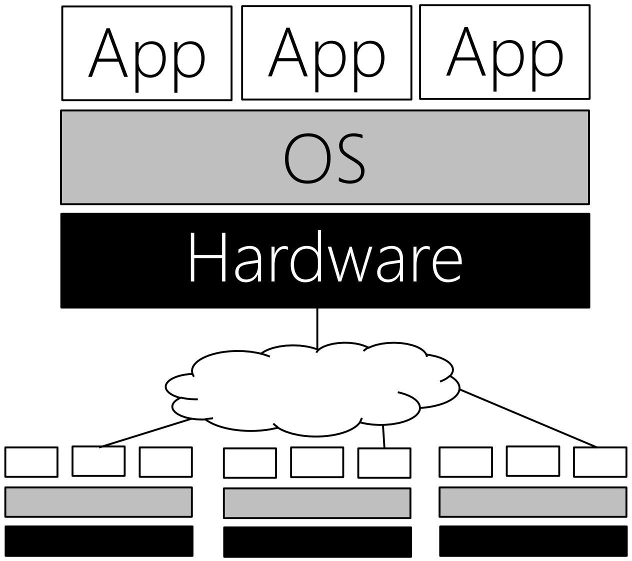

SCAN COMPLETE
PASSWORD PROTECT
1234
567
89
10
HACKING DETECTED

Security

What Hollywood thinks computer security is about

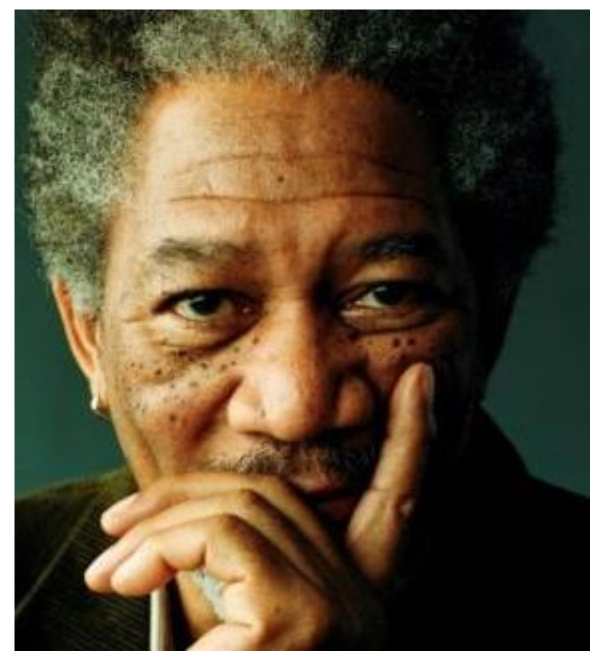
Understanding how systems work



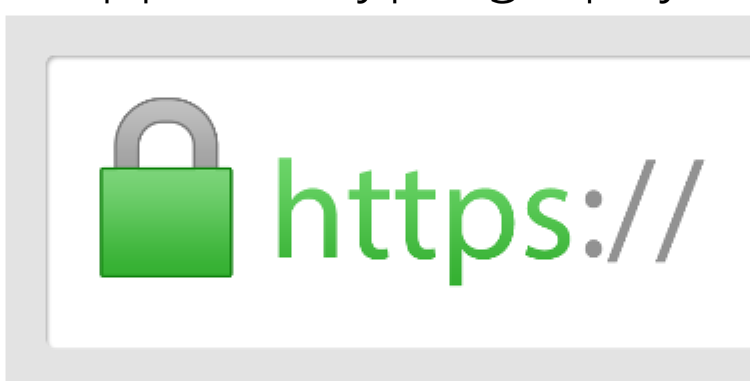
Theoretical cryptography

$$
\begin{aligned}
\Pr[\quad & \tau \leftarrow \mathsf{KeyGen}_1(1^\lambda); \tau = (\tau_{\mathcal{S}}, \tau_{\mathcal{E}}); \\
& (EK, VK) \leftarrow \mathsf{KeyGen}_2(\tau, R); \\
& (\mathbf{D}, \pi; \boldsymbol{\chi}, \mathbf{o}) \leftarrow (\mathcal{A}(EK, R) \,\|\, \mathcal{E}(EK, R, \tau_{\mathcal{E}})) : \\
& (\exists b \in [\ell].\ \mathsf{Verify}(VK_b, D_b^{(t)}) \wedge D_b^{(t)} \neq \mathsf{Digest}(EK_b, \chi_b^{(t)}, o_b^{(t)})) \vee \\
& (\forall b \in [\ell].\ \mathsf{Verify}(VK_b, D_b^{(t)}) \wedge \mathsf{Verify}(VK, \mathbf{D}, \pi) \wedge \boldsymbol{\chi} \notin R) \\
] = & \ \mathtt{negl}(\lambda).
\end{aligned}
$$

