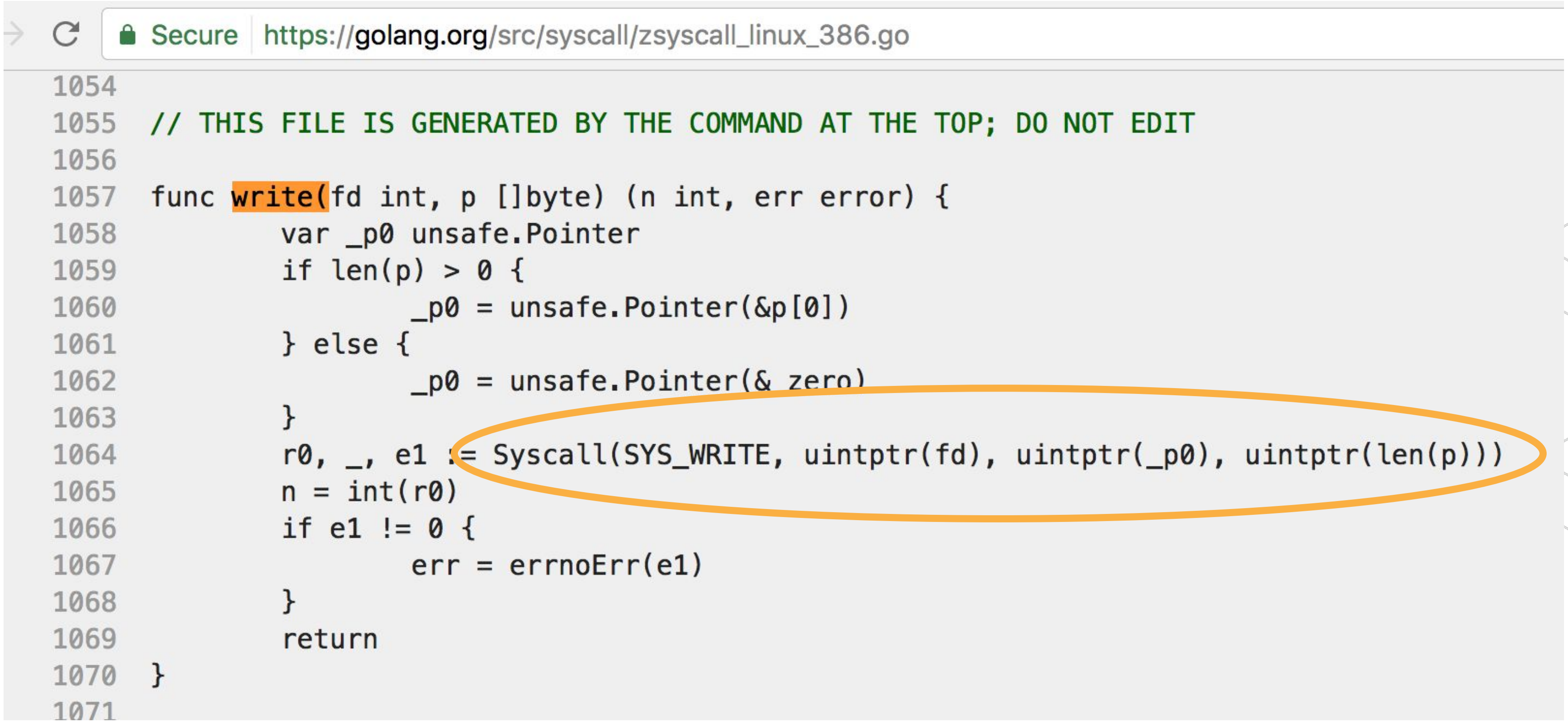
Index

Overview ▾

Package syscall contains an interface to the low-level operating system primitives. The details vary depending on the underlying system, and by default, godoc will display the syscall documentation for the current system. If you want godoc to display syscall documentation for another system, set $GOOS and $GOARCH to the desired system. For example, if you want to view documentation for freebsd/arm on linux/amd64, set $GOOS to freebsd and $GOARCH to arm. The primary use of syscall is inside other packages that provide a more portable interface to the system, such as "os", "time" and "net". Use those packages rather than this one if you can. For details of the functions and data types in this package consult the manuals for the appropriate operating system.

