Isolation and System Calls

Dongyoon Lee

Summary of last lectures

- Getting, building, and exploring the Linux kernel
  - `git`, `tig`, `make`, `make modules`, `make modules_install`, `make install`, `vim`, `emacs`, LXR, `cscope`, `ctags`, `tmux`
- *Don’t try to master them at once. Instead gradually get used to them.*

How to read kernel code (top-down)

- **E.g., ext4 file system**

1. General understanding on file systems in OS ← any OS text book
2. File system in Linux kernel ← Ch. 13 in our text book
3. Check kernel Documentation and Ext4 on-disk layout
4. Read the ext4 kernel code
    - Module by module (e.g., dir, file, block management)
    - Start from a system call (e.g., how write() is implemented?)
5. Search LWN to check the latest changes → E.g., ext4 encryption support

How to read kernel code (bottom-up)

- **Use function tracer**
  - `ftrace`: function tracer framework
  - `perf tools`: `ftrace` front end
- `kernel/funcgraph`
  - trace a graph of kernel function calls, showing children and times

```
# ./funcgraph -Htp 5363 vfs_read
Tracing "vfs_read" for PID 5363... Ctrl-C to end.
# tracer: function_graph
#
#     TIME        CPU  DURATION            FUNCTION CALLS
#      |          |     |   |               |   |   |   |
 1728.478683 |   0)                 |  vfs_read() {
 1728.478690 |   0)                 |    rw_verify_area() {
 1728.478691 |   0)                 |      security_file_permission() {
 1728.478692 |   0)                 |        selinux_file_permission() {
```

How to navigate kernel code

```
$ KBUILD_ABS_SRCTREE=1 make ARCH=x86_64 cscope tags -j2
  # KBUILD_ABS_SRCTREE=1 # use absolute path
  # ARHC=x86_64          # select CPU architcture
  # cscope               # build cscope database
  # tags                 # build ctag database
  # -j2                  # concurrently index source code using 2 CPUs

$ vim
  # :tag <symbol>        # search symbol definition
  # :cs find s <symbol>  # find uses of symbol
  # Ctrl-]               # search symbol definition on the cursor
  # Ctrl-t               # returning after a tag jump
  # :bp :bn              # nativate back and forth between files
```

Reference: [Vim Tips Wiki: Browsing programs with tag]()

What's operating system (again)?

- OS design focuses on:
  - **Abstracting** the hardware for convenience and portability
  - **Multiplexing** the hardware among multiple applications
  - **Isolating** applications that might contain bugs
  - Allowing **sharing** among applications

Today: isolation and system calls

- How to isolate user applications from the kernel?
- How to safely access the kernel from user application?

The unit of isolation: “process”

- Prevent process X from wrecking or spying on process Y
  - (e.g., memory, cpu, FDs, resource exhaustion)
- Prevent a process from wrecking the operating system itself
  - (i.e. from preventing kernel from enforcing isolation)
- In the face of bugs or malice
  - (e.g. a bad process may try to trick the h/w or kernel)

Isolation mechanisms in operating systems

1. User/kernel mode flag (aka ring)
2. Address spaces (later)
3. Timeslicing (later)
4. System call interface

Hardware isolation in x86 (aka ring)

