Linux Kernel Futex Fun: Exploiting CVE-2014-3153

Dougall Johnson

azimuth SECURITY

Overview

- Futex system call
- Kernel implementation
- CVE-2014-3153
- My approach to exploiting it

Futexes

- “Fast user-space mutexes”
- 32-bit integer in shared memory
- Designed to be used entirely in user-space unless contended
- When the lock is contended, the **futex** system call is used

Futex Syscall

```
int futex(int *uaddr, int op, int val,
          const struct timespec *timeout,
          int *uaddr2, int val3);
```

- Action depends on the **op** argument
- The arguments can be unused, or cast to different types
- No glibc wrapper, need to use the **syscall** function to invoke it: **syscall(SYS_futex, ...)**

FUTEX_WAIT and FUTEX_WAKE

- When lock acquisition fails, the thread makes the futex(…, FUTEX_WAIT, …) system call, which sleeps the thread

- When the lock is released, the owner will make the futex(…, FUTEX_WAKE, …) system call, which will wake up any waiters

FUTEX_REQUEUE

- Thundering herd problem: **FUTEX_WAKE** wakes up several processes, all of which attempt to acquire another futex

- Instead, **FUTEX_REQUEUE** moves a number of waiters to another futex without waking them

PI futexes

- “Priority inheritance” futexes are semantically different, but similar

- The user-space futex value is zero for unlocked, or holds the thread ID of the owner

- The **FUTEX_LOCK_PI** and **FUTEX_UNLOCK_PI** calls are used instead of wait and wake

- Unlocking a PI-futex wakes only the highest priority waiter

FUTEX_CMP_REQUEUE_PI

- Avoid “thundering herd” when moving from a non-PI futex to a PI-futex
- Waiters call **FUTEX_WAIT_REQUEUE_PI** to sleep
- Another thread calls **FUTEX_CMP_REQUEUE_PI** to “requeue” the waiters to the PI-futex
- If the PI-futex is unlocked, one of the threads will lock it and wake

Kernel Implementation

- Kernel keeps track of waiters, but forgets about futexes with no waiters
- A **futex_q** structure represents each waiting thread
- An additional **rt_mutex_waiter** structure is used for each thread waiting on a PI futex

futex_q

- Waiters on pi_futexes only in that they have a non-NULL **pi_state** and **rt_waiter**
- Waiters created by **WAIT_REQUEUE_PI** have a **requeue_pi_key** indicating the destination PI-futex
- These structures are only needed while the waiter is waiting, so they are allocated on the thread’s kernel stack

```
struct futex_q {
    plist_node list;

    task_struct *task;
    spinlock_t *lock_ptr;
    futex_key key;
    futex_pi_state *pi_state;
    rt_mutex_waiter *rt_waiter;
    futex_key *requeue_pi_key;
    u32 bitset;
};
```

rt_mutex_waiter

- These are kept in a priority-list on an **rt_mutex**
- The waiter at the start of list will be woken when the PI futex is unlocked
- These are also allocated on the thread's kernel stack

```
struct rt_mutex_waiter {
    plist_node list_entry;
    plist_node pi_list_entry;
    task_struct *task;
    rt_mutex *lock;
}
```

CVE-2014-3153

- Posted to oss-sec mailing list on June 5th
- Explanations was somewhat cryptic:

```
Forbid uaddr == uaddr2 in futex_requeue(..., requeue_pi=1)

If uaddr == uaddr2, then we have broken the rule of only
requeueing from a non-pi futex to a pi futex with this call.
If we attempt this, then dangling pointers may be left for
rt_waiter resulting in an exploitable condition.
```

Huh?

- Requeueing from **uaddr1** to **uaddr2** doesn’t look possible
- **FUTEX_WAIT_REQUEUE_PI** already verifies that **uaddr1 != uaddr2**, and then sets **requeue_pi_key** to the key for **uaddr2**
- **FUTEX_CMP_REQUEUE_PI** fails unless **uaddr2** matches the **requeue_pi_key**
- Even if I could, it wouldn’t necessarily break things

