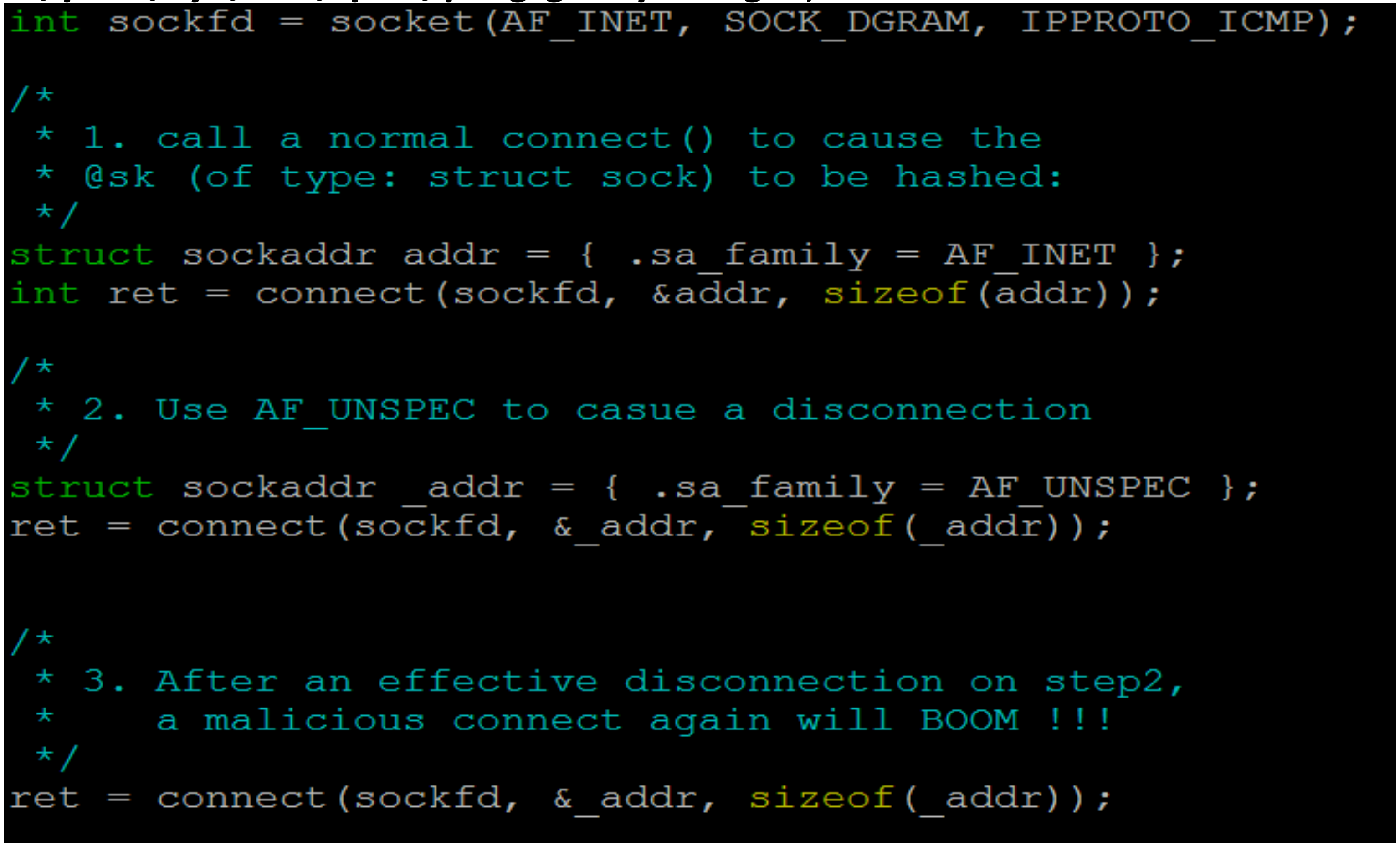
# Analysis of CVE-2015-3636

This CVE was exposed in v4.0 vanilla kernel by Keen Team, and was fixed in v4.1. The security issue was incorporated in to vanilla kernel in v3.0 .

All kernel codes in this slides are all taken from v4.0 .

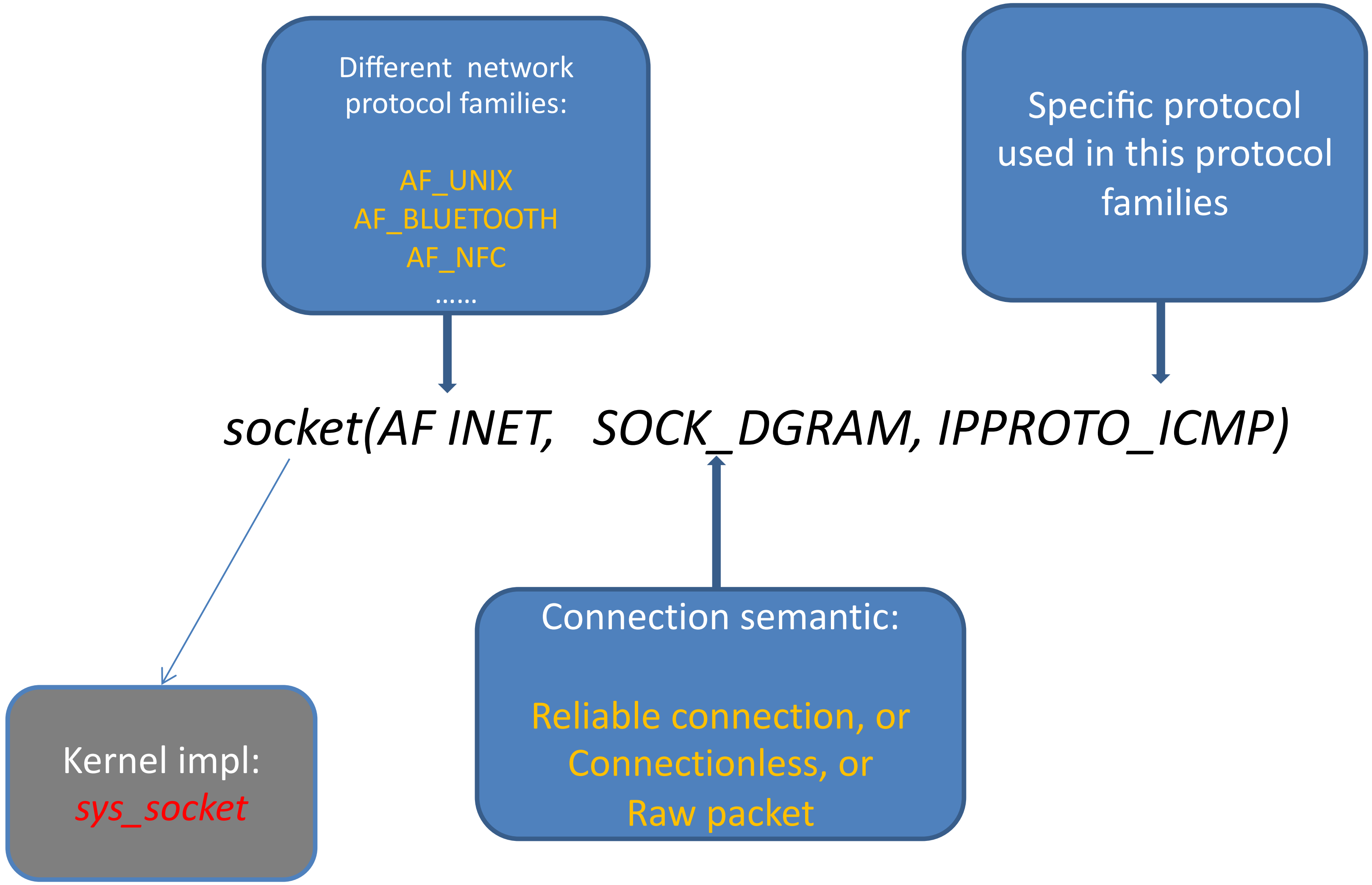
# POC of CVE-2015-3636

- This CVE stems from a UAF bug….
- On Android, even a common user can setup a ping socket ! ( controlled by ***/proc/sys/net/ipv4/ping group range*** )

