

CVE-2016-6187: Exploiting Linux kernel heap off-by-one

by Vitaly Nikolenko (<https://twitter.com/vnik5287>)

🕒 Posted on October 16, 2016 at 8:38 PM

Introduction

I guess the reason I decided to write about this vulnerability is because when I posted it on Twitter, I've received a few DMs saying that either this kernel path wasn't vulnerable (i.e., couldn't see where off-by-1 was) or it wasn't exploitable. The other reason is that I wanted to try the `userfaultfd()` syscall in practice and I needed a real UAF to play with.

First, I don't know if this vulnerability got into any upstream kernels on any major distributions. I've only checked the Ubuntu line and Yakkety wasn't affected. But hey, backports happen quite often :]. The bug was introduced by the [bb646cdb12e75d82258c2f2e7746d5952d3e321a](https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=bb646cdb12e75d82258c2f2e7746d5952d3e321a) (<https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=bb646cdb12e75d82258c2f2e7746d5952d3e321a>) commit and fixed in [30a46a4647fd1df9cf52e43bf467f0d9265096ca](https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=30a46a4647fd1df9cf52e43bf467f0d9265096ca) (<https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/commit/?id=30a46a4647fd1df9cf52e43bf467f0d9265096ca>).

Since I couldn't find a vulnerable Ubuntu kernel, I've compiled the 4.5.1 kernel on Ubuntu 16.04 (x86_64). It's worth mentioning that this vulnerability only affects distributions that use AppArmor by default (such as Ubuntu).

Vulnerability

Writing into `/proc/self/attr/current` triggers the `proc_pid_attr_write()` function. The following is the code before the vulnerability was introduced:

```
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);
    char *page;
    ssize_t length;
    struct task_struct *task = get_proc_task(inode);

    length = -ESRCH;
    if (!task)
        goto out_no_task;
    if (count > PAGE_SIZE) [1]
        count = PAGE_SIZE;

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

    length = -ENOMEM;
    page = (char*)__get_free_page(GFP_TEMPORARY); [2]
    if (!page)
        goto out;

    length = -EFAULT;
    if (copy_from_user(page, buf, count)) [3]
        goto out_free;

    /* Guard against adverse ptrace interaction */
    length = mutex_lock_interruptible(&task->signal->cred_guard_mutex);
    if (length < 0)
        goto out_free;

    length = security_setprocattr(task,
                                  (char*)file->f_path.dentry->d_name.name,
                                  (void*)page, count);

    ...
}
```

The buf parameter represents the user-supplied buffer (with length count) that's being written to /proc/self/attr/current. In [1], the check is performed to ensure that this buffer will fit into a single page (4096 bytes by default). In [2] and [3], a single page is allocated and the user-space buffer is copied into the newly allocated page. This page is then passed to security_setprocattr which represents the LSM hook (AppArmor, SELinux, Smack). In case of Ubuntu, this hook triggers apparmor_setprocattr() function shown below:

```
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') {                                [4]
        if (size == PAGE_SIZE)
            return -EINVAL;
        args[size] = '\0';
    }
    ...
}
```

In [4], if the last byte of the user-supplied buffer is not null and the size of the buffer is not equal to the page size, the buffer is terminated with a null. On the other hand, if the user-supplied buffer exceeds (or equal to) the size of a single page (allocated in [2]), the path is terminated and `-EINVAL` is returned.

The following shows the change (in [3]) to `proc_pid_attr_write()` *after* the vulnerability was introduced:

```

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);
    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;

    length = security_setprocattr(task,
                                  (char*)file->f_path.dentry->d_name.name,
                                  page, count);
    ...
}

