Anatomy of a system call, part 1

lwn.net/Articles/604287/

July 9, 2014

This article was contributed by David Drysdale

System calls are the primary mechanism by which user-space programs interact with the Linux kernel. Given their importance, it's not surprising to discover that the kernel includes a wide variety of mechanisms to ensure that system calls can be implemented generically across architectures, and can be made available to user space in an efficient and consistent way.

I've been working on getting FreeBSD's Capsicum security framework onto Linux and, as this involves the addition of several new system calls (including the slightly unusual `execveat()` system call), I found myself investigating the details of their implementation. As a result, this is the first of a pair of articles that explore the details of the kernel's implementation of system calls (or syscalls). In this article we'll focus on the mainstream case: the mechanics of a normal syscall (`read()`), together with the machinery that allows x86_64 user programs to invoke it. The second article will move off the mainstream case to cover more unusual syscalls, and other syscall invocation mechanisms.

System calls differ from regular function calls because the code being called is in the kernel. Special instructions are needed to make the processor perform a transition to ring 0 (privileged mode). In addition, the kernel code being invoked is identified by a syscall number, rather than by a function address.

Defining a syscall with `SYSCALL_DEFINEn()`

The `read()` system call provides a good initial example to explore the kernel's syscall machinery. It's implemented in `fs/read_write.c`, as a short function that passes most of the work to `vfs_read()`. From an invocation standpoint the most interesting aspect of this code is way the function is defined using the `SYSCALL_DEFINE3()` macro. Indeed, from the code, it's not even immediately clear what the function is called.

```
SYSCALL_DEFINE3(read, unsigned int, fd, char __user *, buf, size_t, count)
{
 struct fd f = fdget_pos(fd);
 ssize_t ret = -EBADF;
 /* ... */
```

These `SYSCALL_DEFINEn()` macros are the standard way for kernel code to define a system call, where the *n* suffix indicates the argument count. The definition of these macros (in `include/linux/syscalls.h`) gives two distinct outputs for each system call.

```
SYSCALL_METADATA(_read, 3, unsigned int, fd, char __user *, buf, size_t, count)
__SYSCALL_DEFINEx(3, _read, unsigned int, fd, char __user *, buf, size_t, count)
{
 struct fd f = fdget_pos(fd);
 ssize_t ret = -EBADF;
 /* ... */
```

The first of these, `SYSCALL_METADATA()`, builds a collection of metadata about the system call for tracing purposes. It's only expanded when `CONFIG_FTRACE_SYSCALLS` is defined for the kernel build, and its expansion gives boilerplate definitions of data that describes the syscall and its parameters. (A separate page describes these definitions in more detail.)

The `__SYSCALL_DEFINEx()` part is more interesting, as it holds the system call implementation. Once the various layers of macros and GCC type extensions are expanded, the resulting code includes some interesting features:

```
asmlinkage long sys_read(unsigned int fd, char __user * buf, size_t count)
 __attribute__((alias(__stringify(SyS_read))));

static inline long SYSC_read(unsigned int fd, char __user * buf, size_t count);
asmlinkage long SyS_read(long int fd, long int buf, long int count);

asmlinkage long SyS_read(long int fd, long int buf, long int count)
{
 long ret = SYSC_read((unsigned int) fd, (char __user *) buf, (size_t) count);
 asmlinkage_protect(3, ret, fd, buf, count);
 return ret;
}

static inline long SYSC_read(unsigned int fd, char __user * buf, size_t count)
{
 struct fd f = fdget_pos(fd);
 ssize_t ret = -EBADF;
 /* ... */
```

First, we notice that the system call implementation actually has the name `SYSC_read()`, but is `static` and so is inaccessible outside this module. Instead, a wrapper function, called `SyS_read()` and aliased as `sys_read()`, is visible externally. Looking closely at those aliases, we notice a difference in their parameter types — `sys_read()` expects the explicitly declared types (e.g. `char __user *` for the second argument), whereas `SyS_read()` just expects a bunch of (long) integers. Digging into the history of this, it turns out that the `long` version ensures that 32-bit values are correctly sign-extended for some 64-bit kernel platforms, preventing a historical vulnerability.

The last things we notice with the `SyS_read()` wrapper are the `asmlinkage` directive and `asmlinkage_protect()` call. The Kernel Newbies FAQ helpfully explains that `asmlinkage` means the function should expect its arguments on the stack rather than in registers, and the generic definition of `asmlinkage_protect()` explains that it's used to prevent the compiler from assuming that it can safely reuse those areas of the stack.

To accompany the definition of `sys_read()` (the variant with accurate types), there's also a declaration in `include/linux/syscalls.h`, and this allows other kernel code to call into the system call implementation directly (which happens in half a dozen places). Calling system calls directly from elsewhere in the kernel is generally discouraged and is not often seen.

Syscall table entries

Hunting for callers of `sys_read()` also points the way toward how user space reaches this function. For "generic" architectures that don't provide an override of their own, the `include/uapi/asm-generic/unistd.h` file includes an entry referencing `sys_read`:

```
#define __NR_read 63
__SYSCALL(__NR_read, sys_read)
```

This defines the generic syscall number `__NR_read` (63) for `read()`, and uses the `__SYSCALL()` macro to associate that number with `sys_read()`, in an architecture-specific way. For example, arm64 uses the `asm-generic/unistd.h` header file to fill out a table that maps syscall numbers to implementation function pointers.