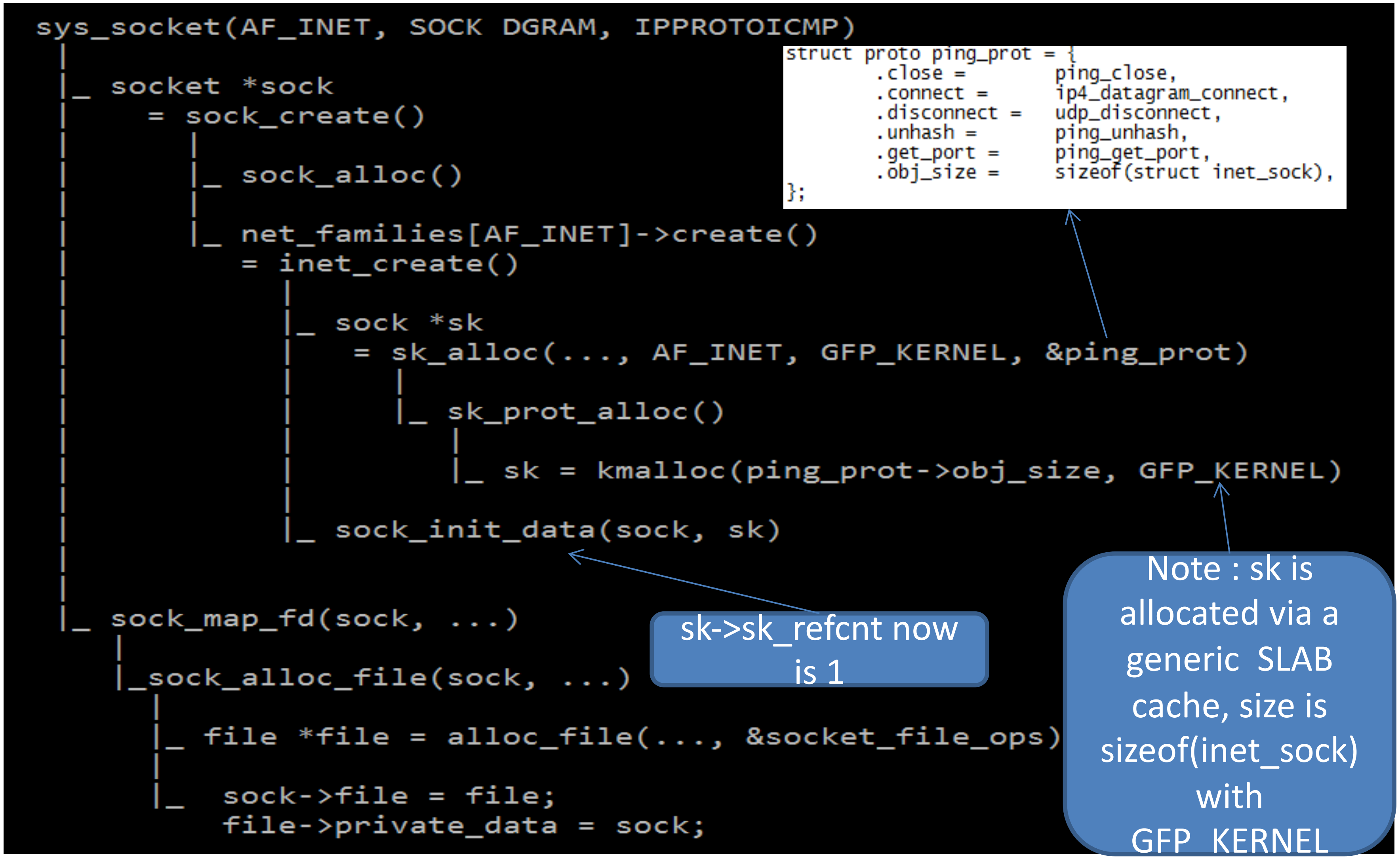


```
int sockfd = socket(AF_INET, SOCK_DGRAM, IPPROTO_ICMP);

/*
 * 1. call a normal connect() to cause the
 * @sk (of type: struct sock) to be hashed:
 */
struct sockaddr addr = { .sa_family = AF_INET };
int ret = connect(sockfd, &addr, sizeof(addr));

/*
 * 2. Use AF_UNSPEC to casue a disconnection
 */
struct sockaddr _addr = { .sa_family = AF_UNSPEC };
ret = connect(sockfd, &_addr, sizeof(_addr));


/*
 * 3. After an effective disconnection on step2,
 *    a malicious connect again will BOOM !!!
 */
ret = connect(sockfd, &_addr, sizeof(_addr));
```

# *socket()* syscall


Different network protocol families:
AF_UNIX
AF_BLUETOOTH
AF_NFC
……
Specific protocol used in this protocol families
*socket(AF INET,   SOCK_DGRAM, IPPROTO_ICMP)*
Kernel impl:
*sys_socket*
Connection semantic:
Reliable connection, or
Connectionless, or
Raw packet

# Implementation of sys_socket()



```
sys_socket(AF_INET, SOCK DGRAM, IPPROTOICMP)
  |
  |_ socket *sock
  |     = sock_create()
  |         |
  |         |_ sock_alloc()
  |         |
  |         |_ net_families[AF_INET]->create()
  |              = inet_create()
  |                  |
  |                  |_ sock *sk
  |                  |    = sk_alloc(..., AF_INET, GFP_KERNEL, &ping_prot)
  |                  |        |
  |                  |        |_ sk_prot_alloc()
  |                  |             |
  |                  |             |_ sk = kmalloc(ping_prot->obj_size, GFP_KERNEL)
  |                  |
  |                  |_ sock_init_data(sock, sk)
  |
  |
  |_ sock_map_fd(sock, ...)
       |
       |_sock_alloc_file(sock, ...)
          |
          |_ file *file = alloc_file(..., &socket_file_ops)
          |
          |_  sock->file = file;
              file->private_data = sock;
```

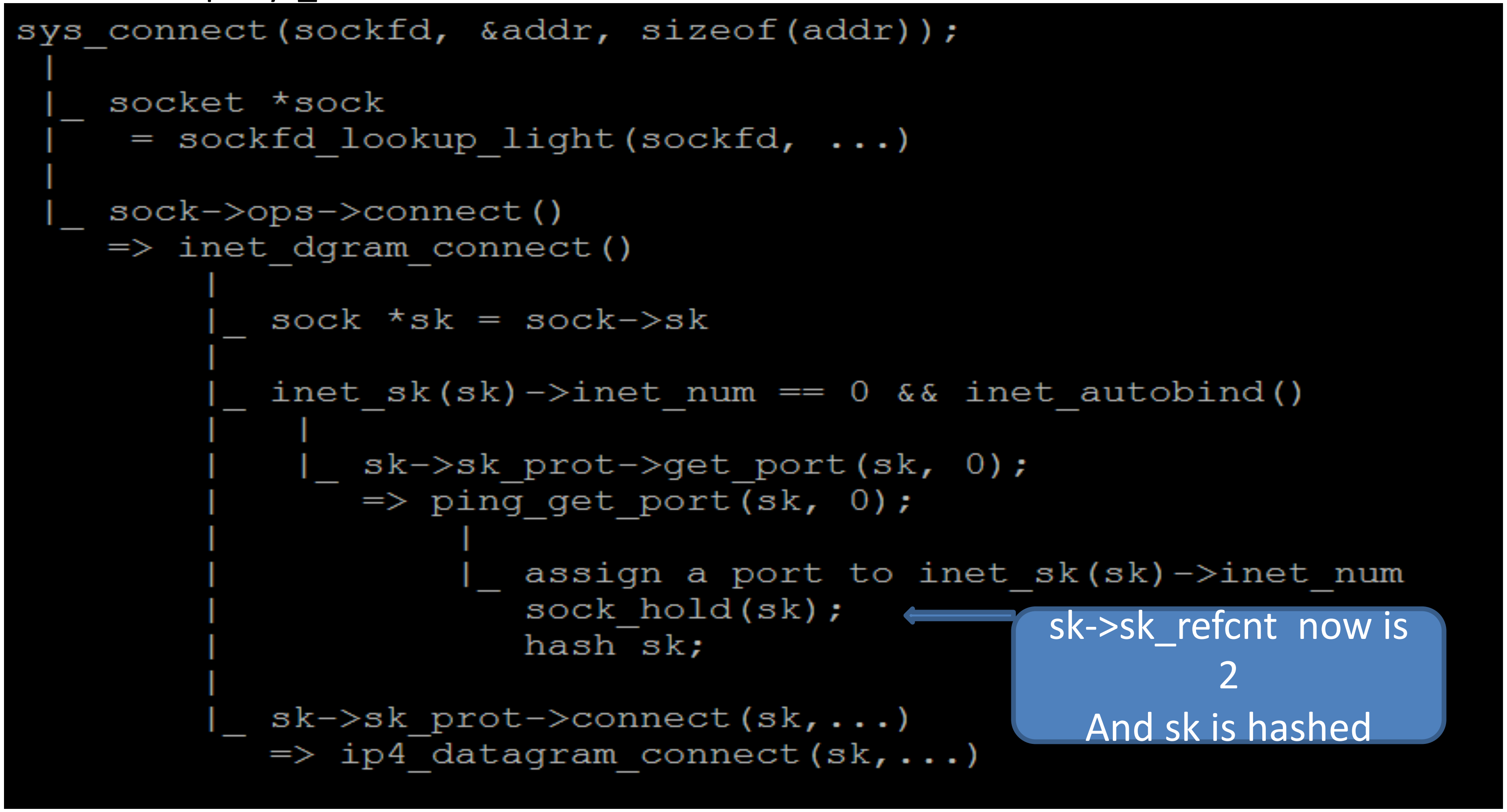
```
struct proto ping_prot = {
        .close =        ping_close,
        .connect =      ip4_datagram_connect,
        .disconnect =   udp_disconnect,
        .unhash =       ping_unhash,
        .get_port =     ping_get_port,
        .obj_size =     sizeof(struct inet_sock),
};
```

sk->sk_refcnt now is 1

Note : sk is allocated via a generic SLAB cache, size is sizeof(inet_sock) with GFP_KERNEL