```

Unlike `__get_free_page()`, `memdup_user()` allocates a block of memory specified by the `count` parameter and copies the user-supplied data into it. Hence, the size of the object being allocated is no longer restricted to 4096 bytes (even though that's still the maximum buffer size). Let's assume that the user-supplied data is 128 bytes in size and the last byte of this buffer is not null. When `apparmor_setprocattr()` is triggered, `args[128]` will be set to 0 because the check is still for `PAGE_SIZE` and not the actual size of the buffer:

```

    if (args[size - 1] != '\0') {
        if (size == PAGE_SIZE)
            return -EINVAL;
        args[size] = '\0';
    }

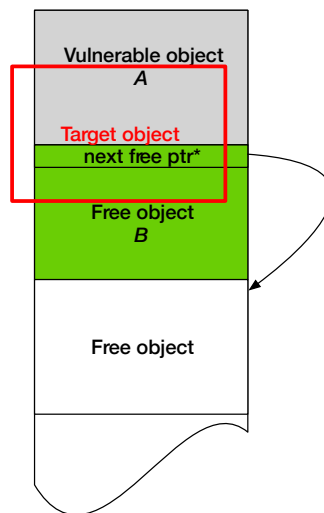
```

Since the objects are allocated dynamically on the heap, the first (least-significant byte) of the next object will be overwritten with a null. The standard technique for placing a target object (containing a function pointer as the first member) right after the vulnerable object won't work here. One idea was to overwrite a reference counter in some object (of the same size as the vulnerable object) and then trigger a UAF (thanks to Nicolas Trippeur (<https://twitter.com/ntrippeur>) for suggesting this). While on the subject of counter overflows, if you'll be at Ruxcon next week, check out my talk ([https://ruxcon.org.au/speakers/#Vitaly Nikolenko](https://ruxcon.org.au/speakers/#Vitaly%20Nikolenko)) on exploiting counter overflows in the kernel. Objects reference counters (represented by the `atomic_t` type = signed int) are generally the first members of the struct. Since counter values are typically under 255 for most objects, overwriting the least-significant byte of such an object would clear the counter and result in a standard UAF. However, to exploit this vulnerability, I've decided to go with a different approach: overwriting SLUB freelist pointers.

Exploitation

The nice thing about this vulnerability is that we control the size of the target object. To trigger the vulnerability, the object size should be set to one of the cache sizes (i.e., 8, 16, 32, 64, 96, etc.). We won't go into details on how the SLUB allocator (default kernel memory allocator on Linux) works. All we need to know is that (different) objects of the same size (in powers of 2) are accumulated into same caches for both general-purpose and special-purpose allocations. Slabs are basically pages in caches that contain objects of the same size. Free objects have a "next free" pointer at offset 0 (by default) pointing to the next free object in the slab.

The idea is to place our vulnerable object (*A*) next to a free object (*B*) in the same slab and then clear the least-significant byte of this "next free" pointer of object *B*. When two new objects are allocated in the same slab, the last object will be allocated over objects *A* and/or *B* depending the original value of the "next free" pointer:



The scenario above (overlapping both *A* and *B* objects) is just one of the possible outcomes. The "shift" value for the target object is 1 byte (0 to 255) and the final target object's position would depend on the original "next free" pointer value and the object size.

Assuming that the target object will overlap both objects *A* and *B*, we would like to control the contents of both of these objects.

At a high level, the exploitation procedure is as follows:

1. Place the vulnerable object *A* next to free object *B* in the same slab
2. Overwrite the least-significant byte of the "next free" pointer in *B*
3. Allocate two new objects in the same slab: the first object will be placed in *B* and the second object will represent our target object *C*
4. If we control the contents of objects *A* and *B*, we can force object *C* to be allocated in user space
5. Assuming object *C* has a function pointer that can be triggered from user space, set this pointer to our privilege escalation payload in user space or possibly a ROP chain (to bypass SMEP).

To perform steps 1-3, sequential object allocations can be achieved using a standard heap exhaustion technique.