Understanding humanity



Applied cryptography

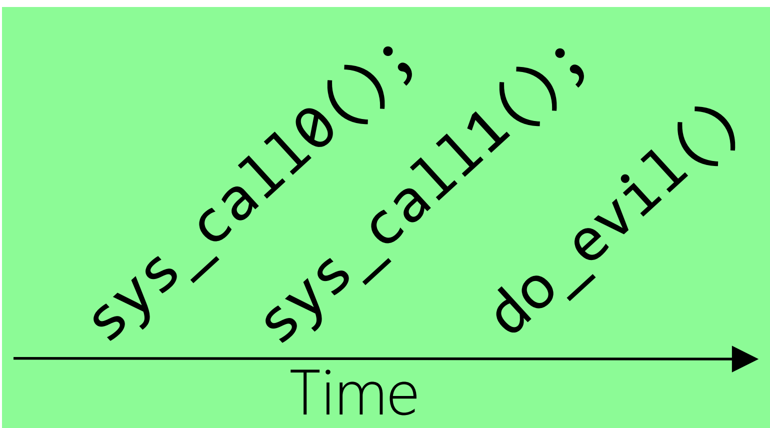


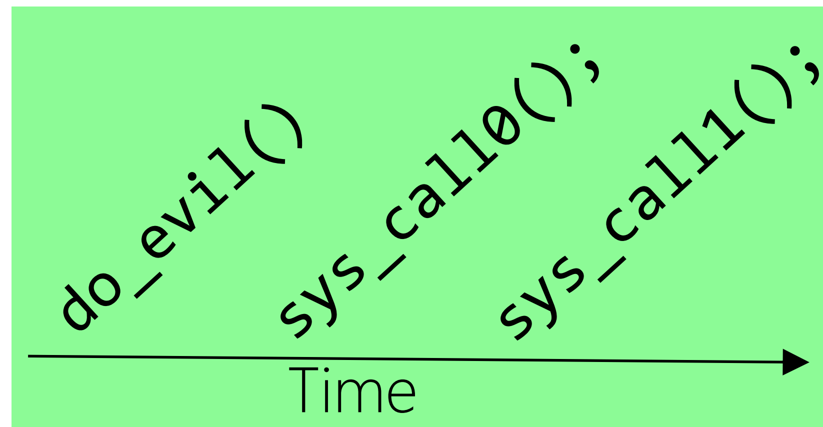
What computer security is really about

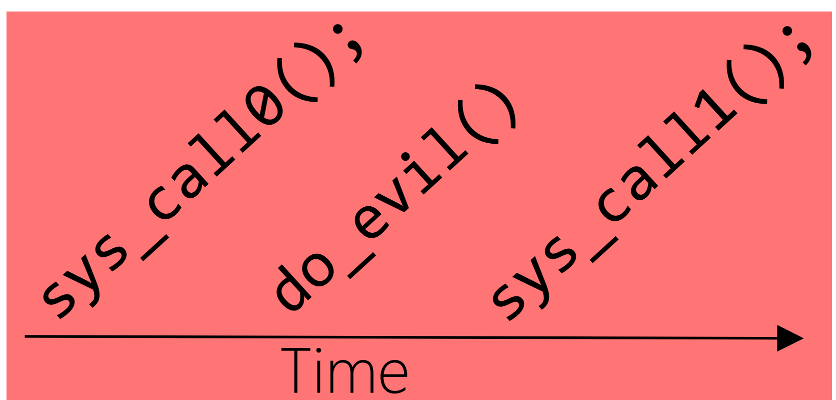
Race Conditions: Non-atomic System Call Pairs

```
//Process X
sys_call0();
sys_call1();
```

```
//Process Y
do_evil()
```







Linux: Mapping Humans to Privileges

- Each user has a user ID (UID), with root having UID 0
  - Each user also has a group ID (GID), but we'll mostly ignore groups today
- Each file has:
  - Read/write/execute permissions for the file's owner, the file's group, and the world (i.e., all other users)
  - Set-user-id bit: 1 if the file should be executed with the owner's permissions (shows up as "s" instead of "x" in "ls –l" output); 0 if the file should be executed with launching user's permissions

```
$ ls -l a.out
-rwxr-xr-x   1 mickens  somegrp 524288 Jan 19 11:35
```

User

Group

World

Owner

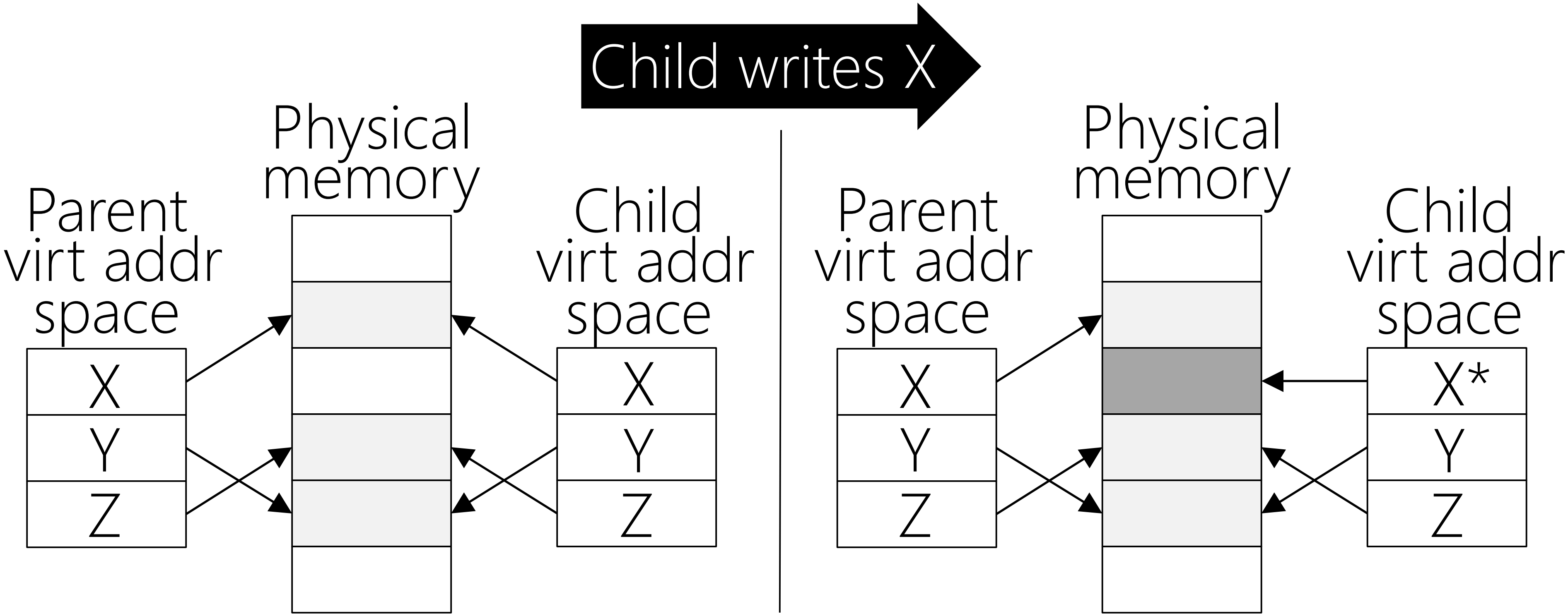
Owner group

Linux: Mapping Humans to Privileges

- Each process has a bunch of IDs, including:
  - Real UID: The UID of the process owner
  - Effective UID: The UID that the kernel checks when validating access permissions
- On fork(), the child inherits the UIDs of its parent
- On exec(progName), the process keeps its UIDs unless the progName file has the set-user-ID bit set, in which case the effective UID of the process is set to the UID of the binary's owner
  - Ex: The `passwd` command needs to update a user's entry in /etc/shadow file
  - The /etc/shadow file is sensitive and should only be modified by root, but regular users need to be able to update their passwords!
  - So, the `passwd` binary is owned by root, but has a set-user-ID bit of 1