# Step 1:After socket(), a normal connect()

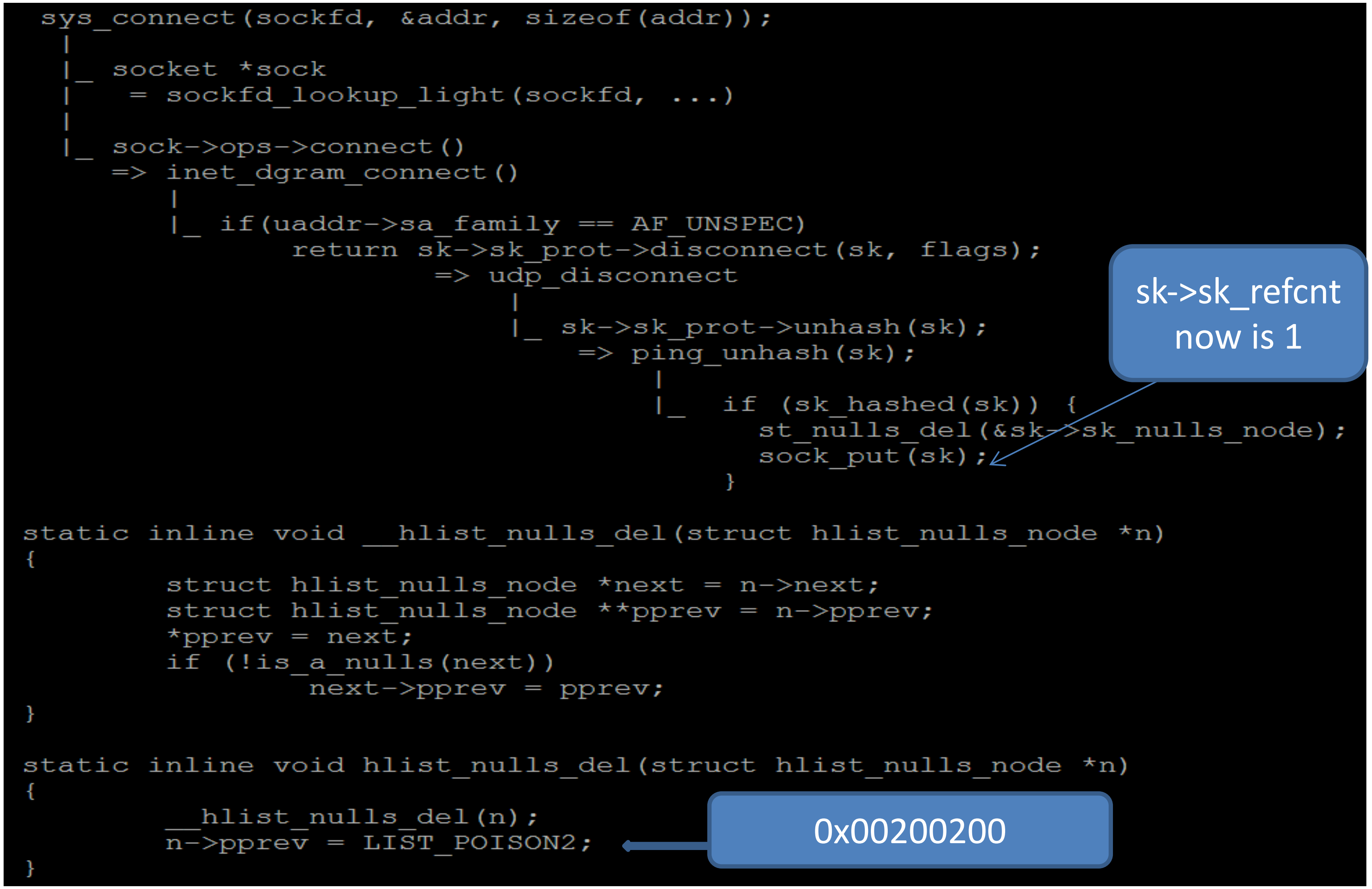
- struct sockaddr addr = { .sa_family = AF_INET };
- int ret = connect(sockfd, &addr, sizeof(addr));
- Kernel impl: sys_connect



```
sys_connect(sockfd, &addr, sizeof(addr));
  |
  |_ socket *sock
  |   = sockfd_lookup_light(sockfd, ...)
  |
  |_ sock->ops->connect()
     => inet_dgram_connect()
              |
              |_ sock *sk = sock->sk
              |
              |_ inet_sk(sk)->inet_num == 0 && inet_autobind()
              |    |
              |    |_ sk->sk_prot->get_port(sk, 0);
              |       => ping_get_port(sk, 0);
              |                |
              |                |_ assign a port to inet_sk(sk)->inet_num
              |                   sock_hold(sk);
              |                   hash sk;
              |
              |_ sk->sk_prot->connect(sk,...)
                 => ip4_datagram_connect(sk,...)
```

## Step 2: AF_UNSPEC connect() effectively disconnect

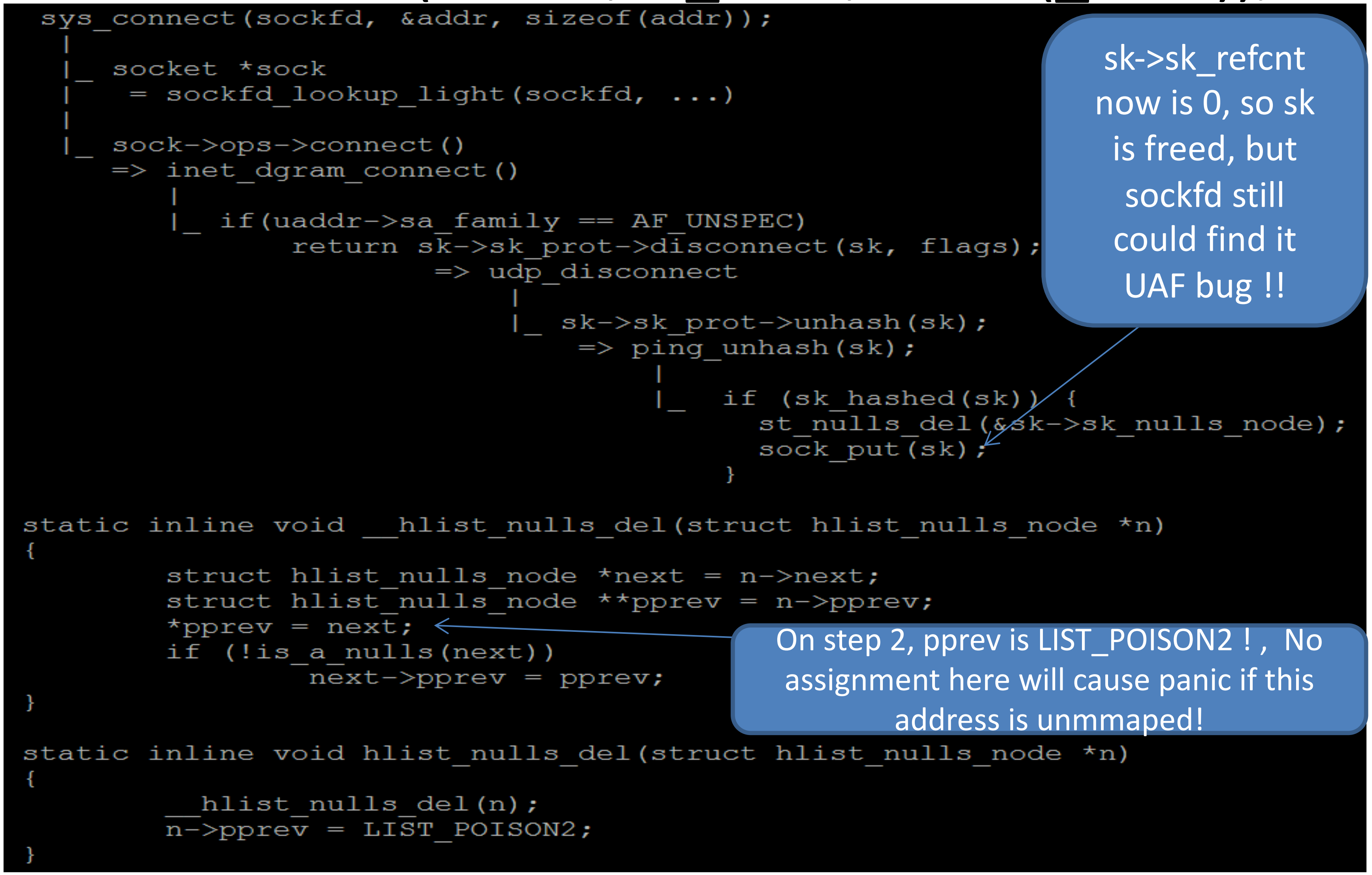
- struct sockaddr _addr = { .sa_family = AF_UNSPEC };
- ret = connect(sockfd, &_addr, sizeof(_addr));



```
 sys_connect(sockfd, &addr, sizeof(addr));
   |
   |_ socket *sock
   |   = sockfd_lookup_light(sockfd, ...)
   |
   |_ sock->ops->connect()
      => inet_dgram_connect()
          |
          |_ if(uaddr->sa_family == AF_UNSPEC)
                 return sk->sk_prot->disconnect(sk, flags);
                         => udp_disconnect
                              |
                              |_ sk->sk_prot->unhash(sk);
                                  => ping_unhash(sk);
                                      |
                                      |_  if (sk_hashed(sk)) {
                                            st_nulls_del(&sk->sk_nulls_node);
                                            sock_put(sk);
                                          }

static inline void __hlist_nulls_del(struct hlist_nulls_node *n)
{
        struct hlist_nulls_node *next = n->next;
        struct hlist_nulls_node **pprev = n->pprev;
        *pprev = next;
        if (!is_a_nulls(next))
                next->pprev = pprev;
}

static inline void hlist_nulls_del(struct hlist_nulls_node *n)
{
        __hlist_nulls_del(n);
        n->pprev = LIST_POISON2;
}
```

# Step 3: One more AF_UNSPEC malicious connect()…

- ret = connect(sockfd, &_addr, sizeof(_addr));



```
 sys_connect(sockfd, &addr, sizeof(addr));
   |
   |_ socket *sock
   |   = sockfd_lookup_light(sockfd, ...)
   |
   |_ sock->ops->connect()
      => inet_dgram_connect()
         |
         |_ if(uaddr->sa_family == AF_UNSPEC)
               return sk->sk_prot->disconnect(sk, flags);
                    => udp_disconnect
                         |
                         |_ sk->sk_prot->unhash(sk);
                              => ping_unhash(sk);
                                  |
                                  |_  if (sk_hashed(sk)) {
                                         st_nulls_del(&sk->sk_nulls_node);
                                         sock_put(sk);
                                      }

static inline void __hlist_nulls_del(struct hlist_nulls_node *n)
{
        struct hlist_nulls_node *next = n->next;
        struct hlist_nulls_node **pprev = n->pprev;
        *pprev = next;
        if (!is_a_nulls(next))
                next->pprev = pprev;
}

static inline void hlist_nulls_del(struct hlist_nulls_node *n)
{
        __hlist_nulls_del(n);
        n->pprev = LIST_POISON2;
}
```

sk->sk_refcnt now is 0, so sk is freed, but sockfd still could find it UAF bug !!