Triggering the Vulnerability

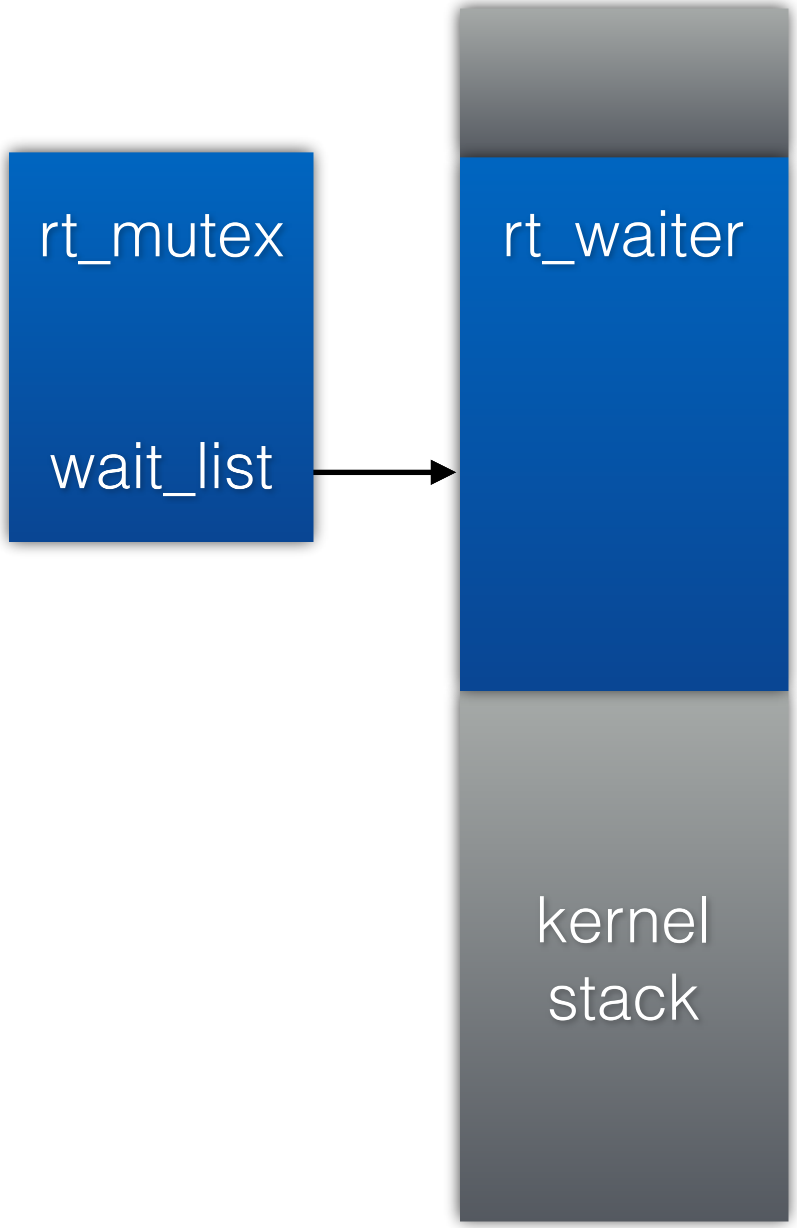
- The **requeue_pi_key** field is never cleared, so I can requeue twice to the same destination

- By setting the value to zero (unlocked) in memory, the thread will be resumed as though it had never joined the **rt_mutex_waiter** list

```
Thread A: futex_wait_requeue_pi( &futex1, &futex2 )
Thread B: futex_lock_pi( &futex2 )
Thread B: futex_cmp_requeue_pi( &futex1, &futex2 )
Thread B: futex2 = 0
Thread B: futex_cmp_requeue_pi( &futex2, &futex2 )
```

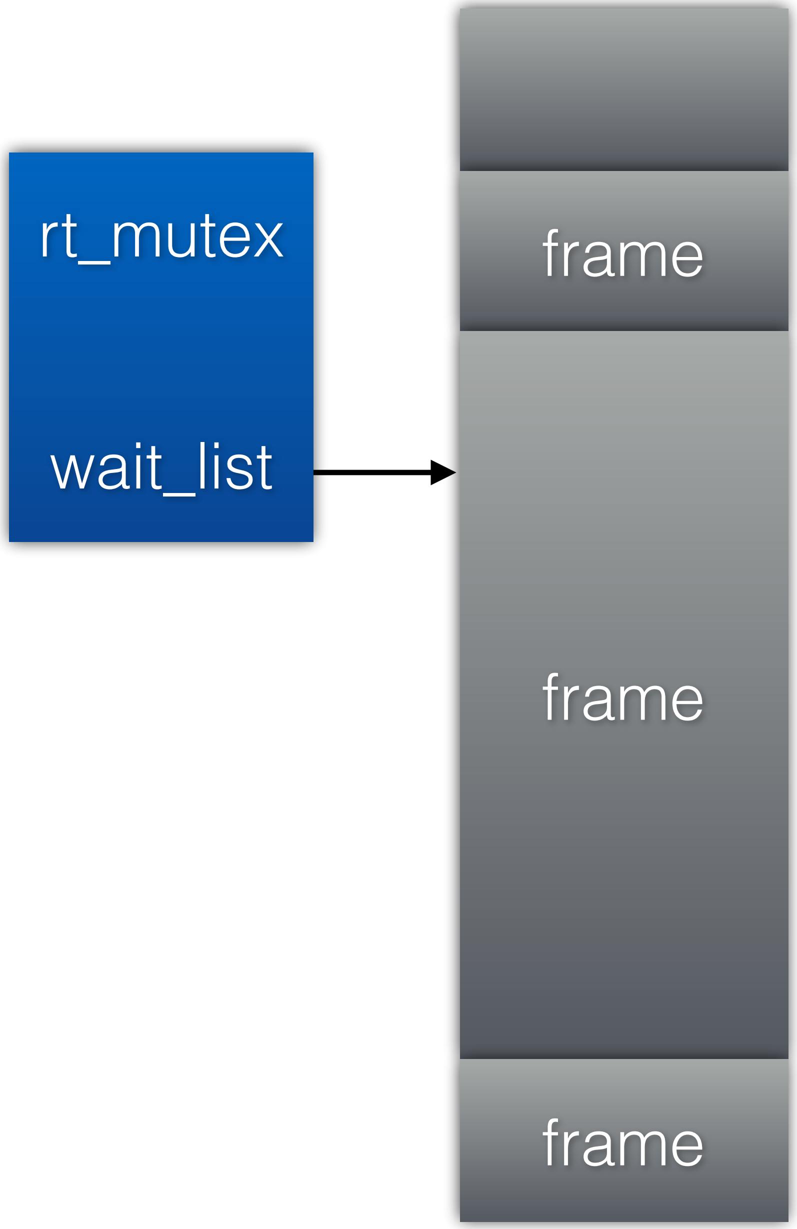
Stack Use-After-Free

- The thread wakes up, resuming execution
- It doesn't unlink the **rt_mutex_waiter** from the list
- Whatever happens to be on the kernel stack will be interpreted as an **rt_mutex_waiter**



