



Linux Kernel Exploitation 101

Spawn your uid=0 like ninja



Indice

- Introduzione teorica al kernel
- Kernel Debugging
- Vulnerabilità comuni
- Mitigation & Exploitation



Cos'è il kernel

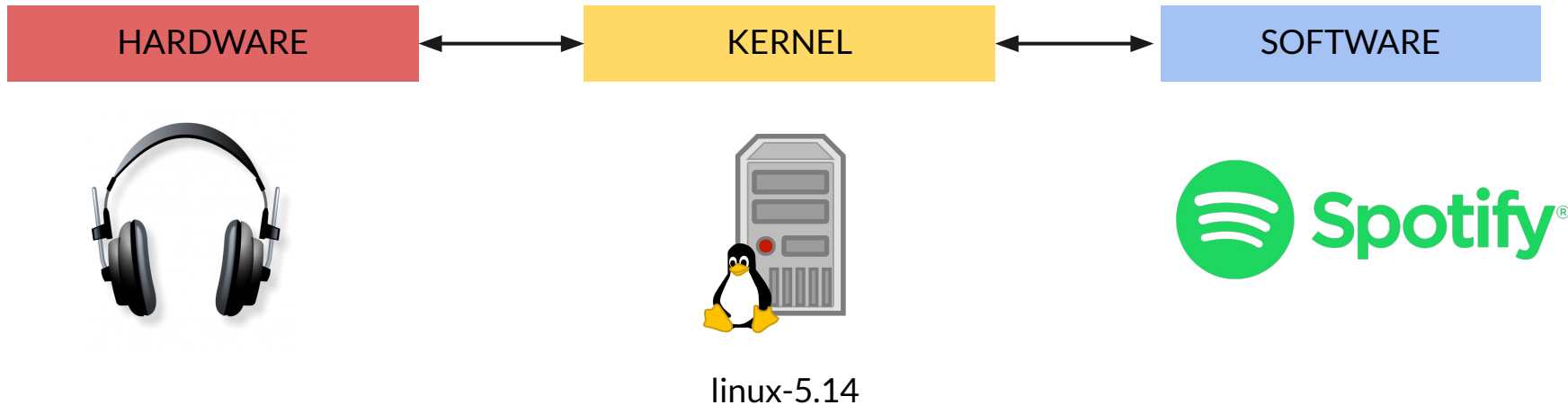
Know your enemy



Comunicazione con HW e SW



Esempio: Vuoi sentire musica da spotify

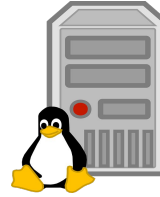


SOFTWARE

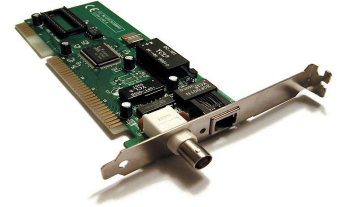
KERNEL

HARDWARE

```
curl https://target.com/
```



linux-5.14

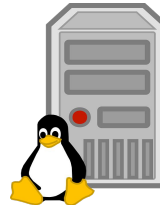


WiFi/ethernet/..

SOFTWARE

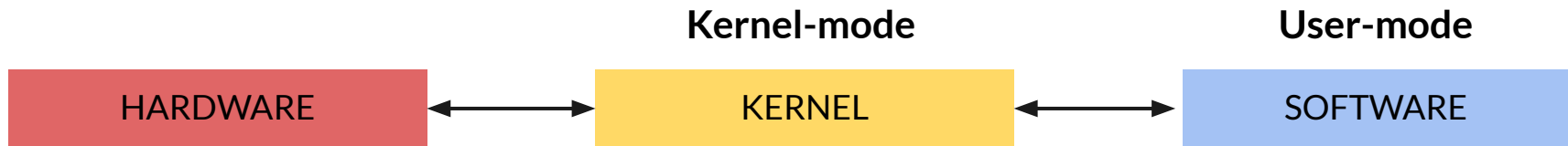
KERNEL

HARDWARE



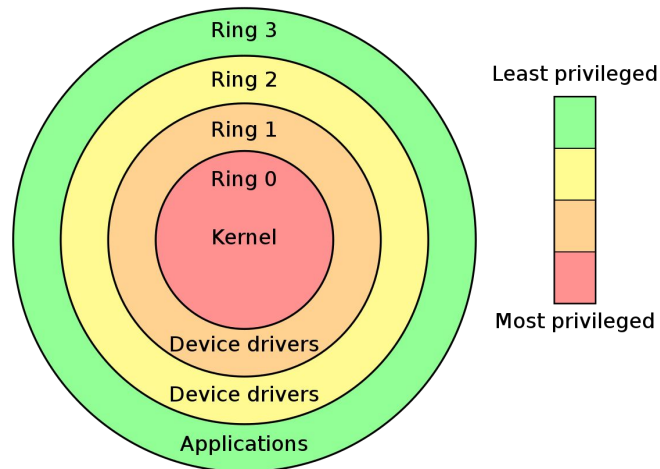
linux-5.14

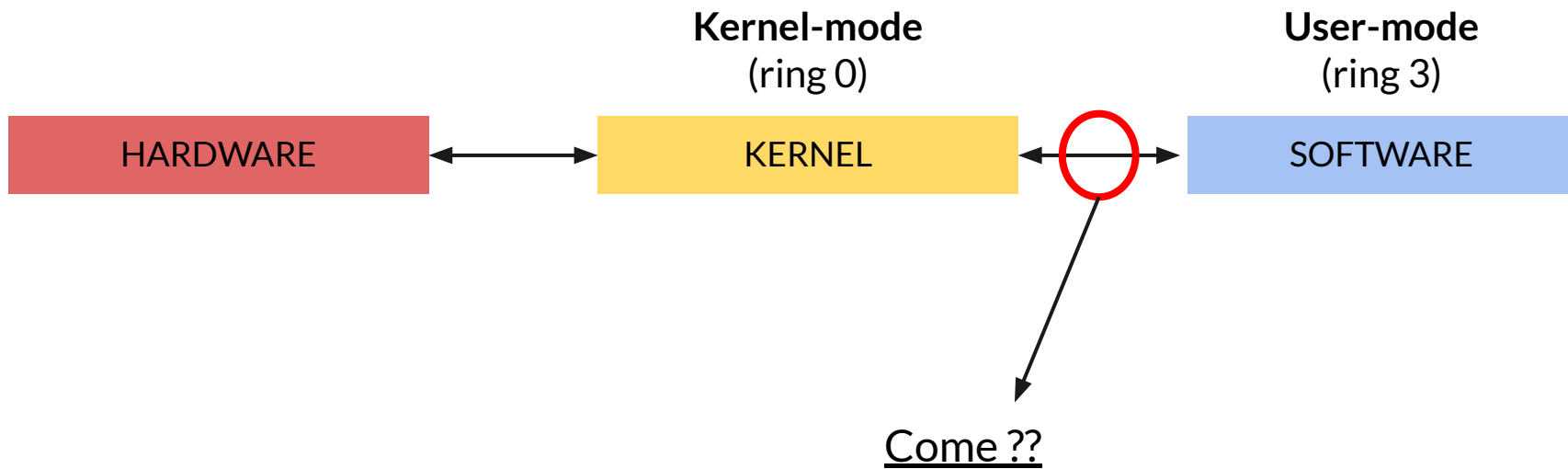




User-Mode vs Kernel-Mode

- User-Mode (aka User-Land / User-Space)
 - Applicazioni software (Spotify, Browser (Chrome, Safari,...), Teams, ..)
 - Privilegi limitati (**ring 3**)
 - Comunica **costantemente** con il kernel (volontariamente e non)
 - root user (**root != kernel**)
- Kernel-Mode (aka Kernel-Land / Kernel-Space)
 - **Ring 0**
 - Driver
 - Interazione con l'hardware
 - Schede grafiche/rete, chip Bluetooth/4G/, mouse, tastiere, periferiche audio
 - Gestione dei privilegi
 - Operazioni più comuni
 - Allocazione memoria per applicazione user-mode
 - Comunicazione tra processi (IPC)
 - Comunicazione network (TCP/IP, ...)
- Ring 1 e 2 nativamente non utilizzati
 - Ring 1 utilizzato da virtualizzazione (VirtualBox, VMWare, ..)







User-mode => Kernel-Mode

- [syscall](#)
 - Wrappers
 - [read](#), write, open, close
 - [socket](#), bind, connect, listen
 - [mmap](#), [mprotect](#)
 - [ioctl](#) (Input Output ConTroL)
 - <https://filippo.io/linux-syscall-table/>



Kernel-Mode => Hardware (Driver)

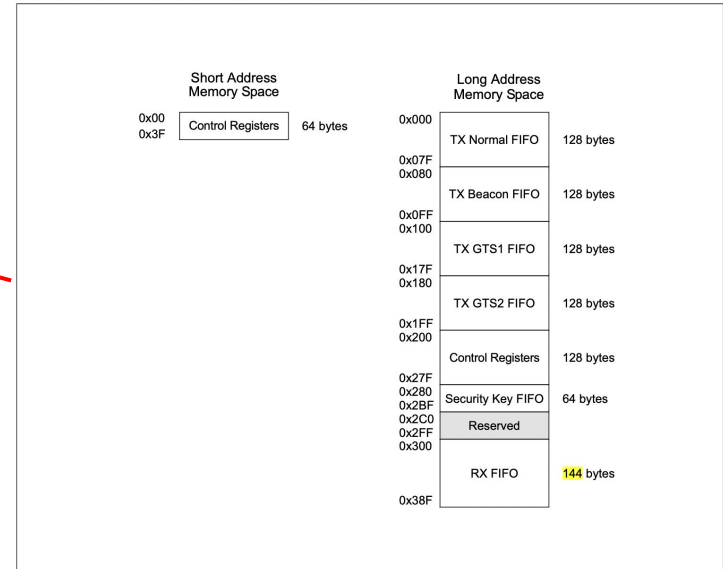
- Comunica direttamente con l'hardware tramite driver
- Un driver è un “modulo” aggiuntivo che stabilisce una comunicazione tra hardware e kernel
 - Il driver viene scritto seguendo il “datasheet” della periferica
- Può fare da “ponte” per l'interazione tra user-mode e hardware
 - Per esempio per la riproduzione di file musicali (sequenza di bit).

Kernel-Mode => Hardware - Esempio MRF24J40

- Implementazione del driver per il microchip MRF24J40 802.15.4
- Datasheet: <https://ww1.microchip.com/downloads/en/DeviceDoc/39776C.pdf>
- Driver: </drivers/net/ieee802154/mrf24j40.c>

```
#define TX_FIFO_SIZE 128 /* From datasheet */  
#define RX_FIFO_SIZE 144 /* From datasheet */  
#define SET_CHANNEL_DELAY_US 192 /* From datasheet */
```

FIGURE 2-6: MEMORY MAP FOR MRF24J40





Hardware => Kernel-Mode

- L'HW si fa "notare" dal kernel utilizzando gli Interrupt
- Una periferica (e.g. una scheda di rete) manda un interrupt alla CPU
- Il kernel gestisce l'interrupt (Interrupt Handling) ed instrada l'informazione dove necessario
 - Per esempio, se si tratta di un pacchetto arrivato ad una scheda di rete, questo dovrà essere elaborato in base al tipo di protocollo.
 - I tasti che premiamo sulla tastiera mandando un Interrupt all'Interrupt Handler
- Diversi tipi di Interrupt
 - Maskable/Non Maskable
 - Classificati per priority



Kernel Memory

Stack, Context, Heap (SLA|O|UB)

Stack vs Heap



Stack

- Statico
 - Le allocazioni sono determinate a compile-time
- Size-limited
- Più veloce
- Salva variabili locali alla funzione
- I registri RSP & RBP delimitano lo stack
- Ogni funzione ha il proprio stack

Heap

- Dinamico
 - Allocazioni determinate runtime
- alloc/free manuali
 - kmalloc/kfree
- Frammentazione

Heap allocations / Memory Management



- SLAB
 - Precursore
 - EOL (verrà rimosso)
 - Termine generico anche nell'implementazione di SLOB/SLUB
- SLOB
 - Ottimizzato per sistemi Embedded
- SLUB
 - Linux > 2.6.12

SLUB



- PAGE_SIZE allocations (4096)
- Divisione per grandezza
 - 8, 16, 32, 64, 128 ... 4k, 8k (x86_64)
 - 128, 256, 512, .. 6k, 8k (ARM64)
- Divisione per tipo
 - GFP_KERNEL / GFP_KERNEL_ACCOUNT
 - kmalloc-16 / kmalloc-cg-16 / special caches
- Chunk
 - Un oggetto in memoria
- slab/cache
 - Un insieme di allocazioni dello stesso tipo
 - e.g. kmalloc-8 può avere N slabs che contengono ognuno (PAGE_SIZE / 8) chunks

SLUB / partial slabs



slab 1



slab 2



slab 3



SLUB / partial slabs



slab 1



slab 2



slab 3



SLUB / partial slabs



partial slab 1

slab 1



partial slab 2

slab 2



slab 3



SLUB / partial slabs



slab 1



partial slab 2

slab 2



slab 3



SLUB / partial slabs



slab 1



partial slab 2

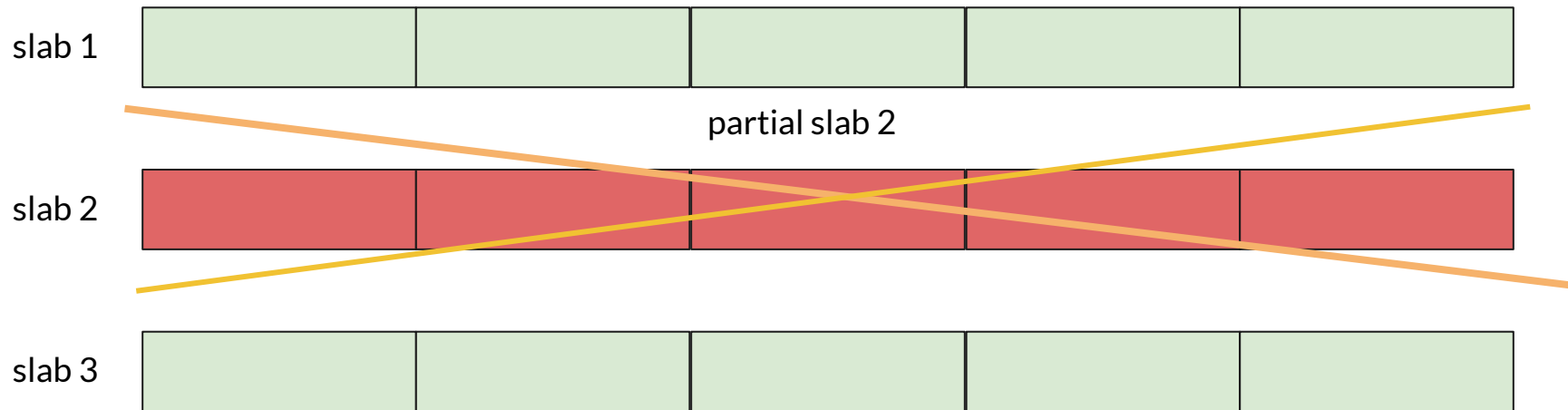
slab 2



slab 3



SLUB / partial slabs



SLUB



- APIs
 - [kmalloc\(\)](#)
 - flags: GFP_KERNEL | GFP_KERNEL_ACCOUNT | ...
 - kfree()
 - Special-purpose caches
 - kmem_cache_create()
 - kmem_cache_alloc()
- [struct kmem_cache](#) / [struct kmem_cache_cpu](#)
 - Struttura che descrive la cache

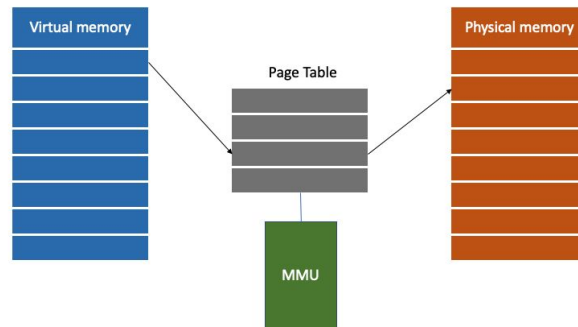
SLUB - Useful files



- `/proc/slabinfo`
- `/sys/kernel/slab/`
- `/sys/kernel/slab/<CACHE>/partial`
- `/sys/kernel/slab/<CACHE>/objects_size`
- `/sys/kernel/slab/<CACHE>/objs_per_slab`