Next, we need to pick the right object size. Objects that are larger than 128 bytes (i.e., kmalloc caches 256, 512, 1024, etc.) won't work here. Let's assume that the start slab address is `0x1000` (note that slab start addresses are aligned to the page size and sequential object allocations are contiguous). The following C program lists the allocations for a single page given the object size:

```
// page_align.c
#include

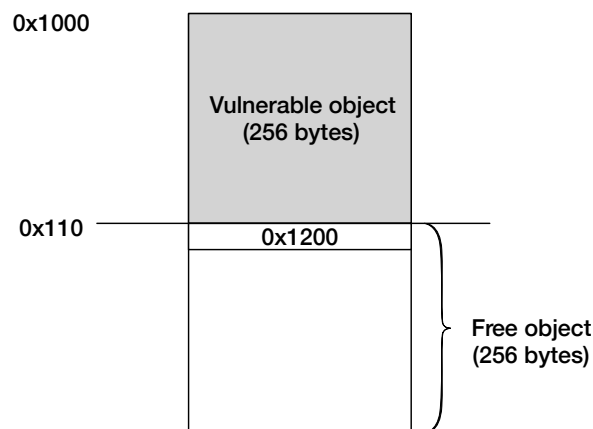
int main(int argc, char **argv) {
    int i;
    void *page_begin = 0x1000;

    for (i = 0; i < 0x1000; i += atoi(argv[1]))
        printf("%p\n", page_begin + i);
}
```

For objects that are 256 bytes (or > 128 and <= 256 bytes), we have the following pattern:

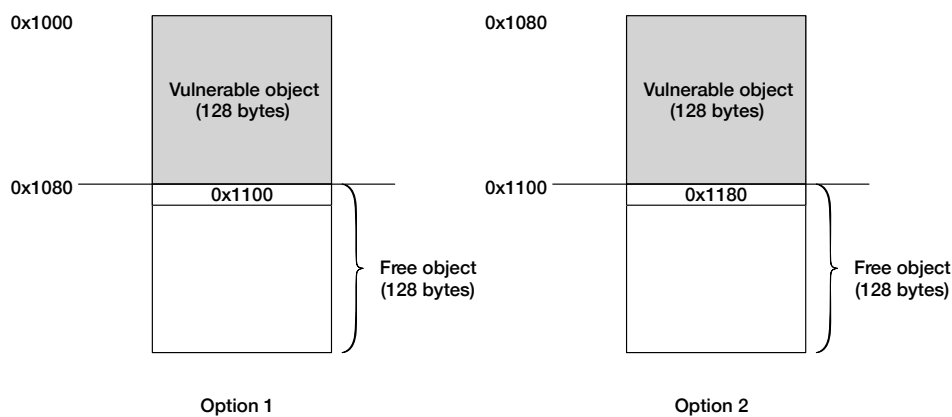
```
vnik@ubuntu:~$ ./align 256
0x1000
0x1100
0x1200
0x1300
0x1400
0x1500
0x1600
0x1700
0x1800
...
```

The least significant byte for all allocations in the slab is 0 and overwriting the "next free" pointer of the adjacent free object with a null will have no effect:



For the 128-byte cache, there're two possible options:

```
vnik@ubuntu:~$ ./align 128
0x1000
0x1080
0x1100
0x1180
0x1200
0x1280
0x1300
0x1380
0x1400
...
```



The first option is similar to the 256-byte example above (the least-significant byte of "next free" pointer is already 0). The second option is interesting because overwriting the least-significant byte of "next free" pointer will point it to the free object itself. Allocating some object (*A*) with the first 8 bytes set to some (fixed) user-space memory address, followed by the allocation of the target object (*B*), will place object *B* at the user-controlled memory address in user space. This is probably the best option both in terms of reliability and ease of exploitation:

1. There's a 50/50 chance of success. If it's the first option, there's no crash and we can try again.
2. Finding an object with some user-space address (first 8 bytes) that would be placed in the `kmaloc-128` cache is not that hard.

Despite this being the best approach, I've decided to go with 96-byte objects and glue it all together with `msgsnd()` heap exhaustion/spraying. The main (and only) reason for this is that I've already found a target object that I wanted to use and the size of that object happened to be 96 bytes. Thanks to Thomas Pollet (<https://twitter.com/Tohmaxx>) for helping find the right heap objects and automate this tedious process with `gdb/python` at runtime!