Kernel Stack Manipulation

- Subsequent syscalls will use the same memory for stack frames
- Many syscalls place data from user-space on the stack, or data which is otherwise predictable
- By making some sequence of system calls, and performing futex operations, this can be exploited



Exploitation

- Technique that can work reliably without precise knowledge of the kernel stack
- Turn this vulnerability into two useful primitives, to allow leaking and arbitrary memory corruption

Getting Started

- After triggering the vulnerability, all sorts of things cause crashes, even exiting the program
- Kernel crashes can make development really painful
- “I HAVE NO TOOLS BECAUSE I’VE DESTROYED MY TOOLS WITH MY TOOLS”
- Finally managed to get crash dumps using a virtual serial port in VMware

Preparing the List

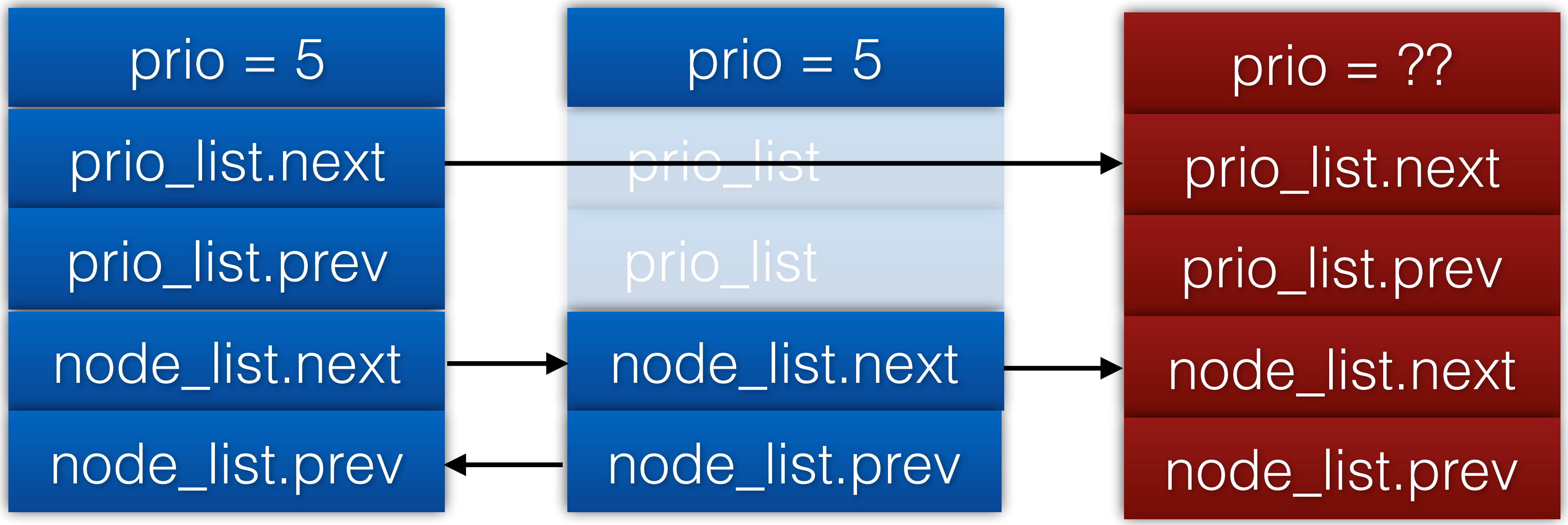
- In theory waiters can be added or removed from the corrupt list

- In practice, **rt_mutex_top_waiter** verifies the first item in the list and crashes all the time:
  BUG_ON(w->lock != lock)

- Need to insert nodes before the invalid node so that the list head is valid

- Use **nice** to order nodes, and **FUTEX_LOCK_PI** to add them to the **rt_mutex_waiter** list

plist



prio = 5
prio_list.next
prio_list.prev
node_list.next
node_list.prev

prio = 5
prio_list
prio_list
node_list.next
node_list.prev

prio = ??
prio_list.next
prio_list.prev
node_list.next
node_list.prev

```
struct plist_node {
   int prio;
   struct list_head prio_list;
   struct list_head node_list;
};
```