On step 2, pprev is LIST_POISON2 ! , No assignment here will cause panic if this address is unmmaped!

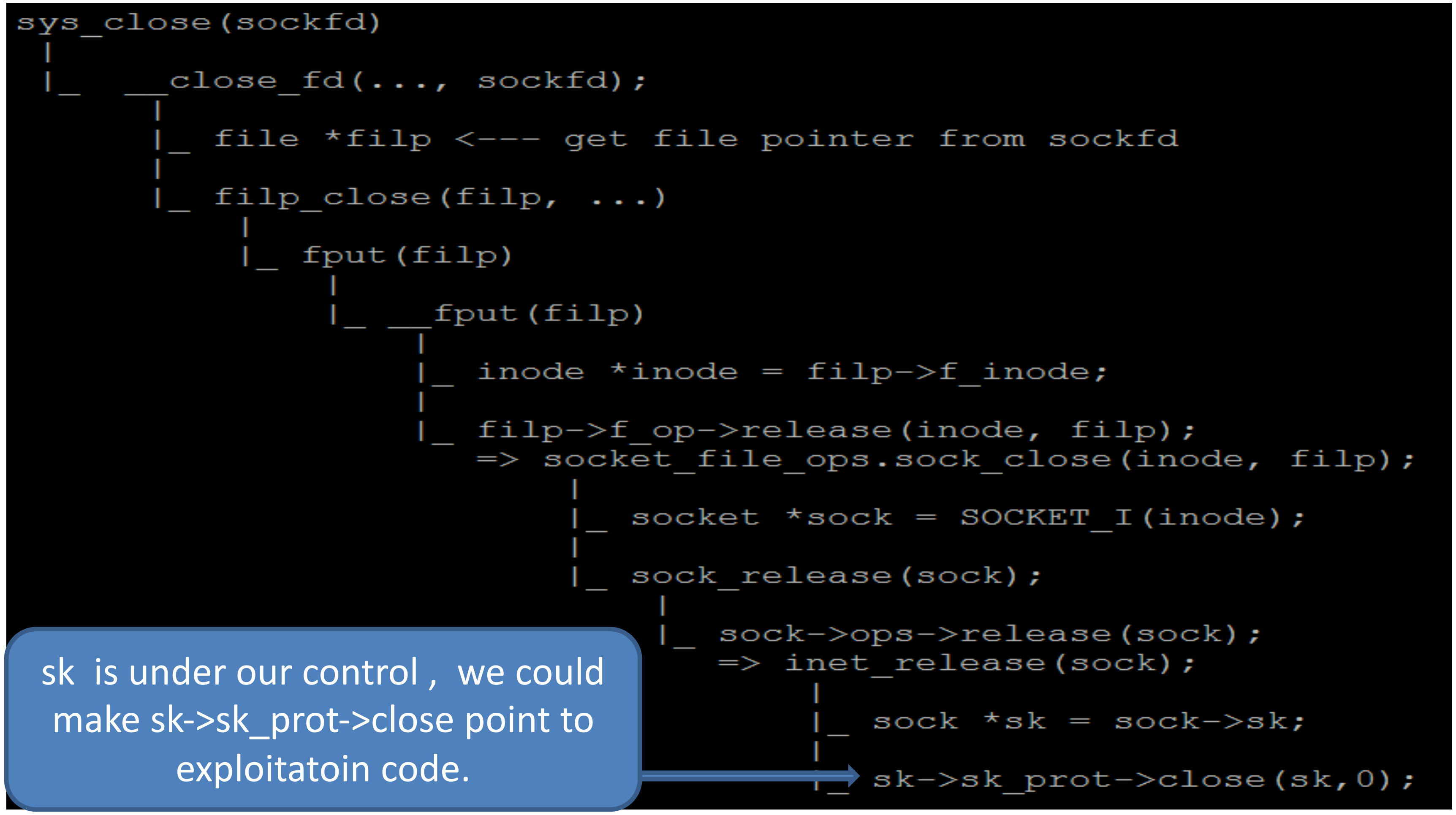
## Avoid panicking when assign to address LIST_POISON2 on step 3?

- LIST_POISON2 is 0x200200(~slightly larger than 2MB)
- If not mmap'ed, write to this addres on step 3 will BOOM.
- For *mmap*(), there is a '*/proc/sys/vm/mmap_min_addr*' tunable to decide the lowest virtual address permitted to mmap, default to 4K.
- On x86_32/x86_64, it is advisable to set to <=16K
- On arm/arm64, it is advisable to set to <=8K
- So, 0x200200 is no doubt above this limit and could be mmap!

- Solution: just *mmap* this address should avoid panic on step 3

## Ok, no panic, then how to exploit this UAF bug?

- Thought: use this freed sk( struct sock) to hijack the PC
- *Close()* could be one viable way! [kernel impl: *sys_close()*]



```
sys_close(sockfd)
 |
 |_   __close_fd(..., sockfd);
          |
          |_ file *filp <--- get file pointer from sockfd
          |
          |_ filp_close(filp, ...)
                |
                |_ fput(filp)
                      |
                      |_ __fput(filp)
                            |
                            |_ inode *inode = filp->f_inode;
                            |
                            |_ filp->f_op->release(inode, filp);
                               => socket_file_ops.sock_close(inode, filp);
                                     |
                                     |_ socket *sock = SOCKET_I(inode);
                                     |
                                     |_ sock_release(sock);
                                           |
                                           |_ sock->ops->release(sock);
                                              => inet_release(sock);
                                                    |
                                                    |_ sock *sk = sock->sk;
                                                    |
                                                    |_ sk->sk_prot->close(sk,0);
```

sk is under our control , we could make sk->sk_prot->close point to exploitatoin code.

# Where sk locates

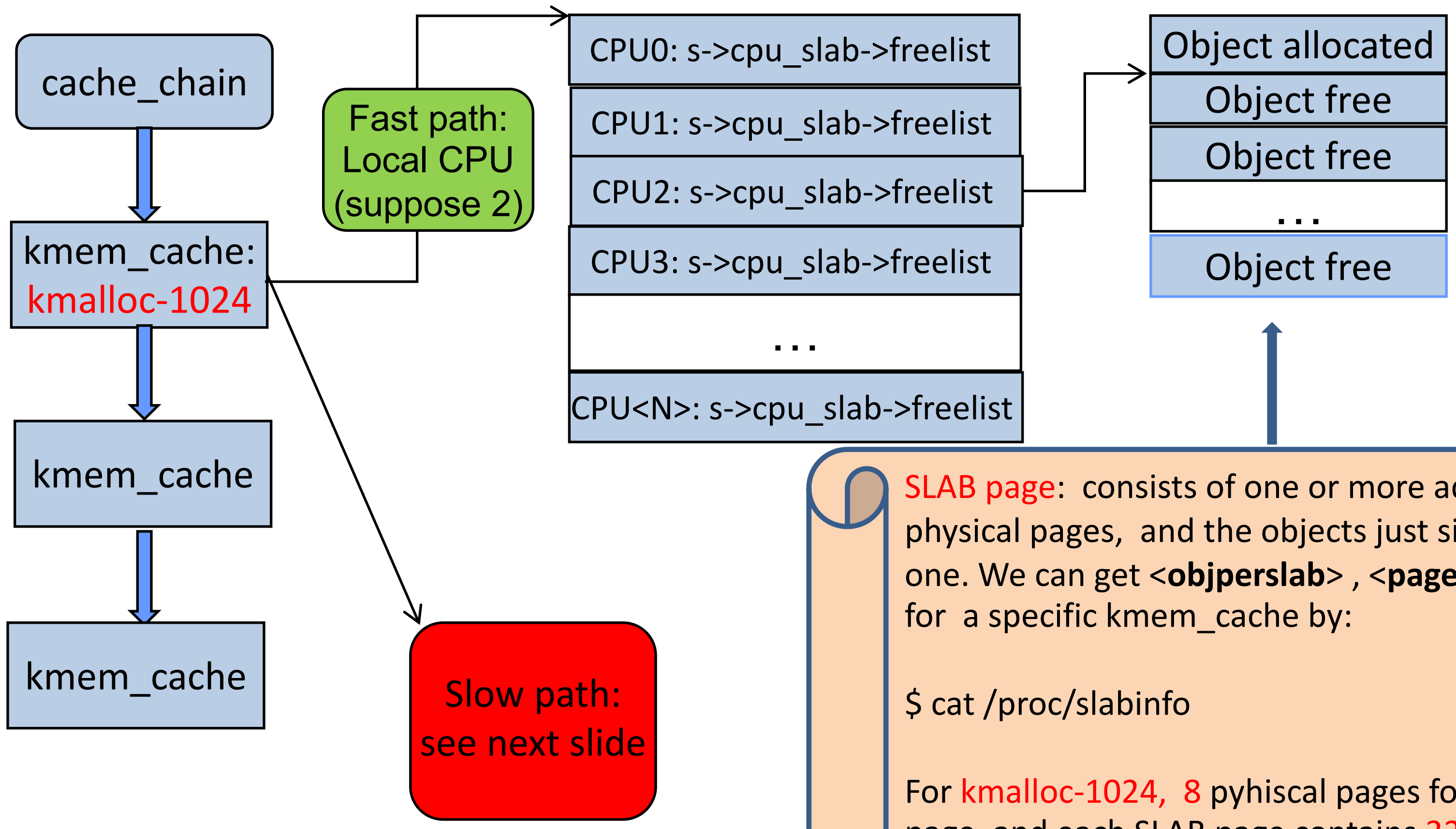
- Recall how sk is allocated:

  ***sk = kmalloc(ping_prot->obj_size, GFP_KERNEL)***
- sk is a generic SLAB cache
- ***ping_prot->obj_size == sizeof(inet_sock)***
  - Larger than 512 bytes, but smaller than 1024 bytes
  - There is no dedicated ***kmem_cache*** to allocate from(

    ***ping_prot->slab == NULL***)
  - so it will be allocated from generic *kmalloc-1024* SLAB cache(SLAB always rounds it to nearest pow-of-2)
- The underlying implementation of SLAB is SLUB
- For rest of this slides, I will call pages that contain SLAB cache as SLAB page, though the implementation is SLUB.

# SLUB implementation under the hood

Allocation path: kmalloc(), **kmem_cache *s**



SLAB page: consists of one or more adjacent physical pages, and the objects just sit one by one. We can get <**objperslab**> , <**pagesperslab**> for a specific kmem_cache by:

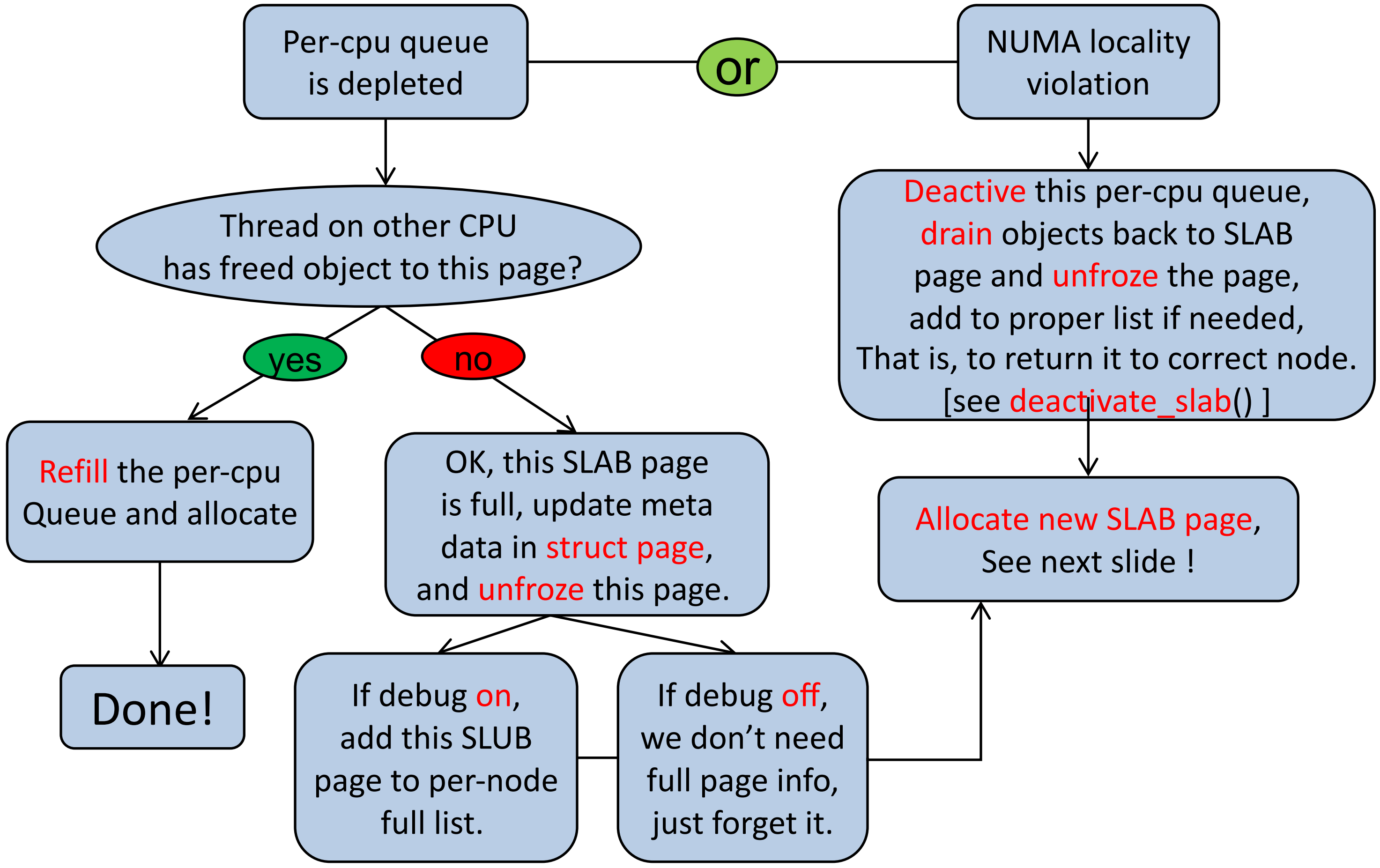
```
$ cat /proc/slabinfo
```

For kmalloc-1024, 8 pyhiscal pages form a SLAB page, and each SLAB page contains 32 objects:

$8 * 4096 = 32 * 1024$

# SLUB implementation under the hood(cont.)

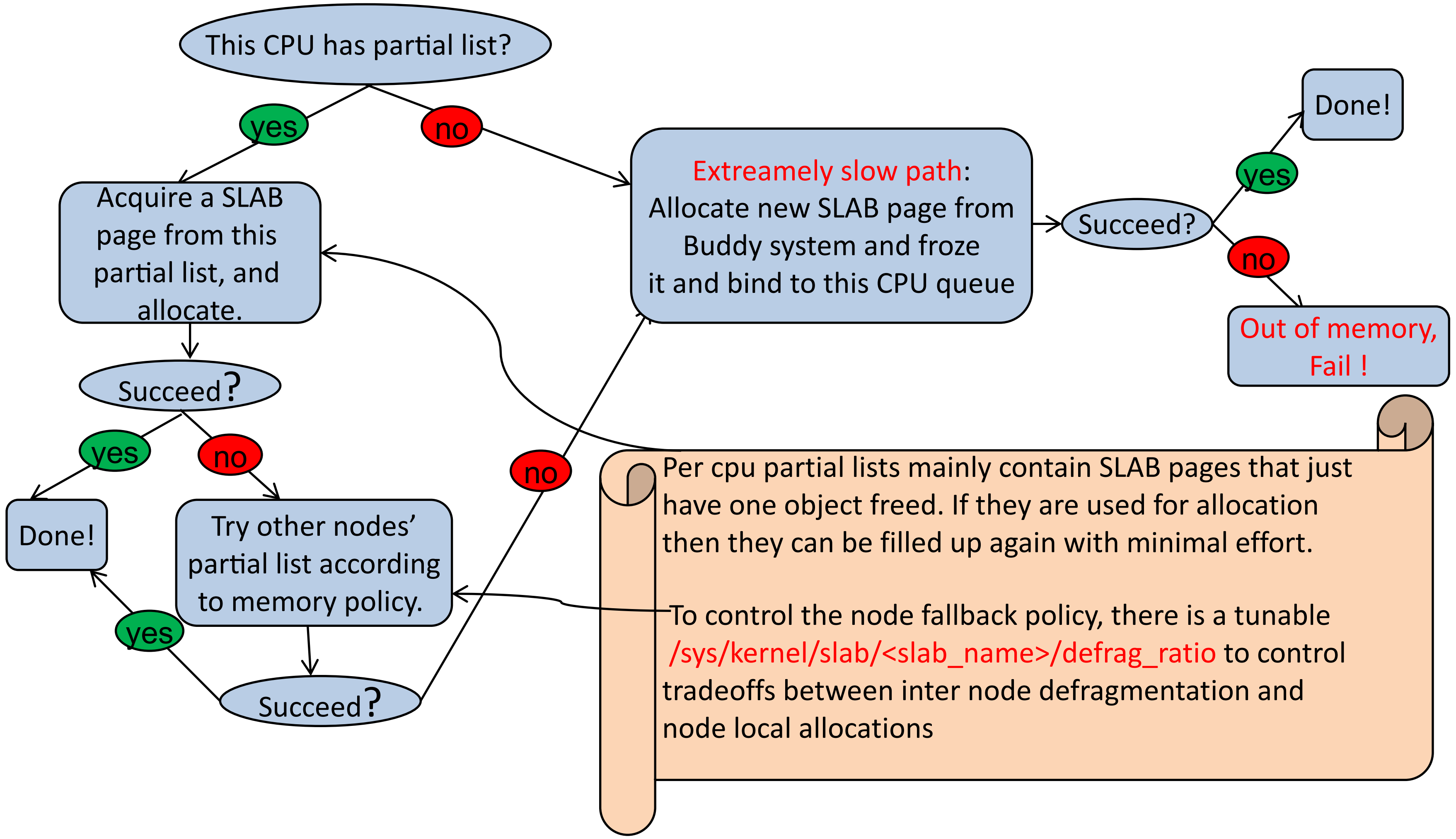
- Enter allocation slow path when:



Per-cpu queue is depleted
or
NUMA locality violation
Thread on other CPU has freed object to this page?
yes
no
Refill the per-cpu Queue and allocate
Done!
OK, this SLAB page is full, update meta data in struct page, and unfroze this page.
If debug on, add this SLUB page to per-node full list.
If debug off, we don't need full page info, just forget it.
Deactive this per-cpu queue, drain objects back to SLAB page and unfroze the page, add to proper list if needed, That is, to return it to correct node. [see deactivate_slab() ]
Allocate new SLAB page, See next slide !