However, there're obvious downsides to using 96-byte objects; the main one is in exploit reliability. The idea is to exhaust the slabs (i.e., fill in the partial slabs) using the standard `msgget()` technique with 48-byte objects (the other 48 bytes are used for the message header). This will serve as a heap spray as well since we control a half (48 bytes) of the `msg` object. We also control the contents of the vulnerable object (data written to `/proc/self/attr/current` from user space). If the target object is allocated so that its first 8 bytes are overlapped with our controlled data, then the exploit will succeed. On the other hand, if these 8 bytes are overlapped with the `msg` header (that we don't control), this will result in a page fault but the kernel is likely to recover by itself. Based on my analysis, there're a couple of cases where the "next free" pointer would overlap with the random `msg` header of the previously allocated object.

There're some tricks to improve the reliability of the exploit however.

Target object

For the target object, I've used `struct subprocess_info` which is exactly 96 bytes in size. To trigger the allocation of this object, the following socket operation can be used with a random protocol family:

```
socket(22, AF_INET, 0);
```

Socket family 22 doesn't exist but module autoloading will still be triggered reaching the following function in the kernel:

```

int call_usermodehelper(char *path, char **argv, char **envp, int wait)
{
    struct subprocess_info *info;
    gfp_t gfp_mask = (wait == UMH_NO_WAIT) ? GFP_ATOMIC : GFP_KERNEL;

    info = call_usermodehelper_setup(path, argv, envp, gfp_mask,      [6]
                                   NULL, NULL, NULL);

    if (info == NULL)
        return -ENOMEM;

    return call_usermodehelper_exec(info, wait);                    [7]
}

```

`call_usermodehelper_setup` [6] will then allocate the object and initialise its fields:

```

struct subprocess_info *call_usermodehelper_setup(char *path, char **argv,
                                                char **envp, gfp_t gfp_mask,
                                                int (*init)(struct subprocess_info *info, struct cred *new),
                                                void (*cleanup)(struct subprocess_info *info),
                                                void *data)
{
    struct subprocess_info *sub_info;
    sub_info = kzalloc(sizeof(struct subprocess_info), gfp_mask);
    if (!sub_info)
        goto out;

    INIT_WORK(&sub_info->work, call_usermodehelper_exec_work);
    sub_info->path = path;
    sub_info->argv = argv;
    sub_info->envp = envp;

    sub_info->cleanup = cleanup;
    sub_info->init = init;
    sub_info->data = data;

out:
    return sub_info;
}

```

Once the object is initialised, it will be passed to `call_usermodehelper_exec` in [7]:

```

int call_usermodehelper_exec(struct subprocess_info *sub_info, int wait)
{
    DECLARE_COMPLETION_ONSTACK(done);
    int retval = 0;

    if (!sub_info->path) {                                         [8]
        call_usermodehelper_freeinfo(sub_info);
        return -EINVAL;
    }

    ...
}

```

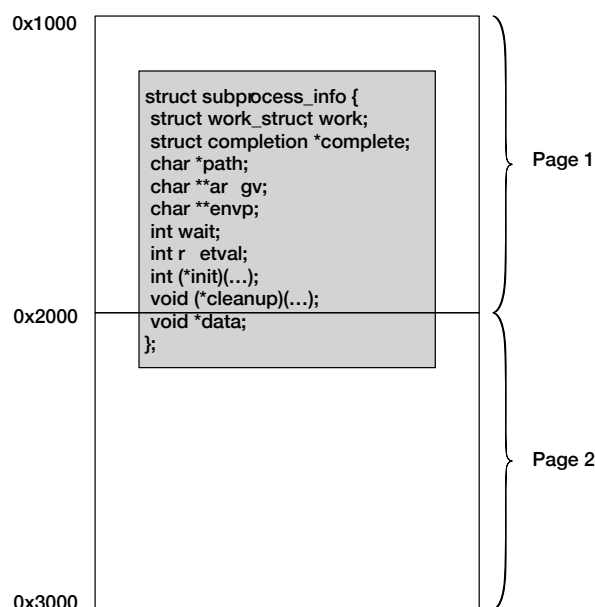
If the `path` variable is null [8], then the `cleanup` function is executed and the object is freed:


```
static void call_usermodehelper_freeinfo(struct subprocess_info *info)
{
    if (info->cleanup)
        (*info->cleanup)(info);
    kfree(info);
}
```