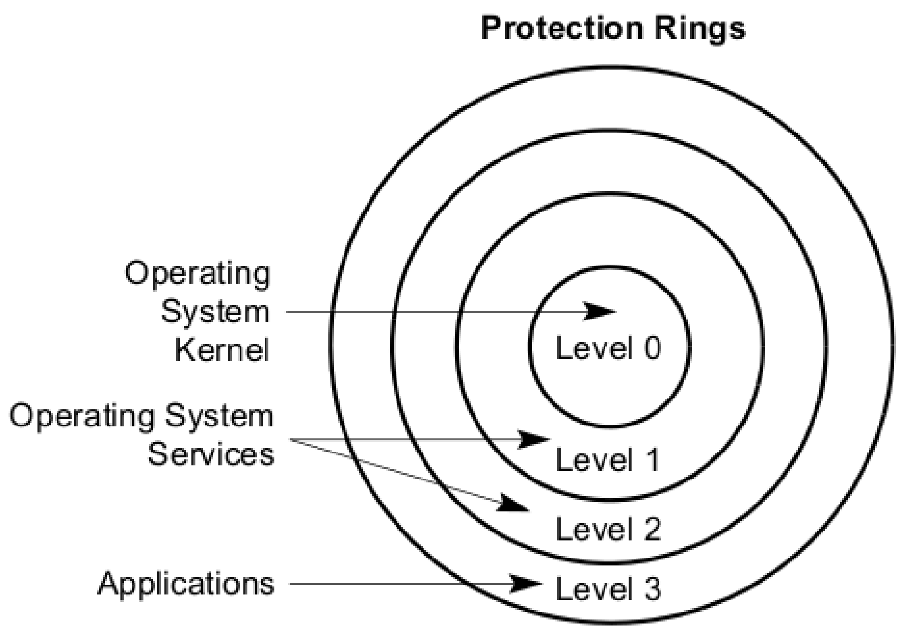


Figure 5-3. Protection Rings

- **Q: How isolation is enforced in x86?**

Segmentation in x86

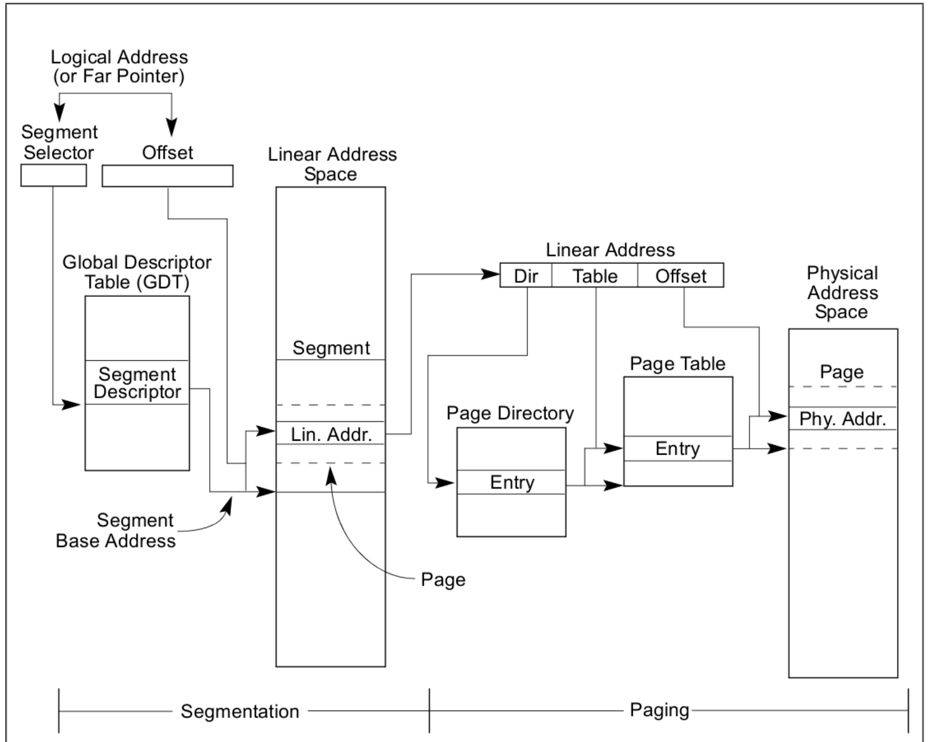


Figure 3-1. Segmentation and Paging

Segmentation in x86_64

3.2.4 Segmentation in IA-32e Mode

In IA-32e mode of Intel 64 architecture, the effects of segmentation depend on whether the processor is running in compatibility mode or 64-bit mode. In compatibility mode, segmentation functions just as it does using legacy 16-bit or 32-bit protected mode semantics.

In 64-bit mode, segmentation is generally (but not completely) disabled, creating a flat 64-bit linear-address space. The processor treats the segment base of CS, DS, ES, SS as zero, creating a linear address that is equal to the effective address. The FS and GS segments are exceptions. These segment registers (which hold the segment base) can be used as additional base registers in linear address calculations. They facilitate addressing local data and certain operating system data structures.

Note that the processor does not perform segment limit checks at runtime in 64-bit mode.