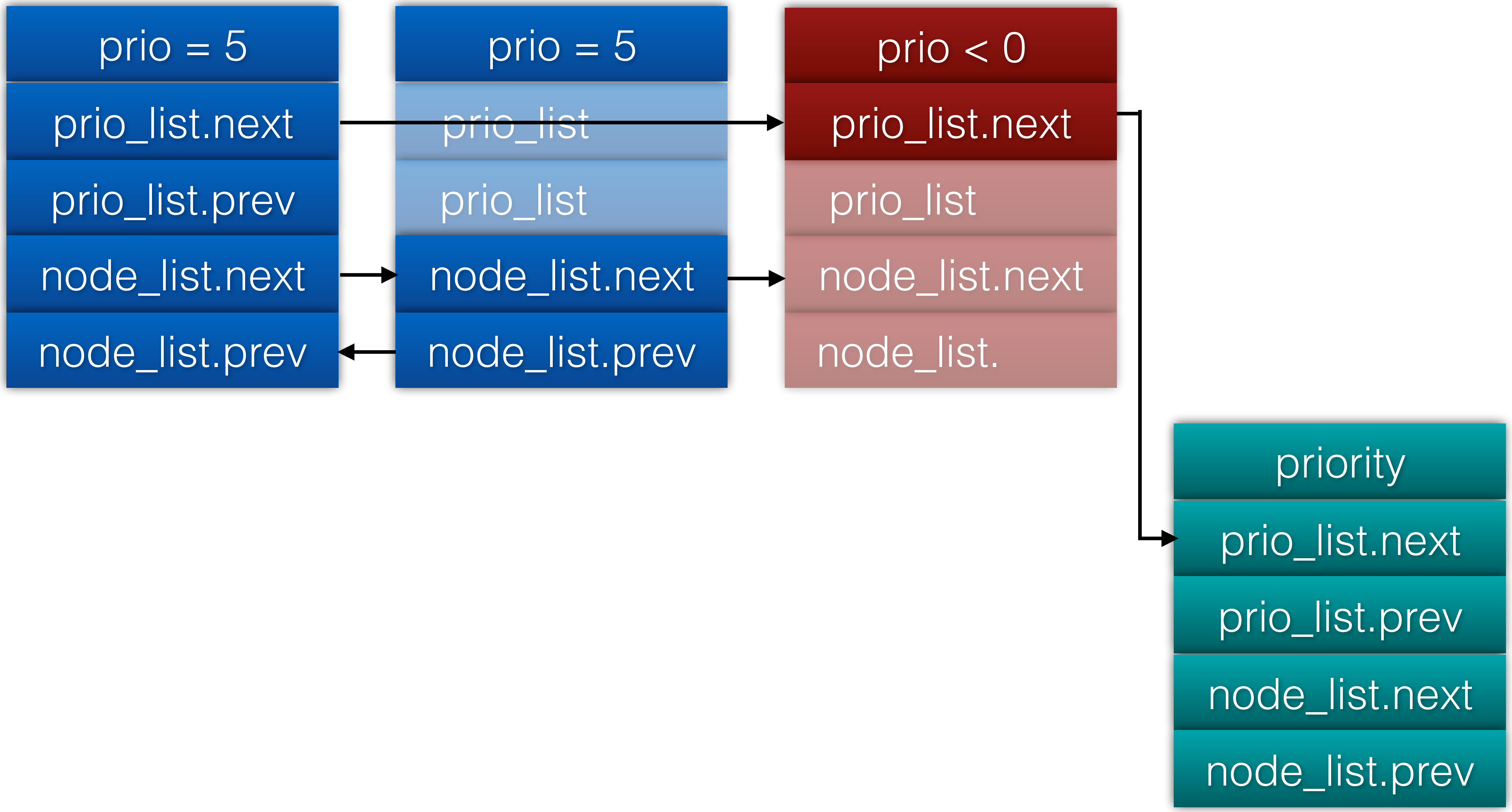
32-bit Linux Memory Split

- Kernel memory is 0xC0000000 and higher
- User memory is 0xBFFFFFFF and lower
- Kernel code can read and write user memory directly
- (Well, not *all* 32-bit Linux, but generally)

Manipulating the stack

- The kernel stack can be manipulated with system calls, for example **select** stores user controlled data on the stack
- Stack layout is unpredictable, unlike a use after free on the heap
- Fill the stack with a repeated value to overwrite both
- Use a value which is both a negative integer, and a user-space pointer (0x80000000 - 0xBFFFFFFF)
- The **prior** will be negative, and the **next** pointer will go to a fake user-space node