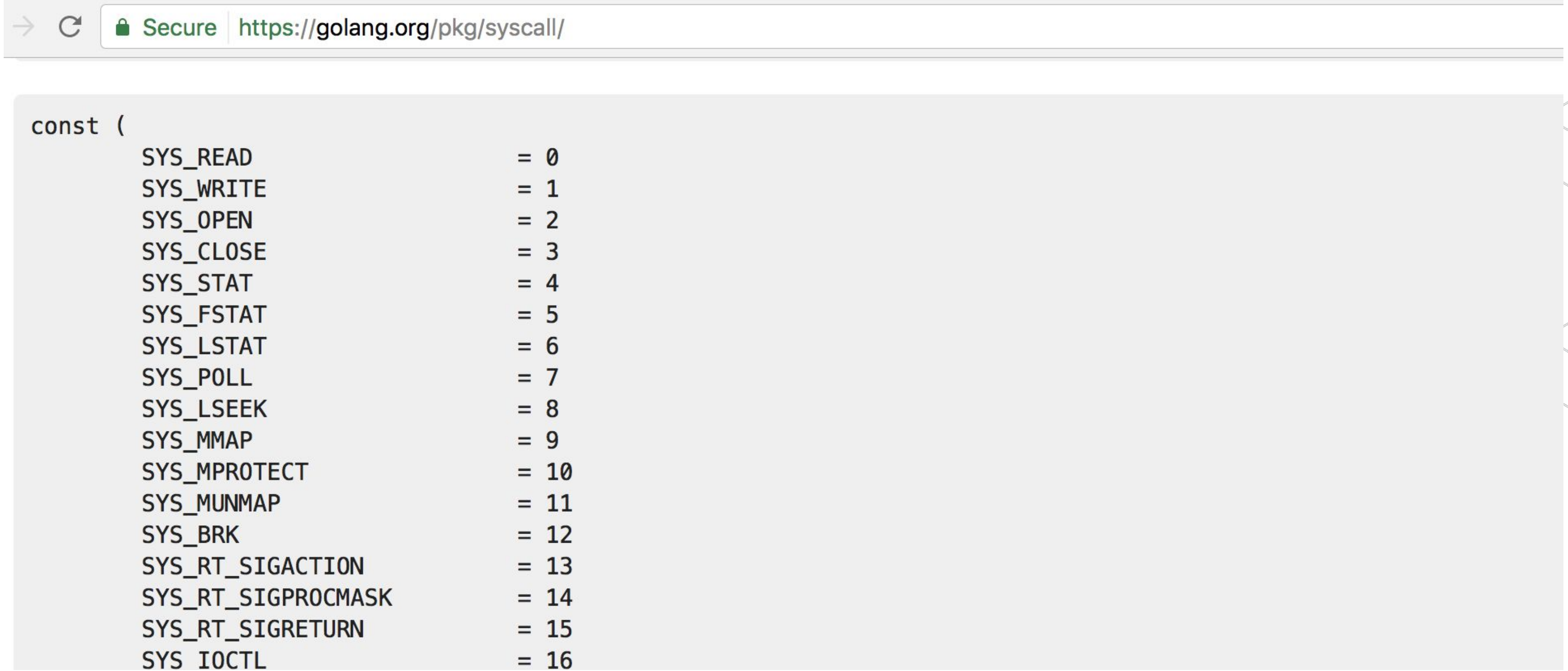
Golang syscall package

- OS-specific files
  - e.g. https://golang.org/src/syscall/syscall_linux.go
- Autogenerated files
  - e.g. https://golang.org/src/syscall/zsyscall_linux_386.go



