

Advanced Operating Systems (and System Security)

MS degree in Computer Engineering

University of Rome Tor Vergata



Lecturer: Francesco Quaglia

## Cross ring data move

1. Segmentation based protection breaks
2. Kernel level actual data move facilities
3. Enhanced hardware/software data move support

# User/kernel interactions so far

- We can change execution flow between user and kernel
- The effects are
  - ✓ the switch of segmentation information (CS, DS ....)
  - ✓ the switch of the CPL
- We can use CPU general purpose registers to
  - ✓ Post register-fitting input data to the kernel
  - ✓ Get register-fitting results from the kernel
- What about the need for exchanging larger data sets?
  - ✓ see, e.g., Posix `read()`/`write()`, or Win-API `ReadFile()`/`WriteFile()`

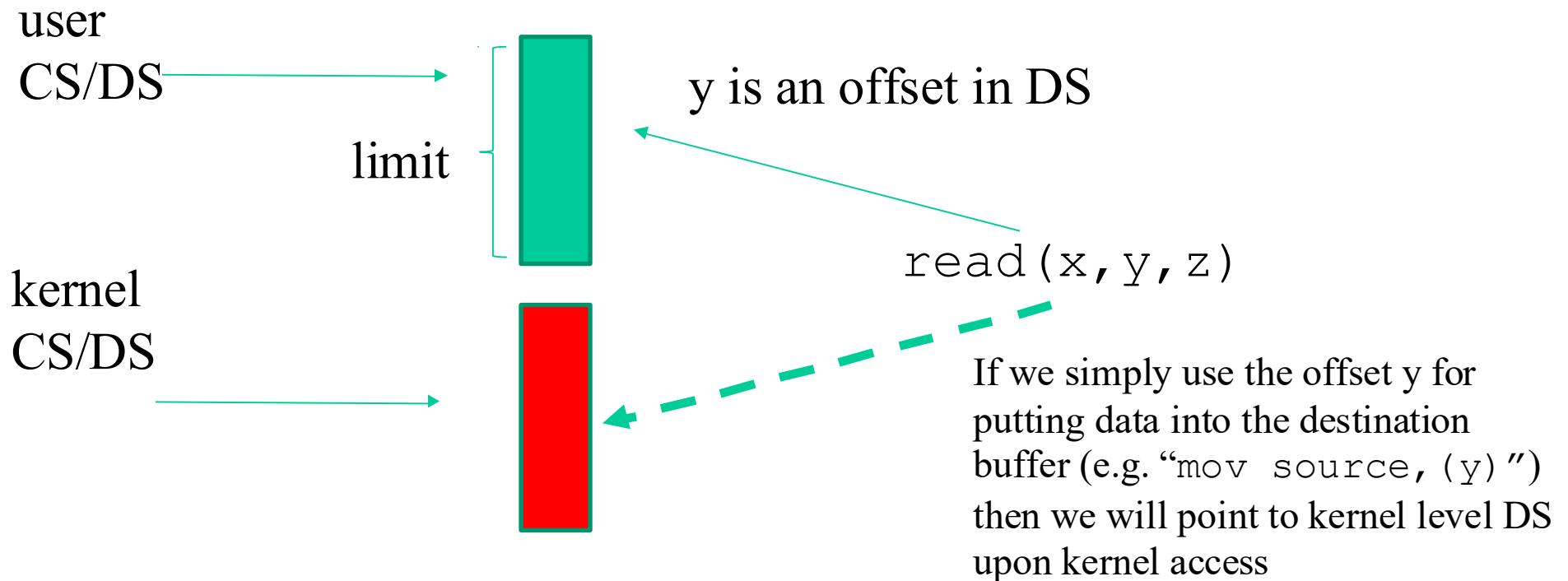
# Usage of pointers

- Clearly, to exchange larger data sets between user and kernel software we use buffers, hence pointers
- Pointers fully break the ring-based protection model
  - ✓ A pointer value can be defined at user level
  - ✓ The actual pointed content can be (over)written or read executing at kernel level
  - ✓ Without additional mechanisms, kernel software can be tampered
- The actual solution to this problem depends on a lot of factors
  - ✓ Actual segmentation support in the hardware
  - ✓ Absence or presence of additional protection mechanisms in the hardware

# The case of flexible segmentation

- This is x86 protected mode segmentation
- We can make, e.g., CS and DS point to whatever we want in the linear address space
- Actual advantages and problems:
  - ✓ Segment full separation in the address space will allow protecting illegal read/writes from kernel segments
  - ✓ We need a mechanism for making this protection occur seamless to the software development process

