Fuzzing Testing Of Linux Kernel System Calls

Yunfan Li, Xiang Li, Zhicheng Fu
Oregon State University
Group 35

June 10, 2015

Abstract

The main purpose of this report is how to use fuzzing technique to find bugs of Linux kernel system calls. We use C programming language to implement a smart fuzzing. The fuzzing application can against Linux kernel system calls which will find system vulnerabilities. This report will introduce a brief overview of fuzzing history and definition. Then we will give the process of the fuzzing test as well. At the end of this report, we will explain the test results and analysis for the Linux kernel system call.

1 Introduction

1.1 The requirement of the assignment

Fuzzing is a popular methods to test the bugs in a system. In this assignment, we need to implement a fuzzing tool which is used to test the bugs of Linux system kernel. We plan to implement a smart fuzzing tool, which is written by C programming language. And we are going to test seven system calls in our fuzzer.

1.2 The introduction of our fuzzing

In our fuzzing test, we choose seven system calls of Linux, which are sys_umask, sys_chdir, sys_mkdir, sys_rmdir, sys_open, sys_creat, sys_getcwd. We make our fuzzer in smart method, which means that the fuzzer know which system calls are going to be tested. According to these specific calls, the fuzzer will send specific data to these calls and then check if the return value is the same as we predicate or there is some error information sent back. On the other hand, if the system gets serious trouble because of the incorrect input of the system calls, that means the system has bugs needed to be fixed. When testing the specific system call, we provided the data which should be incorrect for this specific call. This data should cover all incorrect type of data for the call.

1.3 Testing environment

We will use the Linux 3.14.24 version as the fuzzing test environment.

1.4 Bullete

To illustrate our understanding of fuzzing, the process of how do we implement our own fuzzing application and how do we use our own fuzzing to attack the Linux system, we organize the rest of this paper as follows:

- The Definition of Fuzzing
- Fuzzing Testing
- Result
- Conclusion

2 The Definition of Fuzzing

2.1 Definition of fuzzing

Fuzzing is a testing technique used to find vulnerabilities in applications or in systems.[4] Fuzzing could be described as black-box software testing technique.[5] It can detect abnormal results by providing a group of random

inputs to the target software. From these abnormal results which are generated from software, we can discover software vulnerabilities.

2.2 Brief history of fuzzing

The earliest fuzzing concept can be traced back 1988.[6] The professor Barton Miller at the University of Wisconsin first put forward fuzzing idea. In 1989, professor Barton Miller used an fuzzing in his advanced operating systems course to test the robustness of UNIX applications. About 1999, the University of Oulu start to build PROTOS test suite. With PORTOS mature in 2002 , Dave Aitel GPL agreement to publish an open source fuzzing device called SPIKE. From 2005 to now, fuzzing is booming in commercial space. Although fuzzy testing has so far achieved some results, it is still in the preliminary stage.[6]

2.3 Categories of fuzzing

Software testing is divided into black box, white-box and gray box testing. There are two kind of fuzz testing. The first one is general fuzzing. Mostly, general fuzzing is black box, also called automatic fuzzing. That is randomly generating the input parameters for an application. The input is disorganized and no logical. This fuzzing donot have to guess which data can lead to damage, but input as much of the clutter data into the program. And the second one is smart fuzzing. Whitebox fuzzing or smart fuzzing is a systematic methodology that is used to find buffer overruns (remote code execution); unhandled exceptions, read access violations (AVs), and thread hangs (permanent denial-of-service); leaks and memory spikes (temporary denial-of-service); and so forth.[1, 2]

2.4 What kinds of fuzzing we used

We use smart fuzzing as our fuzzing test tool. This kind of fuzzing recommend us analyzing the parameters of the system calls first and list the possibilities of cases that will cause the system crash. Although all the inputs of the Linux are random, these data also are to be consider potential threats that will cause the system to crash.

Next, we will discuss the thinking of how did we implement this fuzzing and how did we use this fuzzing to test the Linux kernel system calls.

3 Fuzzing Test

What we did in this assignment is we build an external Linux module whose name is fuzzing module, then load this external Linux module into the Linux running kernel. Then our fuzzing module will test 7 system calls automatically. In order to do this, we must firstly find how to make our code as a Linux kernel module, then need to consider how to achieve our thinking of fuzzing in kernel space. In this part we will essentially focus on how we approach to our goal step by step.

3.1 Linux module

A module in Linux is a mechanism that can be loaded or unloaded from Linux kernel on demand [3]. There are two kinds of modules, one is called modules living in source tree and another is called external modules. The fuzzing module cannot live in Linux source tree because it is not a functional module, once it started up, this module will consume a great deal of system resources. Also the fuzzing module will call system calls a bunch of times when it is running, such frequently interrupt in hardware will reduce the efficiency of system. So according to these two drawbacks, we think that we need to develop an external module so that we can decide when to load this fuzzing module in the running kernel. After it finished, we can unload it from the running kernel so that this part of code will not keep running and delay the system efficiency of response.

We need to include init.h, kernel.h and module.h for our program to make it as a module.

For a module program, at least two functions are needed as the requirement of Linux kernel module. They are init function and exit function. The format for each function is module_name_init and module_name_exit. In our program, there are also two functions we need to call as the entrance and exit for a module, they are module_init() and module_exit().