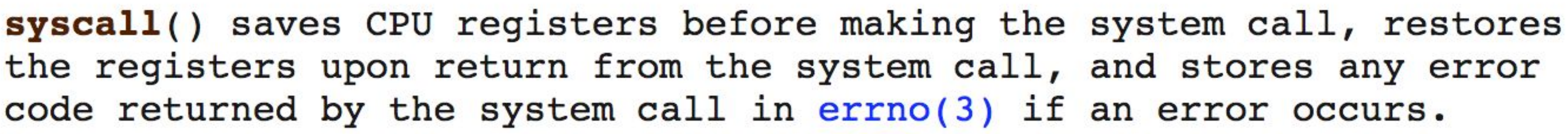
```go

// THIS FILE IS GENERATED BY THE COMMAND AT THE TOP; DO NOT EDIT

func write(fd int, p []byte) (n int, err error) {
	var _p0 unsafe.Pointer
	if len(p) > 0 {
		_p0 = unsafe.Pointer(&p[0])
	} else {
		_p0 = unsafe.Pointer(&_zero)
	}
	r0, _, e1 := Syscall(SYS_WRITE, uintptr(fd), uintptr(_p0), uintptr(len(p)))
	n = int(r0)
	if e1 != 0 {
		err = errnoErr(e1)
	}
	return
}

```

Syscall codes



```
const (
        SYS_READ                   = 0
        SYS_WRITE                  = 1
        SYS_OPEN                   = 2
        SYS_CLOSE                  = 3
        SYS_STAT                   = 4
        SYS_FSTAT                  = 5
        SYS_LSTAT                  = 6
        SYS_POLL                   = 7
        SYS_LSEEK                  = 8
        SYS_MMAP                   = 9
        SYS_MPROTECT               = 10
        SYS_MUNMAP                 = 11
        SYS_BRK                    = 12
        SYS_RT_SIGACTION           = 13
        SYS_RT_SIGPROCMASK         = 14
        SYS_RT_SIGRETURN           = 15
        SYS_IOCTL                  = 16
```

Making a syscall

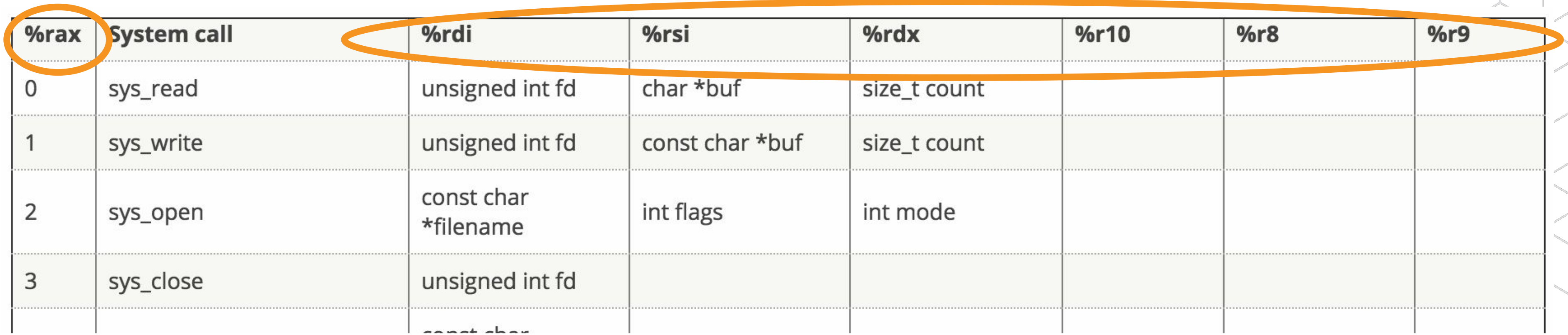


syscall() saves CPU registers before making the system call, restores the registers upon return from the system call, and stores any error code returned by the system call in errno(3) if an error occurs.