# A scheme



If we use pure compiler-selected segmentation  
then the ring model is broken

# A solution

- Pieces of kernel code for moving data cross user/kernel must be “handcrafted” (since choices involving segments must be carefully handled – not solely based on compilers)
- We can use a programmable segment selector (e.g. FS) to do this
  - ✓ map FS to the user DS
  - ✓ move data using the pointer ‘y’ applying the displacement to FS
- These operations are generally called ‘segmentation fixup’
- Clearly they have a cost in terms of processor state setup for carrying out the memory copy

# Solution details

user  
CS/DS



limit



kernel  
CS/DS



y is an offset in DS



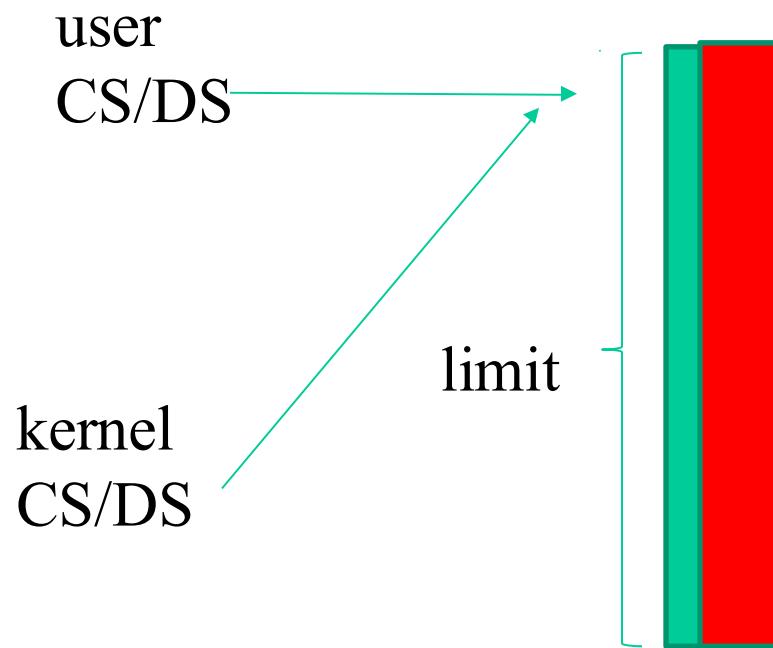
read(x, y, z)

- 1) Trap to kernel
- 2) Materialize data to be delivered into the buffer cache (or other kernel buffers)
- 3) Set FS base to user DS base
- 4) Execute a memory copy module based on the "mov source, FS:(y)" pattern
- 5) Restore FS to the original content

# The case of “constrained” segmentation

- This is x86 long mode segmentation
- This is also x86 protected mode with classical mapping of user/kernel CS, DS, SS, ES to base 0x0
- Making FS point to the base of “user DS” does not work (it fails)
- The offset ‘y’ will still apply to kernel DS
- Hence the “`mov source, FS: (y)`” construct may lead to write kernel level memory pages, depending on the value of ‘y’

# A representation of the failure



$y$  is an offset in DS

read( $x, y, z$ )

- 1) Trap to kernel
- 2) Materialize data into the buffer cache (or other kernel buffers)
- 3) Set FS base to user DS base
- 4) Execute a memory copy module based on the "`mov source, FS:(y)`" pattern
- 5) Restore FS to the original content

# Actual solutions with constrained segmentation

- Where to point for a user/kernel data exchange operation is not only defined by the processor state (and its relation to parameters passed to the kernel)
- It is determined by the kernel software
- The determination is actuated per each individual address space the kernel is managing
- Hence each thread has its limitations on where pointers can be redirected for user/kernel data move
- When an operation is requested, the data move fixup inspects the per-thread limitations to determine if the operation is “legit”

# Per-thread memory limits in Linux

- Each thread management metadata keep a field called `addr_limit`
- It is embedded into a struct (in a field called `seg`) which can be read via the kernel API `get_fs()`
- [It can also be updated](#) to a generic value ‘x’ via the kernel API `set_fs(x)` ... but only up to kernel < 5.9
- All the kernel services that implement user/kernel data move make a check on `addr_limit`
- If the memory area (based on passed pointer and size of the destination/source buffer) is not within `addr_limit` the service does not (or partially) perform(s) memory copy

## Example of addr\_limit read

```
unsigned long limit;  
.....  
limit = (unsigned long)get_fs().seg;  
printk("limit is %p\n", limit);
```

Currently the limit in Linux is set to 0x00007fffffff000 which is the lower half of the x86 long mode canonical addressing form

# addr\_limit update vs security

- Updates of `addr_limit` are typically infrequent (if not executed at all) operations
- At the same time enabling the update of `addr_limit` allows a thread to execute highly critical tasks (read/write) related to the access to kernel level zones
- The current plan in Linux (since kernel 5.9) has been the one of eliminating this value from updatable thread management data
- The limit will be then identified on the basis of a non-modifiable compile time defined value

# User/kernel level data move API

```
unsigned long copy_from_user(void *to, const void *from,  
                           unsigned long n)
```

Copies n bytes from the user address(from) to the kernel address space(to).

```
unsigned long copy_to_user(void *to, const void *from, unsigned  
                           long n)
```

Copies n bytes from the kernel address(from) to the user address space(to).

```
void get_user(void *to, void *from)
```

Copies an integer value from userspace (from) to kernel space (to).

```
void put_user(void *from, void *to)
```

Copies an integer value from kernel space (from) to userspace (to).