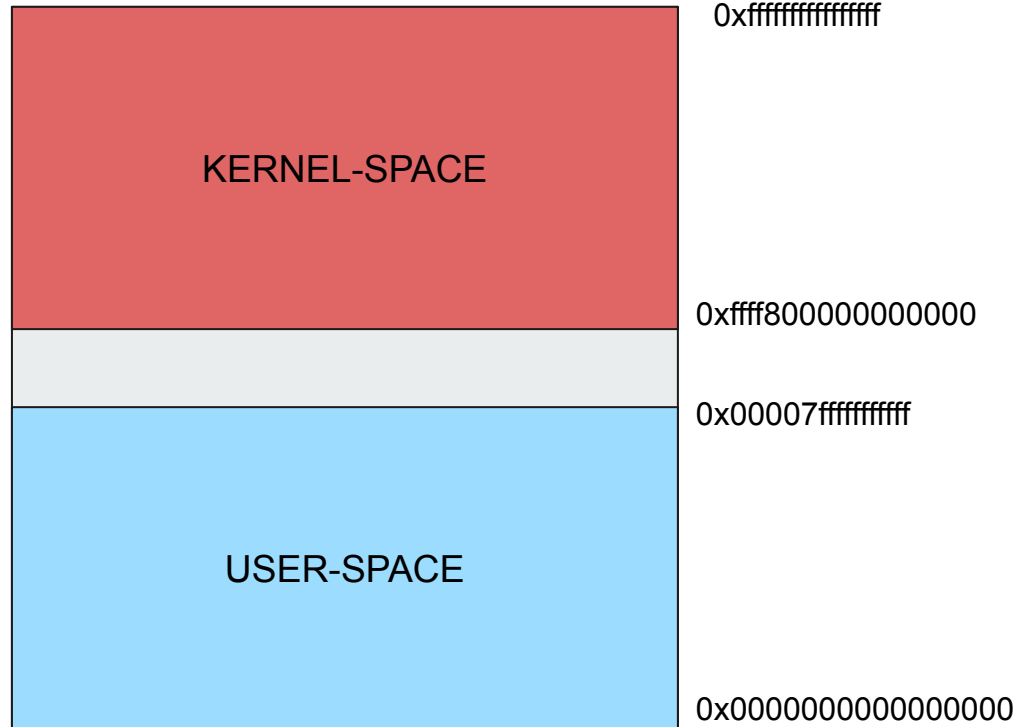
Pointers: Kernel vs User

- [Documentation/x86/x86_64/mm.rst](#)
- 4-level page tables
 - Più utilizzato su x86_64 (in base alla CPU)
 - Userspace
 - `0x0000000000000000 => 0x00007fffffffffff`
 - Kernel
 - `0xffff800000000000 => 0xffffffffffffffff`
- 5-level page tables
 - [Documentation/x86/x86_64/mm.rst#L75](#)
- Configuration file: [arch/x86/Kconfig](#)



```
root@kernel-exploitation:~/hacking/kernel# cat /boot/config-5.4.0-122-generic | grep PGTA
CONFIG_PGTABLE_LEVELS=4
root@kernel-exploitation:~/hacking/kernel#
```

Kernel mapped on users process



Kernel memory mapped on user-space

- È possibile dunque accedere alla memoria del kernel tramite user-mode?
 - NO!
 - Se sì, è una vulnerabilità (o post-exploitation).
- Anche le operazioni da kernel-mode a user-mode sono controllate.
 - <https://elixir.bootlin.com/linux/v5.19-rc7/source/lib/usercopy.c#L10>
 - https://elixir.bootlin.com/linux/latest/source/include/asm-generic/access_ok.h#L31

```
__copy_from_user(void *to, const void __user *from, unsigned long n)
{
    unsigned long res = n;
    might_fault();
    if (likely(access_ok(from, n))) {
        instrument_copy_from_user(to, from, n);
        res = raw_copy_from_user(to, from, n);
    }
    if (unlikely(res))
        memset(to + (n - res), 0, res);
    return res;
}
```



```
static inline int __access_ok(const void __user *ptr, unsigned long size)
{
    unsigned long limit = TASK_SIZE_MAX;
    unsigned long addr = (unsigned long)ptr;

    if (IS_ENABLED(CONFIG_ALTERNATE_USER_ADDRESS_SPACE) ||
        !IS_ENABLED(CONFIG_MMU))
        return true;

    return (size <= limit) && (addr <= (limit - size));
}
```

copy_[from|to]_user



- copy from user

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

- “Copy a block of data from user space”

- copy from user

- “Copy a block of data from user space, **with less checking**” (no *access_ok*)

- copy to user

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

- “Copy a block of data into user space.”

- copy to user

- “Copy a block of data into user space, **with less checking**” (no *access_ok*)

copy_[from|to]_user checks



- Puntatori
 - Che i due parametri “to” e “from” siano validi (tramite *access_ok()*)
 - **copy_from_user**
 - **from**: user pointer (< 0x00007fffffffffff)
 - **copy_to_user**
 - **to**: user pointer (< 0x00007fffffffffff)
- CONFIG_HARDENED_USERCOPY
 - Controlli sulla grandezza dei chunk



Common vulnerabilities

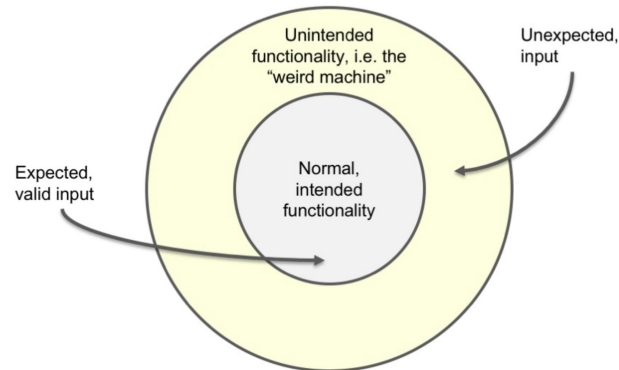


Common vulnerabilities (memory corruptions)

- Stack Overflow
- Heap Overflow/Underflow
 - Large overflows / off-by-one / Integer Overflow...
- Use-After-Free
 - UAF read/write
 - UAF self-referencing pointer (linked lists)
- Double Free
- Race Conditions / refcount
 - => UAF/Hep ovf/..

Weird machine

- Che cos'è una corruzione di memoria?
 - Si accede (r|w|x) ad una zona di memoria in maniera non prevista dal programma
 - Mancati controlli (e.g. di una size)
 - Errati controlli (assunzioni non corrette)
- Come si sfrutta?
 - **GIOCANDO** e **MANIPOLANDO** la memoria ed il programma
 - Creando una **Weird Machine**
 - Far fare ad un programma ciò per cui non è stato progettato (e.g. eseguire uno shellcode/ROP => RCE).
 - Bypass mitigazioni correnti



Common Mitigations



- Comuni
 - kASLR
 - SMAP
 - SMEP
- Extra
 - CONFIG_FG_KASLR
 - CONFIG_STATIC_USERMODEHELPER
 - CONFIG_SLAB_FREELIST_RANDOM/HARDENED
 - CONFIG_DEBUG_LIST
 - CONFIG_HARDENED_USERCOPY
 - CONFIG_ARM64_UAO
 - CONFIG_FORTIFY_SOURCE
 - CONFIG_MEMCG
 - ..



Mitigazioni == Impossibile scrivere exploit ?

- NO !
- Ma ci sono dei cambiamenti
 - Ogni singola vulnerabilità fa caso a se
 - ~~Stack Overflow => Overwrite RIP => shellcode => PWN~~
 - ~~Heap Overflow => Unlinking => RIP control => PWN~~
- Sempre **piú difficile** (nuove mitigation appaiono ogni anno)
- Ma anche **piú divertente** =) (e stressante)

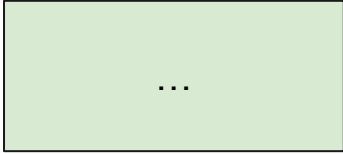


Heap Overflow

Heap Overflow

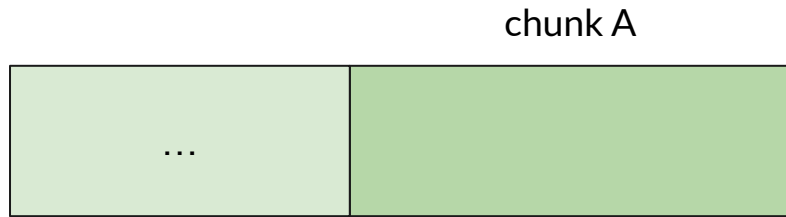


- Concetto base
 - Oggetto **A** sovrascrive oggetto **B** (allocato successivamente all'oggetto A)
- In memoria, **oggetto** prende il nome di **chunk**



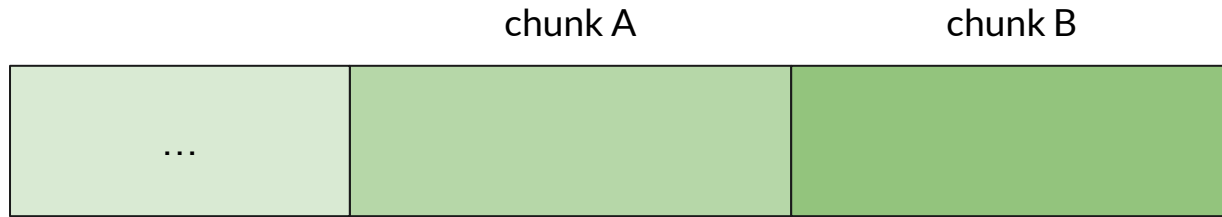
Heap Overflow

```
A = kmalloc(32);
```



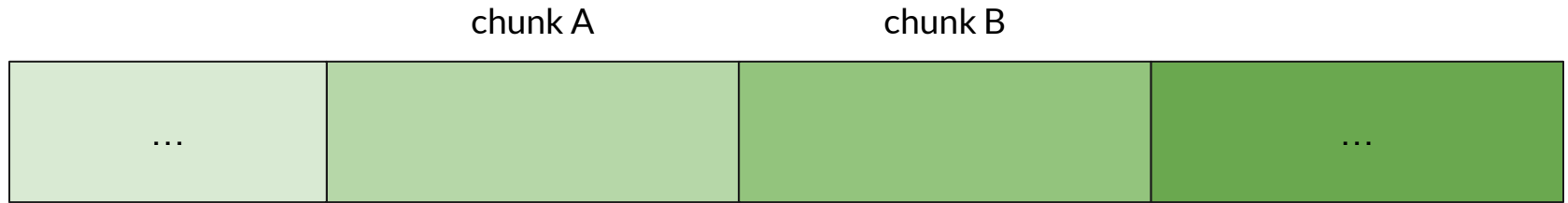
Heap Overflow

```
A = kmalloc(32);  
B = kmalloc(32);
```



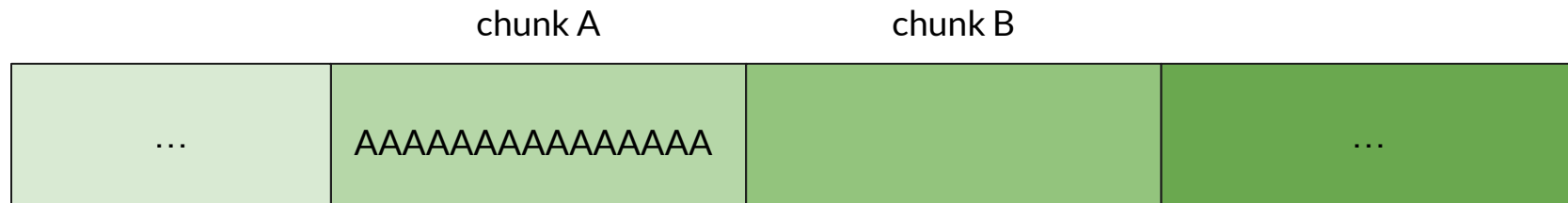
Heap Overflow

```
A = kmalloc(32);  
B = kmalloc(32);  
.. kmalloc(...)... // Altre allocazioni
```



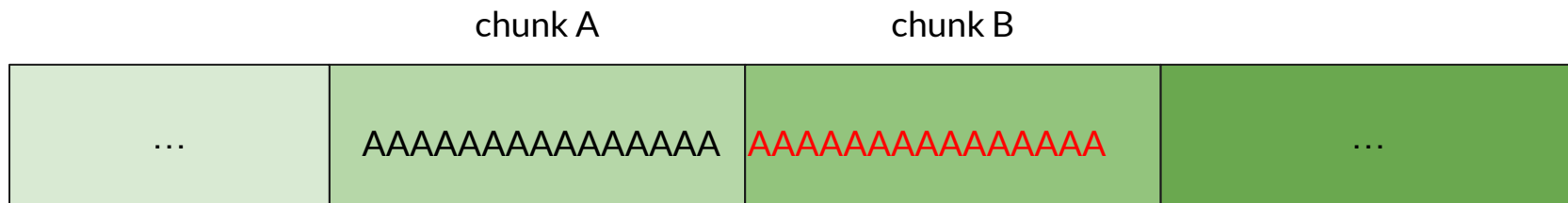
Heap Overflow

```
A = kmalloc(32);  
B = kmalloc(32);  
.. kmalloc(...)... // Altre allocazioni  
memset(A, 0x41, 32);
```



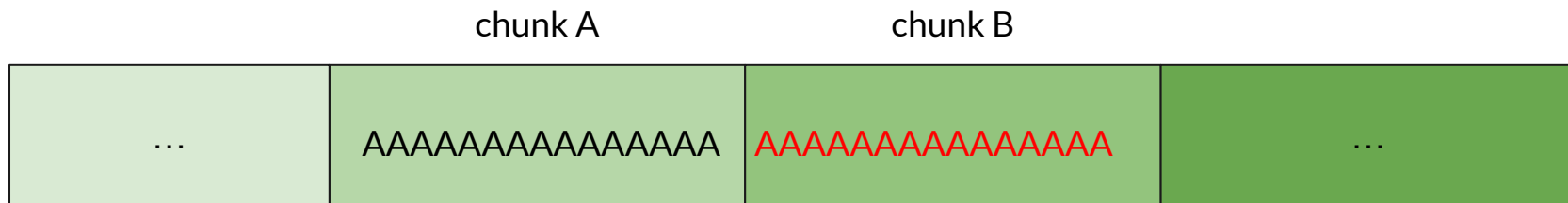
Heap Overflow

```
A = kmalloc(32);  
B = kmalloc(32);  
.. kmalloc(...)... // Altre allocazioni  
memset(A, 0x41, 32);  
memset(A, 0x41, 64);
```



Heap Overflow

```
A = kmalloc(32);  
B = kmalloc(32);  
.. kmalloc(...) // Altre allocazioni  
memset(A, 0x41, 32);  
memset(A, 0x41, 64);  
B->function() // => Corrupted from chunk A => RIP = 0x4141414141414141
```



A => **Target** object
B => **Victim** object

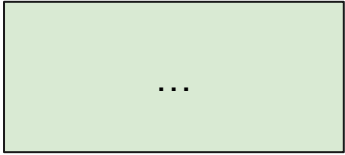
- Target** object
- Oggetto dove si presenta la vulnerabilità
- Victim** object
- Oggetto scelto arbitrariamente dall'attaccante (per essere corrotto ad-hoc)

Heap Overflow

Use-After-Free



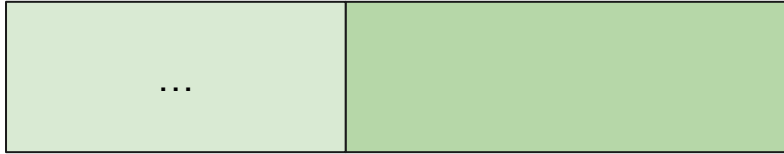
- Concetto base
 - Un oggetto, precedentemente de-allocato (free), viene riutilizzato



Use-After Free

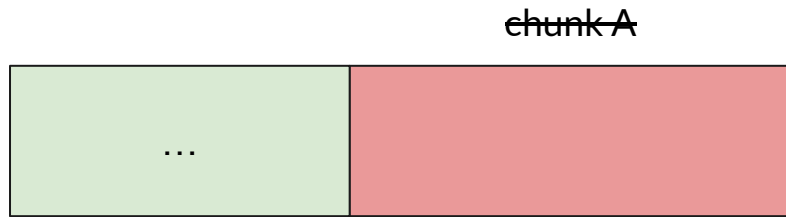
```
A = kmalloc(32);
```

chunk A



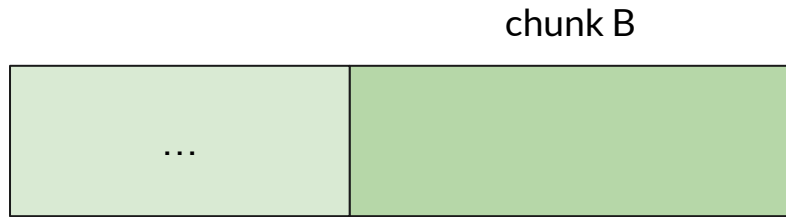
Use-After Free

```
A = kmalloc(32);  
free(A);
```



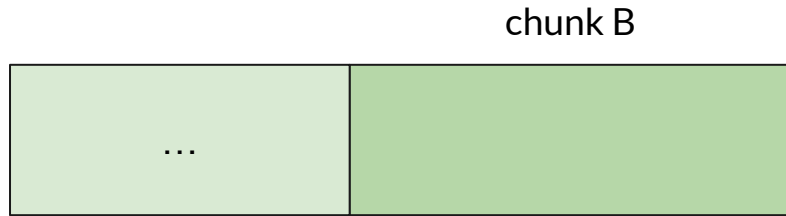
Use-After Free

```
A = kmalloc(32);  
free(A);  
B = kmalloc(32);
```



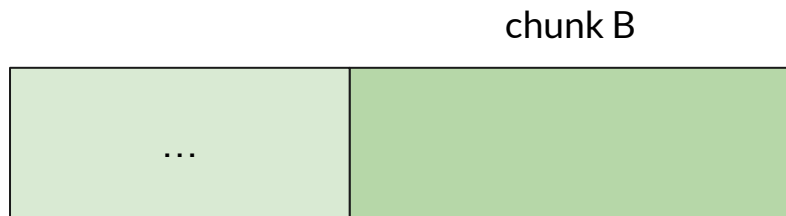
Use-After Free

```
A = kmalloc(32);  
free(A);  
B = kmalloc(32);  
A->function(32); // Access undefined values => RIP: ????
```