Making New Processes: fork()

- fork() allows a parent process to create a child process
  - Abstractly speaking, child gets a copy of parent's address space
  - Linux's fork() uses copy-on-write pages: avoids synchronous copy of the entire address space, and only incurs copy overhead for pages which actually diverge between parent and child



- BSD's old fork() did full, synchronous copy

The Drunk Uncle Named BSD vfork()

- vfork() was intended for the situation in which the parent does a fork() and the child does an "immediate" exec()
- After vfork(), parent is suspended; child executes using parent's address space and thread-of-control until exec() is called
- When child calls exec(), BSD makes a new address space for the child and copies file descriptors, current working directory, etc. like a regular fork()
- Child process is supposed to not modify anything in the parent's address space before calling exec() or otherwise undefined demons happen
- BSD BE SERIOUS UGGHHH IMPLEMENT COW

The Drunk Uncle Named BSD vfork()

Problem: vfork()+exec() is not atomic!

```
//Suppose that the parent process is
//run as the privileged "root" user.
//Non-root users can't send signals
//to root processes.
pid_t pid = vfork();
if(pid == 0){
  //In child (parent is suspended); drop
  //privilege so child lacks root powers
  setuid(nonroot_user);
```

A process owned by nonroot_user can send SIGSTOP to child; child is blocked, and so is the privileged parent—denial of service!

```
  execve(path, NULL, NULL);
  exit(1);  //Kill child if execve()
            //fails and we get here.
}
```

The Drunk Uncle Named BSD vfork()

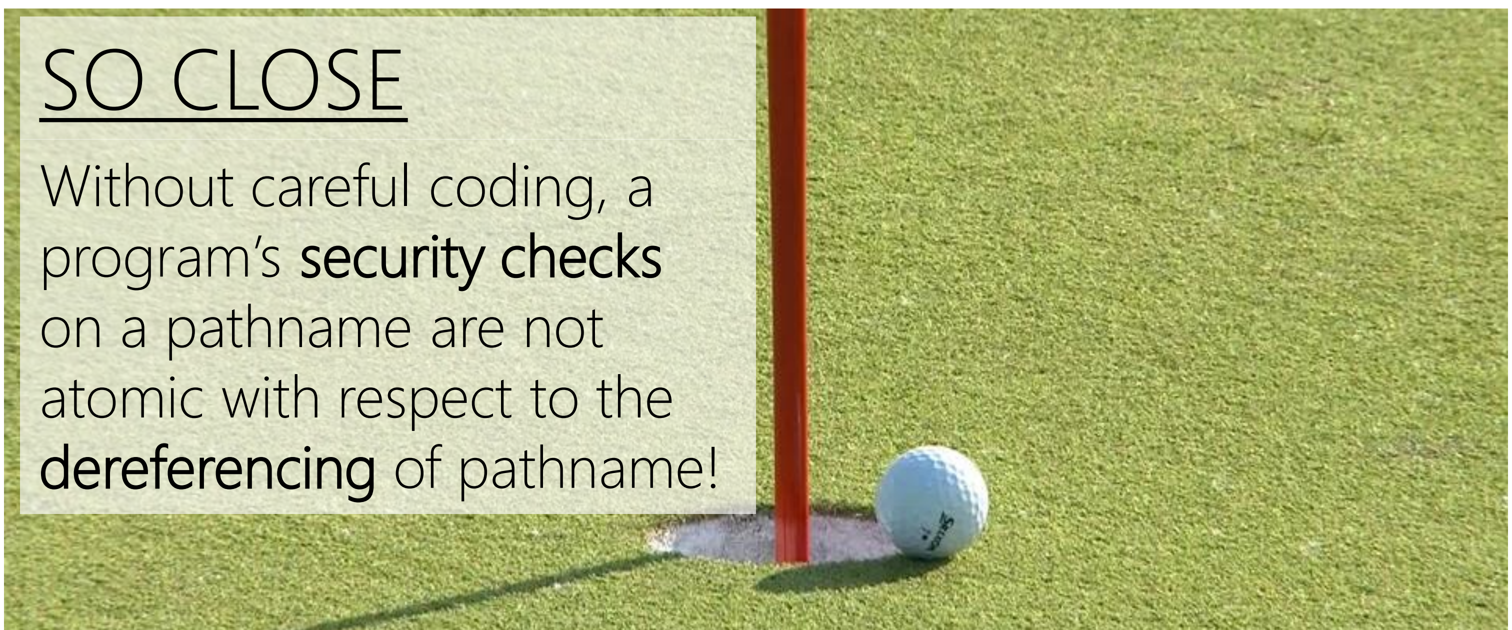
Problem: vfork()+exec() is not atomic!

Solution for priority inversion: Use fork()+exec(), and implement COW to make fork() fast.

[fork()+exec() still isn't atomic, but the denial of service is now fixed since the parent isn't blocked waiting for the child.]

Fun With Symlinks

- Symbolic link: special file whose contents are name of another file
  - Ex: **`link -s /etc/shadow /tmp/foo`**
  - **`/tmp/foo`** is now an alias for **`/etc/shadow`**
  - When a process P wants to read/write/execute **`/tmp/foo`**, the kernel checks whether P has the appropriate access permissions for the underlying path **`/etc/shadow`**
  - A program can check whether path refers to symlink or regular file
  - Sounds pretty secure, right?



```c
//Imagine that a mailserver runs with
//root privileges.
char *filename = "/home/loki/mailbox";
char *new_msg = get_new_msg_for(“loki");
if(new_msg == NULL){
    return;
}
int new_msg_len = strlen(new_msg);

//Write the new message into Loki's
//mailbox, but we're paranoid! Only
//write the new message if Loki's
//mailbox path isn't a symlink.
struct stat lstat_info;
int fd;
lstat(filename, &lstat_info);
if(!S_ISLNK(lstat_info.st_mode)){
   //Window of vulnerability
   fd = open(filename, O_RDWR);
   write(fd, new_msg, new_msg_len);
   close(fd);
}
```

```
$ rm /home/loki/mailbox
$ ln –s /etc/shadow
       /home/loki/mailbox
```