# User/kernel level data move API

```
long strncpy_from_user(char *dst, const char *src, long count)
```

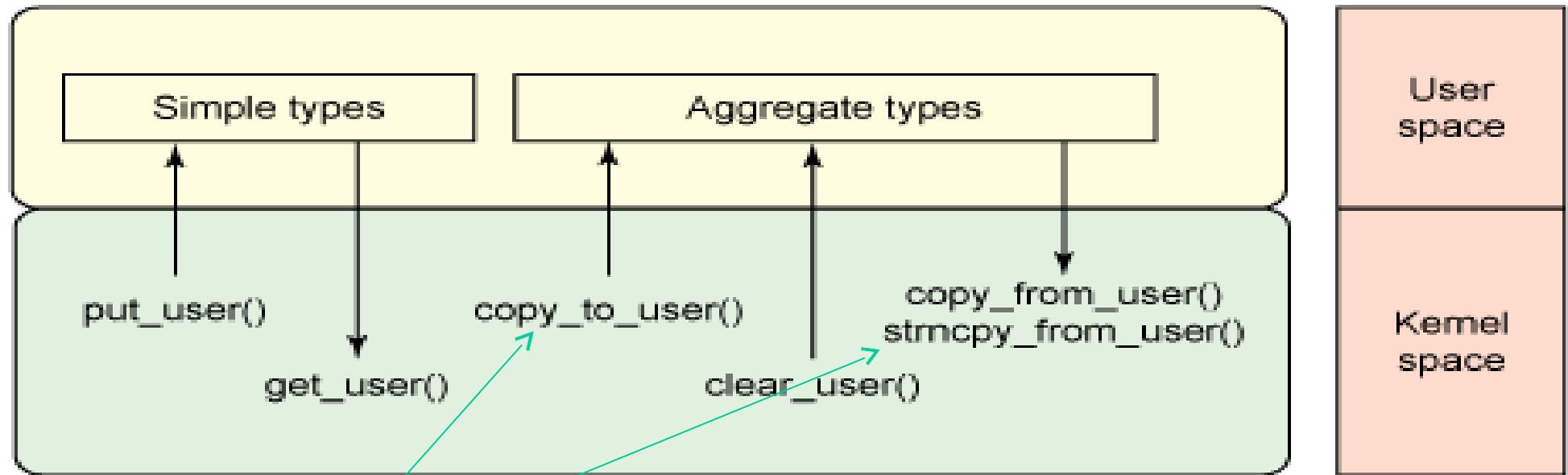
Copies a null terminated string of at most count bytes long from userspace (src) to kernel space (dst)

```
int access_ok(int type, unsigned long addr, unsigned long size)
```

Returns nonzero if the userspace block of memory is valid and zero otherwise

These data move operations deal with already mapped regions – the results returned indicates the residual bytes of the data move operation, not the amount of data actually moved

# A scheme



These functions return the residuals  
(bytes not managed)

Most of them ground on  
`access_ok()`

The actual copy operation may lead the thread to sleep  
(we will be back to this issue when talking of contexts)