If we could overwrite the `cleanup` function pointer (remember that this object is now allocated in user space), then we'll have arbitrary code execution with `CPL=0`. The only problem is that `subprocess_info` object allocation and freeing happens on the same path. One way to modify the object's function pointer is to somehow suspend the execution before `info->cleanup(info)` gets called and set the function pointer to our privilege escalation payload. I could have found other objects of the same size with two "separate" paths for allocation and function triggering but I needed a reason to try `userfaultfd()` and the page splitting idea.

The `userfaultfd` syscall (<https://www.kernel.org/doc/Documentation/vm/userfaultfd.txt>) can be used to handle page faults in user space. We can allocate a page in user space and set up a handler (as a separate thread); when this page is accessed either for reading or writing, execution will be transferred to the user-space handler to deal with the page fault. There's nothing new here and this was mentioned by Jann Hornh (<https://bugs.chromium.org/p/project-zero/issues/detail?id=808>).

The SLUB allocator accesses the object (first 8 bytes to update the cache freelist pointer) before it's allocated. Hence, the idea is to split the `subprocess_info` object over two contiguous pages so that all object fields except say the last one (i.e., `void *data`) will be placed in the same page:



Then we would set up the user-space page fault handler to deal with PFs in the second page. When `call_usermodehelper_setup` gets to assigning `sub_info->data`, execution will be transferred to the user-space PF handler (where we can change previously assigned `sub_info->cleanup` function pointer). This approach would've worked if the target object was allocated with `kmalloc`. Unlike `kmalloc`, `kzalloc` uses `memset(..., 0, size(...))` to zero out the object after the allocation. Unlike `glibc`, kernel's `memset` implementation is pretty straightforward (i.e., setting single bytes sequentially):

```
void *memset(void *s, int c, size_t count)
{
    char *xs = s;

    while (count--)
        *xs++ = c;
    return s;
}
EXPORT_SYMBOL(memset);
```

This means that setting the user-space PF handler on the second page will no longer work because a PF will be triggered by `memset`. However, it's still possible to bypass this by chaining user-space page faults:

1. Allocate two consecutive pages, split the object over these two pages (as before) and set up the page handler for the second page.
2. When the user-space PF is triggered by `memset`, set up another user-space PF handler but for the first page.
3. The next user-space PF will be triggered when object variables (located in the first page) get initialised in `call_usermodehelper_setup`. At this point, set up another PF for the second page.
4. Finally, the last user-space PF handler can modify the `cleanup` function pointer (by setting it to our privilege escalation payload or a ROP chain) and set the `path` member to 0 (since these members are all located in the first page and already initialised).

Setting up user-space PF handlers for already "page-faulted" pages can be accomplished by `munmapping/mapping` these pages again and then passing them to `userfaultfd()`. The PoC for 4.5.1 can be found here ([/exploits/matreshka.c](#)). There's nothing specific to the kernel version though (it should work on all vulnerable kernels). There's no privilege escalation payload but the PoC will execute instructions at the user-space address `0xdeadbeef`.




Conclusion

There're possibly easier ways to exploit this vulnerability but I just wanted to make my found target object "work" with `userfaultfd`. Clean-up is missing but since we're allocating IPC msg objects, it's not very important and there're a few easy ways to fixate the system.

Articles (</feed.atom>)

- » [September 2018 \(1\)](#)
- » [October 2016 \(1\)](#)
- » [June 2016 \(2\)](#)
- » [January 2016 \(3\)](#)
- » [October 2015 \(1\)](#)
- » [July 2014 \(1\)](#)
- » [June 2014 \(3\)](#)

Contact

-  Follow me on Twitter (<https://twitter.com/vnik5287>)
-  Email ([mailto:vnik \[at\] cyseclabs.com](mailto:vnik[at]cyseclabs.com))
-  GPG Key ([/vnik.asc](#))