Making a syscall

- Set registers up with syscall ID (%rax on x86) & parameters
- Trap - transition to kernel - run syscall code
- Result returned in %rax (x86)

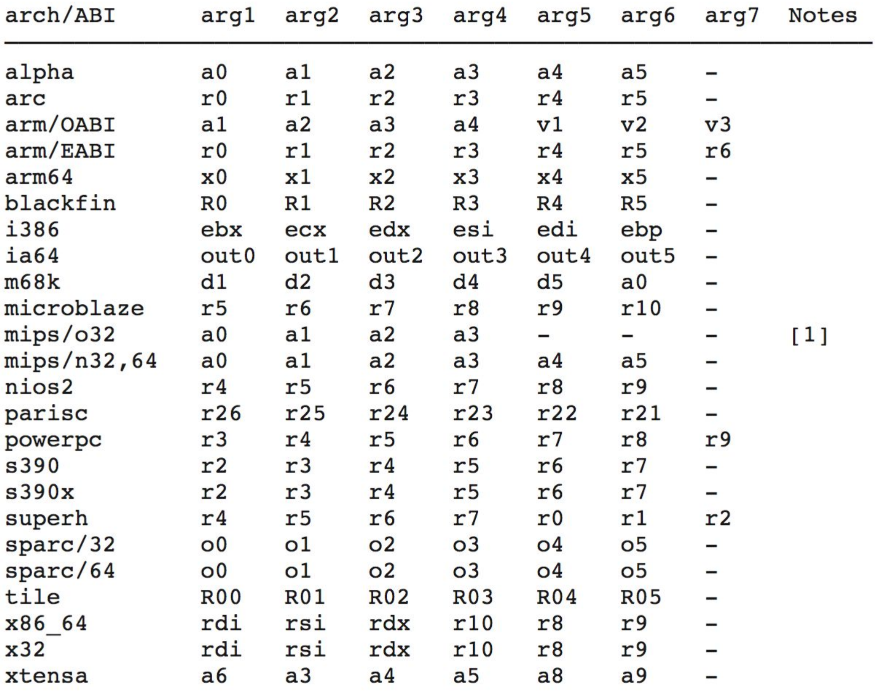
x86 64 table from blog.rchapman.org



%rax	System call	%rdi	%rsi	%rdx	%r10	%r8	%r9
0	sys_read	unsigned int fd	char *buf	size_t count			
1	sys_write	unsigned int fd	const char *buf	size_t count			
2	sys_open	const char *filename	int flags	int mode			
3	sys_close	unsigned int fd					
		const char					

Making a syscall

- Different architectures, same approach



arch/ABI	arg1	arg2	arg3	arg4	arg5	arg6	arg7	Notes
alpha	a0	a1	a2	a3	a4	a5	-	
arc	r0	r1	r2	r3	r4	r5	-	
arm/OABI	a1	a2	a3	a4	v1	v2	v3	
arm/EABI	r0	r1	r2	r3	r4	r5	r6	
arm64	x0	x1	x2	x3	x4	x5	-	
blackfin	R0	R1	R2	R3	R4	R5	-	
i386	ebx	ecx	edx	esi	edi	ebp	-	
ia64	out0	out1	out2	out3	out4	out5	-	
m68k	d1	d2	d3	d4	d5	a0	-	
microblaze	r5	r6	r7	r8	r9	r10	-	
mips/o32	a0	a1	a2	a3	-	-	-	[1]
mips/n32,64	a0	a1	a2	a3	a4	a5	-	
nios2	r4	r5	r6	r7	r8	r9	-	
parisc	r26	r25	r24	r23	r22	r21	-	
powerpc	r3	r4	r5	r6	r7	r8	r9	
s390	r2	r3	r4	r5	r6	r7	-	
s390x	r2	r3	r4	r5	r6	r7	-	
superh	r4	r5	r6	r7	r0	r1	r2	
sparc/32	o0	o1	o2	o3	o4	o5	-	
sparc/64	o0	o1	o2	o3	o4	o5	-	
tile	R00	R01	R02	R03	R04	R05	-	
x86_64	rdi	rsi	rdx	r10	r8	r9	-	
x32	rdi	rsi	rdx	r10	r8	r9	-	
xtensa	a6	a3	a4	a5	a8	a9	-	