The password file will get overwritten with the contents of Loki’s new email. By the way, LOKI CAN EMAIL HIMSELF.

```
char *filename = "/home/loki/mailbox";
char *new_msg = get_new_msg_for("loki");
if(new_msg == NULL){return;}
int new_msg_len = strlen(new_msg);

struct stat lstat_info;
struct stat fstat_info;
int fd;

lstat(filename, &lstat_info);      //Doesn't follow
                                   //symlinks
//Attacker window to corrupt the fd that will
//be opened by the mailserver.
fd = open(filename, O_RDWR);       //*Does* follow
        //symlinks--"dereferences" symlink to get
        //fd which represents the symlink target
fstat(fd, &fstat_info);
if (lstat_info.st_mode == fstat_info.st_mode &&
    lstat_info.st_ino  == fstat_info.st_ino  &&
    lstat_info.st_dev  == fstat_info.st_dev){
    write(fd, new_msg, new_msg_len);
}
close(fd);
```

Ensures that the filename used to open the file wasn't a symlink!

Other Defenses

Linux added several new system calls to reduce the number of race conditions developers must check

```
//Ten directory race conditions
//to check.
int fd0 = open(“dirName/0.txt”);
int fd1 = open(“dirName/1.txt”);
          . . .
int fd9 = open(“dirName/9.txt”);
```

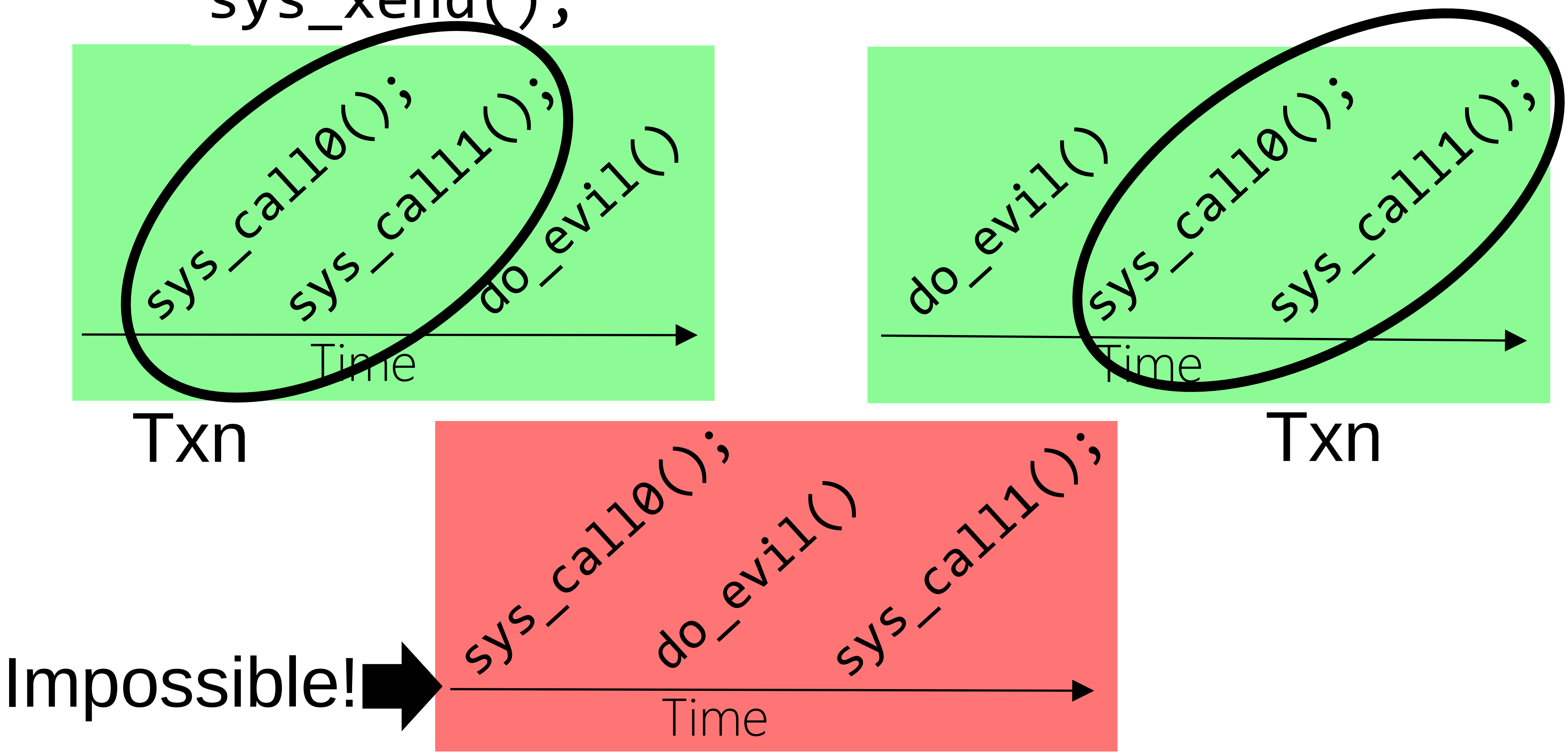
```
//One directory race condition
//to check.
int dirFd = open(“dirName”);
int fd0 = openat(dirFd, “0.txt”);
int fd1 = openat(dirFd, “1.txt”);
          . . .
int fd9 = openat(dirFd, “9.txt”);
```

Once process has an fd for a path, symlink shenanigans won’t change the inode that the file descriptor points to!