# Overall view of the API actions

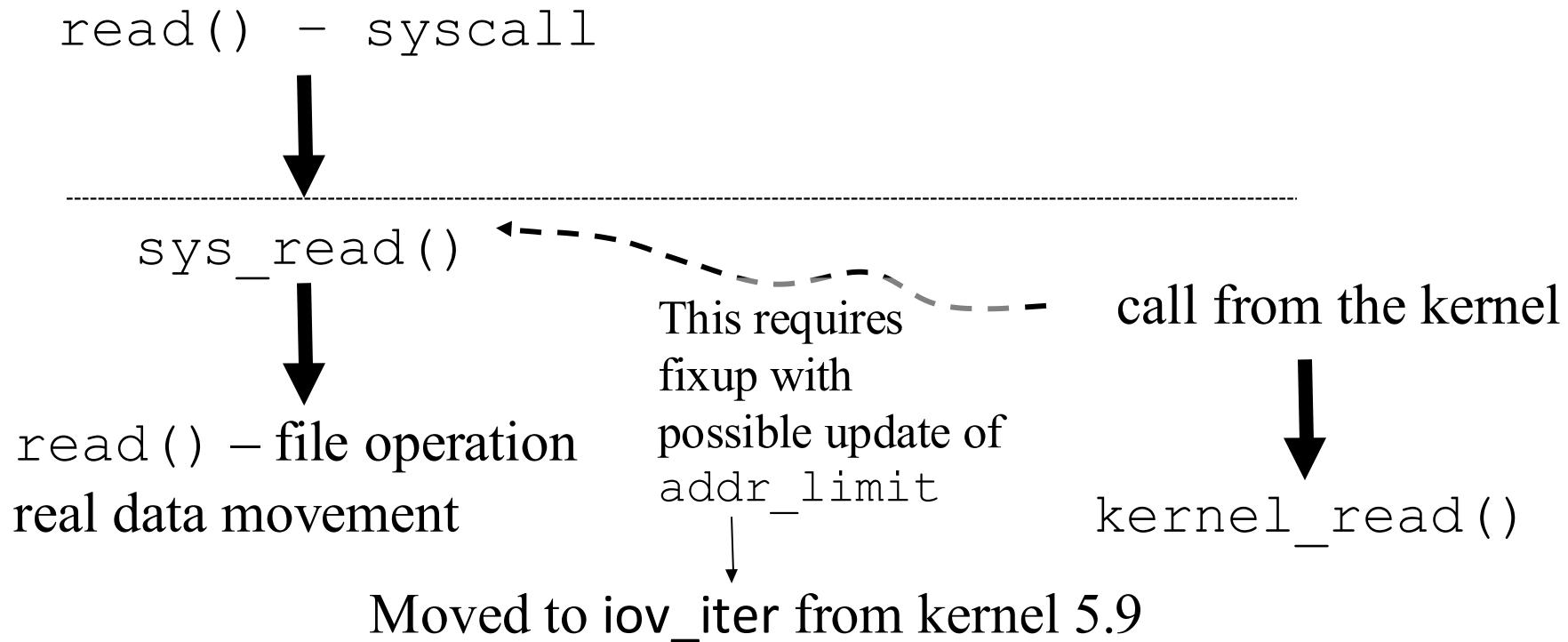
- Segment fixup (if segmentation takes a real role in the composition of the addresses)
- Check on address ranges related to user level
  - ✓ The actual depth of check may depend on the specific implementation (namely on the kernel version)
  - ✓ E.g., the process memory map might be checked or not
- Note: associating physical to virtual memory is demanded to the page-fault handler
  - ✓ Performance impact due to (possible) non-atomicity while finalizing the handling

# Service redundancy approaches

- Check and fixup are required only in case we need to link activities across different privilege levels within the ring model (as when calling system calls)
- Particularly, this occurs when the execution semantic crosses the boundaries of individual segments
- Bypassing check and fixup when no crossing of segment boundaries occurs takes place via “service redundancy” (for performance reasons)
- The kernel layer entails an internal API for executing activities that are typically triggered when running in user mode

# Classical examples

- `kernel_read()` is a redundancy for `read()`
- `kernel_write()` is a redundancy for `write()`



# `memcpy` with tampered pointers

- Clearly, the usage of fixup based APIs for data movement does not break the ring model under normal operating conditions
- What if a `memcpy()` is called by the kernel, with arbitrary pointers after a subversion (speculative or not) or in presence of bugs?
- In more dated processor/kernel versions we could do nothing
- In more modern processors/kernels we have an additional security oriented hardware support, which leads to **constrained supervisor mode!!**

# The actual hardware support on x86

- SMAP (Supervisor Mode Access Prevention)
  - ✓ It blocks data access to user pages when running at CPL 0
- SMEP (Supervisor Mode Execution Prevention)
  - ✓ It blocks instruction fetches from user pages when running at CPL 0
- Two bits in CR4 (21 and 20) activate them
- They can be temporary disabled (e.g. setting the AC bit in EFLAGS for the case of SMAP)

## `copy_to_user` timeline (as a reference example)

- Check within per-thread limit
- Determine the legal amount of data to be copied
- Disable SMAP (via the AC flag through the `stac` x86 instruction)
- Make the copy (may wait but not SEGFAULT)
- Enable SMAP again (via the AC flag through the `clac` x86 instruction)

## access\_OK limitations

- The determination of the legal amount of data to be copied requires inspecting the memory map (via `*mm`) of the running thread
- Various additional machine instructions used just to move data between kernel and user
  - ✓ Interactions with suboptimal usage of I/O services (e.g. byte rather than segment reads/writes)
- `mm` inspection may have linear (non-constant) cost

# Newer approaches - kernel masked SEGFAULTS

- Access OK control only checks the `addr_limit`
- If `addr_limit` is OK then the memory copy is directly executed
- If and only if some user page not mapped (or not compliant with the protection requested by the memory copy) is touched we have a SEGFAULT from kernel software (RIP points to a kernel page)
- The philosophy is the one of speeding up the normal scenario

# Kernel masked SEGFAULTS details