Manipulating the stack



prio = 5
prio_list.next
prio_list.prev
node_list.next
node_list.prev

prio = 5
prio_list
prio_list
node_list.next
node_list.prev

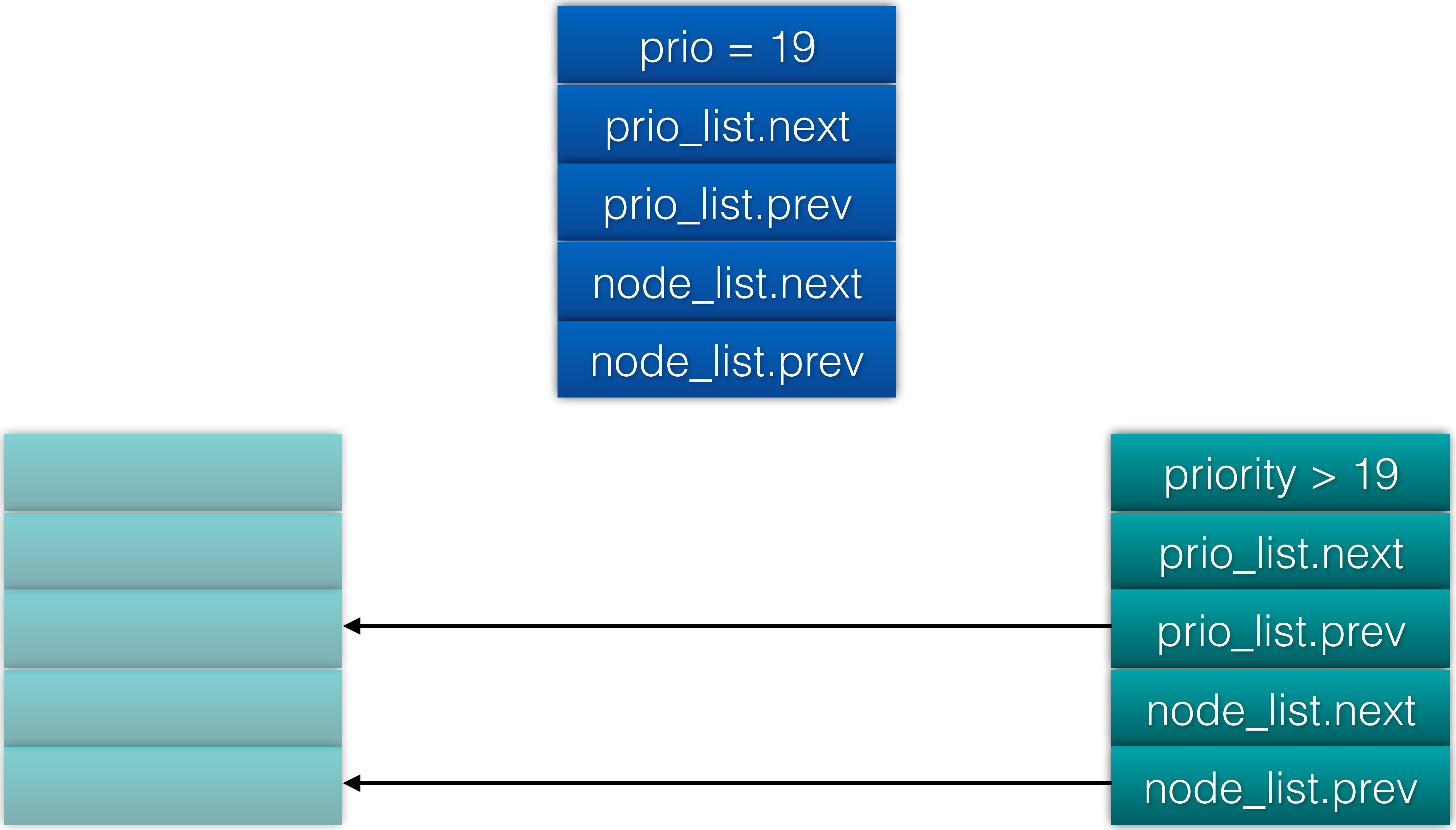
prio < 0
prio_list.next
prio_list
node_list.next
node_list.

priority
prio_list.next
prio_list.prev
node_list.next
node_list.prev

Node Insertion

- Priority list insertion first walks the **prio_list** until a higher priority value is found
- It then inserts the new node before that (using the **prev** pointers)
- By inserting a low priority node, it will traverse the “freed” node and be inserted before the user-space node