Syscalls as a portability layer

- Implement syscalls interface = emulate Linux
  - Just one syscall function - can implement a subset
- Bash shell on Windows

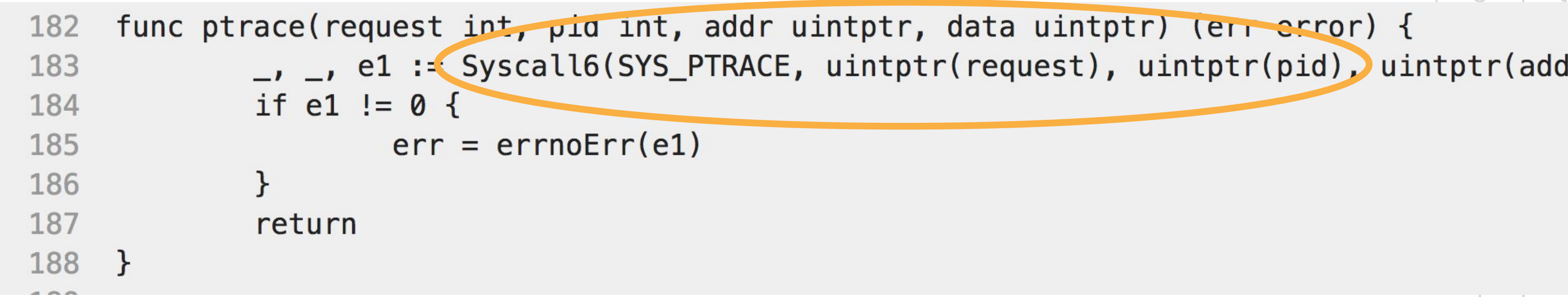
What syscalls are being called?

- Linux *strace*
  - *strace -c* for a summary

ptrace

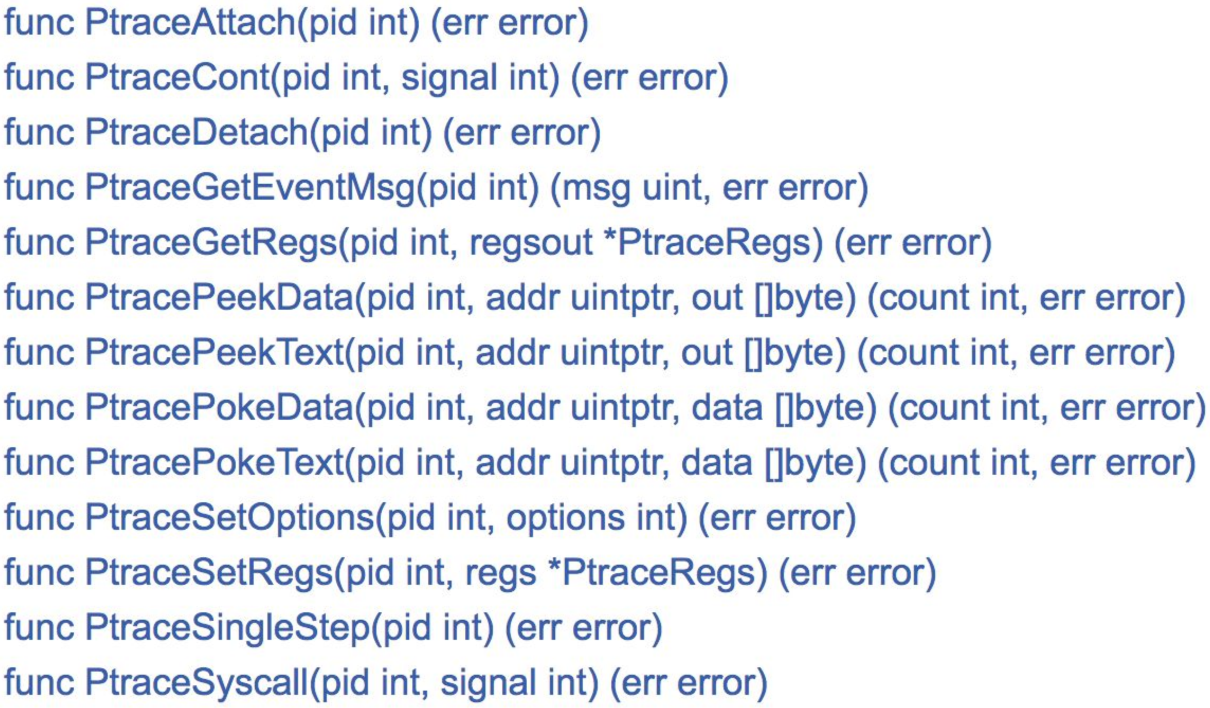
The **ptrace**() system call provides a means by which one process (the "tracer") may observe and control the execution of another process (the "tracee"), and examine and change the tracee's memory and registers. It is primarily used to implement breakpoint debugging and system call tracing.