- Segmentation in 64-bit mode is generally disabled
- Two important features in segmentation are:
  - checking privilege level → ring 0, 3
  - implementing thread-local storage (TLS) → `fs`, `gs`

Privilege levels of a segment

- CPL (current privilege level)
    - the privilege level of currently executing program
    - bits 0 and 1 in the `%cs` register
- RPL (requested privilege level)
    - an override privilege level that is assigned to a segment selector
    - a segment selector is a part (16-bit) of segment registers (e.g., `ds`, `fs`), which is an index of a segment descriptor and RPL
- DPL (descriptor privilege level)
    - the privilege level of a segment

How isolation is enforced in x86?

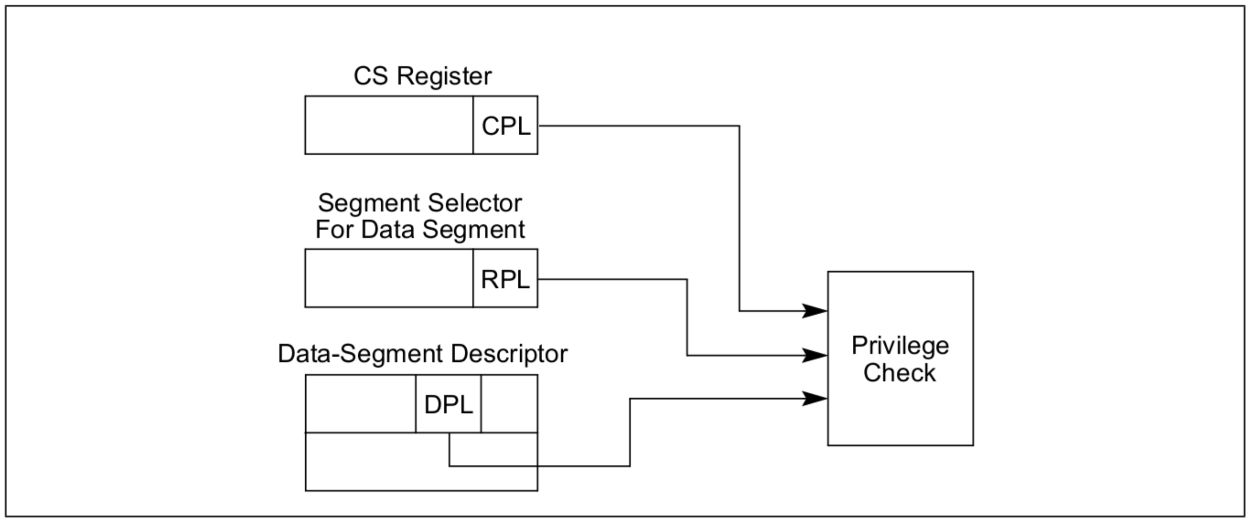


Figure 5-4. Privilege Check for Data Access

- Access is granted if `DPL` >= `RPL` and `DPL` >= `CPL`

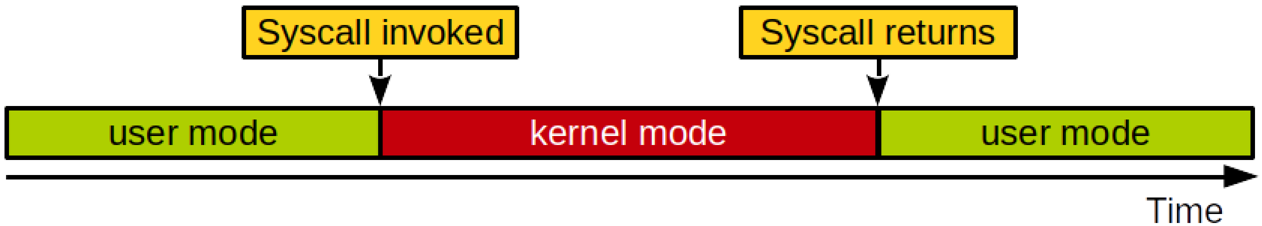
What does “ring 0” protect?

- Protects everything relevant to isolation
    - writes to `%cs` (to defend CPL)
    - every memory read/write
    - I/O port accesses
    - control register accesses (`eflags`, `%fs`, `%gs`, …)

How to switch b/w rings (ring 0 ↔ ring 3)?