# SLUB implementation under the hood(cont.)

- Allocation extremely slow path: allocate a new SLAB page


This CPU has partial list?
yes
no
Acquire a SLAB page from this partial list, and allocate.
Succeed?
yes
no
Done!
Try other nodes' partial list according to memory policy.
Succeed?
yes
no
Extreamely slow path: Allocate new SLAB page from Buddy system and froze it and bind to this CPU queue
Succeed?
yes
Done!
no
Out of memory, Fail !

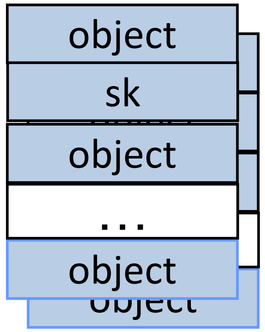
Per cpu partial lists mainly contain SLAB pages that just have one object freed. If they are used for allocation then they can be filled up again with minimal effort.

To control the node fallback policy, there is a tunable /sys/kernel/slab/<slab_name>/defrag_ratio to control tradeoffs between inter node defragmentation and node local allocations

# Can we refill the freed sk struct by heap overflow ?

- As proposed by the white paper[1], to allocate(sendmsg) objects adjacent to sk, and try to overflow it.
- difficulties:
  - kmalloc-1024 is a generic cache, where other objects(512B < size <=1024B) could be allocated.
  - On SMP preempt-able devices, we(attack thread) could be preempted(and migrated) and could not guarantee we allocate on the same per-CPU queue as sk.
  - We also could not guarantee we allocate adjacent to sk('cause other could preempt us and also allocate from kmalloc-1024)

  .....
- All in all, to hijack an isolated heap on an multiple CPUs device is not trivial. Now turn to other direction…

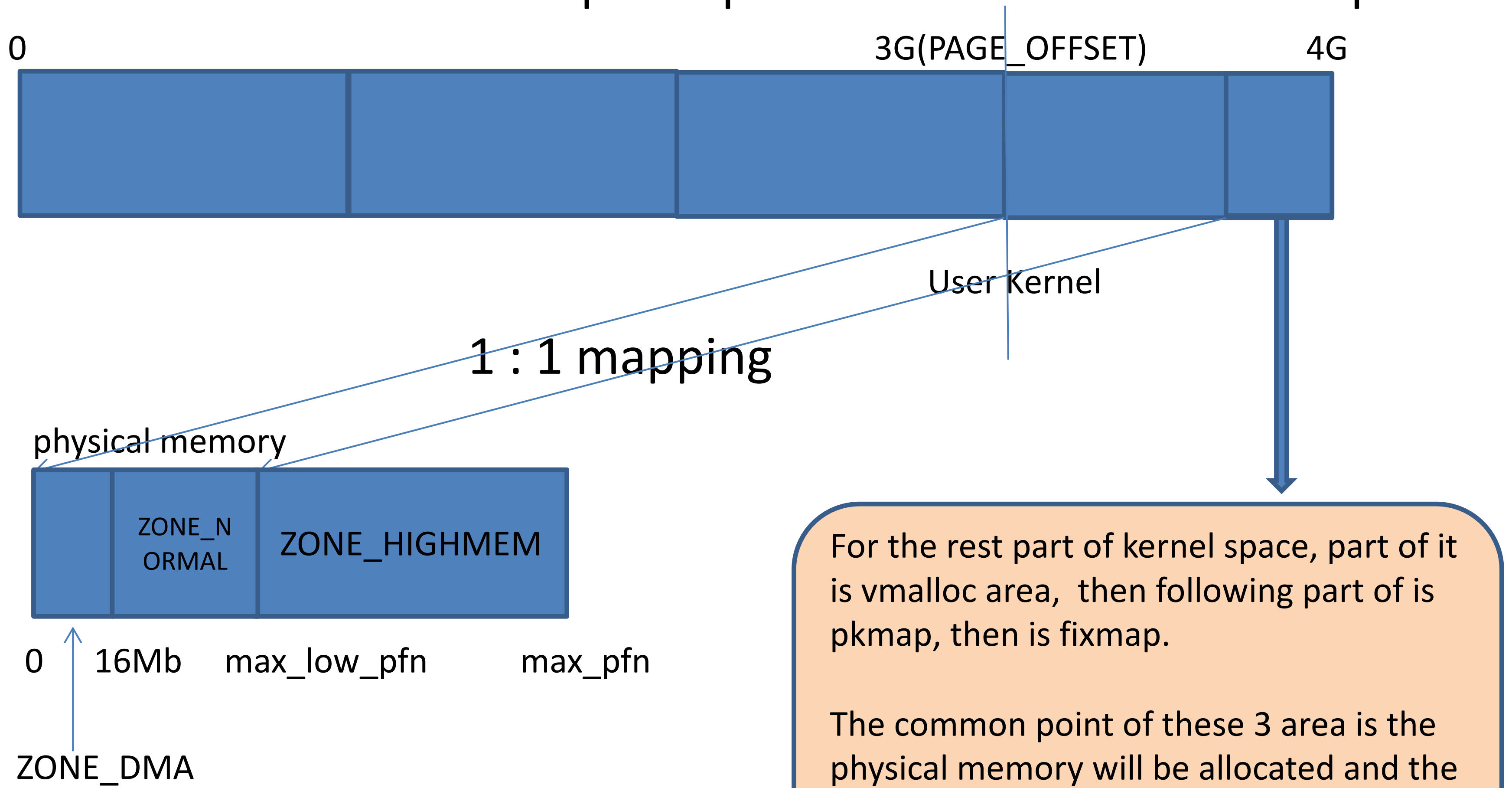


# Physmap, *a.k.a.* kernel Direct Mapping

- What is Physmap?
  - A universal practice adopted by most main-stream modern operating systems to directly map parts of, or all of physical memory into kernel virtual space, i.e. to pre-fill page tables for that topical memory area, to accelerate kernel access.
  - And it is directly 1:1 shift mapping, so it is trivial and fast for kernel to get the physical address (just an offset calculation)
  - For x86_32 / arm, kernel will map physical memory address [0, max_low_pfn) to kernel space during booting (x86: start_kernel -> steup_arch -> init_mem_mapping)
  - For x86_64 / arm64, kernel will map physical memory address [0, max_pfn) to kernel space during booting (x86_64: start_kernel -> steup_arch -> init_mem_mapping)

# How Physmap is implemented in Linux?

- Take a 3:1 user-kernel space split 32 bit kernel as an example

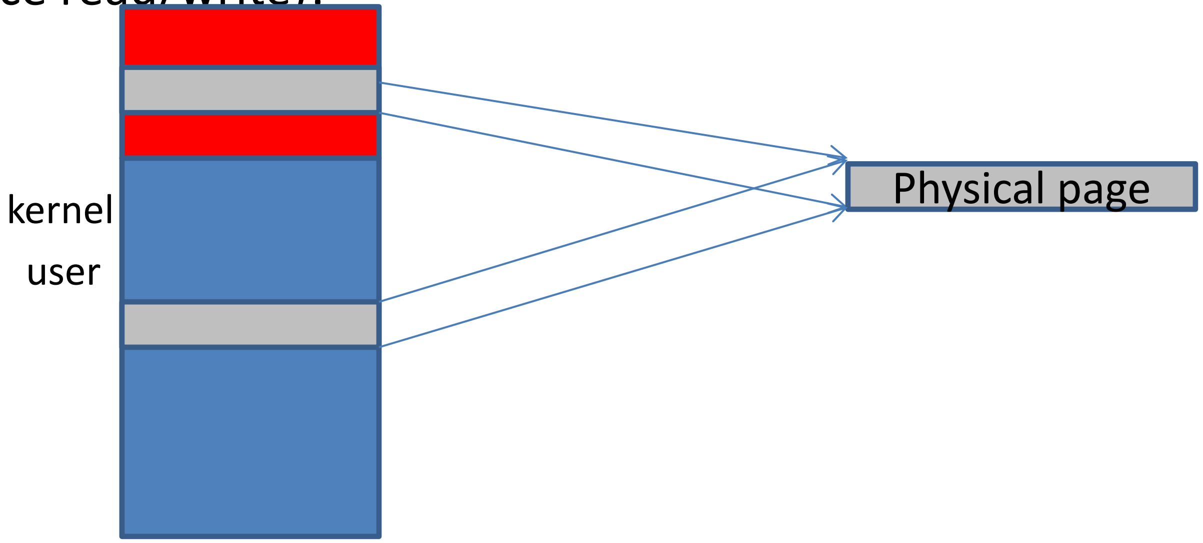


For the rest part of kernel space, part of it is vmalloc area, then following part of is pkmap, then is fixmap.