Use-After Free

```
A = kmalloc(32);  
free(A);  
B = kmalloc(32);  
A->function(32); // Access undefined values => RIP: ????
```



A => **Target** object
B => **Victim** object

Target object

- Oggetto dove si presenta la vulnerabilità

Victim object

- Oggetto scelto arbitrariamente dall'attaccante (per essere corrotto ad-hoc)

Use-After Free



Common Mitigations

KASLR, SMAP, SMEP

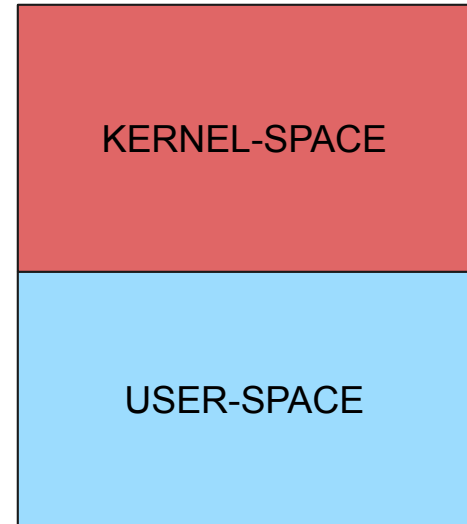
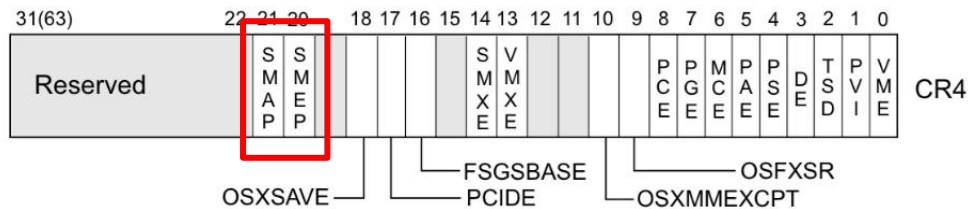


KASLR

- Randomizzazione degli indirizzi
- Richiede, quasi sempre, la necessità di una Information Disclosure
 - Per ottenere altri indirizzi
- Esempio
 - Un leak di un pointer nella sezione .text del kernel permette di ottenere tutti gli altri indirizzi
- Scenario
 - Pre-ASLR
 - Arbitrary Write => Write ad un indirizzo conosciuto => uid=0
 - ASLR
 - Arbitrary Write
 - Se possibile trasformarlo in un Arbitrary/OOB read => uid=0
 - Oppure necessità di un'altra vulnerabilità che permette Information Disclosure => uid=0

SMAP / SMEP (x86_64)

- SMAP (Supervisor Mode **A**ccess Prevention)
 - Genera un “fault” se si **accede** alla memoria user-space
 - “Allows supervisor mode programs to optionally set user-space memory mappings so that access to those mappings from supervisor mode will cause a trap” (Wikipedia)
- SMEP (Supervisor Mode **E**xecution Prevention)
 - Genera un “fault” se si **esegue** dalla memoria user-space
- Gestito dal registro CR4





Come fa `copy_from/to|user` a funzionare?

- Disabilitando temporaneamente SMAP (dal registro CR4)

STEPS (esempio)

1. `copy_from_user()`
2. [Disable SMAP](#) (ASM_STAC)
3. Interazione con user-space
4. [Enable SMAP](#) (ASM_CLAC)