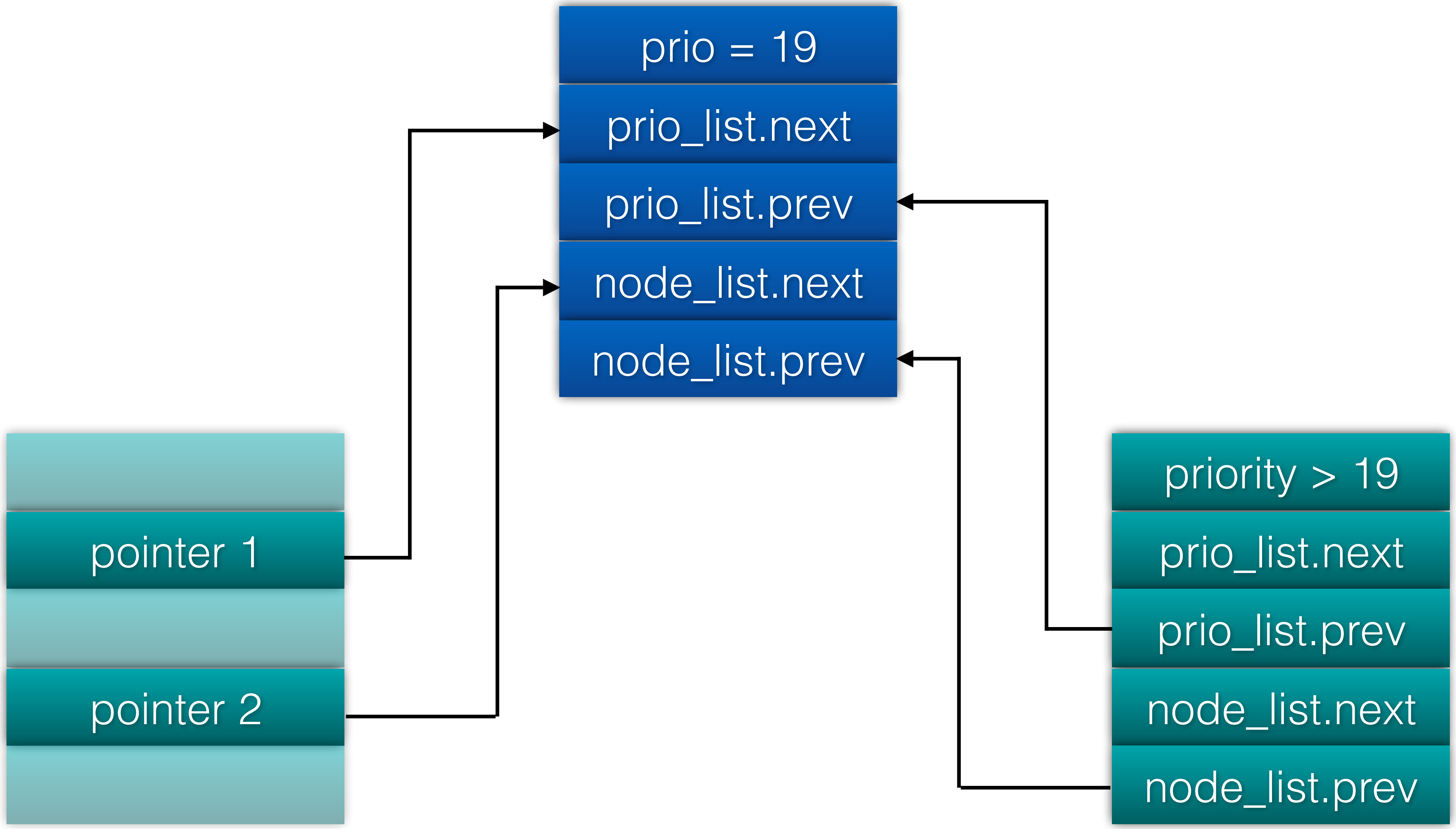
list_add_tail



prio = 19
prio_list.next
prio_list.prev
node_list.next
node_list.prev

priority > 19
prio_list.next
prio_list.prev
node_list.next
node_list.prev

list_add_tail



prio = 19
prio_list.next
prio_list.prev
node_list.next
node_list.prev

pointer 1
pointer 2

priority > 19
prio_list.next
prio_list.prev
node_list.next
node_list.prev

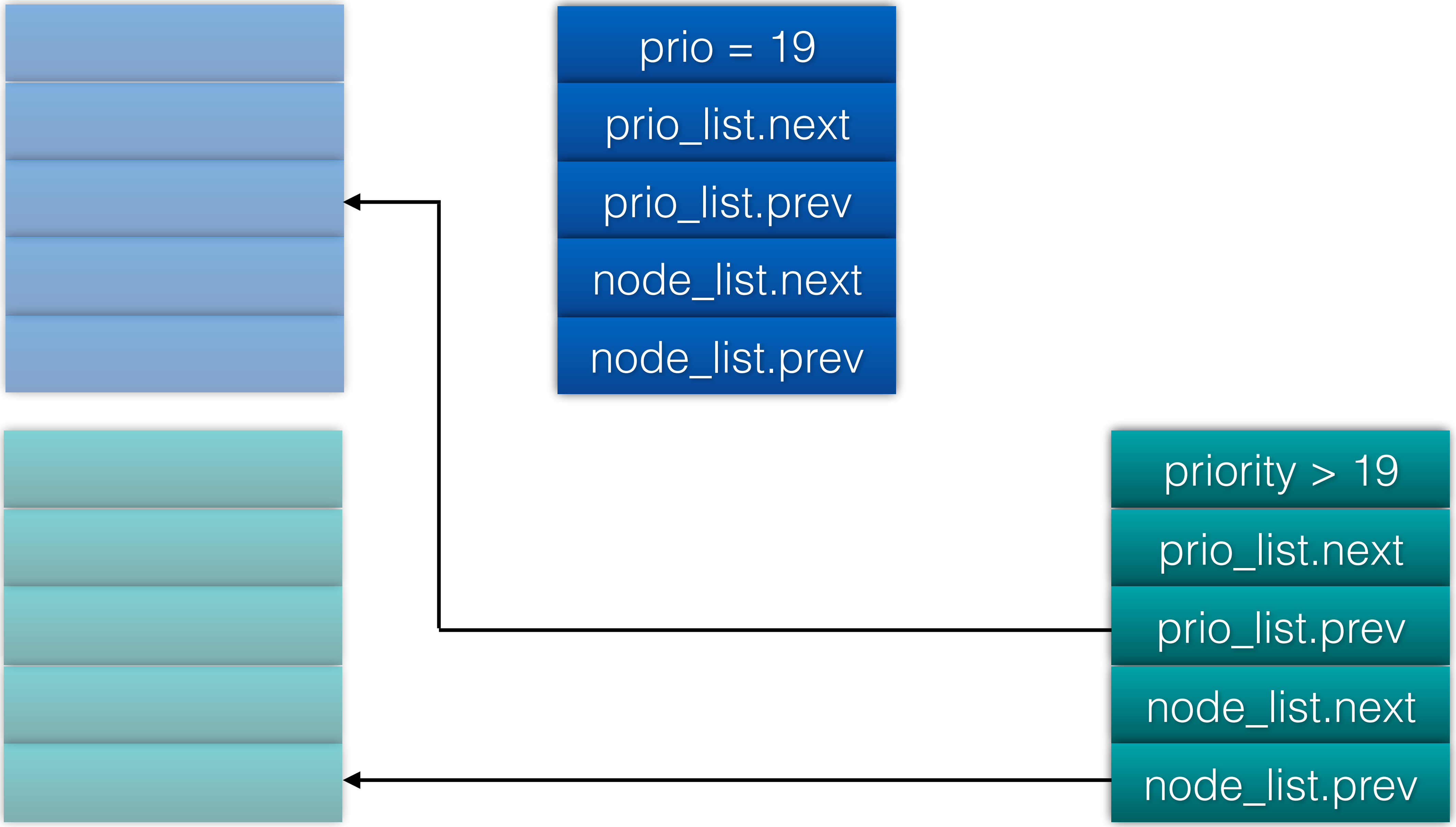
Information leak

- Populate the stack with a pointer to a user-space node (that doubles as a negative number)
- Insert a node (**FUTEX_LOCK_PI**) with a priority 19 so that it will be inserted adjacent to the user-space node
- Pointers to a kernel stack are written into user-space memory

