CVE-2016-6187 Exploiting Linux kernel heap off-by-one 利用堆大小差一错误爆破 Linux 内核（下）

Target object

因为目标对象，我使用了 `struct subprocess_info` 结构，正是 96 字节大小。为了触发这个对象的分配，下面的套接字操作可以使用一个随机的协议家族：

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

套接字族 22 不存在但是模块自动加载会触发到内核中下面的函数：

```
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] 然后会分配对象和初始化它的字段：

```
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(⊂_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;
}
```

一旦对象被初始化，这将绕过 `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;
        }
...
}
```

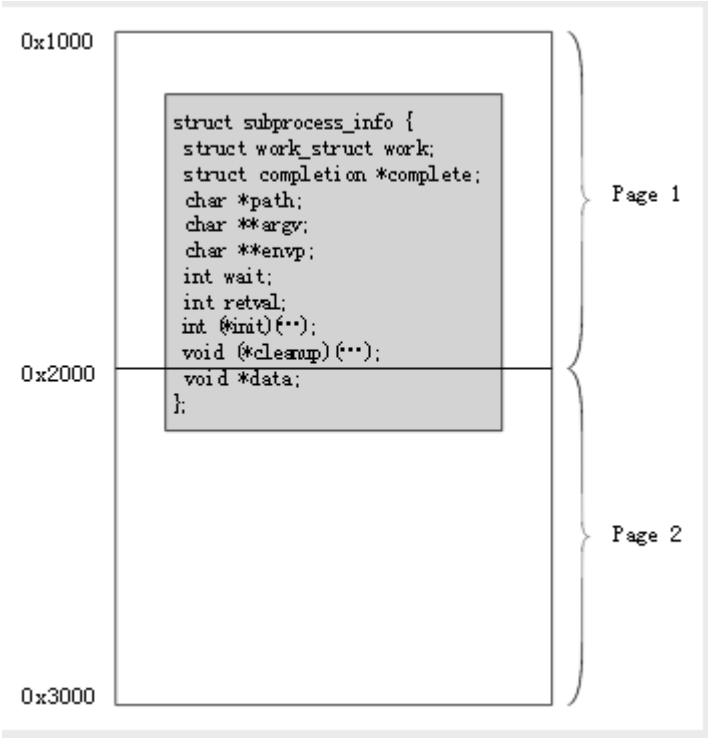
如果路径变量为 null[8]，然后 `cleanup` 函数被执行并且对象被释放：

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

如果我们覆盖了 `cleanup` 函数指针(记住对象现在在用户空间被分配),然后我们随着 CPL=0 就有了任意代码执行。仅有的一个问题是 `subprocess_info` 对象分配和释放在同样的路径。在 `info->cleanup)(info)` 被调用并且设置函数指针到我们的权限提升 payload 之前修改对象函数指针的一个方法是以某种方法停止执行。我本可以找到其他同样大小的因为分配和函数触发的两种“分开”路径，但是我需要一个理由去尝试 `userfaultfd()` 和这个页面分裂的想法。

Userfaultfd 系统调用能够被用来处理用户空间中的页面错误。我们可以在用户空间分配一个页面并且设置一个处理器（当做一个分线程）；当这个页面因为读或写被访问，执行会被转移到用户空间处理器去处理页面错误。这里没有新鲜的并且这是被 Jann Hornh 所提到的。

SLUB 分配器在被分配之前访问对象（首 8 个字节去更新缓存 freelist 指针）。因此，这个主意就是分离开 `subprocess_info` 对象到两个连续的页面以便所有对象字段除了说这最后一个（如 `void *data`）将会被放在同样的页：



然后我们会设置用户空间页面错误处理器去处理在第二页的 PF。当 `call_usermodehelper_setup` 去设定 `sub_info->data` ，代码被转移到用户空间 PF 处理器（在那里我们可以改变先前设定的 `sub_info->cleanup` 函数指针）。如果目标被 `kmalloc` 所分配，这个方法会起作用。不像 `kmalloc`，`kzalloc` 在分配之后使用 `memset(..., 0, size(...))` 归零对象。不想 glibc，内核的杂类函数实现是十分简洁直接的（例如设置连续化的单个字节）：

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

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

这意味着设置在第二页的用户空间 PF 处理器将不再起作用，因为一个 PF 将会被杂项函数触发。然而，这仍然有可能被束缚用户空间页面错误所绕过：

1. 分配两个连续页面，分割对象到这两个页面（如之前的）并且为第二个页面设置页面处理器。
2. 当用户空间 PF 被杂项函数触发，为第一页设置另一个用户空间 PF 处理器。
3. 当对象变量在 `call_usermodehelper_setup` 中初始化了，那么接下来的用户空间 PF 将会触发。这时候设置为第二个页面设置另一个 PF。
4. 最终，最后一个的用户空间 PF 处理器可以修改 `cleanup` 函数指针（通过设置它指向我们的权限提升 payload 或者 ROP 链）并且设置 `path` 成员为 0（因为这些成员在第一页被分配并且已经初始化了）。

因为“页面错误”的页面能够通过再次去除内存页映射/映射这些页实现，设置用户空间 PF 处理器。并且然后传递它们到 userfaultfd()。4.5.1 版本的 POC 能够在这里被找到。尽管对于内核版本没有什么特殊的（它应该可以工作在所有含有漏洞的内核）。这里没有权限提升 payload 但是这 POC 会在用户空间地址 `0xdeadbeef` 执行指令。

Conclusion

这有可能是更容易利用此漏洞的方法，但是我仅仅想我只想让我发现的目标对象随着 `userfaultfd` “工作”。清理机制缺失是因为我们是分配 IPC msg 对象，这不是非常重要并且有一些简单的方法稳固系统。

Update - 18/10/2016

Qihoo 360 反应他们的工作是独立的并且他们从来没有引用过公共资源。我想他们说的应该没错。

未完待续……

译者：rodster

校对：song

原文链接：**https://cyseclabs.com/blog/cve-2016-6187-heap-off-by-one-exploit**

原文作者：Vitaly Nikolenko