SMEP / Supervisor Mode **Execution** Prevention



SMEP / Supervisor Mode **Execution** Prevention



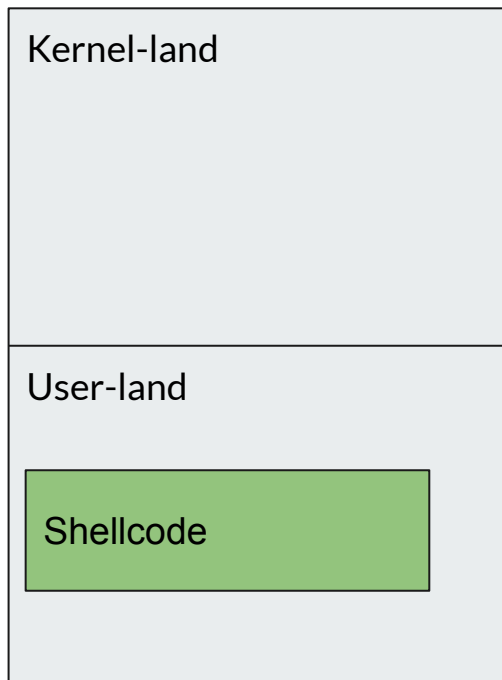
RIP control



SMEP / Supervisor Mode **Execution** Prevention

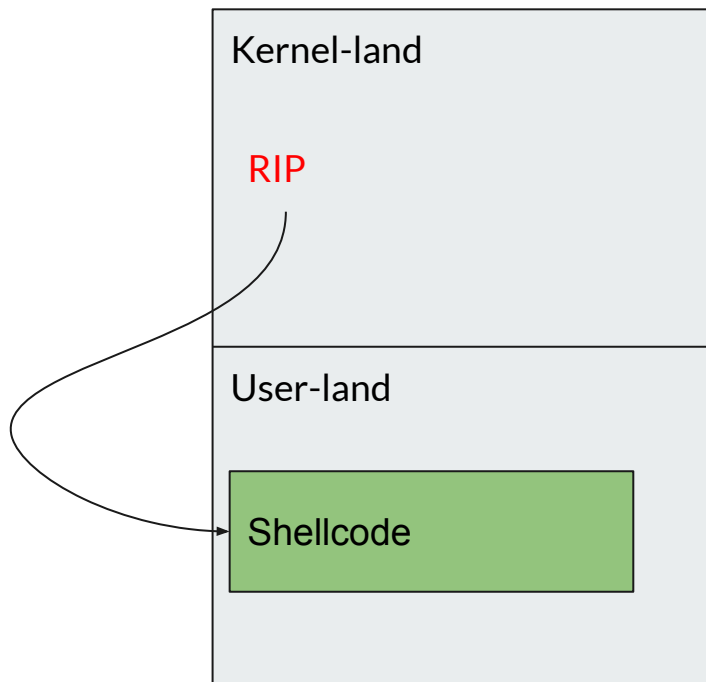


RIP control

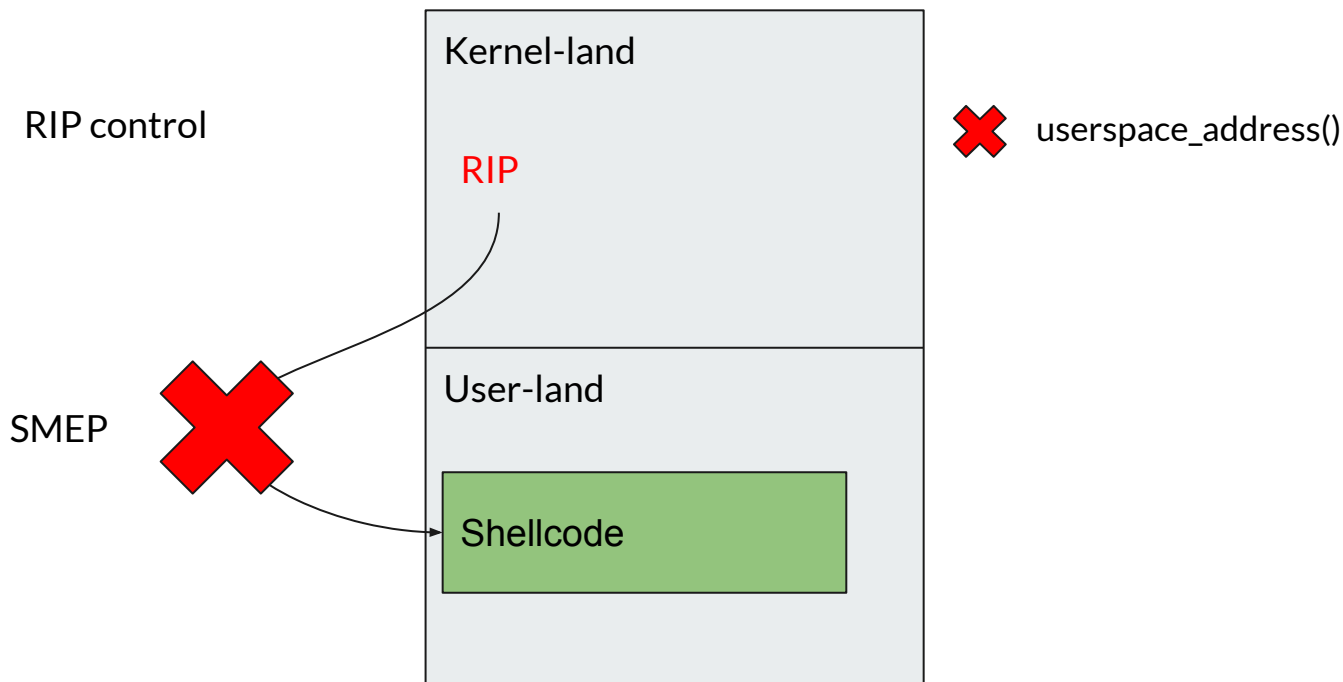


SMEP / Supervisor Mode **Execution** Prevention

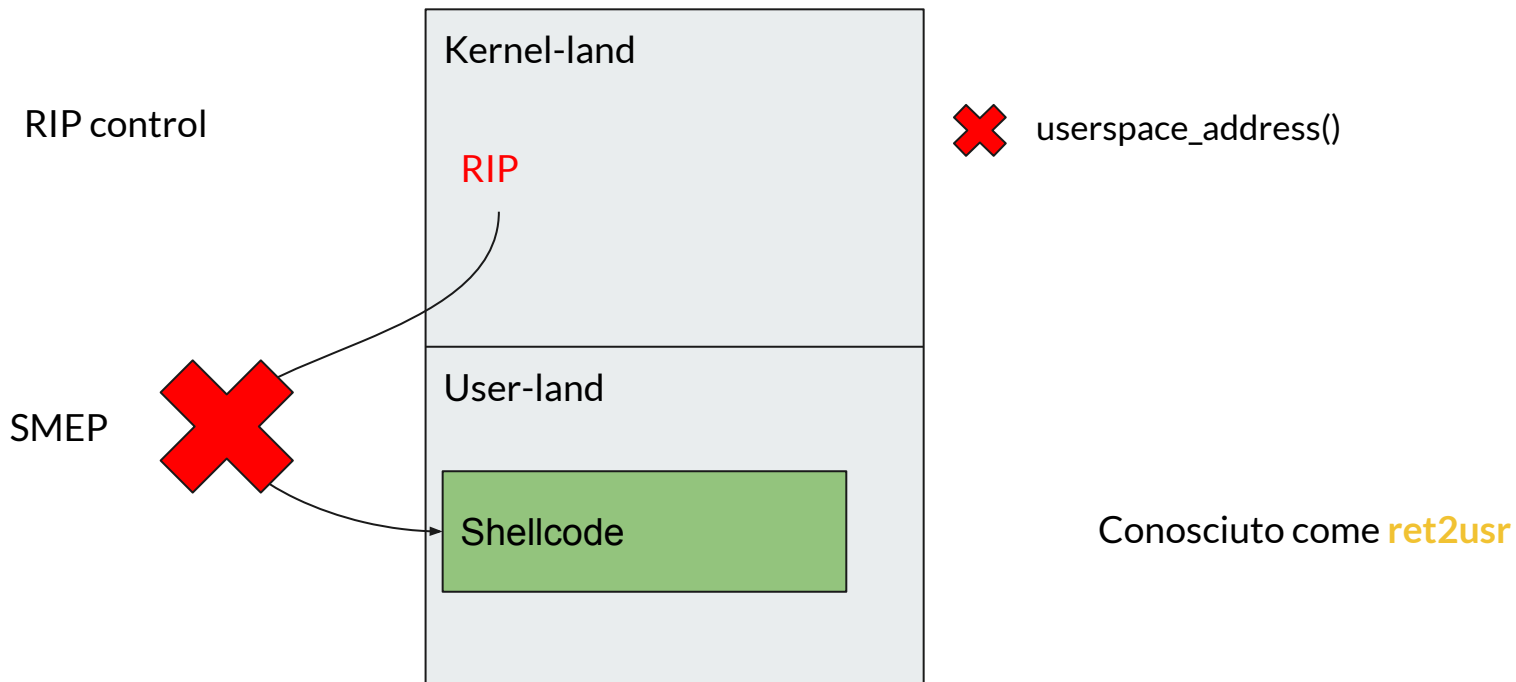
RIP control



SMEP / Supervisor Mode **Execution** Prevention

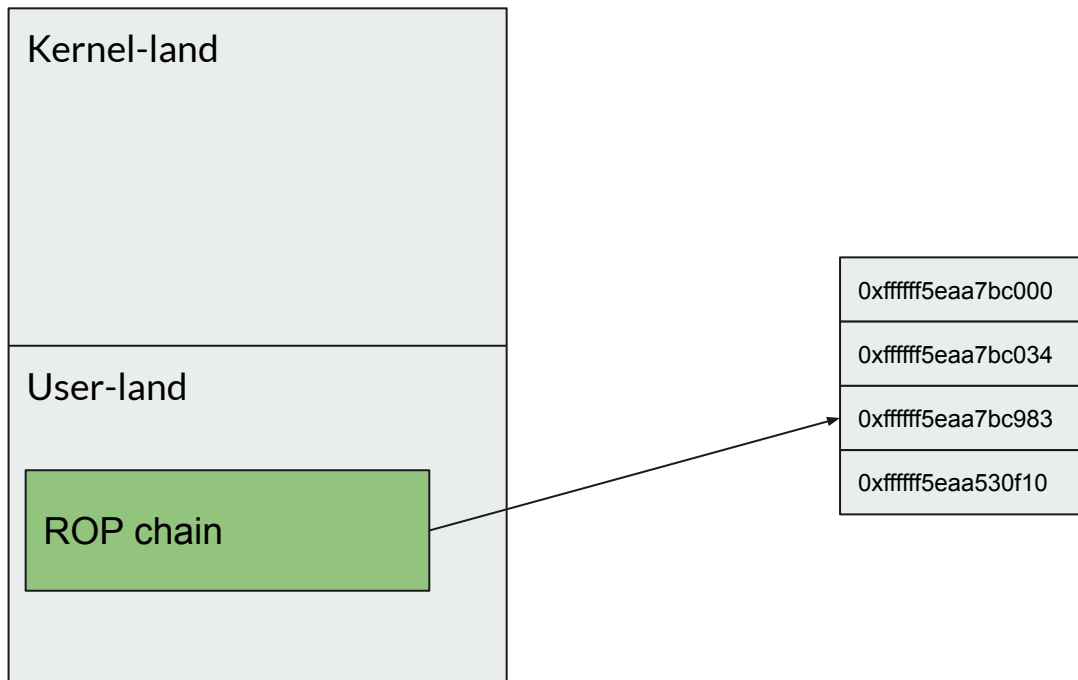


SMEP / Supervisor Mode **Execution** Prevention

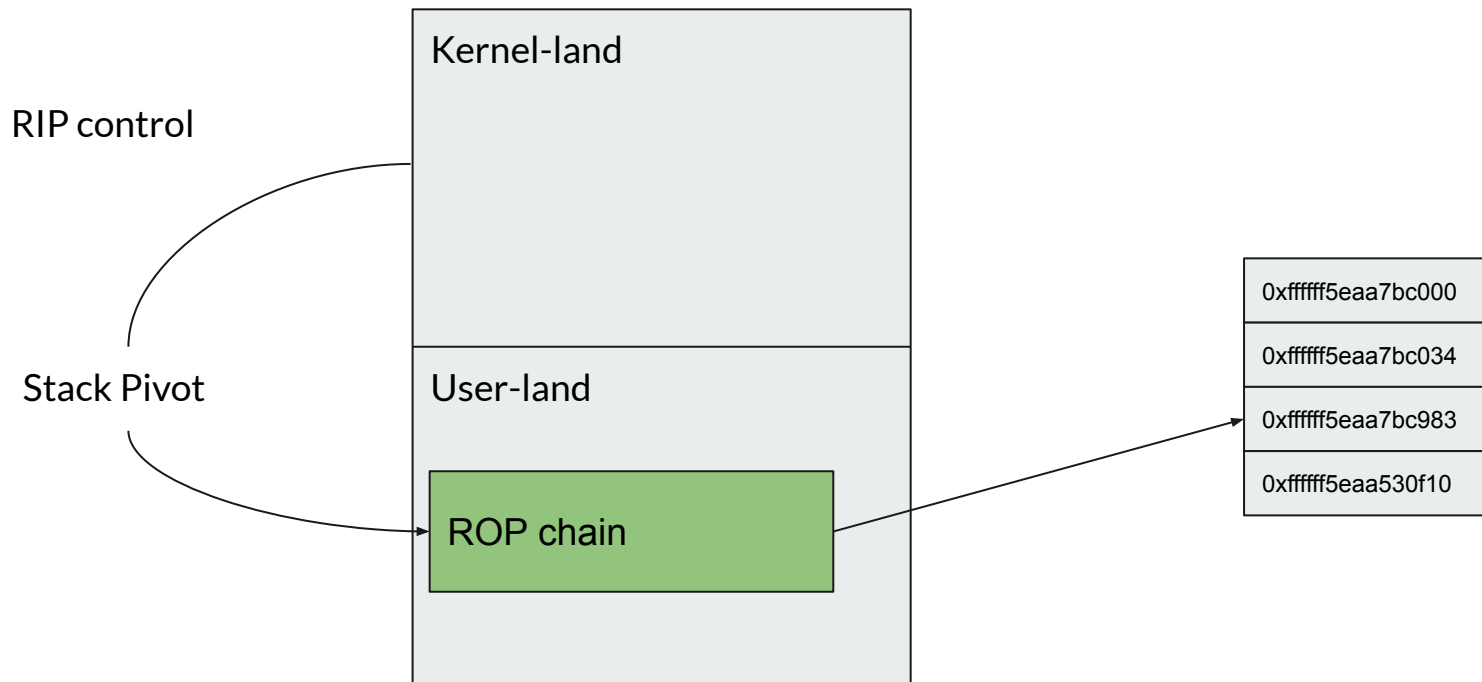


SMAP / Supervisor Mode **Access** Prevention

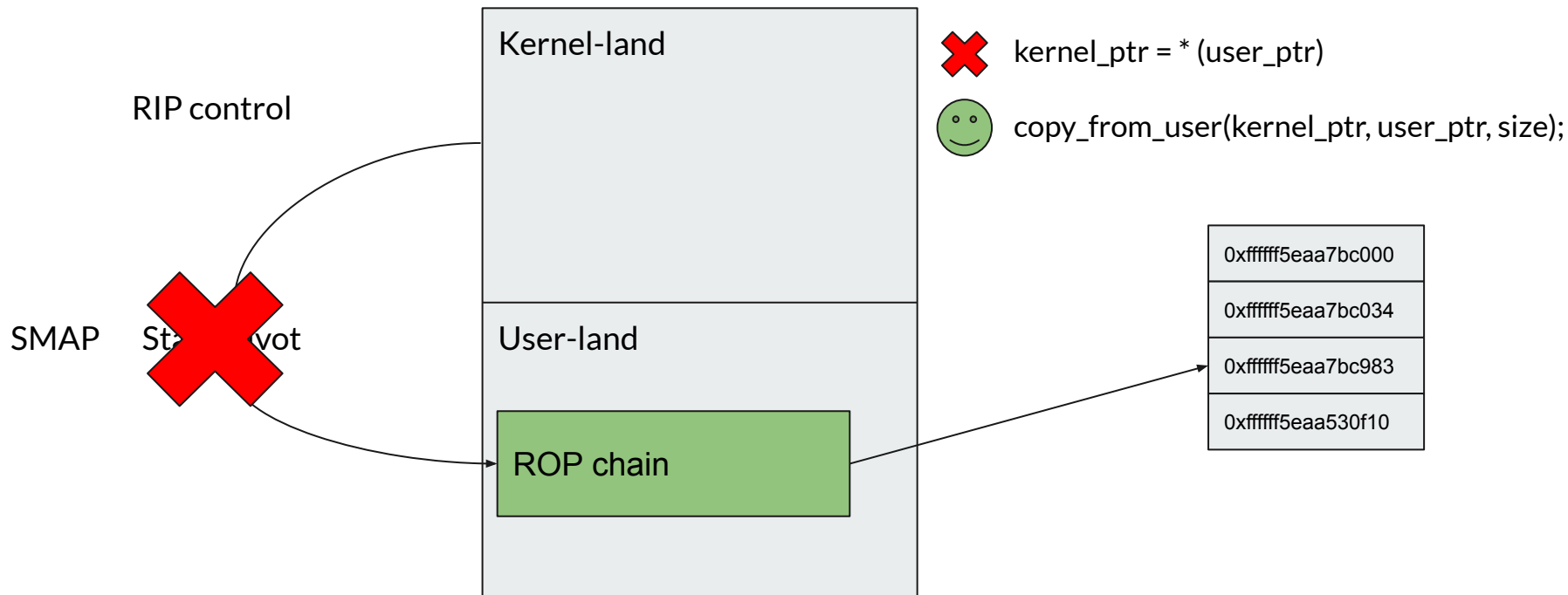
RIP control



SMAP / Supervisor Mode **Access** Prevention



SMAP / Supervisor Mode **Access** Prevention





SMAP/SMEP

- SMAP/SMEP sono mitigation **Intel**
 - e.g. anche Windows
- **ARM** ha il corrispettivo PXN/PAN
 - Stesso concetto, implementazione kernel/cpu diversa



Exploitation techniques

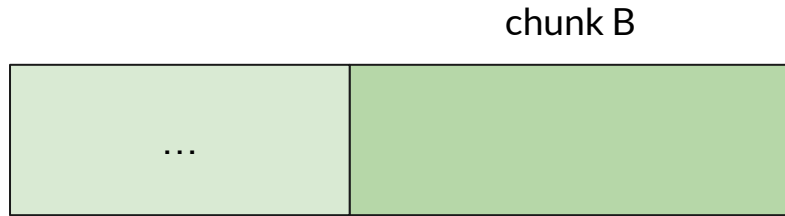
Attack kernel structures



Exploitation strategies

- Dipende dalla vulnerabilità
- L'obiettivo (di solito e se possibile) è costruirsi una primitiva R/W
 - Se hai r/w sul kernel, la root é (generalmente) solo questione di tempo (e di bypass di mitigation)
- Caso comune
 - Si parte da una vuln con r/w limitato per arrivare ad una arbitrary R/W
 - Da un primitiva write è possibile costruirsi una R/W
 - Se controlli il RIP (e.g. tramite la corruzione di una funzione kernel)
 - Possibile costruire una ROP/JOP a seconda delle mitigation
- Si ha quasi sempre bisogno di un information leak
 - Per bypassare KASLR
- Chaining di più vulnerabilità

```
A = kmalloc(32);  
free(A);  
B = kmalloc(32);  
A->function(32); // Access undefined values => RIP: ????
```



A => **Target** object
B => **Victim** object

Target object

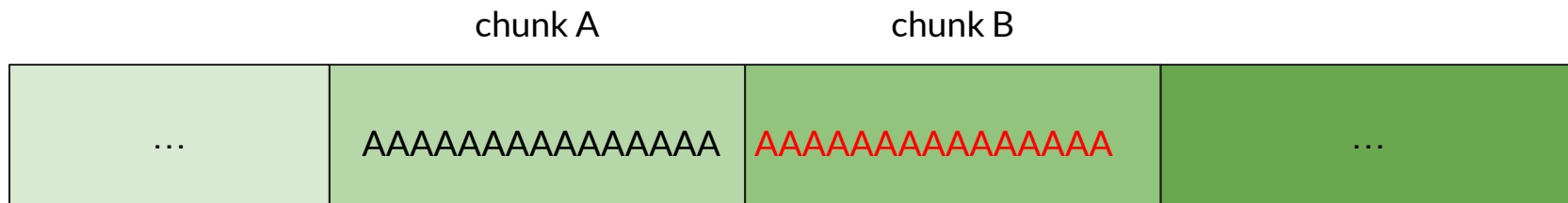
- Oggetto dove si presenta la vulnerabilità

Victim object

- Oggetto scelto arbitrariamente dall'attaccante (per essere corrotto ad-hoc)

Use-After Free

```
A = kmalloc(32);  
B = kmalloc(32);  
.. kmalloc(...) // Altre allocazioni  
memset(A, 0x41, 32);  
memset(A, 0x41, 64);  
B->function() // => Corrupted from chunk A => RIP = 0x4141414141414141
```



A => **Target** object
B => **Victim** object

Target object

- Oggetto dove si presenta la vulnerabilità

Victim object

- Oggetto scelto arbitrariamente dall'attaccante (per essere corrotto ad-hoc)

Heap Overflow

Victim Object



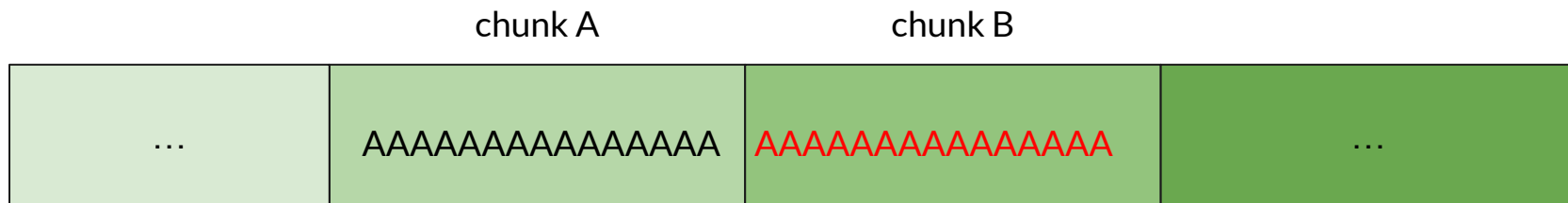
- Un oggetto che viene utilizzato per ottenere altre primitive
 - R/W/X
 - [link 1](#) / [link 2](#)
 - shm_file_data
 - msg_msg
 - Tty_struct
 - ...
- Utilizzi
 - Heap Overflow
 - Oggetto che viene allocato adiacente all'oggetto vulnerabile per essere sovrascritto
 - UAF
 - Oggetto che viene rimpiazzato

Victim Object



- Un oggetto che viene utilizzato per ottenere altre primitive
 - R/W/X
 - [link 1](#) / [link 2](#)
 - shm_file_data
 - msg_msg
 - Tty_struct
 - ...
- Utilizzi
 - Heap Overflow
 - Oggetto che viene allocato adiacente all'oggetto vulnerabile per essere sovrascritto
 - UAF
 - Oggetto che viene rimpiazzato

```
A = kmalloc(32);  
B = kmalloc(32);  
.. kmalloc(...) // Altre allocazioni  
memset(A, 0x41, 32);  
memset(A, 0x41, 64);  
B->function() // => Corrupted from chunk A => RIP = 0x4141414141414141
```



A => **Target object**
B => **Victim object**

Target object

- Oggetto dove si presenta la vulnerabilità

Victim object

- Oggetto scelto arbitrariamente dall'attaccante (per essere corrotto ad-hoc)

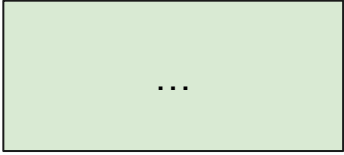
Heap Overflow



Victim Object

- Un oggetto che viene utilizzato per ottenere altre primitive
 - R/W/X
 - [link 1](#) / [link 2](#)
 - Shm_file_data / msg_msg / tty_struct / ..
- Utilizzi
 - Heap Overflow
 - Oggetto che viene allocato adiacente all'oggetto vulnerabile per essere sovrascritto
 - UAF
 - Oggetto che viene rimpiazzato
- **Prerequisito**
 - L'oggetto deve essere allocato nella stessa cache
 - Target obj = kmalloc-32
 - Victim obj = kmalloc-32 (**must**)
 - Altrimenti cross-cache attacks
 - <https://duasynt.com/blog/linux-kernel-heap-feng-shui-2022>
 - Non essere de-allocati prima di sfruttare la vulnerabilità

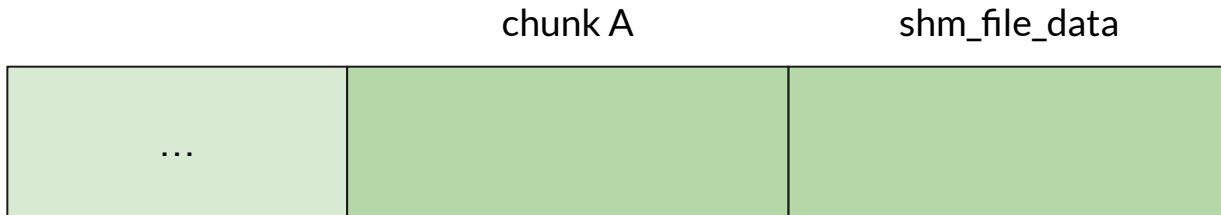
kmalloc-32



Heap Overflow (Read)

kmalloc-32

```
struct shm_file_data {  
    int id;  
    struct ipc_namespace *ns;  
    struct file *file;  
    const struct vm_operations_struct *vm_ops;  
};
```

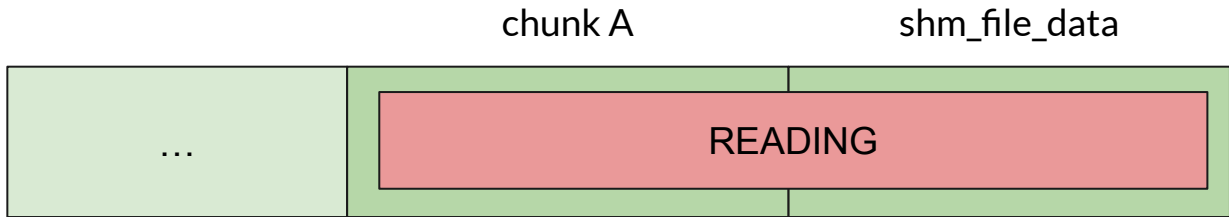


Heap Overflow (Read)

kmalloc-32

```
struct shm_file_data {  
    int id;  
    struct ipc_namespace *ns;  
    struct file *file;  
    const struct vm_operations_struct *vm_ops;  
};
```

read(A, 64); // Read chunk A and shm_file_data
// => leak vm_ops (kernel .data) => **ASLR DEFEATED**

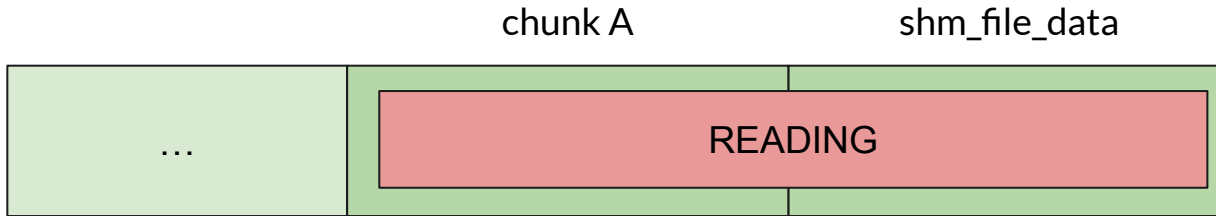


Heap Overflow (Read)

kmalloc-32

```
struct shm_file_data {  
    int id;  
    struct ipc_namespace *ns;  
    struct file *file;  
    const struct vm_operations_struct *vm_ops;  
};
```

read(A, 64); // Read chunk A and shm_file_data
// => leak vm_ops (kernel .data) => **ASLR DEFEATED**



Stesso concetto per **Write Overflow**

Heap Overflow (Read)



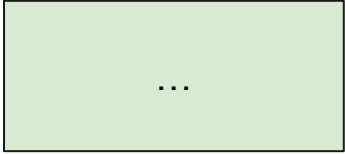
Other common vulnerabilities

Double Free, Double Fetches



Double Free

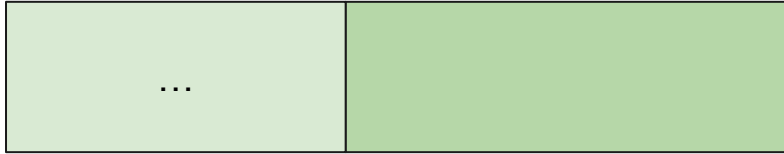
- `kfree()` lo stesso indirizzo due volte



Double Free

```
A = kmalloc(32);
```

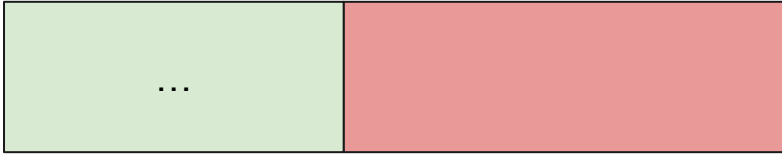
chunk A



Double Free

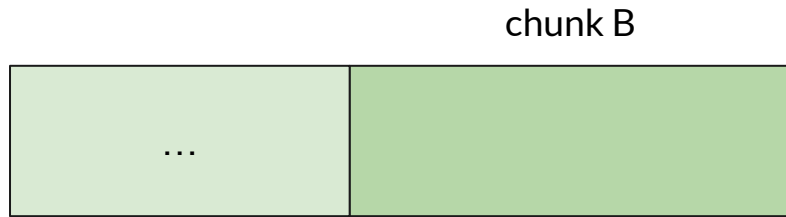
```
A = kmalloc(32);  
kfree(A);
```

chunk A



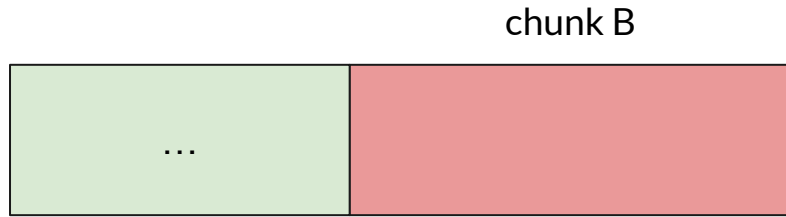
Double Free

```
A = kmalloc(32);  
kfree(A);  
B = kmalloc(32); // replace chunk A
```



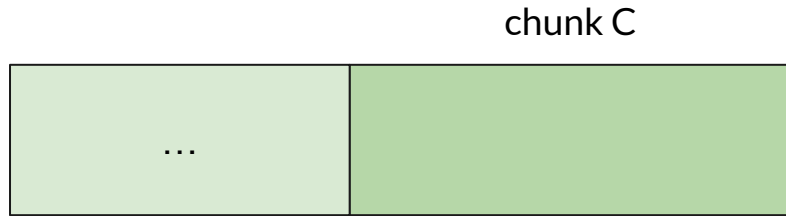
Double Free

```
A = kmalloc(32);  
kfree(A);  
B = kmalloc(32); // replace chunk A  
kfree(A);
```



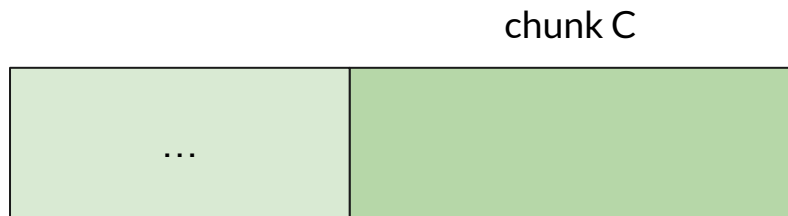
Double Free

```
A = kmalloc(32);  
kfree(A);  
B = kmalloc(32); // replace chunk A  
kfree(A);  
C = kmalloc(32); // replace chunk B/A
```



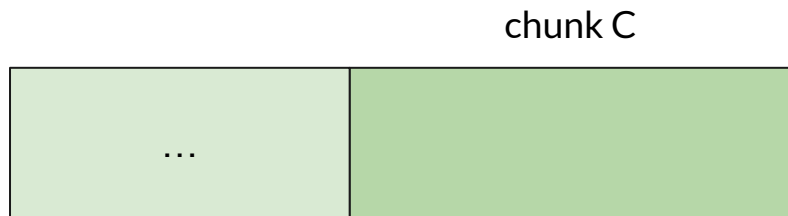
Double Free

```
A = kmalloc(32);  
kfree(A);  
B = kmalloc(32); // replace chunk A  
kfree(A);  
C = kmalloc(32); // replace chunk B/A  
B->function(); // => RIP: ????
```



Double Free

```
A = kmalloc(32);  
kfree(A);  
B = kmalloc(32); // replace chunk A  
kfree(A);  
C = kmalloc(32); // replace chunk B/A  
B->function(); // => RIP: ????
```



I chunk **B** e **C** sono due victim objects (a scelta dell'attaccante).