The common point of these 3 area is the physical memory will be allocated and the page table will be built dynamically.

## Address aliasing

- If a page allocated in user space falls in the Physmap area,  it has two page table entries referred to it:
  - One is the entry that in the kernel direct mapping.
  - The other is the entry that in the user space.
- …. So if we can try to allocate the page frame that contains the freed sk struct, we can overwrite it in user space (we have full control of our user space read/write)!



## The way to create address aliasing!

- Questions need to answer:
  1. Does that very SLAB page fall in Physmap area?( If not, we won't have the address aliasing).
  2. If it does fall in Physmap area, can user allocate that very page frame?

## Q1: Does that very SLAB page fall in Physmap area?

- YES! The SLAB page contains sk struct is located in Physmap area
  - Proof:
    1. sk belongs to *kmalloc-1024* SLAB cache, which is initially allocated during *kmem_cache_init,* with a GFP_NOWAIT, and it translates into ZONE_NORMAL(see gfp_zone())
    2. If *kmalloc-1024* SLAB cache runs out of its cache page, it will allocate new SLAB page from buddy allocator. And recall that we allocate sk with:

       sk = kmalloc(ping_prot->obj_size, GFP_KERNEL),

       in which GFP_KERNEL is used when allocating new page from buddy allocator, and it also translates into ZONE_NORMAL.
    3. kernel page allocator zone policy is : cadidate zone <= suggested zone, so here only possible candidate zones are:

       ZONE_NORMAL -> ZONE_DMA

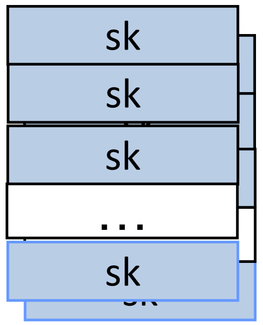
## Q2: Can user allocate that very page?

- In order to create address aliasing, user space must allocate that SLAB page contains sk(F.e. calling *mmap()*)
- Recall that sk allocates from *kmalloc-1024*, a 8-page SLAB cache, which contain 8*4096/1024 = 32 objects, among them one object is sk.
- Even if we has freed sk, but if at that same page there are other objects allocated to other threads(it is a generic cache), then this SLAB page won't be freed to buddy allocator, thus user won't be able to allocate this page.
- … So the trick is to spray the SLAB page, allocate(by socket()) as more as possible sk's, and then free them, then we are likely to free a complete SLAB page that were all sk's !!!

# Create address aliasing

1. Spray SLAB cache by calling socket() multiple times.
2. As long we create enough sk's and latter free them, we will finally sprayed with one SLAB page with all sk's.
3. We normally free most of these sk's, but we then create vulnerable sk's by using the two times connection method.
4. If a whole SLAB page full of sk's now get all freed, SLUB will return the SLAB page(8 adjacent page) to buddy allocator.
5. User uses mmap() to try to allocate these pages,

   and hopefully we will create address aliasing !!!



*kmalloc-1024*

## How do we know we have created address aliasing?

- Use the trick described in white paper[1], we use 8 dwords data to spray the mmap area, among the data we put the key values to control the flow (to hijack ***sk->sk_prot***).
  - why 8 dwords? What is the layout like? (for discussion when interview)
- Among the data, there is also some magic value. And we use ***ioctl(sockfd, SIOCGSTAMPNS, (struct timespec*))*** every several sprays, to leak out ***sk->sk_stamp***, and compare it with our magic value, thus we know whether we have allocated that very page.
- When we have got that page, we have controlled the ***sk->sk_prot*** to our controlled area(user space), where we will deviate the fake 'close' function pointer to help us escalate to root privilege.

# How to root?

- One general way is to exploit a **kernel mode** write bug (so we can arbitrarily write) to modify critical credential information which is maintained in kernel space, to gain the highest privilege.
- In Linux, this means to write current thread's(attack process) ***task_struct.cred.{uid, gid}*** to 0.

➔ So since we could hijack the ***sk->sk_prot->close*** function pointer, we should try to steer it to reach this goal, either

- direct write this struct field, or
- call ***commit_creds(prepare_kernel_cred())*** [if viable]

# Closer look at what we have now

1. We can use ***sys_close()***, which will calls **sk->sk_prot->close()**, and it runs in kernel mode!
2. We have successfully created address aliasing of page that contains **sk**, and hijacked **sk->sk_prot** field to point to some place where we place a fake **'close'** pointer.

=> ***sk->sk_prot*** is under our control, and

***->close*** is also under our control.

**What to write to these two pointers matters** !

# What to write matters

- Consideration on what to write to **sk->sk_prot** and **->close :**
  - Can these two pointers point to kernel space address? Theoretically can, only if they fall in the Physmap area and we create address aliasing in user space. But this is non-trivial. Won't consider this option.
  - Can these two pointers point to user space address? Sure, we have full control over our user space. But…
    - Can we read user space address from kernel mode(in ***sys_close***, because we need read these two pointers before calling ***->close***)?

      Thankfully yes, because most Android devices don't ship with Arm CPUs that have **PAN(Privileged Access Never)** on.
    - Can we call ***->close*** which points to user space address from kernel mode(in ***sys_close***)?

      It depends on whether CPU has **PXN(Privileged eXecute Never)** on or not.

## PXN: can or can not access user space

- **PXN off**

  Trivial ! ***ret2usr*** attack! Since ***sys_close()*** runs in kernel mode, it can write to kernel address. We can make ***->close*** point to user space where contains code like:

  *current_thread_info()->task->cred.uid =0;*

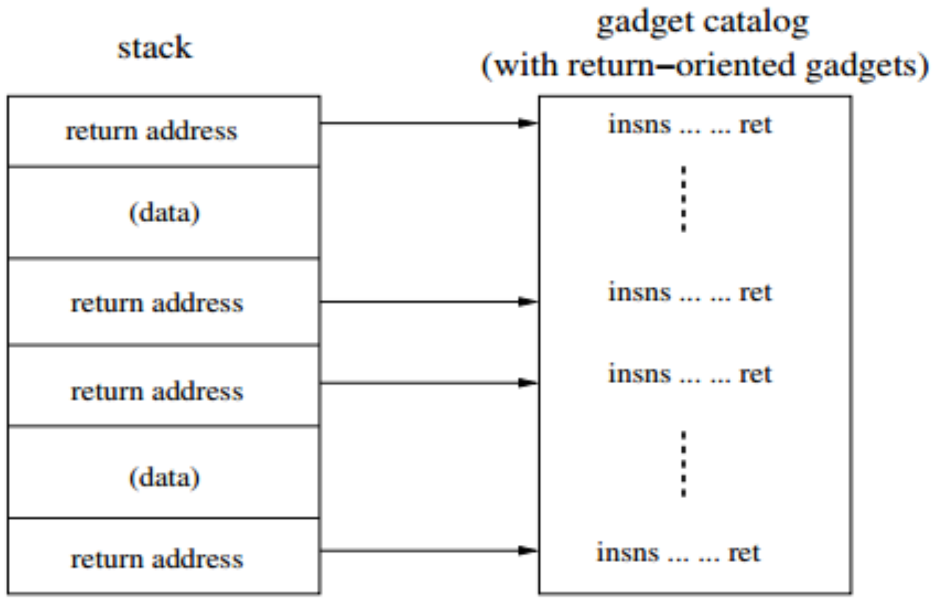
  *current_thread_info()->task->cred.uid =0;*

  **Done! We are root!**

- **PXN on**
  - We can't execute code in user space.
  - We can't even exploit the Physmap method and make ***->close*** point to kernel space aliasing adress, since the W^X mechanism is in effect.
  - We have to resort to a powerful weapon call ROP, or better, JOP!

# ROP

- **ROP(Return Oriented Programming)**
  A method to sidestep the W^X defense. To piece together the shellcode by repurposing current runnable code snippets in text section or mapped library(libc for example), which ends with ***ret***. Each snippet is called ***gadget****.* By putting these ***gadget starting*** addess on stack, so a ***ret*** in previous ***gadget*** will pop the stack top and the drive control flow to next ***gadget .***