Race Conditions: Non-atomic System Call Pairs

```
//Process X
sys_xbegin();
sys_call0();
sys_call1();
sys_xend();
```

```
//Process Y
do_evil()
```



Race Conditions At The Hardware Level

```
//Assume that memory
//locations [x] and [y]
//both initialized to 0
mov 1 --> [x]                mov 1 --> [y]
mov [y] --> %eax             mov [x] --> %ebx
```

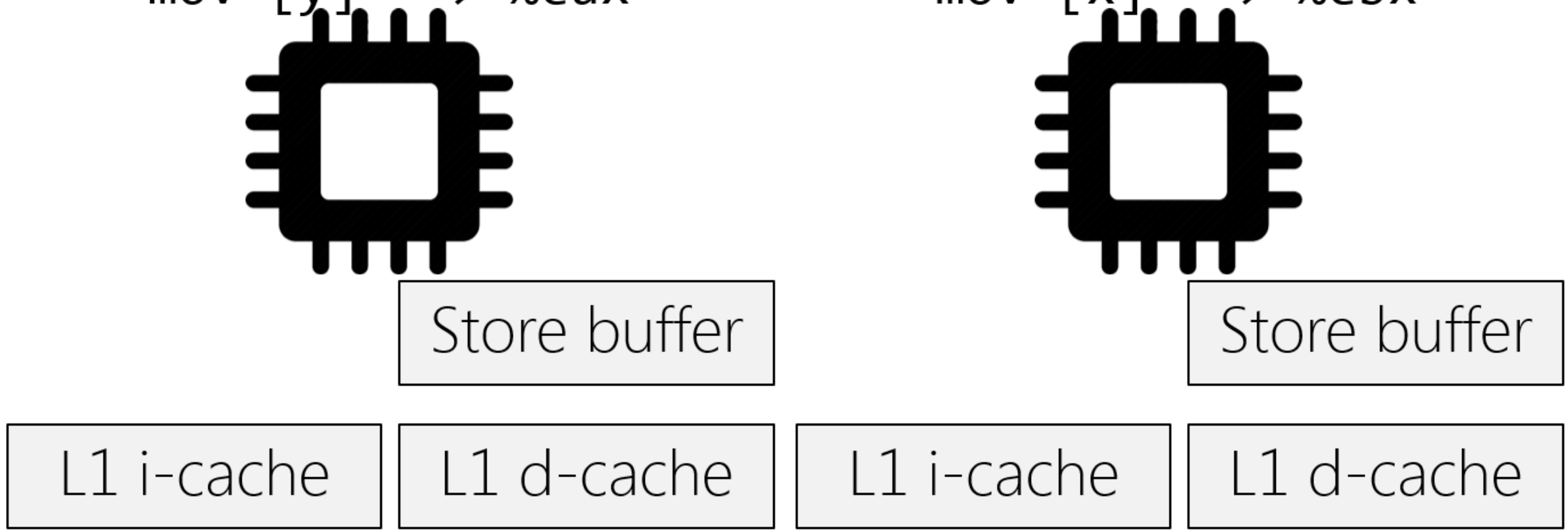
x86 allows both threads to read 0 (which would be impossible if both threads run instructions sequentially (but may be interrupted)).

L1 i-cache	L1 d-cache	L1 i-cache	L1 d-cache
L2 cache		L2 cache	
L3 cache			
RAM			

```
//Assume that memory
//locations [x] and [y]
//both initialized to 0
```

```
mov 1 --> [x]
mov [y] --> %eax
```

```
mov 1 --> [y]
mov [x] --> %ebx
```



x86 allows <u>both</u> threads to read 0 (which would be impossible if both threads run instructions sequentially (but may be interrupted)).

Per-core store buffer allows a core to write to memory and immediately continue execution as the write is asynchronously pushed through the memory hierarchy.

A core sees its own writes immediately, but may not be immediately aware of other cores' writes.

How Hardware Shows Its Love For Us

- Modern processors do unholy things to improve performance
  - Per-core store buffers avoid the need for a core to stall on a synchronous write through the memory hierarchy
  - A core may also reorder instructions (e.g., non-dependent loads and stores) to increase parallelism of hardware usage

```
//Load is dependent on the
//store, so hardware will
//*not* reorder the memory
//operations.
x = 42;
y = x;
```

How Hardware Shows Its Love For Us

- Modern processors do unholy things to improve performance
  - Per-core store buffers avoid the need for a core to stall on a synchronous write through the memory hierarchy
  - A core may also reorder instructions (e.g., non-dependent loads and stores) to increase parallelism of hardware usage

```
//First core
while(shouldStop == 0){;}
printf("%d", answer);
```

```
//Second core
answer = 42;
shouldStop = 1;
```

This code may not print 42!

No data dependency w.r.t. the local core. So, hardware may reorder the stores!

How Hardware Shows Its Love For Us

- Modern processors do unholy things to improve performance
  - Per-core store buffers avoid the need for a core to stall on a synchronous write through the memory hierarchy
  - A core may also reorder instructions (e.g., non-dependent loads and stores) to increase parallelism of hardware usage

```
//First core
while(shouldStop == 0){;}
mem_barrier();
printf("%d", answer);
```

```
//Second core
answer = 42;
mem_barrier();
shouldStop = 1;
```

- Software needs a way to force sequential memory semantics when necessary (e.g., to implement locks), so hardware provides instructions which make it easier for software to reason about memory
  - Ex: Atomic instructions like compare-and-exchange, LL/SC
  - Ex: A memory barrier forces all instructions *before* the barrier to complete before any instruction *after* the barrier

It's hard for developers to find all of the places in which memory-ordering primitives are necessary!

Linux TLB Flushing Logic on x86

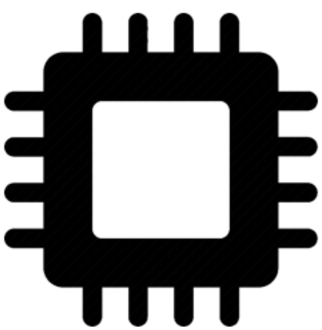
- When the kernel changes a page table mapping, the kernel must flush the local core's TLB to remove any stale entry
    - On a multi-core system, the local core must also send an IPI to other cores so that those cores can flush their TLBs too

Linux TLB Flushing Logic on x86

- When the kernel changes a page table mapping, the kernel must flush the local core’s TLB to remove any stale entry

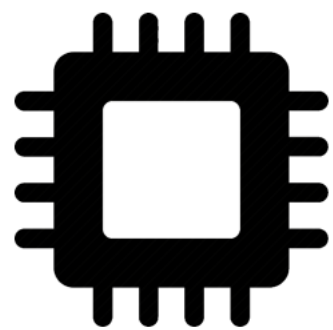
```
//Core A has locked a page
//table, wants to update it.
A1: Change page table
A2: Flush local TLB if
    necessary
A3: Check whether Core B is
    using page table; if so,
    send TLB_flush IPI
```

```
//Core B wants to context switch
//to a process which uses the
//page table that A modifies.
B1: Atomically set bit in page
    table saying “Core B is
    using the page table”
B2: Load %cr3 to set page table
    pointer (flushing local TLB
    as side effect)
B3: Run code (filling TLB as
    side effect)
```



Core A

On x86, assigning to %cr3 is “serializing,” i.e., an implicit memory barrier. So, B1—B3 won’t be reordered by HW.