Double Free



TOCTOU

- Time-Of-Check to Time-Of-Use
- Generalmente causata da una **Race Condition**
- Concetto
 - Lo stato di un oggetto cambia dopo essere stato controllato, appena prima del suo utilizzo



TOCTOU

sizeof(user_input) = 20

```
kernel_buffer = kmalloc(32);
if (sizeof(user_input) > 32){
    return -EINVAL; // Input troppo grande
}
// Altre operazioni
copy_from_user(kernel_buffer, user_input, sizeof(user_input));
```



TOCTOU

sizeof(user_input) = 20

```
kernel_buffer = kmalloc(32);
if (sizeof(user_input) > 32){
    return -EINVAL; // Input troppo grande
}
// Altre operazioni
copy_from_user(kernel_buffer, user_input, sizeof(user_input));
```



TOCTOU

sizeof(user_input) = 20

```
kernel_buffer = kmalloc(32);  
if (sizeof(user_input) > 32){  
    return -EINVAL; // Input troppo grande  
}  
// Altre operazioni  
copy_from_user(kernel_buffer, user_input, sizeof(user_input));
```



TOCTOU

sizeof(user_input) = 20

```
kernel_buffer = kmalloc(32);
if (sizeof(user_input) > 32){
    return -EINVAL; // Input troppo grande
}
// Altre operazioni
copy_from_user(kernel_buffer, user_input, sizeof(user_input));
```



TOCTOU

sizeof(user_input) = 60

```
kernel_buffer = kmalloc(32);  
if (sizeof(user_input) > 32){  
    return -EINVAL; // Input troppo grande  
}  
// Altre operazioni  
copy_from_user(kernel_buffer, user_input, sizeof(user_input));
```



TOCTOU

sizeof(user_input) = 60

```
kernel_buffer = kmalloc(32);  
if (sizeof(user_input) > 32){  
    return -EINVAL; // Input troppo grande  
}  
// Altre operazioni  
copy_from_user(kernel_buffer, user_input, sizeof(user_input)); => OOB Write
```



Victim Objects: msg_msg




msg_msg

- Primitives
 - Out Of Bounds Read (da un OOB write)
 - Arbitrary Write
- Fa parte di System-V IPC
 - Implementa **code di messaggi**, shared memory e semafori
 - Dipende da CONFIG_SYSVIPC
 - Abilitato di default (non su Android)



System-V IPC: Code di messaggi


- `msgget()` per creare o collegarsi ad una coda
- Non accessibili tramite file descriptors
 - `write => msgsnd()` => **Inserire** un messaggio nella coda
 - `read => msgrcv()` => **Leggere** un messaggio nella coda
 - `ioctl => msgctl()` => **Controllare** la coda



```
int send_msg(){
    int msgtype = 1;
    int msgkey = 0x1337;
    int qid = msgget(msgkey, IPC_CREAT | 0666);
    if( qid == -1 ){
        perror("msgget");
        return -1;
    }

    struct msgbuf msg_buf;
    msg_buf.mtype = 1;
    memset(msg_buf.mtext, 0x41, 10);
    printf("sizeof msg_buf: %d \n", sizeof(msg_buf));
    if( msgsnd(qid, &msg_buf, sizeof(msg_buf), IPC_NOWAIT) == -1 ){
        perror("msgsnd");
        return -1;
    }

    return 0;
}
```



```
int get_msg(){
    int msgtype = 1;
    int msgkey = 0x1337;
    int qid = msgget(msgkey, IPC_CREAT | 0666);
    if( qid == -1 ){
        perror("msgget");
        return -1;
    }

    struct msgbuf rec_msg_buf;
    printf("sizeof rec_msg_buf: %d \n", sizeof(rec_msg_buf));
    if (msgrcv( qid, &rec_msg_buf, sizeof(rec_msg_buf), msgtype, MSG_NOERROR | IPC_NOWAIT ) == -1 ){
        if( errno != ENOMSG) {
            perror("msgrcv");
            return -1;
        }
        printf("No message received\n");
        return 0;
    }

    printf("Message received: %s\n", rec_msg_buf.mtext);
    return 0;
}
```



msg_msg kernel Walkthrough

msgsnd/msgrcv syscalls

msgsnd



`msgsnd(qid, &msg_buf, sizeof(msg_buf), IPC_NOWAIT)`

1. `ksys_msgsnd(int msqid, struct msgbuf __user *msgp, size_t msgsz, int msgflg)`
2. `do_msgsnd(msqid, mtype, msgp->mtext, msgsz, msgflg);`
3. `load_msg(mtext, msgsz)`
4. `alloc_msg(msgsz); // MSGMAX 8192`
5. `static struct msg_msg *alloc_msg(size_t len)`
`{`
`alen = min(len, DATALEN_MSG);`
`=> msg = kmalloc(sizeof(*msg) + alen, GFP_KERNEL_ACCOUNT);`
`}`



msgrcv

```
msgrcv( qid, &rec_msg_buf, sizeof(rec_msg_buf), msgtype, MSG_NOERROR | IPC_NOWAIT )
```

- Ottiene la queue tramite **qid**
- ksys_msgrcv => do_msgrcv (ricava il msg dal QID) => do_msg_fill => store_msg (**copy_to_user**) => (if !MSG_COPY) free_msg (**kfree**)



Perché `msg_msg` è una buona primitiva

- Possiamo allocare chunk di una dimensione semi-arbitraria (`kmalloc`)
- Possiamo scrivere in questi chunk valori arbitrari (`copy_from_user`)
- Possiamo leggere da questi chunk (`copy_to_user`)
- Possiamo decidere quando de-allocare i chunk (`kfree`)




Target caches e limitazioni

- OOB read da kmalloc-64 => kmalloc-4096
 - Modificando msg_msg->m_ts
- Arbitrary Free/Write
 - <https://syst3mfailure.io/wall-of-perdition>



msg_msgseg

- Se il messaggio è superiore a 4096
 - Il messaggio restante a 4096 viene suddiviso in segmenti collegati tramite linked-list.



```
struct msg_msg {
    struct list_head m_list;
    long m_type;
    size_t m_ts;          /* message text size */
    struct msg_msgseg *next;
    void *security;
    /* the actual message follows immediately */
};

struct list_head {
    struct list_head *next, *prev;
};

struct msg_msgseg {
    struct msg_msgseg *next;
    /* the next part of the message follows immediately */
};
```

msg_msg allocation

```
struct msg_msg *load_msg(const void __user *src, size_t len)
{
    struct msg_msg *msg;
    struct msg_msgseg *seg;
    msg = alloc_msg(len);
    alen = min(len, DATALEN_MSG);
    /* copy into msg_msg */
    if (copy_from_user(msg + 1, src, alen))
        goto out_err;

    for (seg = msg->next; seg != NULL; seg = seg->next) {
        len -= alen;
        src = (char __user *)src + alen;
        alen = min(len, DATALEN_SEG);
        /* if doesn't fit into msg_msg, copy into segments */
        if (copy_from_user(seg + 1, src, alen))
            goto out_err;
    }
}
```

```
static struct msg_msg *alloc_msg(size_t len)
{
    struct msg_msg *msg;
    alen = min(len, DATALEN_MSG);
    /* Allocate first part of msg_msg */
    msg = kmalloc(sizeof(*msg) + alen, GFP_KERNEL_ACCOUNT);
    /* ... */
    len -= alen;
    while (len > 0) {
        struct msg_msgseg *seg;
        /* if doesn't fit into msg_msg, allocate msg_msgseg */
        alen = min(len, DATALEN_SEG);
        seg = kmalloc(sizeof(*seg) + alen,
            GFP_KERNEL_ACCOUNT);
        /* ... */
        len -= alen;
    }
    /* ... */
}
```

msg_msg allocation

```
struct msg_msg *load_msg(const void __user *src, size_t len)
{
    struct msg_msg *msg;
    struct msg_msgseg *seg;
    msg = alloc_msg(len);
    alen = min(len, DATALEN_MSG);
    /* copy into msg_msg */
    if (copy_from_user(msg + 1, src, alen))
        goto out_err;

    for (seg = msg->next; seg != NULL; seg = seg->next) {
        len -= alen;
        src = (char __user *)src + alen;
        alen = min(len, DATALEN_SEG);
        /* if doesn't fit into msg_msg, copy into segments */
        if (copy_from_user(seg + 1, src, alen))
            goto out_err;
    }
}
```

```
static struct msg_msg *alloc_msg(size_t len)
{
    struct msg_msg *msg;
    alen = min(len, DATALEN_MSG);
    /* Allocate first part of msg_msg */
    msg = kmalloc(sizeof(*msg) + alen, GFP_KERNEL_ACCOUNT);
    /* ... */
    len -= alen;
    while (len > 0) {
        struct msg_msgseg *seg;
        /* if doesn't fit into msg_msg, allocate msg_msgseg */
        alen = min(len, DATALEN_SEG);
        seg = kmalloc(sizeof(*seg) + alen,
            GFP_KERNEL_ACCOUNT);
        /* ... */
        len -= alen;
    }
    /* ... */
}
```

msg_msg allocation

```
struct msg_msg *load_msg(const void __user *src, size_t len)
{
    struct msg_msg *msg;
    struct msg_msgseg *seg;
    msg = alloc_msg(len);
    alen = min(len, DATALEN_MSG);
    /* copy into msg_msg */
    if (copy_from_user(msg + 1, src, alen))
        goto out_err;

    for (seg = msg->next; seg != NULL; seg = seg->next) {
        len -= alen;
        src = (char __user *)src + alen;
        alen = min(len, DATALEN_SEG);
        /* if doesn't fit into msg_msg, copy into segments */
        if (copy_from_user(seg + 1, src, alen))
            goto out_err;
    }
}
```

```
static struct msg_msg *alloc_msg(size_t len)
{
    struct msg_msg *msg;
    alen = min(len, DATALEN_MSG);
    /* Allocate first part of msg_msg */
    msg = kmalloc(sizeof(*msg) + alen, GFP_KERNEL_ACCOUNT);
    /* ... */
    len -= alen;
    while (len > 0) {
        struct msg_msgseg *seg;
        /* if doesn't fit into msg_msg, allocate msg_msgseg */
        alen = min(len, DATALEN_SEG);
        seg = kmalloc(sizeof(*seg) + alen,
            GFP_KERNEL_ACCOUNT);
        /* ... */
        len -= alen;
    }
    /* ... */
}
```

msg_msg allocation

```
struct msg_msg *load_msg(const void __user *src, size_t len)
{
    struct msg_msg *msg;
    struct msg_msgseg *seg;
    msg = alloc_msg(len);
    alen = min(len, DATALEN_MSG);
    /* copy into msg_msg */
    if (copy_from_user(msg + 1, src, alen))
        goto out_err;

    for (seg = msg->next; seg != NULL; seg = seg->next) {
        len -= alen;
        src = (char __user *)src + alen;
        alen = min(len, DATALEN_SEG);
        /* if doesn't fit into msg_msg, copy into segments */
        if (copy_from_user(seg + 1, src, alen))
            goto out_err;
    }
}
```

```
static struct msg_msg *alloc_msg(size_t len)
{
    struct msg_msg *msg;
    alen = min(len, DATALEN_MSG);
    /* Allocate first part of msg_msg */
    msg = kmalloc(sizeof(*msg) + alen, GFP_KERNEL_ACCOUNT);
    /* ... */
    len -= alen;
    while (len > 0) {
        struct msg_msgseg *seg;
        /* if doesn't fit into msg_msg, allocate msg_msgseg */
        alen = min(len, DATALEN_SEG);
        seg = kmalloc(sizeof(*seg) + alen,
            GFP_KERNEL_ACCOUNT);
        /* ... */
        len -= alen;
    }
    /* ... */
}
```

msg_msg allocation

```
struct msg_msg *load_msg(const void __user *src, size_t len)
{
    struct msg_msg *msg;
    struct msg_msgseg *seg;
    msg = alloc_msg(len);
    alen = min(len, DATALEN_MSG);
    /* copy into msg_msg */
    if (copy_from_user(msg + 1, src, alen))
        goto out_err;

    for (seg = msg->next; seg != NULL; seg = seg->next) {
        len -= alen;
        src = (char __user *)src + alen;
        alen = min(len, DATALEN_SEG);
        /* if doesn't fit into msg_msg, copy into segments */
        if (copy_from_user(seg + 1, src, alen))
            goto out_err;
    }
}
```

```
static struct msg_msg *alloc_msg(size_t len)
{
    struct msg_msg *msg;
    alen = min(len, DATALEN_MSG);
    /* Allocate first part of msg_msg */
    msg = kmalloc(sizeof(*msg) + alen, GFP_KERNEL_ACCOUNT);
    /* ... */
    len -= alen;
    while (len > 0) {
        struct msg_msgseg *seg;
        /* if doesn't fit into msg_msg, allocate msg_msgseg */
        alen = min(len, DATALEN_SEG);
        seg = kmalloc(sizeof(*seg) + alen,
            GFP_KERNEL_ACCOUNT);
        /* ... */
        len -= alen;
    }
    /* ... */
}
```

msg_msg allocation

```
struct msg_msg *load_msg(const void __user *src, size_t len)
{
    struct msg_msg *msg;
    struct msg_msgseg *seg;
    msg = alloc_msg(len);
    alen = min(len, DATALEN_MSG);
    /* copy into msg_msg */
    if (copy_from_user(msg + 1, src, alen))
        goto out_err;

    for (seg = msg->next; seg != NULL; seg = seg->next) {
        len -= alen;
        src = (char __user *)src + alen;
        alen = min(len, DATALEN_SEG);
        /* if doesn't fit into msg_msg, copy into segments */
        if (copy_from_user(seg + 1, src, alen))
            goto out_err;
    }
}
```

```
static struct msg_msg *alloc_msg(size_t len)
{
    struct msg_msg *msg;
    alen = min(len, DATALEN_MSG);
    /* Allocate first part of msg_msg */
    msg = kmalloc(sizeof(*msg) + alen, GFP_KERNEL_ACCOUNT);
    /* ... */
    len -= alen;
    while (len > 0) {
        struct msg_msgseg *seg;
        /* if doesn't fit into msg_msg, allocate msg_msgseg */
        alen = min(len, DATALEN_SEG);
        seg = kmalloc(sizeof(*seg) + alen,
            GFP_KERNEL_ACCOUNT);
        /* ... */
        len -= alen;
    }
    /* ... */
}
```

msg_msg allocation

```
struct msg_msg *load_msg(const void __user *src, size_t len)
{
    struct msg_msg *msg;
    struct msg_msgseg *seg;
    msg = alloc_msg(len);
    alen = min(len, DATALEN_MSG);
    /* copy into msg_msg */
    if (copy_from_user(msg + 1, src, alen))
        goto out_err;

    for (seg = msg->next; seg != NULL; seg = seg->next) {
        len -= alen;
        src = (char __user *)src + alen;
        alen = min(len, DATALEN_SEG);
        /* if doesn't fit into msg_msg, copy into segments */
        if (copy_from_user(seg + 1, src, alen))
            goto out_err;
    }
}
```

```
static struct msg_msg *alloc_msg(size_t len)
{
    struct msg_msg *msg;
    alen = min(len, DATALEN_MSG);
    /* Allocate first part of msg_msg */
    msg = kmalloc(sizeof(*msg) + alen, GFP_KERNEL_ACCOUNT);
    /* ... */
    len -= alen;
    while (len > 0) {
        struct msg_msgseg *seg;
        /* if doesn't fit into msg_msg, allocate msg_msgseg */
        alen = min(len, DATALEN_SEG);
        seg = kmalloc(sizeof(*seg) + alen,
            GFP_KERNEL_ACCOUNT);
        /* ... */
        len -= alen;
    }
    /* ... */
}
```