- Controlled transfer: system call
  - `int`, `sysenter` or `syscall` instruction set CPL to 0; change to KERNEL_CS and KERNEL_DS segments
  - set CPL to 3 before going back to user space; change to USER_CS and USER_DS segments
- **Q: How to systematically manage such interfaces?**

System call



- **One and only way for user-space application to enter the kernel to request OS services and privileged operations such as accessing the hard ware**
  - A layer between the hardware and user-space processes
  - An abstract hardware interface for user-space
  - Ensure system security and stability

Examples of system calls

- Process management/scheduling: `fork`, `exit`, `execve`, `nice`, `{get|set}priority`, `{get|set}pid`
- Memory management: `brk`, `mmap`
- File system: `open`, `read`, `write`, `lseek`, `stat`
- Inter-Process Communication: `pipe`, `shmget`
- Time management: `{get|set}timeofday`
- Others: `{get|set}uid`, `connect`
- **Q: Where are system call implementations in Linux kernel?**

Syscall table and syscall identifier

- The syscall table for x86_64 architecture
    - `linux/arch/x86/entry/syscalls/syscall_64.tbl`
- Syscall ID: unique integer ← sequentially assigned

```
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
#
# The abi is "common", "64" or "x32" for this file.
0   common  read            sys_read
1   common  write           sys_write
2   common  open            sys_open
...
332 common  statx           sys_statx
```

`sys_call_table`

- `syscall_64.tbl` is translated to an array of function pointers, `sys_call_table`, upon kernel build
  - `linux/arch/x86/entry/syscalls/syscalltbl.sh`

```
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,
    [2] = sys_open,
    ...
    ...
    ...
};
```

Syscall implementation (e.g., `read`)

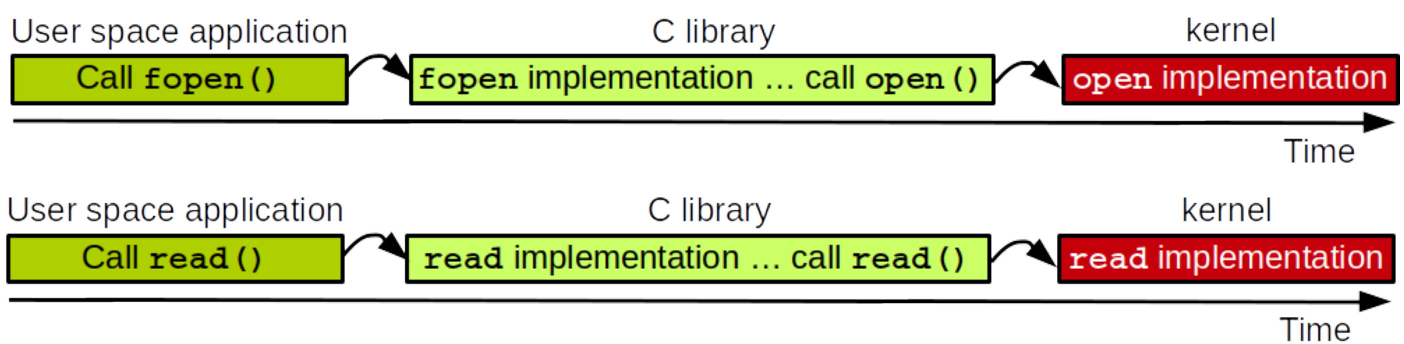
```
# linux/arch/x86/entry/syscalls/syscall_64.tbl
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
0   common  read            sys_read
1   common  write           sys_write
```

```
/* linux/fs/read_write.c */
/* ssize_t write(int fd, const void *buf, size_t count); */
SYSCALL_DEFINE3(write, unsigned int, fd, const char __user *, buf,
        size_t, count)
{
    return ksys_write(fd, buf, count);
}
```

- **Q: How is a system call invoked?**

Invoking a syscall from a user space

- Syscalls are rarely invoked directly
  - Most of them are wrapped by the C library (`libc` , POSIX API)



Invoking a syscall from an application

- A syscall can be directly called through `syscall`
  - See `man syscall` → C library uses `syscall`

```
#include <unistd.h>
#include <sys/syscall.h>   /* for SYS_xxx definitions */

int main(void)
{
    char    msg[] = "Hello, world!\n";
    ssize_t bytes_written;

    /* ssize_t write(int fd, const void *msg, size_t count);      */
    bytes_written = syscall(1, 1, msg, 14);
    /*                      \  \                                  */
    /*                       \  +-- fd: standard output           */
    /*                        +-- write syscall id (or SYS_write) */
    return 0;
}
```