Waking up the thread

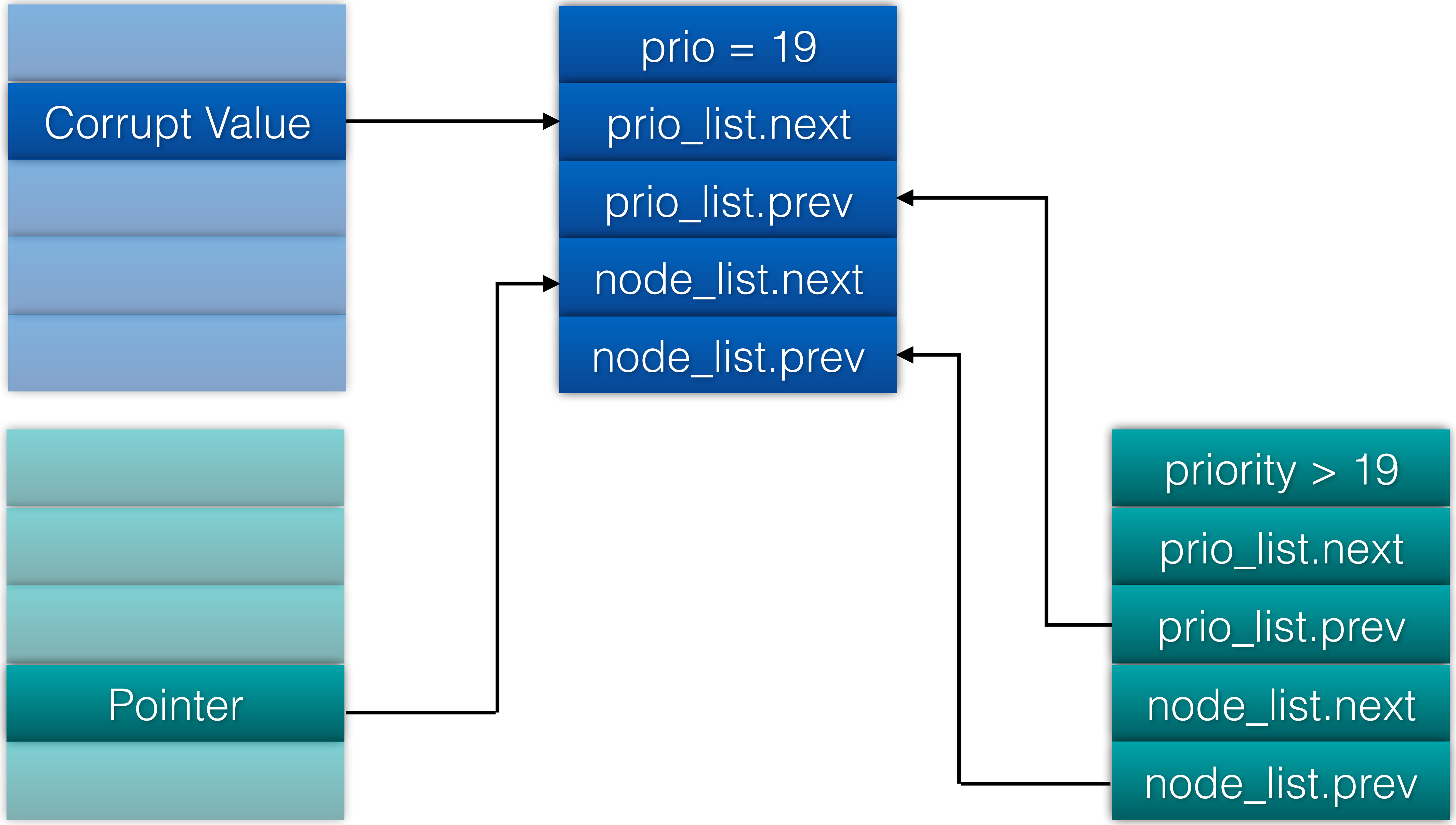
- The thread isn’t actually in the list, so it can’t be woken by unlocking the futex
- It will wake up if I send it a signal, though
- Register a handler for **SIGUSR1**
- Use **pthread_kill** to deliver the signal to the right thread
- The node will be unlinked and execution will resume