struct msg_msg



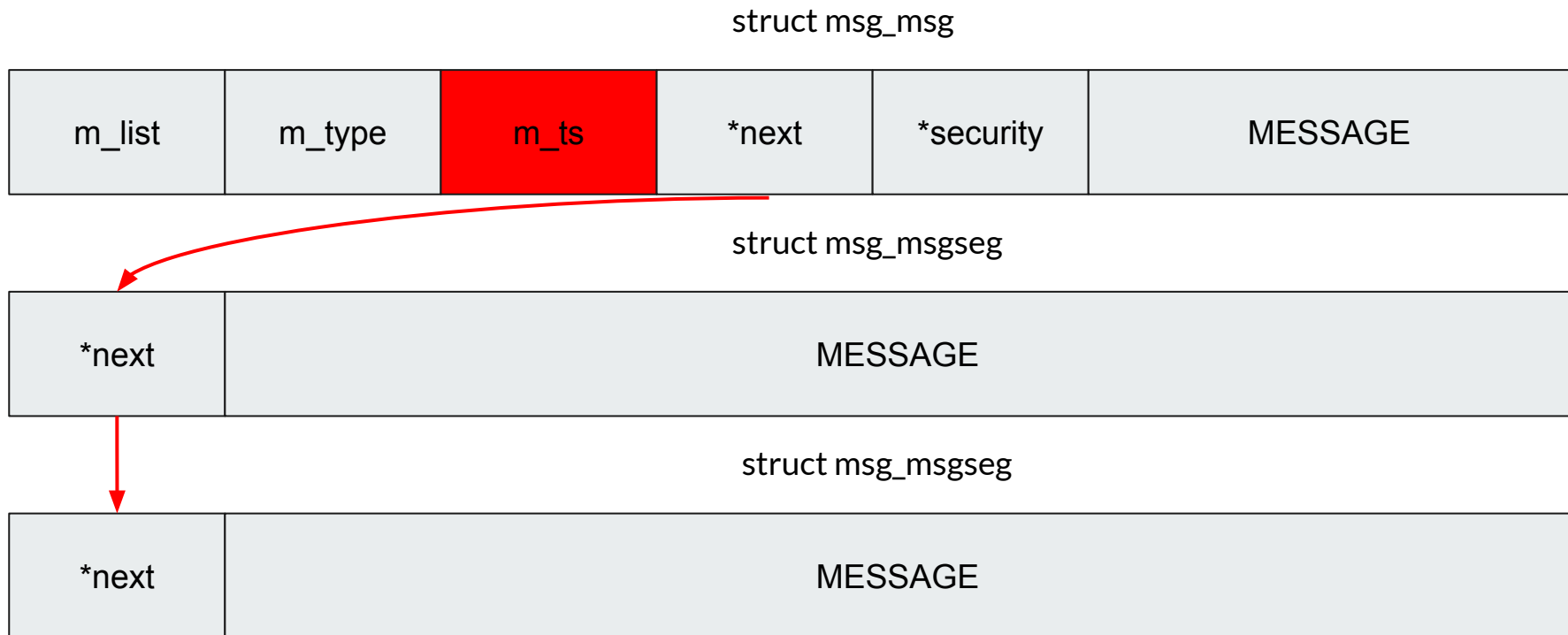
struct msg_msgseg



struct msg_msgseg



OOB READ





Pipe + iovec

Arbitrary Write + Read

Pipe



- Pipe
 - Canale bi-direzionale per comunicazione IPC
 - Due file descriptors
 - pipe[0]
 - Legge dalla pipe
 - pipe[1]
 - Scrive nella pipe

```
int pp[2];  
pipe(pp);  
write(pp[1], write_buffer, 10); // Writes into the pipe pp[1]  
read(pp[0], dest_buffer, 10); // Reads from the pipe pp[0]
```

Pipe - Blocking



- Può essere “bloccata”
 - Se leggiamo da una pipe che non è stata scritta
 - Fino a che non viene scritto nulla in pp[1]

```
int pp[2];  
pipe(pp);  
read(pp[0], dest_buffer, 10); // Blocks: Waits for write data into pp[1]
```



iovec

- R/W in buffer **multipli**
 - readv
 - Legge in buffer multipli
 - writev
 - Scrive in buffer multipli
- Uguale a read/write, ma utilizzando buffer multipli anziché un buffer

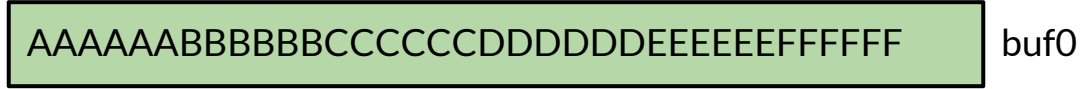
AAAAAABBBBBBCCCCCCDDDDDDDEEEEEEEFFFFFFF

buf0

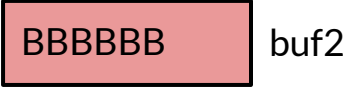
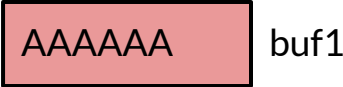
read(...)

AAAAAABBBBBBCCCCCCDDDDDDDEEEEEEEFFFFFFF

buf1



readv(...)



```
struct iovec
{
    void __user *iov_base;
    __kernel_size_t iov_len;
};
```



struct iovec

```
struct iovec
{
    void __user *iov_base;
    __kernel_size_t iov_len;
};
```

```
struct iovec iov_read_buffers[13] = {0};
char read_buffer0[0x100];
memset(read_buffer0, 0x52, 0x100);
iov_read_buffers[0].iov_base = read_buffer0;
iov_read_buffers[0].iov_len = 0x10;
iov_read_buffers[1].iov_base = read_buffer1;
iov_read_buffers[1].iov_len = 0x20;
iov_read_buffers[8].iov_base = read_buffer2;
iov_read_buffers[8].iov_len = 0x30;
iov_read_buffers[12].iov_base = read_buffer3;
iov_read_buffers[12].iov_len = 0x40;
```



iovec kernel heap allocation

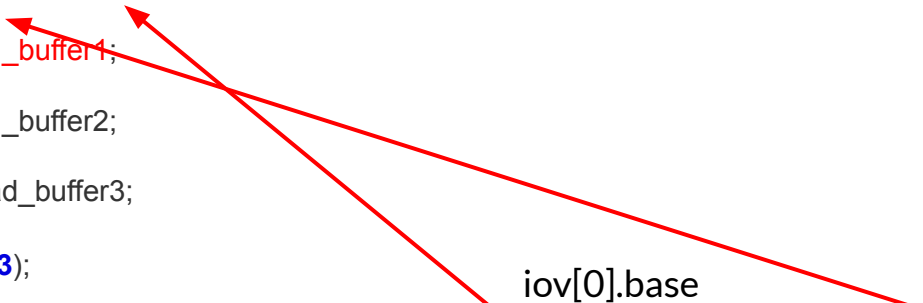
- writev => do_writev => vfs_writev => do_readv_writev => import_iovec => rw_copy_check_uvector

```
ssize_t rw_copy_check_uvector(...) {  
    if (nr_segs > fast_segs) {  
        iov = kmalloc(nr_segs * sizeof(struct iovec), GFP_KERNEL);  
    }  
}
```

- fast_segs = 8
 - <= 8 => Stack
 - > 8 => Heap (kmalloc)
- sizeof(struct iovec) = 16

Kernel Allocation

```
struct iovec iov_read_buffers[13] = {0};
char read_buffer0[0x100];
memset(read_buffer0, 0x52, 0x100);
iov_read_buffers[0].iov_base = read_buffer0;
iov_read_buffers[0].iov_len = 0x10;
iov_read_buffers[1].iov_base = read_buffer1;
iov_read_buffers[1].iov_len = 0x10;
iov_read_buffers[8].iov_base = read_buffer2;
iov_read_buffers[8].iov_len = 0x10;
iov_read_buffers[12].iov_base = read_buffer3;
iov_read_buffers[12].iov_len = 0x10;
readv(pipefd[0], iov_read_buffers, 13);
```



	iov[0].base	iov[0].len
0xffff88003cd7a700:	0x00007ffd22b3ae20	0x0000000000000010
0xffff88003cd7a710:	0x00007ffd22b3ae20	0x0000000000000010
0xffff88003cd7a720:	0x0000000000000000	0x0000000000000000
0xffff88003cd7a730:	0x0000000000000000	0x0000000000000000
0xffff88003cd7a740:	0x0000000000000000	0x0000000000000000



Corrupt iov[1].base for Arbitrary Write


0xffff88003cd7a700:	0x00007ffd22b3ae20	0x0000000000000010
0xffff88003cd7a710:	0x4141414141414141	0x0000000000000010
0xffff88003cd7a720:	0x0000000000000000	0x0000000000000000
0xffff88003cd7a730:	0x0000000000000000	0x0000000000000000
0xffff88003cd7a740:	0x0000000000000000	0x0000000000000000




Corrupt iov[1].base for Arbitrary Write

0xffff88003cd7a700:	0x00007ffd22b3ae20	0x0000000000000010
0xffff88003cd7a710:	0x4141414141414141	0x0000000000000010
0xffff88003cd7a720:	0x0000000000000000	0x0000000000000000
0xffff88003cd7a730:	0x0000000000000000	0x0000000000000000
0xffff88003cd7a740:	0x0000000000000000	0x0000000000000000


- `left = __copy_to_user_inatomic(buf, from, copy);` => INSECURE




```
int pipefd[2];
pipe(pipefd);
struct iovec iov_read_buffers[13] = {0};
char read_buffer0[0x100];
memset(read_buffer0, 0x52, 0x100);
iov_read_buffers[0].iov_base = read_buffer0;
iov_read_buffers[0].iov_len= 0x10;
iov_read_buffers[1].iov_base = read_buffer1;
iov_read_buffers[1].iov_len= 0x10;
iov_read_buffers[8].iov_base = read_buffer2;
iov_read_buffers[8].iov_len= 0x10;
iov_read_buffers[12].iov_base = read_buffer3;
iov_read_buffers[12].iov_len= 0x10;
readv(pipefd[0], iov_read_buffers, 13); // Blocking (use another thread or fork())
/* Corrupt iov[1].base*/
write(pipefd[1], write_buffer, 0x20);
```




```
int pipefd[2];
pipe(pipefd);
struct iovec iov_read_buffers[13] = {0};
char read_buffer0[0x100];
memset(read_buffer0, 0x52, 0x100);
iov_read_buffers[0].iov_base = read_buffer0;
iov_read_buffers[0].iov_len= 0x10;
iov_read_buffers[1].iov_base = read_buffer1;
iov_read_buffers[1].iov_len= 0x10;
iov_read_buffers[8].iov_base = read_buffer2;
iov_read_buffers[8].iov_len= 0x10;
iov_read_buffers[12].iov_base = read_buffer3;
iov_read_buffers[12].iov_len= 0x10;
readv(pipefd[0], iov_read_buffers, 13); // Blocking (use another thread or fork())
/* Corrupt iov[1].base*/
write(pipefd[1], write_buffer, 0x20);
```




```
int pipefd[2];
pipe(pipefd);
struct iovec iov_read_buffers[13] = {0};
char read_buffer0[0x100];
memset(read_buffer0, 0x52, 0x100);
iov_read_buffers[0].iov_base = read_buffer0;
iov_read_buffers[0].iov_len= 0x10;
iov_read_buffers[1].iov_base = read_buffer1;
iov_read_buffers[1].iov_len= 0x10;
iov_read_buffers[8].iov_base = read_buffer2;
iov_read_buffers[8].iov_len= 0x10;
iov_read_buffers[12].iov_base = read_buffer3;
iov_read_buffers[12].iov_len= 0x10;
readv(pipefd[0], iov_read_buffers, 13); // Blocking (use another thread or fork())
/* Corrupt iov[1].base*/
write(pipefd[1], write_buffer, 0x20);
```



```
int pipefd[2];
pipe(pipefd);
struct iovec iov_read_buffers[13] = {0};
char read_buffer0[0x100];
memset(read_buffer0, 0x52, 0x100);
iov_read_buffers[0].iov_base = read_buffer0;
iov_read_buffers[0].iov_len= 0x10;
iov_read_buffers[1].iov_base = read_buffer1;
iov_read_buffers[1].iov_len= 0x10;
iov_read_buffers[8].iov_base = read_buffer2;
iov_read_buffers[8].iov_len= 0x10;
iov_read_buffers[12].iov_base = read_buffer3;
iov_read_buffers[12].iov_len= 0x10;
readv(pipefd[0], iov_read_buffers, 13); // Blocking (use another thread or fork())
/* Corrupt iov[1].base*/
write(pipefd[1], write_buffer, 0x20);
```



```
int pipefd[2];
pipe(pipefd);
struct iovec iov_read_buffers[13] = {0};
char read_buffer0[0x100];
memset(read_buffer0, 0x52, 0x100);
iov_read_buffers[0].iov_base = read_buffer0;
iov_read_buffers[0].iov_len= 0x10;
iov_read_buffers[1].iov_base = read_buffer1;
iov_read_buffers[1].iov_len= 0x10;
iov_read_buffers[8].iov_base = read_buffer2;
iov_read_buffers[8].iov_len= 0x10;
iov_read_buffers[12].iov_base = read_buffer3;
iov_read_buffers[12].iov_len= 0x10;
readv(pipefd[0], iov_read_buffers, 13); // Blocking (use another thread or fork())
/* Corrupt iov[1].base*/
write(pipefd[1], write_buffer, 0x20);
```



```
int pipefd[2];
pipe(pipefd);
struct iovec iov_read_buffers[13] = {0};
char read_buffer0[0x100];
memset(read_buffer0, 0x52, 0x100);
iov_read_buffers[0].iov_base = read_buffer0;
iov_read_buffers[0].iov_len= 0x10;
iov_read_buffers[1].iov_base = read_buffer1;
iov_read_buffers[1].iov_len= 0x10;
iov_read_buffers[8].iov_base = read_buffer2;
iov_read_buffers[8].iov_len= 0x10;
iov_read_buffers[12].iov_base = read_buffer3;
iov_read_buffers[12].iov_len= 0x10;
readv(pipefd[0], iov_read_buffers, 13); // Blocking (use another thread or fork())
/* Corrupt iov[1].base*/
write(pipefd[1], write_buffer, 0x20); => Arbitrary Write
```



Resume

- Combinando pipe + iovec possiamo ottenere Arbitrary Write
 - Corrompendo `iov[N].base`
 - pipe + `readv` + `write`
- Pipe ci permette di “bloccare” l’allocazione di iovec nel kernel
 - Dandoci il tempo di corromperla (e.g. tramite heap overflow/uaf/..)
- “Sbloccandola” tramite una `write`, si ottiene Arbitrary Write
- È possibile anche ottenere Arbitrary Read



Limitation

- Primitive eliminata dal kernel 4.13
- 4.9
 - `left = __copy_to_user_inatomic(buf, from, copy);` => INSECURE
- > 4.13
 - `left = copyout(buf, from, copy);`

```
static int copyout(void __user *to, const void *from, size_t n)
{
    if (access_ok(VERIFY_WRITE, to, n)) {
        kasan_check_read(from, n);
        n = raw_copy_to_user(to, from, n);
    }
    return n;
}
```



modprobe_path

Arbitrary Write => Code Exec



modprobe_path

- Variabile **globale**
- Comando eseguito dal **kernel** (con i privilegi)
 - Se non viene trovato il modo di eseguire un file

```
(gdb) x/s modprobe_path
0xffffffff81e42a80 <modprobe_path>:    "/sbin/modprobe"
```

kernel-land

- **SYSCALL_DEFINE3(execve, ..)=> do_execve
=> do_execveat_common => exec_binprm
=> search_binary_handler =>
__request_module => call_modprobe =>
**call_usermodehelper_setup =>
call_usermodehelper_exec****
- *// call_usermodehelper_exec - start a
usermode application*
 - **Tramite work queue**

[illegible]



modprobe_path + Arbitrary Write

- Default
 - `char modprobe_path[KMOD_PATH_LEN] = "/sbin/modprobe";`
- Arbitrary Write
 - `char modprobe_path[KMOD_PATH_LEN] = "/tmp/script";`
 - Tramite arbitrary write modifichiamo la variabile **globale** di modprobe_path
 - Permette di eseguire script come **root**

kernel-land

- **SYSCALL_DEFINE3(execve, ..)=> do_execve
=> do_execveat_common => exec_binprm
=> search_binary_handler =>
__request_module => call_modprobe =>
**call_usermodehelper_setup =>
call_usermodehelper_exec****
- *call_usermodehelper_exec - start a
usermode application*
 - **Tramite work queue**

[illegible]



Limitation

- CONFIG_STATIC_USERMODEHELPER / CONFIG_STATIC_USERMODEHELPER_PATH
 - modprobe_path read-only
- Non è più possibile sovrascrivere la variabile
 - Se è settata l'opzione
- Linux > 4.11