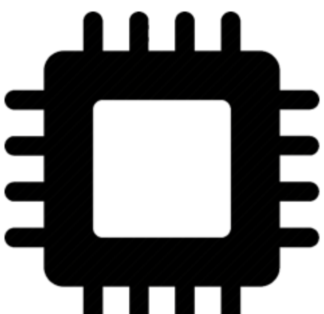
Core B

Linux TLB Flushing Logic on x86

- When the kernel changes a page table mapping, the kernel must flush the local core’s TLB to remove any stale entry

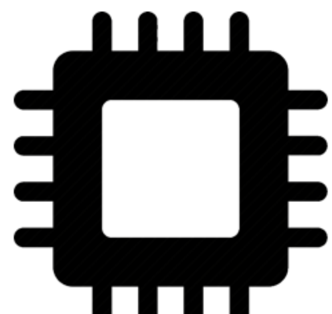
```
//Core A has locked a page
//table, wants to update it.
A1: Change page table
A2: Flush local TLB if
    necessary
A3: Check whether Core B is
    using page table; if so,
    send TLB_flush IPI
```

```
//Core B wants to context switch
//to a process which uses the
//page table that A modifies.
B1: Atomically set bit in page
    table saying “Core B is
    using the page table”
B2: Load %cr3 to set page table
    pointer (flushing local TLB
    as side effect)
B3: Run code (filling TLB as
    side effect)
```



Core A

TLB flush is serializing, BUT IT MAY NOT OCCUR.



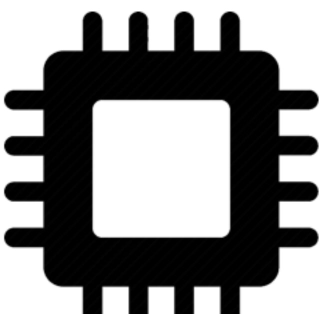
Core B

Linux TLB Flushing Logic on x86

- When the kernel changes a page table mapping, the kernel must flush the local core’s TLB to remove any stale entry

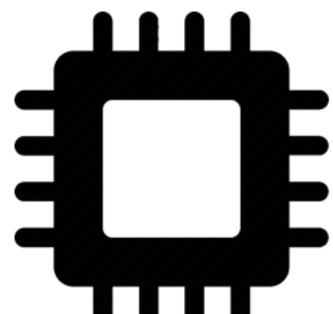
```
//Core A has locked a page
//table, wants to update it.
A1: Change page table
A2: Flush local TLB if
    necessary
A3: Check whether Core B is
    using page table; if so,
    send TLB_flush IPI
```

```
//Core B wants to context switch
//to a process which uses the
//page table that A modifies.
B1: Atomically set bit in page
    table saying “Core B is
    using the page table”
B2: Load %cr3 to set page table
    pointer (flushing local TLB
    as side effect)
B3: Run code (filling TLB as
    side effect)
```



Core A

A1 is ordinary store, and A3 is ordinary load, so if A2 doesn’t execute, HW may reorder A1 and A3: Core A won’t send IPI to Core B!



Core B

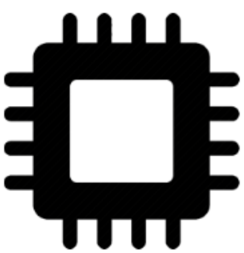
Linux TLB Flushing Logic on x86

- When the kernel changes a page table mapping, the kernel must flush the local core’s TLB to remove any stale entry

```
//Core A has locked a page
//table, wants to update it.
A1: Change page table
A2: Flush local TLB if
    necessary
A3: Check whether Core B is
    using page table; if so,
    send TLB_flush IPI
```

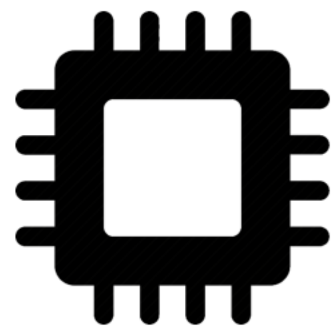
Solution: Always force a barrier here.

Race: Core B may fill TLB with stale virt->phys mappings (B3) before updated mappings hit page table (reordered A1). If stale mappings allow writing, and Core A reuses phys page for kernel data (e.g., page table), Core B can steal secrets or escalate privilege!



Core A

```
//Core B wants to context switch
//to a process which uses the
//page table that A modifies.
B1: Atomically set bit in page
    table saying “Core B is
    using the page table”
B2: Load %cr3 to set page table
    pointer (flushing local TLB
    as side effect)
B3: Run code (filling TLB as
    side effect)
```



Core B

Race Conditions Caused By Poor Locking Discipline

Race Condition in Linux's ELF Loader

- Just like OS161, Linux has code to load an ELF binary into memory, parse it, and initialize various in-memory regions for the stack, heap, and code
  - Each process structure has a **`struct mm_struct`** representing the process' address space
  - Kernel must grab **`mm_struct::mmap_sem`** before modifying the process' regions (there's one **`struct vm_area_struct`** for each region)

```
struct mm_struct{
    struct vm_area_struct *mmap;  /* List of VMAs */
    pgd_t *pgd;                   /* Page table directory pointer,
                                   * i.e., the thing that gets
                                   * assigned to %cr3 */
    struct rw_semaphore mmap_sem; /* Protects VMAs */
    //...etc...
};
```

```c
//In Linux 2.4.28, the mmap_sem lock was released too early!
static int load_elf_library(struct file *file){
    down_write(&current->mm->mmap_sem);
    //Update the code VMA for the ELF library.
    error = do_mmap(file,
                    ELF_PAGESTART(elf_phdata->p_vaddr),
                    (elf_phdata->p_filesz +
                        ELF_PAGEOFFSET(elf_phdata->p_vaddr)),
                    PROT_READ | PROT_WRITE | PROT_EXEC,
                    MAP_FIXED | MAP_PRIVATE | MAP_DENYWRITE,
                    (elf_phdata->p_offset -
                        ELF_PAGEOFFSET(elf_phdata->p_vaddr)));
    up_write(&current->mm->mmap_sem);
    if(error != ELF_PAGESTART(elf_phdata->p_vaddr))
        goto out_free_ph;

    elf_bss = elf_phdata->p_vaddr + elf_phdata->p_filesz;
    padzero(elf_bss);

    len = ELF_PAGESTART(elf_phdata->p_filesz + elf_phdata->p_vaddr +
                        ELF_MIN_ALIGN - 1);
    bss = elf_phdata->p_memsz + elf_phdata->p_vaddr;
    if(bss > len) //Create the data VMA for the ELF library.
        do_brk(len, bss - len);  //do_brk() assumes mmap_sem is held!
```

```
unsigned long do_brk(unsigned long addr, unsigned long len){
    //Allocate a new VMA to represent the new
    //extension to the process' address space.
    struct vm_area_struct *vma = kmem_cache_alloc(vm_area_cachep,
                                                   SLAB_KERNEL);
    if(!vma)
        return -ENOMEM;

    //Update VMA bookkeeping.
    vma->vm_mm = mm;
    vma->vm_start = addr;
    vma->vm_end = addr + len;
    vma->vm_flags = flags;
    vma->vm_page_prot = protection_map[flags & 0x0f];
    //Other bookkeeping values updated . . .

    //Add the new VMA to the process' set of VMAs
    //(the VMAs are stored in a red-black tree).
    vma_link(mm, vma, prev, rb_link, rb_parent);

    return addr;
}
```