ptrace



```
func ptrace(request int, pid int, addr uintptr, data uintptr) (err error) {
        _, _, e1 := Syscall6(SYS_PTRACE, uintptr(request), uintptr(pid), uintptr(add
        if e1 != 0 {
                err = errnoErr(e1)
        }
        return
}
```

ptrace



```
func PtraceAttach(pid int) (err error)
func PtraceCont(pid int, signal int) (err error)
func PtraceDetach(pid int) (err error)
func PtraceGetEventMsg(pid int) (msg uint, err error)
func PtraceGetRegs(pid int, regsout *PtraceRegs) (err error)
func PtracePeekData(pid int, addr uintptr, out []byte) (count int, err error)
func PtracePeekText(pid int, addr uintptr, out []byte) (count int, err error)
func PtracePokeData(pid int, addr uintptr, data []byte) (count int, err error)
func PtracePokeText(pid int, addr uintptr, data []byte) (count int, err error)
func PtraceSetOptions(pid int, options int) (err error)
func PtraceSetRegs(pid int, regs *PtraceRegs) (err error)
func PtraceSingleStep(pid int) (err error)
func PtraceSyscall(pid int, signal int) (err error)
```

Let’s build our own strace!

With hat tips to @mlowicki and @nelhage

Catching system calls with ptrace

- PTRACE_SYSCALL

  Restart the stopped tracee … but arrange for the tracee to be stopped at the next entry to or exit from a system call

  From the tracer's perspective, the tracee will appear to have been stopped by receipt of a SIGTRAP.

Two stops for PTRACE_SYSCALL

- The tracee enters syscall-enter-stop just prior to entering any system call … the tracee enters syscall-exit-stop when the system call is finished
- Syscall-enter-stop and syscall-exit-stop are indistinguishable from each other by the tracer.
- The tracer needs to keep track of the sequence of ptrace-stops