However, we're going to concentrate on the x86_64 architecture, which does not use this generic table. Instead, x86_64 defines its own mappings in `arch/x86/syscalls/syscall_64.tbl`, which has an entry for `sys_read()`:

```
0 common read   sys_read
```

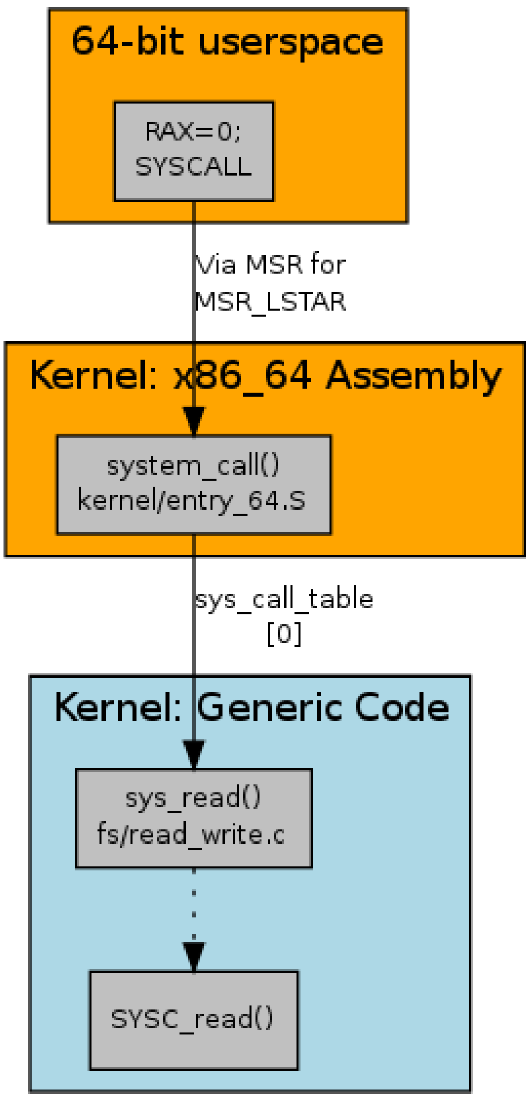
This indicates that `read()` on x86_64 has syscall number 0 (not 63), and has a `common` implementation for both of the ABIs for x86_64, namely `sys_read()`. (The different ABIs will be discussed in the second part of this series.) The `syscalltbl.sh` script generates `arch/x86/include/generated/asm/syscalls_64.h` from the `syscall_64.tbl` table, specifically generating an invocation of the `__SYSCALL_COMMON()` macro for `sys_read()`. This header file is used, in turn, to populate the syscall table, `sys_call_table`, which is the key data structure that maps syscall numbers to `sys_`*`name`*`()` functions.

x86_64 syscall invocation

Now we will look at how user-space programs invoke the system call. This is inherently architecture-specific, so for the rest of this article we'll concentrate on the x86_64 architecture (other x86 architectures will be examined in the second article of the series). The invocation process also involves a few steps, so a clickable diagram, seen at left, may help with the navigation.

In the previous section, we discovered a table of system call function pointers; the table for x86_64 looks something like the following (using a GCC extension for array initialization that ensures any missing entries point to `sys_ni_syscall()`):

```
    asmlinkage const sys_call_ptr_t
sys_call_table[__NR_syscall_max+1] = {
     [0 ... __NR_syscall_max] =
&sys_ni_syscall,
     [0] = sys_read,
     [1] = sys_write,
     /*... */
    };
```



For 64-bit code, this table is accessed from arch/x86/kernel/entry_64.S, from the system_call assembly entry point; it uses the RAX register to pick the relevant entry in the array and then calls it. Earlier in the function, the SAVE_ARGS macro pushes various registers onto the stack, to match the asmlinkage directive we saw earlier.

Moving outwards, the system_call entry point is itself referenced in syscall_init(), a function that is called early in the kernel's startup sequence:

```
    void syscall_init(void)
    {
     /*
      * LSTAR and STAR live in a bit
strange symbiosis.
      * They both write to the same
internal register. STAR allows to
      * set CS/DS but only a 32bit
target. LSTAR sets the 64bit rip.
      */
     wrmsrl(MSR_STAR,  ((u64)__USER32_CS)<<48  | ((u64)__KERNEL_CS)<<32);
     wrmsrl(MSR_LSTAR, system_call);
     wrmsrl(MSR_CSTAR, ignore_sysret);
     /* ... */
```

The wrmsrl instruction writes a value to a model-specific register; in this case, the address of the general system_call syscall handling function is written to register MSR_LSTAR (0xc0000082), which is the x86_64 model-specific register for handling the SYSCALL instruction.

And this gives us all we need to join the dots from user space to the kernel code. The standard ABI for how x86_64 user programs invoke a system call is to put the system call number (0 for read) into the RAX register, and the other parameters into specific registers (RDI, RSI, RDX for the first 3 parameters), then issue the SYSCALL instruction. This instruction causes the processor to transition to ring 0 and invoke the code referenced by the MSR_LSTAR model-specific register — namely system_call. The system_call code pushes the registers onto the kernel stack, and calls the function pointer at entry RAX in the `sys_call_table` table — namely `sys_read()`, which is a thin, `asmlinkage` wrapper for the real implementation in `SYSC_read()`.

Now that we've seen the standard implementation of system calls on the most common platform, we're in a better position to understand what's going on with other architectures, and with less-common cases. That will be the subject of the second article in the series.