Invoking a syscall from an application

- x86_64 architecture has a `syscall` instruction

```
.data

msg:
    .ascii "Hello, world!\n"
    len = . - msg

.text
    .global _start

_start:
    mov  $1, %rax      # syscall id: write
    mov  $1, %rdi      # 1st arg: fd (standard output)
    mov  $msg, %rsi    # 2nd arg: msg
    mov  $len, %rdx    # 3rd arg: length of msg
    syscall            # switch from user space to kernel space

    mov  $60, %rax     # syscall id: exit
    xor  %rdi, %rdi    # 1st arg: 0
    syscall            # switch from user space to kernel space
```

Transition from a user space to kernel space

- x86 instruction for system call
  - `int $0x80` : raise a software interrupt 128 (old)
  - `sysenter` : fast system call (x86_32)
  - `syscall` : fast system call (x86_64)
- Passing a syscall ID and parameters
  - syscall ID: `%rax`
  - parameters (x86_64): `rdi` , `rsi` , `rdx` , `r10` , `r8` and `r9`

Handling the syscall interrupt

- The kernel syscall interrupt handler, system call handler
  - `entry_SYSCALL_64` at [linux/arch/x86/entry/syscall/entry_64.S]() (`do_syscall_64`)
- `entry_SYSCALL_64` is registered at CPU initialization time
  - A handler of `syscall` is specified at a `IA32_LSTAR MSR` register
  - The address of `IA32_LSTAR MSR` is set to `entry_SYSCALL_64` at boot time: [syscall_init() at linux/arch/x86/kernel/cpu/common.c]()

Handling the syscall interrupt

- `entry_SYSCALL_64` invokes the entry function for the syscall ID
    - `call do_syscall_64`
    - `regs->ax = sys_call_table[nr](regs);`

```
# linux/arch/x86/entry/syscalls/syscall_64.tbl
# 64-bit system call numbers and entry vectors
#
# The format is:
# <number> <abi> <name> <entry point>
0   common  read            sys_read
1   common  write           sys_write
```

```
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,
```

Returning from the syscall interrupt

- x86 instruction for system call
  - `iret`: interrupt return (x86-32 bit, old)
  - `sysexit`: fast return from fast system call(x86-32 bit)
  - `sysret`: return from fast system call (x86-64 bit)

Syscall example: `gettimeofday`

- `man gettimeofday`

```
NAME
       gettimeofday, settimeofday - get / set time

SYNOPSIS
       #include <sys/time.h>

       int gettimeofday(struct timeval *tv, struct timezone *tz);

       int settimeofday(const struct timeval *tv, const struct timezone *tz);

DESCRIPTION
       The functions gettimeofday() and settimeofday() can get and the time
       as well as a timezone.  The tv argument is a struct timeval (as specified
       in <sys/time.h>):

           struct timeval {
               time_t      tv_sec;     /* seconds */
               suseconds_t tv_usec;    /* microseconds */
           };
```

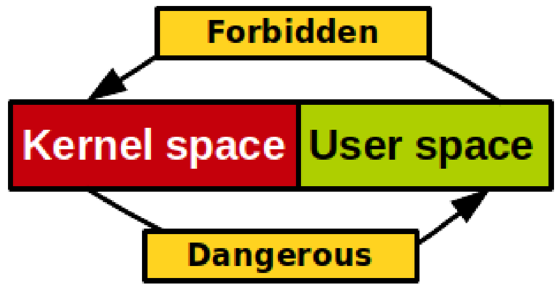
Example C code

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

int main(void)
{
    struct timeval tv;
    int ret;

    ret = gettimeofday(&tv, NULL);
    if(ret == -1)
    {
        perror("gettimeofday");
        return EXIT_FAILURE;
    }

    printf("Local time:\n");
    printf(" sec:%lu\n", tv.tv_sec);
    printf(" usec:%lu\n", tv.tv_usec);

    return EXIT_SUCCESS;
}
```

Kernel implementation: `sys_gettimeofday`

```
/* linux/kernel/time/time.c */
/* SYSCALL_DEFINE2: a macro to define a syscall with two parameters */
SYSCALL_DEFINE2(gettimeofday, struct timeval __user *, tv,
        struct timezone __user *, tz) /* __user: user-space address */
{
    if (likely(tv != NULL)) { /* likely: branch hint */
        struct timespec64 ts;