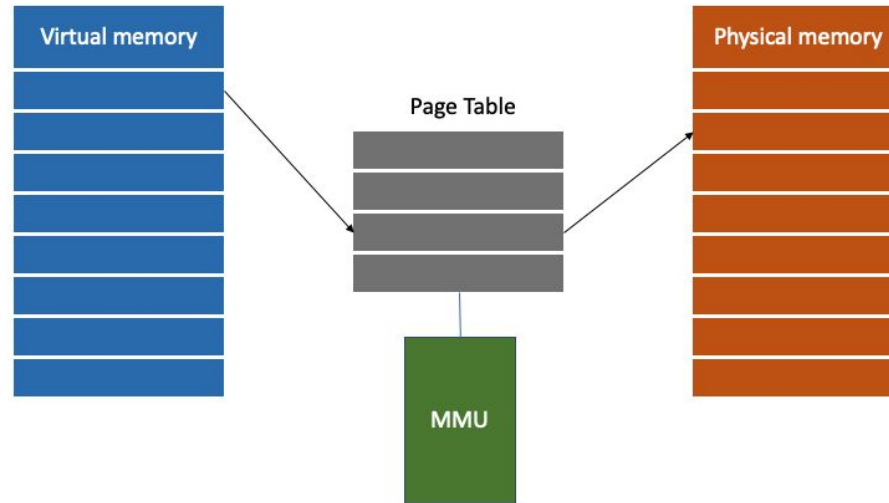
```
struct subprocess_info *call_usermodehelper_setup(const char *path, ..)
{
    // ...
#ifdef CONFIG_STATIC_USERMODEHELPER
    sub_info->path = CONFIG_STATIC_USERMODEHELPER_PATH;
#else
    sub_info->path = path;
#endif
    // ...
}
```



Userfaultd (EXTRA)

Lazy Page Allocations

Virtual / Physical Memory





Page Allocations

- Tramite user-land possiamo chiedere memoria al kernel
 - Es. malloc utilizza brk/sbrk
- Il kernel alloca memoria fisica (physical memory) e la mappa nel processo user-space
 - Dato che la memoria potrebbe non essere utilizzata nella sua interezza, il kernel utilizza un trick chiamato **Lazy Page Allocation**



Lazy Page Allocations

- Il kernel non alloca memoria fisica (physical memory) appena viene richiesta
 - “Segna” soltanto l’indirizzo virtuale
- Quando la memoria viene utilizzata => la CPU genera un fault => Il kernel alloca la memoria richiesta
- Evita di allocare memoria non utilizzata
 - Ottimizzazione



Userfaultd

- Gestire il Page Fault da userland !
- La CPU genera un fault => Da user-land possiamo gestirlo



Esempio

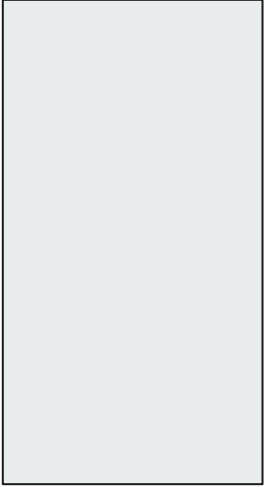
- User-land
 - `addr = mmap(0, 4096, .. | MAP_ANONYMOUS | .., -1, 0)`
 - Demand-zero page
 - Non allocata in physical memory
 - `syscall(..., addr)`
- Kernel
 - `copy_from_user(kernel_buf, addr, 30)`
 - **TRIGGER PAGE FAULT** appena si accede alla memoria non allocata
 - Possiamo gestire il fault da userland! (**userfaultd**)



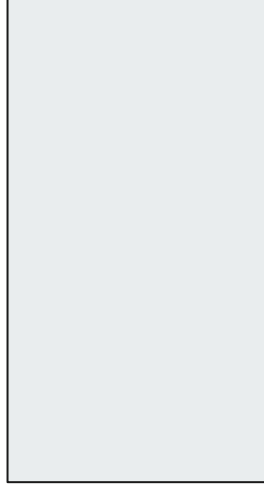
Cosa ci permette di fare

- Essendo il kernel multi-threading
- **Interrompere il kernel thread !**
 - Quando il kernel prova ad accedere alla memoria
 - Il page fault handler user-land viene richiamato
 - Il kernel attende la conclusione della funzione user-land