Corruption Primitive

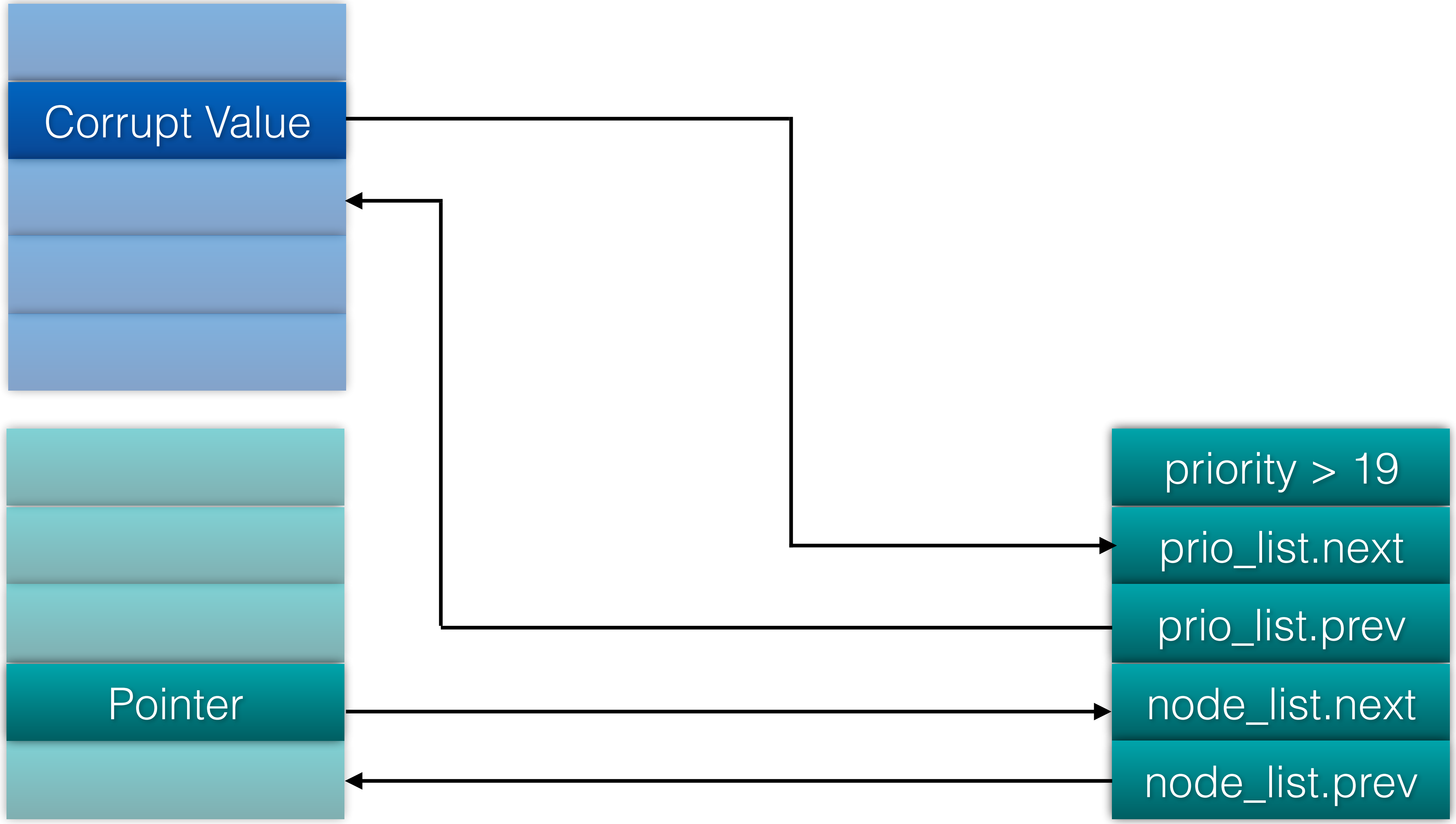

prio = 19
prio_list.next
prio_list.prev
node_list.next
node_list.prev

priority > 19
prio_list.next
prio_list.prev
node_list.next
node_list.prev

Corruption Primitive



Corruption Primitive


Corrupt Value

Pointer

priority > 19

prio_list.next

prio_list.prev

node_list.next

node_list.prev

Finishing the Exploit

- Use these primitives to bypass SMEP and PXN
- Get root
- Clean up kernel memory so the process doesn't crash at exit

SMEP / PXN

- First tried jumping to user space code
- Map the node as RWX and write the pointer over a return address on the stack
- Nope :(
- **Supervisor Mode Execution Prevention** stops user-space code from being executed on x86
- **Privileged Execute Never** is a funny name for exactly the same thing on ARM

addr_limit

- The **addr_limit** value is used by the kernel to validate user-space virtual addresses provided to system calls
- Its value is generally 0xc000000
- If the value is larger, then system calls will accept pointers to kernel memory
- Found in the **thread_info** structure at the top of each kernel stack

Unaligned Write

- Because the value I can write to kernel space is actually a user-space pointer, I can't write a value bigger than 0xC0000000
- Instead, write a value like 0xB000FFFF at offset 2 from the **addr_limit**
- This sets the value of **addr_limit** 0xFFFF0000

Arbitrary Read / Write

- Now we can use kernel-space addresses in system calls
- Use **pipe** to create a pair of file descriptors
- **write** to one then **read** from the other, using kernel-space and user-space addresses

Get Root

- Search the **task_struct** to find the credentials
- Set the uid/gid to zero
- Set the capability bits

Clean up

- Surprisingly hard
- Critically important - the VM still need to be rebooted every time I test something
- Iterate through the **rt_mutex_waiter** list fixing each node to point to the right place

DEMO

Thoughts

- Surprisingly complex problems in seemingly simple functionality

- Older mitigation bypasses still work

Thanks

- Dan Rosenberg and Fionnbharr Davies for helping
- Mark Dowd and John McDonald for giving me time to write this presentation

Questions?

azimuth
SECURITY