        ktime_get_real_ts64(&ts);
        if (put_user(ts.tv_sec, &tv->tv_sec) ||
            put_user(ts.tv_nsec / 1000, &tv->tv_usec))
            return -EFAULT;
    }
    if (unlikely(tz != NULL)) {
        /* memcpy to usr-space memory */
        if (copy_to_user(tz, &sys_tz, sizeof(sys_tz)))
            return -EFAULT;
    }
    return 0;
}
```

User-space vs. kernel-space memory



- User space cannot access kernel memory
- Kernel code must never blindly follow a pointer into user-space
  - Accessing incorrect user address can make kernel crash!
- **Q: How to prevent a user-space access kernel-space memory?**
- **Q: How to safely access user-space memory?**

`copy_{from|to}_user`

```
/* copy user-space memory to kernel-space memory */
static inline
long copy_from_user(void *to, const void __user *from, unsigned long n);

/* copy kernel-space memory to user-space memory */
static inline
long copy_to_user(void __user *to, const void *from, unsigned long n);
```

- Is the provided user-space memory is legitimate?
  - If not, raise an illegal access error
- Does the user-space memory exist?
  - If swapped out, kernel accesses the user-space memory after swap in so the process can sleep

Implementing a new system call

1. Write your syscall function
    - Add to the existing file or create a new file
    - Add your new file into the kernel Makefile
2. Add it to the syscall table and assign an ID
    - linux/arch/x86/entry/syscalls/syscall_64.tbl
3. Add its prototype in linux/include/linux/syscalls.h
4. Compile, reboot, and run
    - Touching the syscall table will trigger the entire kernel compilation

Implementing a new system call

- Example: syscall implemented in linux sources in linux/my_syscall/my_func.c
- Creatre a linux/my_syscall/Makefile

```
obj-y += my_func.o
```

- Add `my_syscall` in linux/Makefile

```
core-y      += kernel/ certs/ mm/ fs/ ipc/ security/ crypto/ block/ my_syscall/
```

Why not to implement a system call

- **Pros:** Easy to implement and use, fast
- **Cons:**
    - Needs an official syscall number
    - Interface cannot change after implementation
    - Must be registered for each architecture
    - Probably too much work for small exchanges of information
- **Alternative:**
    - Create a device node and `read()` and `write()`
    - Use `ioctl()`

Improving system call performance

- System call performance is critical in many applications
  - Web server: `select()`, `poll()`
  - Game engine: `gettimeofday()`
- **Hardware:** add a new fast system call instruction
  - `int 0x80` → `syscall`

Improving system call performance

- **Software:** vDSO (virtual dynamically linked shared object)
  - A kernel mechanism for exporting a kernel space routines to user space applications
  - No context switching overhead
  - E.g., `gettimeofday()`
    - the kernel allows the page containing the current time to be mapped read-only into user space
- **Software:** FlexSC: Exception-less system call, OSDI 2010

Project: common mistakes #1

- Direct operation on userspace string
  - **DONT** `strlen(user_str)`
  - **DO** `strlen_user`, `strncpy_from_user`
  - **DO** `copy_from_user`, `copy_to_user`

Project: common mistakes #2

- **DO** Use `SYSCALL_DEFINE` macro

```
/* !!! WRONG !!! DO NOT USE !!! */
asmlinkage long sys_my_syscall2(char *user_str)
{
    /* ... */
}

/* !!! CORRECT !!! */
SYSCALL_DEFINE1(my_syscall2, char __user *, user_str)
{
    /* ... */
}
```

Project: common mistakes #3

- Potential kernel stack overflow
- **DONT** use a stack (function local) array

```
/* !!! WRONG !!! DO NOT USE !!! */
SYSCALL_DEFINE1(my_syscall2, char __user *, user_str)
{
    int str_len = strlen_user(user_str);
    char str_buffer[str_len]; /* What happen if str_len is 16KB? */
    /* ... */
}
```

Next lecture

- Kernel Data Structures

Further readings

- LWN: Anatomy of a system call: part 1 and part2
- LWN: On vsyscalls and the vDSO
- Linux Inside: system calls
- Linux Performance Analysis: New Tools and Old Secrets