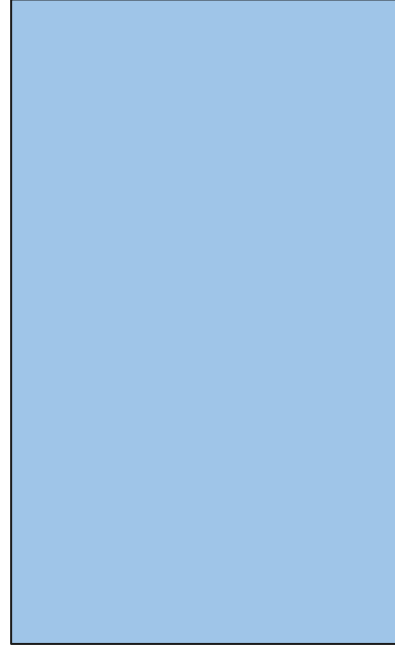
User process



Kernel



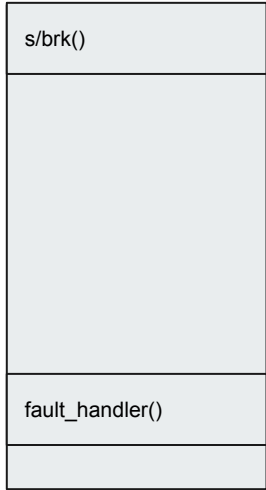
Virtual Memory



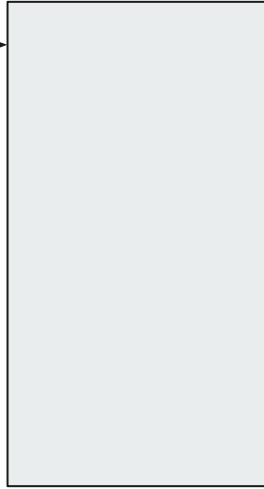
Physical Memory



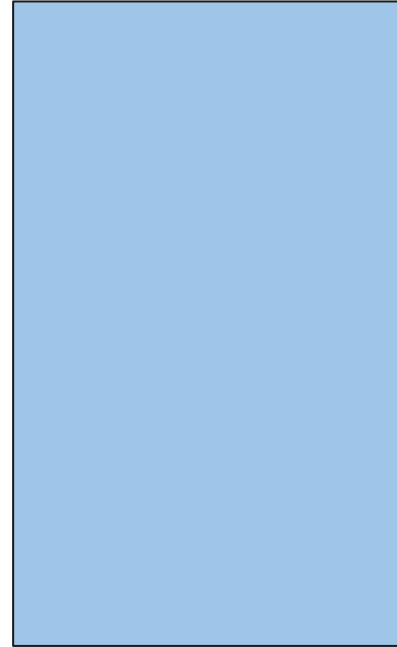
User process



Kernel



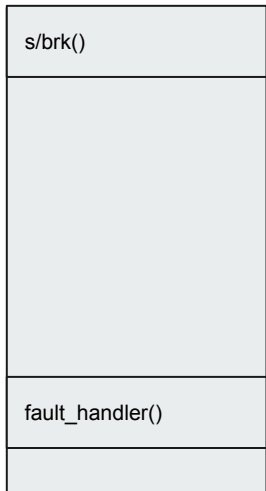
Virtual Memory



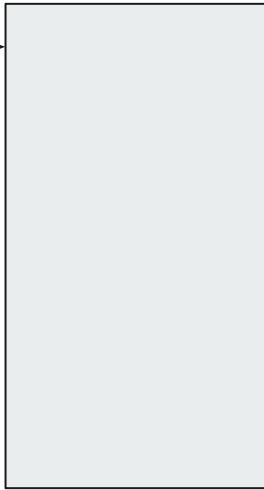
Physical Memory



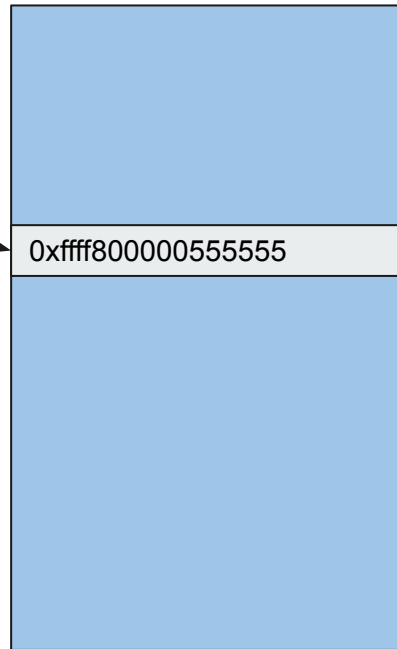
User process



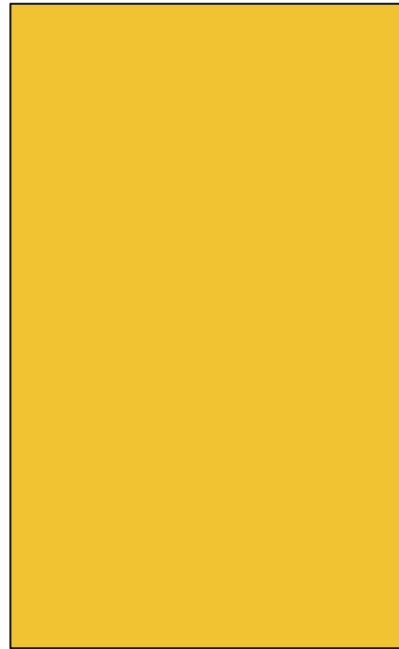
Kernel

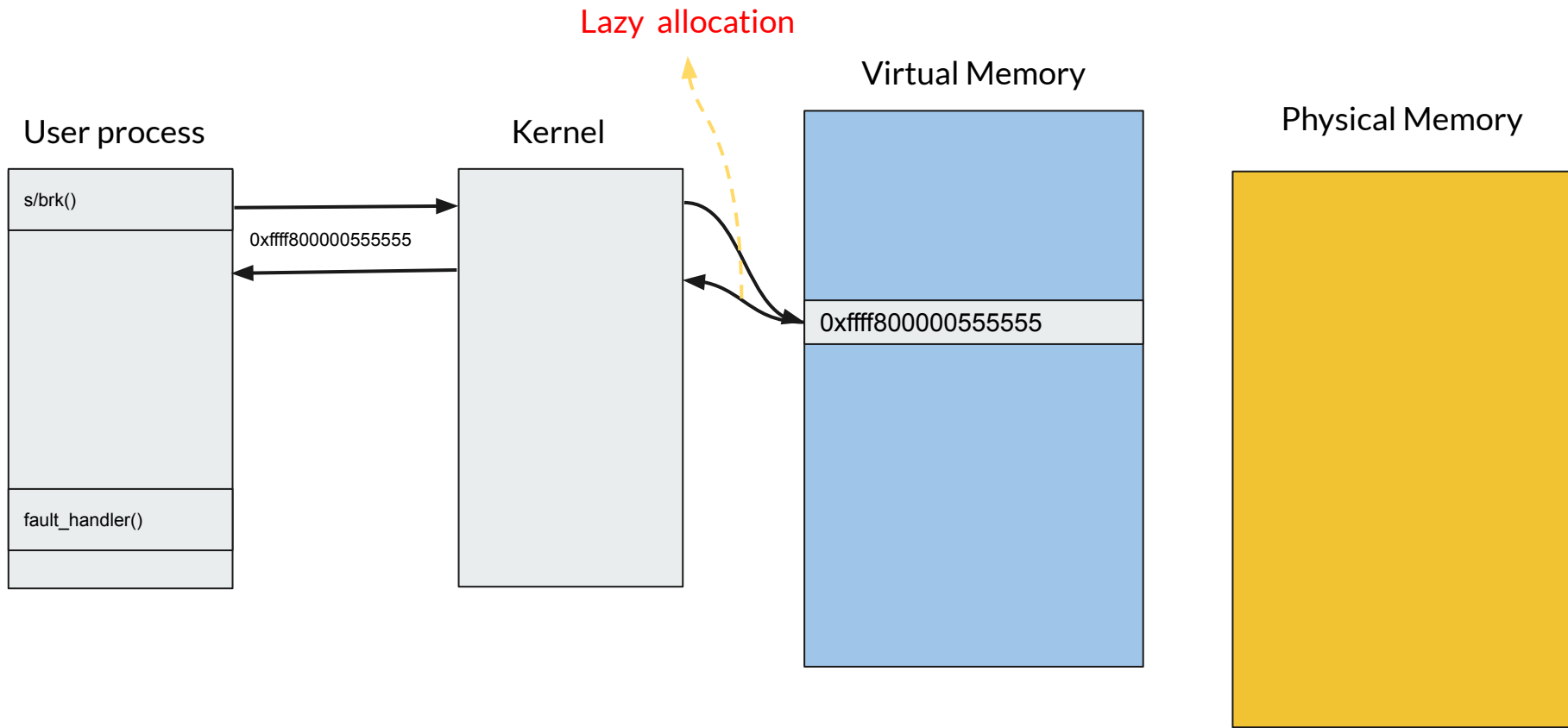


Virtual Memory

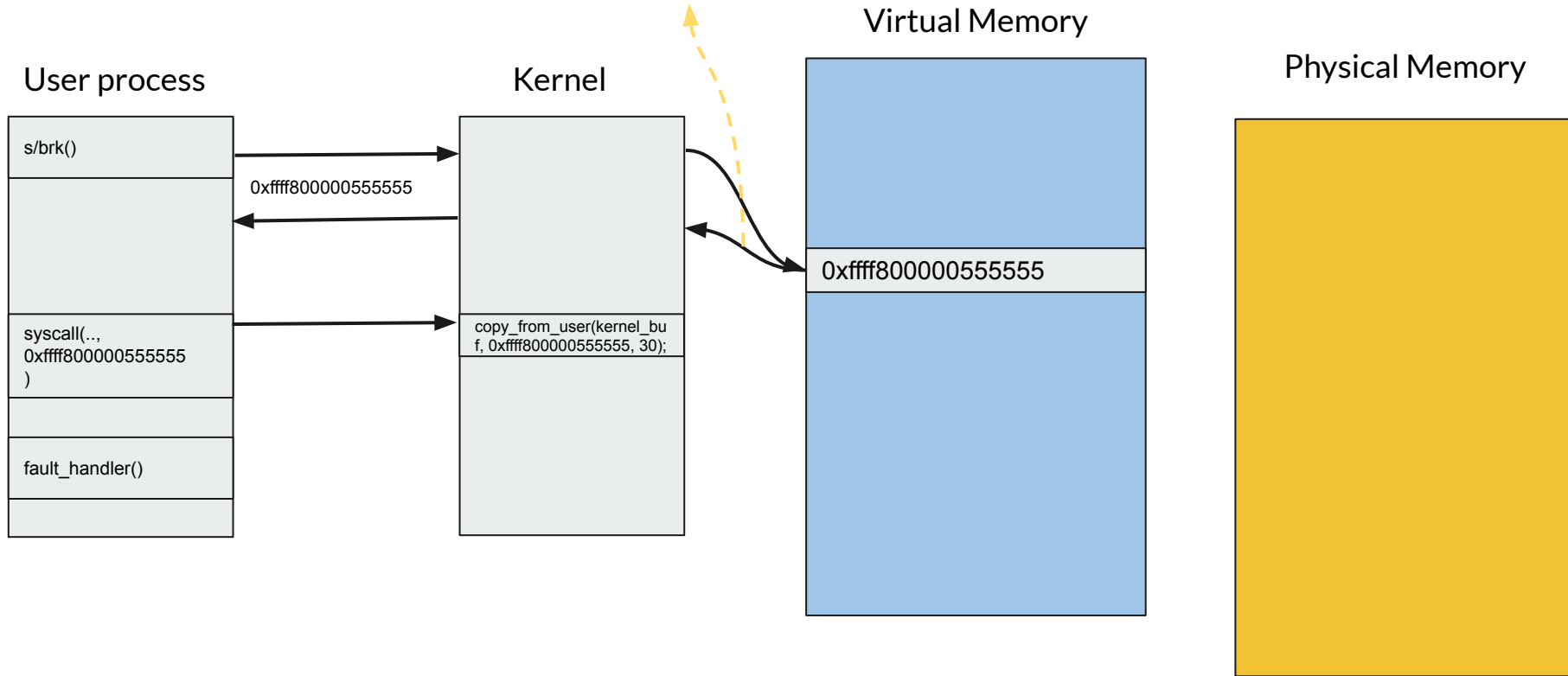


Physical Memory

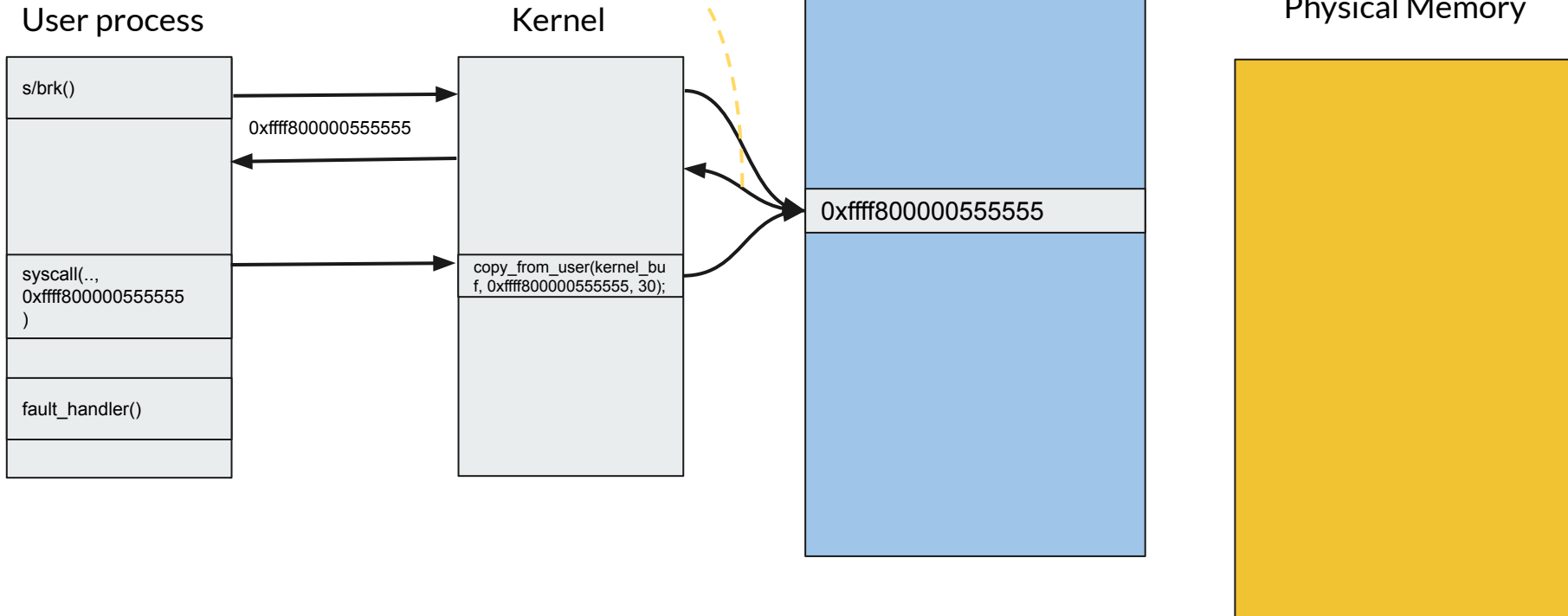


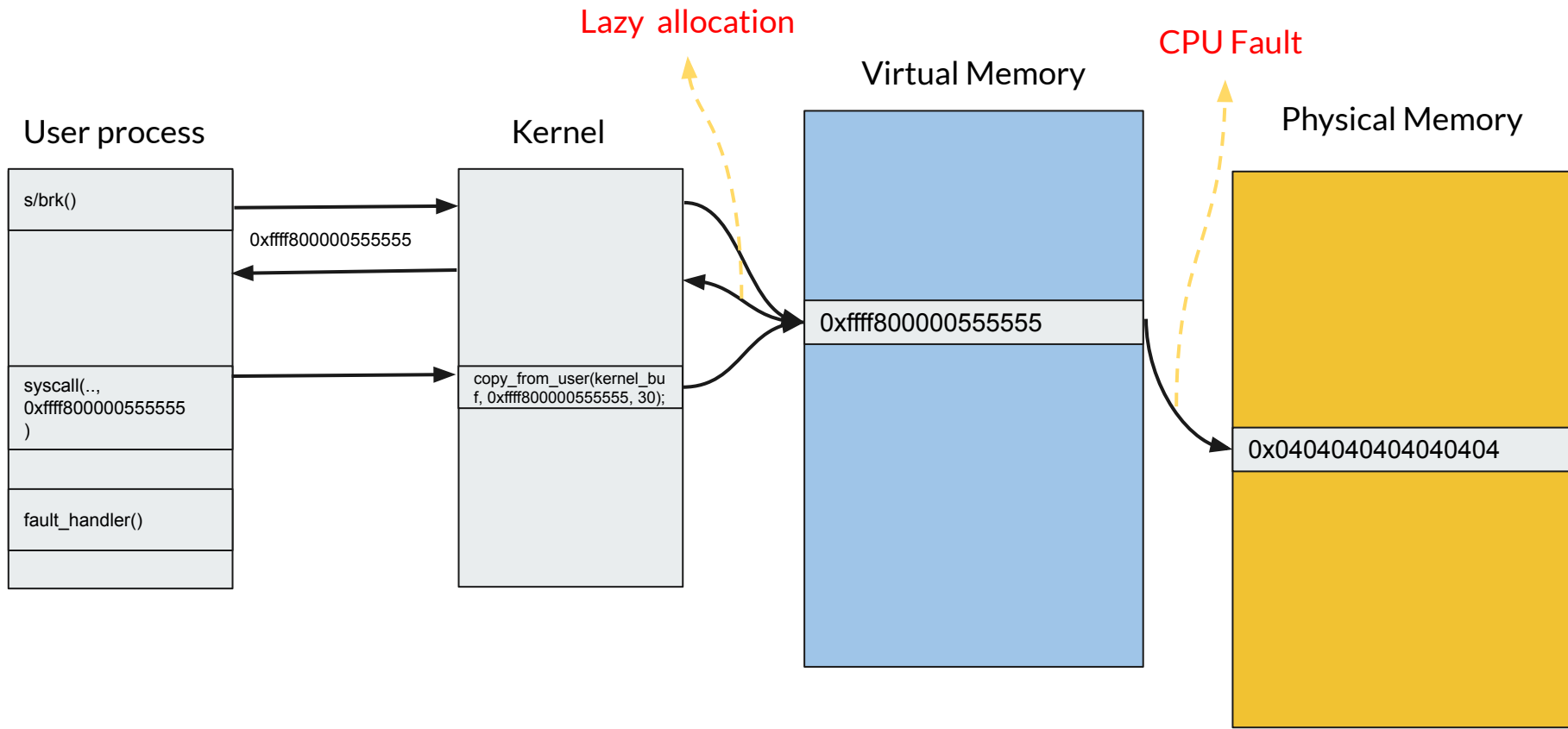


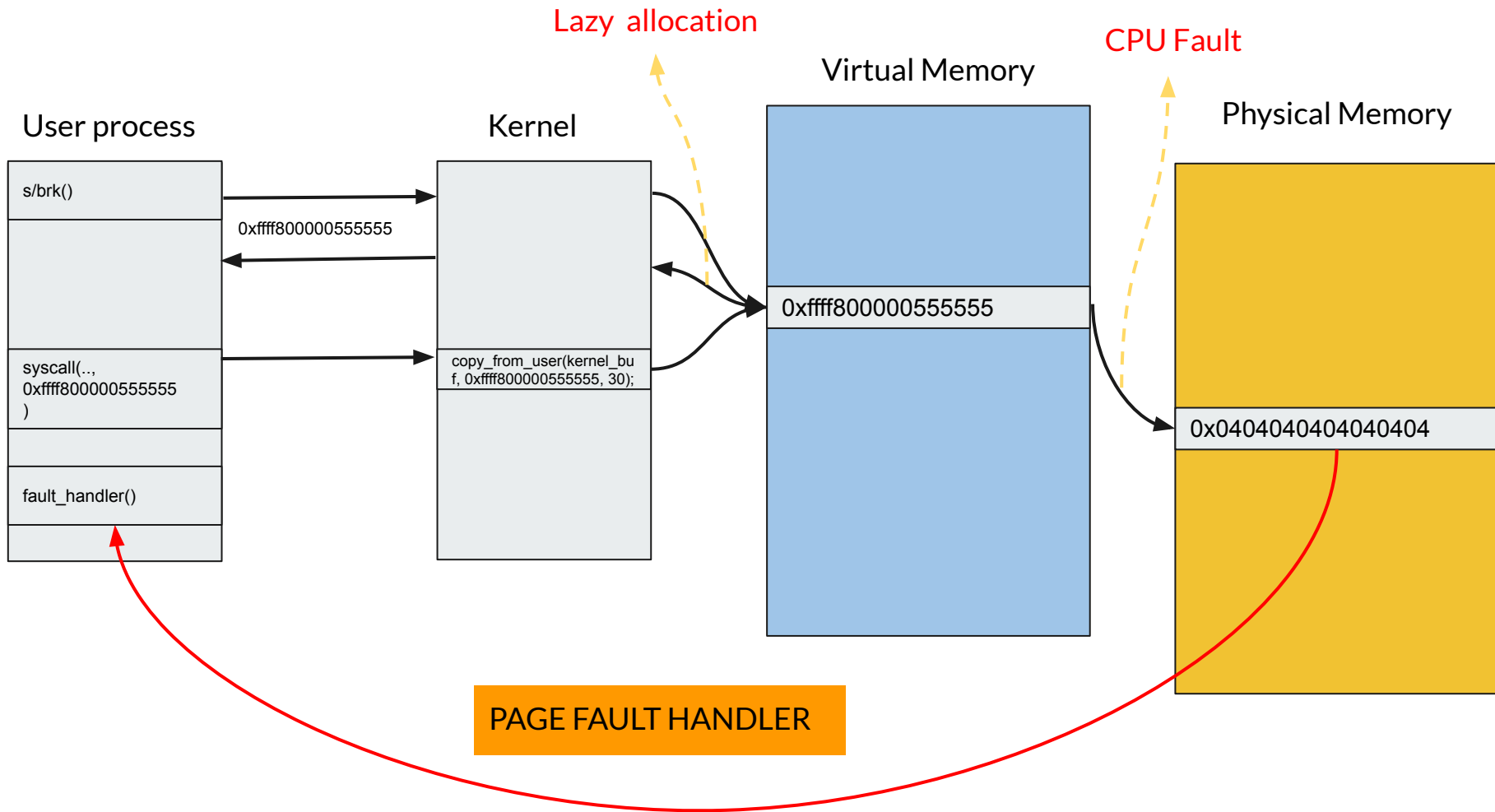
Lazy allocation



Lazy allocation







Lazy allocation

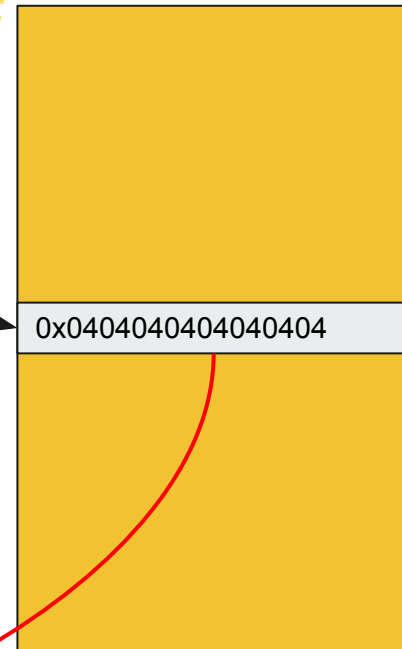
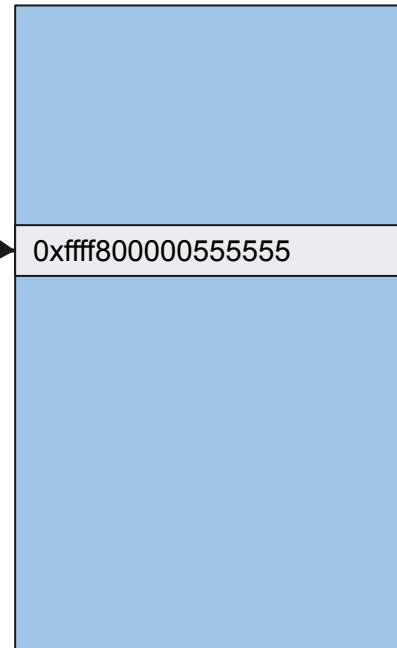
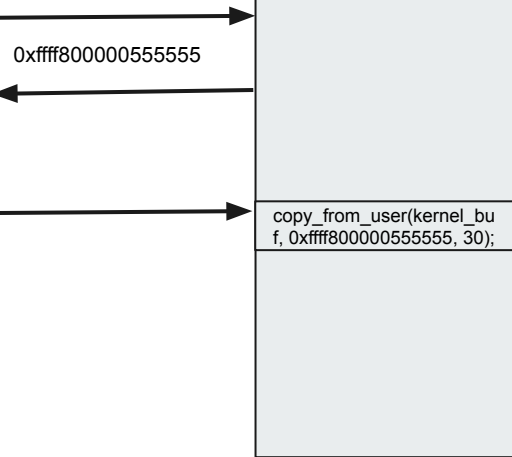
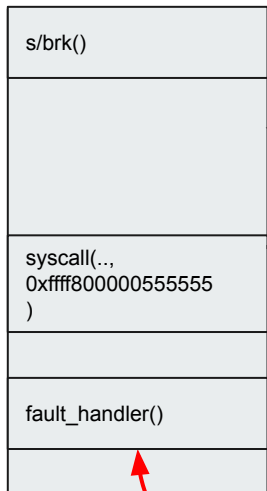
CPU Fault

Virtual Memory

Physical Memory

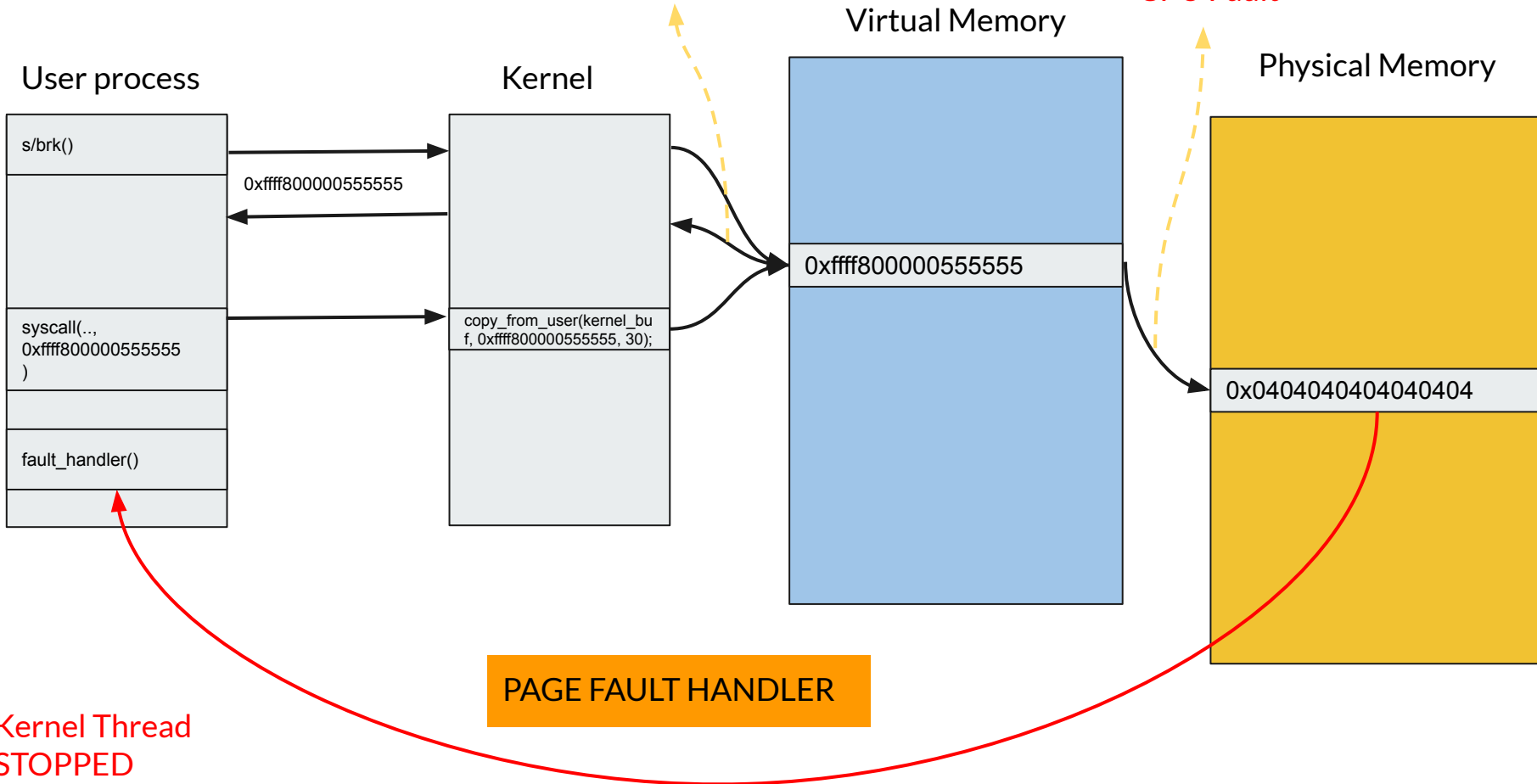
User process

Kernel



Kernel Thread
STOPPED

PAGE FAULT HANDLER





Cosa ci permette di fare

- Limitata ad utenti non privilegiati dal kernel 5.11
 - `/proc/sys/vm/unprivileged_userfaultfd` a 0
- FUSE



Limitation

- Essendo il kernel multi-threading
- **Interrompere il kernel thread !**
 - Quando il kernel prova ad accedere alla memoria
 - Il page fault handler user-land viene richiamato
 - Il kernel attende la conclusione della funzione user-land
- E se mentre il thread è interrotto, alteriamo la memoria del kernel?
 - Quando quello riprenderà, utilizzerà dati diversi da quelli precedenti
 - Tramite una vulnerabilità
- Utile per estendere il “Time Window” delle Race Condition