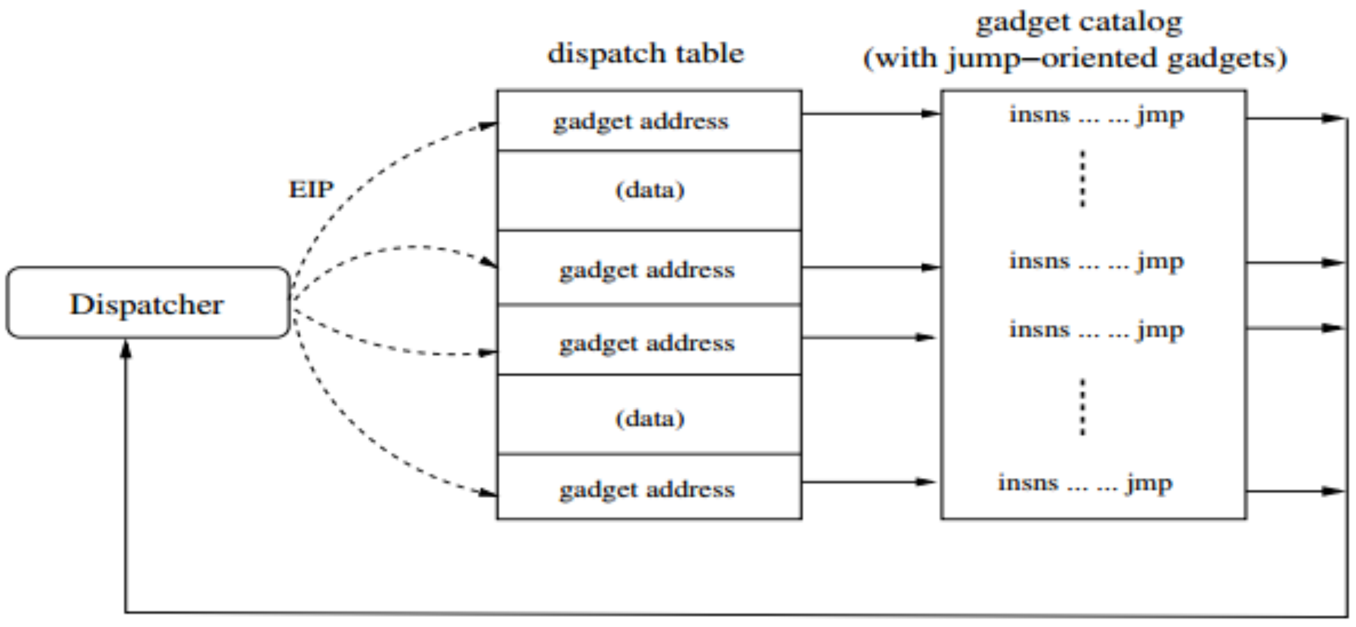


(a) The ROP model

# JOP

- **JOP(Jump Oriented Programming)**

  Also repurpose code ***gadget*** , but eliminate the need of a stack. There is a ***dispatcher gadget***, driving control flow to different ***functional gadgets***, which perform basic logical function. And each ***functional gadgets*** will jump to the ***dispatcher gadget.***



(b) The JOP model

# Favor JOP over ROP?

- Limitation of ROP
  - ROP needs a stack. In this case, we shall first pivot the kernel stack pointer to a user space page, where locates the required *gadget* address and register content setup.
- Is it dangerous for a deviating kernel stack pointer?
  - Subtle:

    The ongoing syscall may be interrupted, and CPU hardware context will be saved on current 'kernel stack', and transfer to a irq stack, previous stack pointer will be saved. After return from interrupt, stack pointer will be restored. All these seem OK? ! (for discussion)

## Final step to root

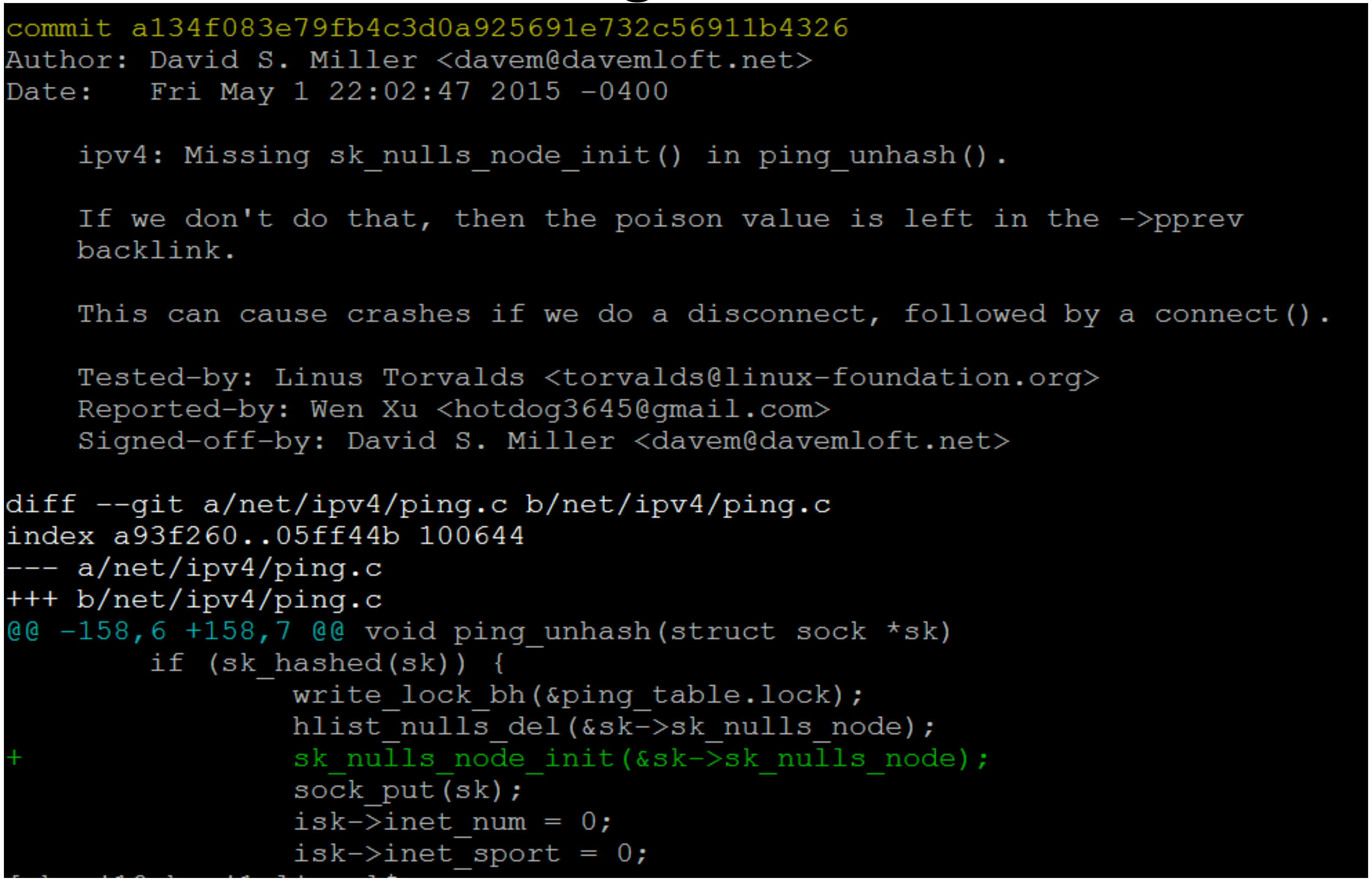
- To leak out the SP value to get *thread_info,* then *task_struct* could be fetched, thus the *cred* struct, then set *cred.{uid, gid}* to 0, to make us the root user!
- So *gadgets* should be carefully chosen to achieve this goal.
- So either PXN is on or off, we can successfully set us to root user !

## And the *addr_limit*?

- We have achieved the root privilege. But even a root user, could not access kernel space when it goes back to user mode.
- What is ***addr_limit*** and why it matters?
  - Per thread limit in ***thread_info*** to avoid kernel from being fooled to overwrite kernel space(user fakes a kernel address as user address)
  - it is checked in ***copy_{to, from}_kernel*** function
  - Sometimes kernel needs do kernle-to-kernel copy by reusing some syscall interfaces that call ***copy_{to, from}_kernel,*** it has to temporarily set ***addr_limit*** to 0(x86 sets it to KERNEL_DS(-1UL), because Arm will substract 1 from ***addr_limit*** before using it to check the limit, so set it to 0 here)
  - So after setting ***addr_limit*** to 0, even we are in user mode, we can freely access kernel space, that is arbitrary read/write.

# How to plug this hole?

- In v4.1, this CVE bug is fixed



```
commit a134f083e79fb4c3d0a925691e732c56911b4326
Author: David S. Miller <davem@davemloft.net>
Date:   Fri May 1 22:02:47 2015 -0400

    ipv4: Missing sk_nulls_node_init() in ping_unhash().

    If we don't do that, then the poison value is left in the ->pprev
    backlink.

    This can cause crashes if we do a disconnect, followed by a connect().

    Tested-by: Linus Torvalds <torvalds@linux-foundation.org>
    Reported-by: Wen Xu <hotdog3645@gmail.com>
    Signed-off-by: David S. Miller <davem@davemloft.net>

diff --git a/net/ipv4/ping.c b/net/ipv4/ping.c
index a93f260..05ff44b 100644
--- a/net/ipv4/ping.c
+++ b/net/ipv4/ping.c
@@ -158,6 +158,7 @@ void ping_unhash(struct sock *sk)
        if (sk_hashed(sk)) {
                write_lock_bh(&ping_table.lock);
                hlist_nulls_del(&sk->sk_nulls_node);
+               sk_nulls_node_init(&sk->sk_nulls_node);
                sock_put(sk);
                isk->inet_num = 0;
                isk->inet_sport = 0;
```

# Why the fix plug the hole?

- The ***sk->sk_nulls_node.pprev*** will be set to NULL, instead of **0x200200**. And NULL is absolutely not visitable nor mappable, so it will crash after we ***connect*** for the second time before we have the chance to trigger the UAF bug.
- For the same reason, in git commit 8a5e5e02fc83 the **LIST_POISON{1,2}** value is also changed to from 0x200200 to 0x200 to defend this kind of bugs.
- On PCs, deprive common users of creating ping sockets by tune ***/proc/sys/net/ipv4/ping_group_range*** , which is also could be used to plug the hole.

# References

- [1] https://www.blackhat.com/docs/us-15/materials/us-15-Xu-Ah-Universal-Android-Rooting-Is-Back-wp.pdf
- [2]https://www.comp.nus.edu.sg/~liangzk/papers/asiaccs11.pdf