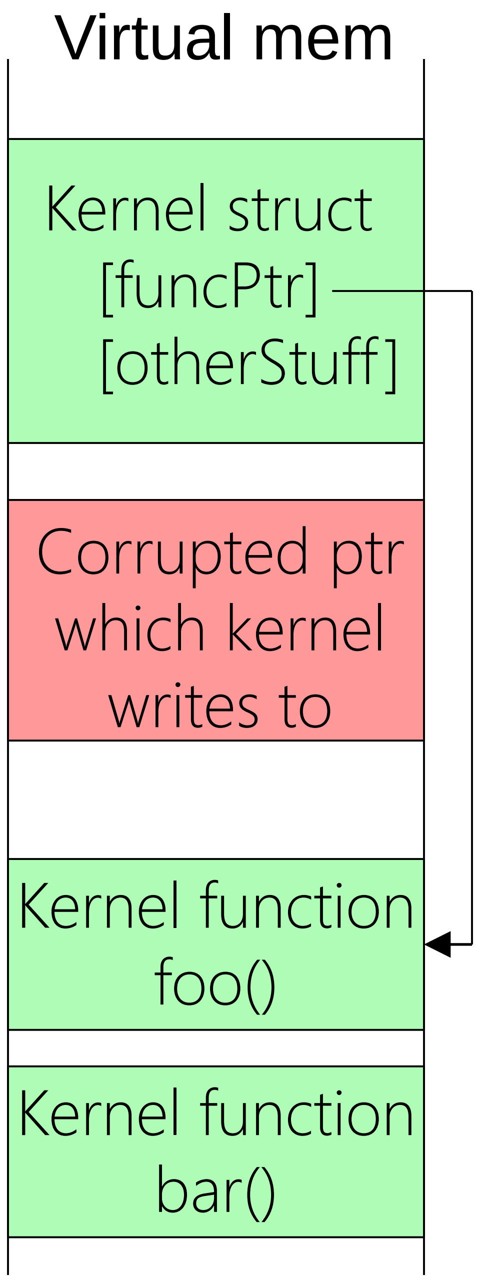
The kernel may sleep during memory alloc!

```
unsigned long do_brk(unsigned long addr, unsigned long len){
    //Allocate a new VMA to represent the new
    //extension to the process' address space.
    struct vm_area_struct *vma = kmem_cache_alloc(vm_area_cachep,
                                               SLAB_KERNEL);
    if(!vma)
        return -ENOMEM;

    //Update VMA bookkeeping.
    vma->vm_mm = mm;
    vma->vm_start = addr;
    vma->vm_end = addr + len;
    vma->vm_flags = flags;
    vma->vm_page_prot = protection_ma
    //Other bookkeeping values updated

    //Add the new VMA to the process' set
    //(the VMAs are stored in a red-black tree).
    vma_link(mm, vma, prev, rb_link, rb_parent);

    return addr;
}
```

Attacker launches a program with several threads that allocate a lot of memory and call sys_uselib() with a carefully-crafted ELF binary. Goal: Trigger race condition and corrupt memory in attacker-controlled way!

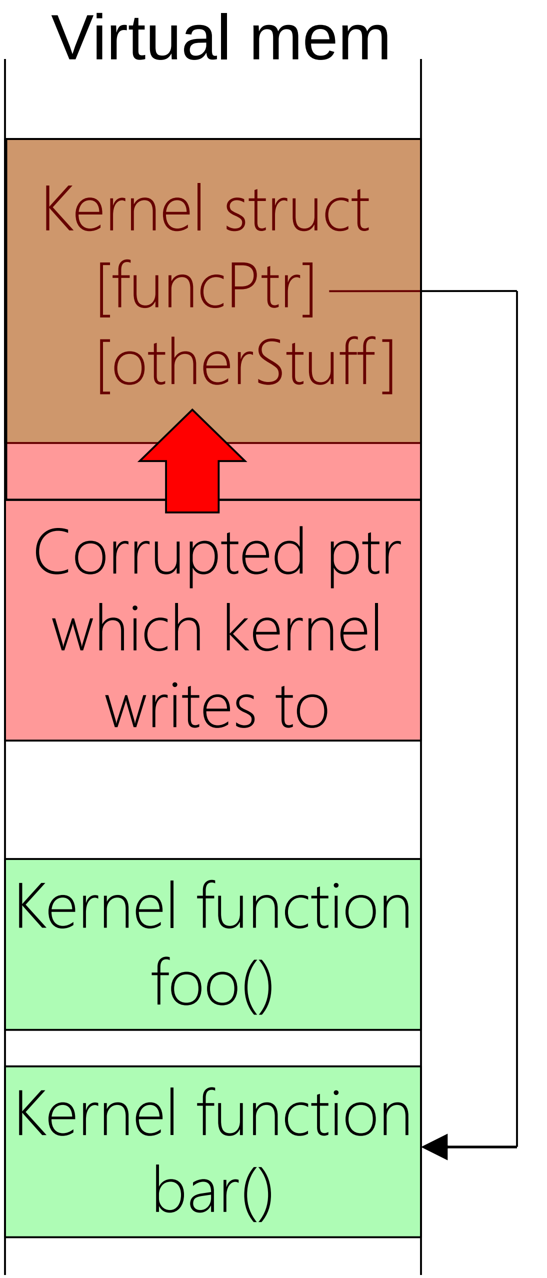
```
vma_link(mm, vma, prev, rb_link, rb_parent);
```