Syscalls and security

Security profiles & microservices

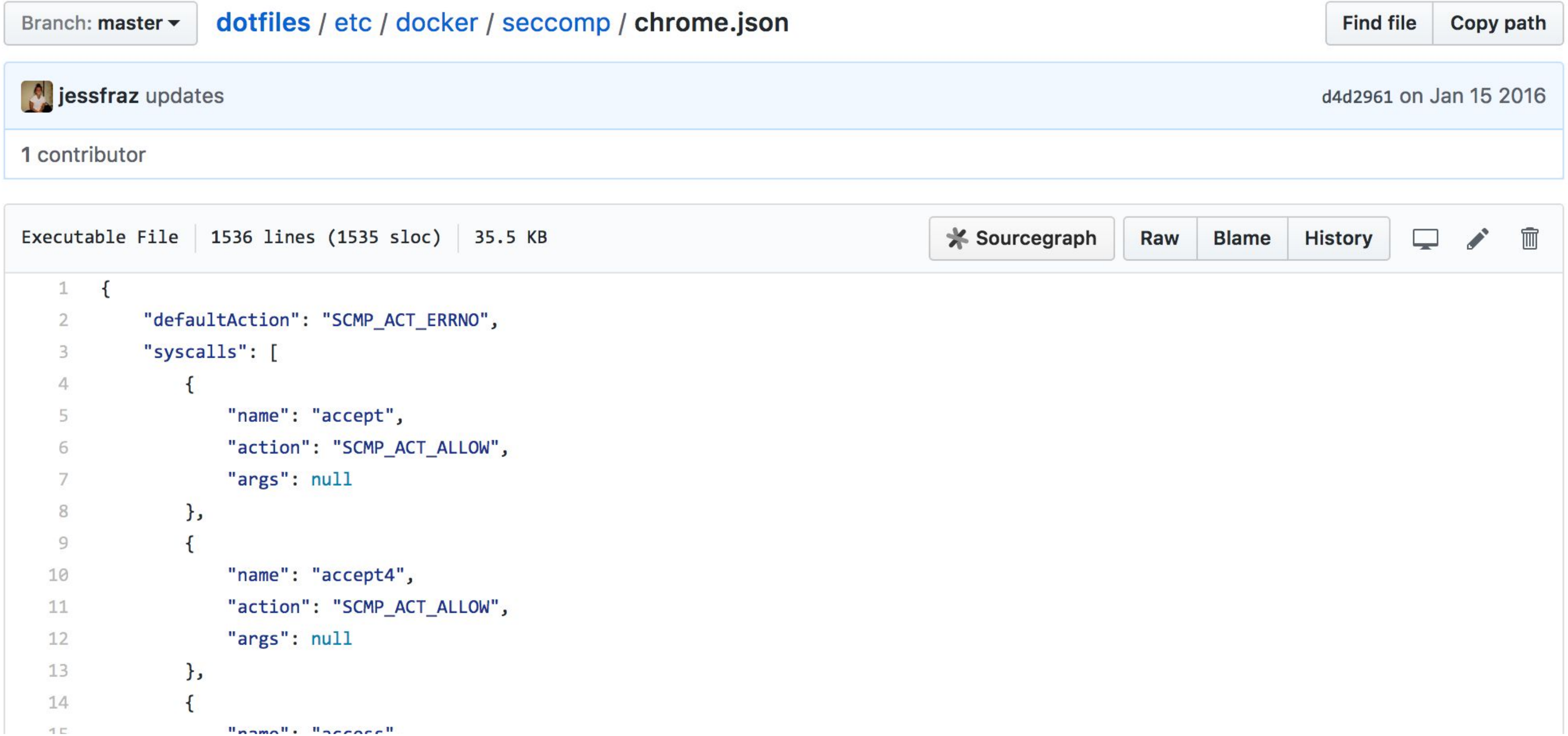
- Microservice only performs small set of functions
- “Least privilege”

Security profiles & microservices

- Seccomp restricts permitted syscalls

```
$ docker run \
--security-opt seccomp=/path/sc_profile.json hello-world
```

Security profiles and containers



dotfiles / etc / docker / seccomp / chrome.json

jessfraz updates d4d2961 on Jan 15 2016

1 contributor

Executable File | 1536 lines (1535 sloc) | 35.5 KB

```
{
    "defaultAction": "SCMP_ACT_ERRNO",
    "syscalls": [
        {
            "name": "accept",
            "action": "SCMP_ACT_ALLOW",
            "args": null
        },
        {
            "name": "accept4",
            "action": "SCMP_ACT_ALLOW",
            "args": null
        },
        {
            "name": "access",
```

Let’s try it!

Syscalls

- Your interface into the kernel
  - even if you’re not using them directly
- Portability
  - running Linux on different hardware
  - emulation
- Strace and ptrace
  - see / manipulate syscalls
- Security
  - limiting which syscalls are permitted

code will be at github.com/lizrice/strace-from-scratch

@LizRice | @AquaSecTeam