- `rb_link` and `rb_parent` are pointers which `vma_link()` uses to write kernel memory
  - By controlling how the race condition corrupts memory, the attacker can overwrite pointers with attacker-controlled values
  - Later, when kernel writes memory through corrupted pointers, the kernel will write to memory addresses of the attacker's choosing
  - If the attacker supplies the data for the write, the attacker also chooses the new contents of those memory addresses
  - Ex: Overwriting a kernel function pointer to trick the kernel into invoking the wrong function

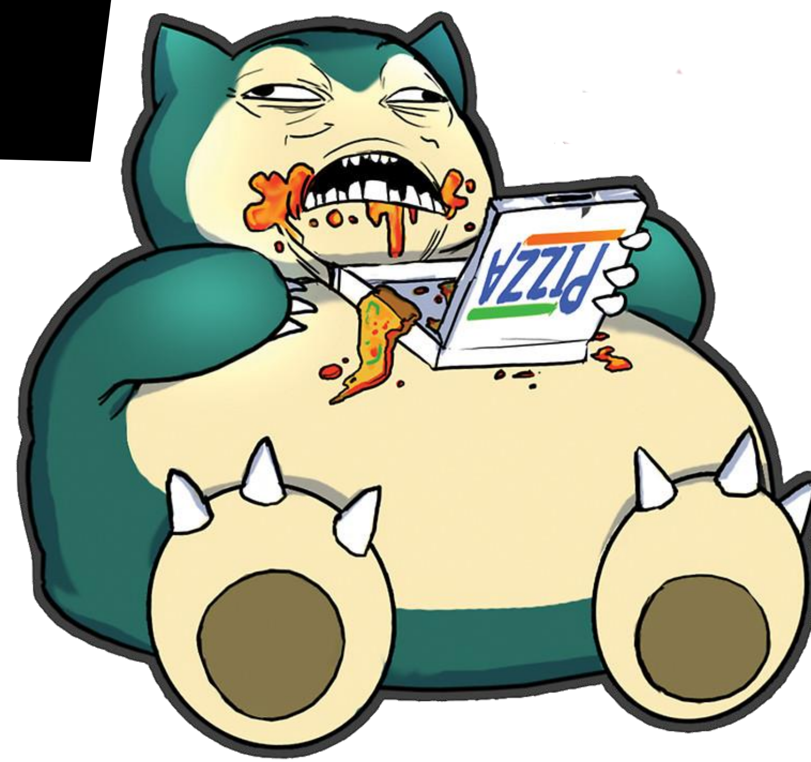


`vma_link(mm, vma, prev, rb_link, rb_parent);`

- `rb_link` and `rb_parent` are pointers which `vma_link()` uses to write kernel memory
  - By controlling how the race condition corrupts memory, the attacker can overwrite pointers with attacker-controlled values
  - Later, when kernel writes memory through corrupted pointers, the kernel will write to memory addresses of the attacker's choosing
  - If the attacker supplies the data for the write, the attacker also chooses the new contents of those memory addresses
  - Ex: Overwriting a kernel function pointer to trick the kernel into invoking the wrong function



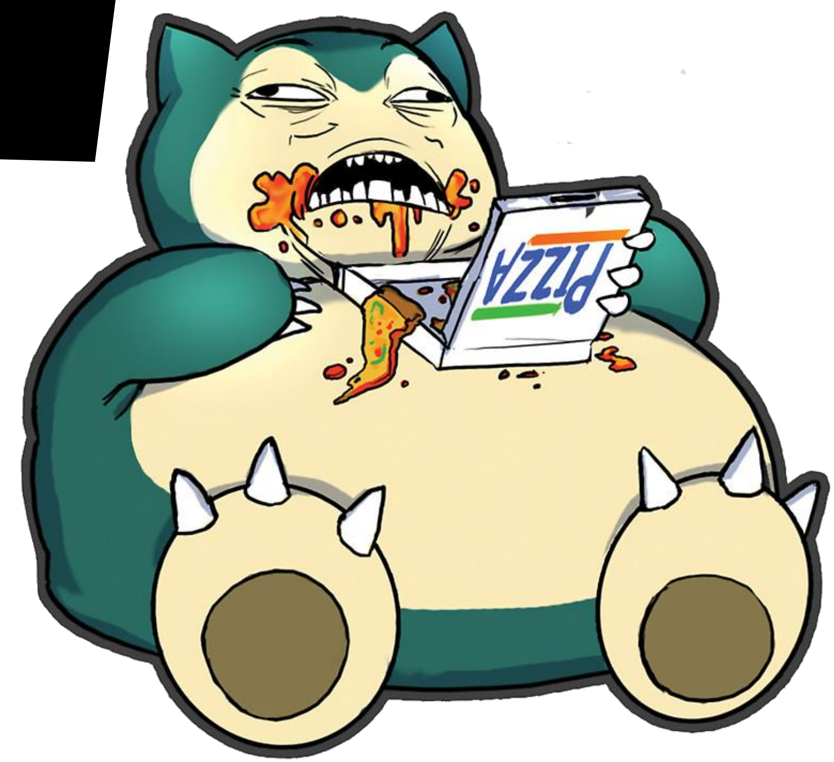
HOW DO WE
PREVENT THIS?



HOW DO WE PREVENT THIS?

- In the case of this specific ELF vulnerability, the fix is to hold mmap_sem while the data region for the library is created

```
//Also, in do_brk() . . .
/*
 * mm->mmap_sem is required to
 * protect against another thread
 * changing the mappings while we
 * sleep (on kmalloc for one).
 */
verify_mmap_write_lock_held(mm);
```



HOW DO WE PREVENT THIS?

- In the case of this specific ELF vulnerability, the fix is to hold mmap_sem while the data region for the library is created
- However, memory corruption vulnerabilities are often tricky to prevent; they represent a large fraction of all security holes!

TAKE CS 263 TO LEARN MORE

FALL 2016