3.2 Linux syscalls in kernel space

As we already known that a syscall can be used through syscall() function in user space. But syscall() function has some of protection mechanism that can protect the system call from dirty arguments. So we need to avoid to use syscall() function and find a way that can call syscalls from kernel space. After research information from library and Internet, we found that there is a array in kernel that contains all pointers to each syscall, this array is called sys_call_table.

sys_call_table will live in a certain physical address when a kernel is running. And the physical address we can find it from a file locate at the root directory of Linux source tree which name is System.map. In this file, there are thousands of specific physical addresses that each of them point to a particular variable. Among all these variables, there is a variable named sys_call_table, if we use cat command, we will get the result. From the results we can see two important information, the first is the physical address for sys_call_table is 0xc17882c0, and the second is this variable and its physical address is only readable. The reason why it is only readable is because a bug happened on versions before 2.6, so from the version 2.6, Linux changed this variable to only readable. Through sys_call_table, we can get entries for each syscall and use the entry we get to call the syscalls from kernel space.

3.3 Syscalls analysis

In order to test system calls with fuzzing method, we need to firstly analyze the system calls. For example, we need to know how many input value the system calls need, what the input type is, what the meaning of the input value. After that we will give some incorrect data for the parameter of these system calls. If there would generate several errors or warnings, that means the system is steady. If the system directly core dump, that means the system is not steady enough and there are some bugs inside the system.

3.3.1 sys_umask()

sys_umask() is a system function which is used to set the umask of the current process. This system call has only one input data, whose type is int with a name of mask. The mechanism of this function is that let o777 do and (&) operation with mask by bit. Then let umask value equals to the result. When system considering umask, it only care about the last three bytes. That means the highest byte of int is useless is this case. This system call will return the value of the old umask value, which is also an int type data. When system check the input of sys_umask, there will be never an error. That means no matter what is given to this function, it will success forever. It is amazing but it could happen. The reason is that even you give data with a type of char, it will calculate umask value on bit level. And with the help of o777, it will cover the high bits.

Here is the declaration of sys_umask:

int sys_umask(int mask);

In old Linux version, this call is implemented by doing bit and operation with o777. But in recent Linux version, it does bit and operation will a macro, which is actually o777.

When we test it, we can give some random data and check if the system would generate an error. Actually, there should not be any error. We use system random data generation method to get random data and put it as the input of sys_umask.

3.3.2 sys_chdir()

sys_chdir is a system call which could change the current working path of a process. It is declared in the fs/open.c file. It only has one parameter in order to identify the new working path of current process. If it successes, it will return zero. Otherwise, it will return a negative value indicating a specific error. It has only two type of errors, which are ENOENT and ENOTDIR.

The declaration of this function is:

int sys_chdir(const char * filename);

When we test it with our fuzzing method, we will test it with some specific data. For example, we will test ".", which is treated as the path of current working path. ".."is the parent path of the current path of the process. On the os-server of our university, " "would be the userâĂŹs home path, but it is not a general symbol. So the compiler would not recognize it. More than this, we would test an input pathname which does not allow users to visit it. In this case, there should come an error.

3.3.3 sys_mkdir()

sys_mkdir() is a system function which is used to create a new directory in the given path. This function has two parameters, which are named as pathname and mode inside the function. The pathname is a const char pointer. That means this input value is a pointer to a string and cannot be changed in the function. The second parameter is an int type value, which is used to confirm the permission of the directory that this function is going to create. This mode value use the same mechanism as the second parameter if sys_creat. The return value of sys_mkdir is an int. If the call successes, it will return a zero. On the other hand, if this call fails, it will return a negative value. Actually, it will return different error flags, which indicate different problems, such as ENOENT, EPERM, ENOSPC, etc. With the different kind of incorrect input, this function will return the cooperating flags.

The function is declared as following:

int sys_mkdir(const char * pathname, int mode);

When we test it with our fuzzer, we will give some incorrect data as its input. For example, we could pass an int type of value as the input of the first parameter. This is similar to the test method of the first parameter of sys_creat. Additionally, there is something different. We might test a situation that the directory does not allow users to create new directories in the path which the pathname (first parameter of sys_mkdir) specifies.

When we consider the second parameter, we use the same method of testing the second parameter of sys_creat. That means this parameter will never fail. The value will be shadowed by current value of umask.

3.3.4 sys_rmdir()

sys_rmdir is a system call which is used to delete a directory. This function only has one parameter. This value is named as pathname in the function. And it indicates the path that the function is going to delete. The return value of sys_rmdir would be zero if the function successes. On the contrary, if the function fails, it will return a negative value, which could be negative EPERM, negative ENOENT, negative ENOTDIR or negative ENOTEMPTY. These are macros defined in include/errno.h. They represent 1, 2, 20, 39.

The declaration of this function is like:

int sys_rmdir(const char * pathname);

When we test it, we could pass a void pointer as the first parameter. Or other fuzzing methods similar to testing the first parameter of sys_creat. The difference is that we will test a state that there is no specific directory in the

described path.

3.3.5 sys_open()

sys_open() is a system call which is used to open an existed file in disk. Or it could create a new file. This system call has three input. The first one is given as the filename, which is in string type. But because C language does not have a type of string, it is given as a pointer of char. That means the compiler will identify the filename from the address that that pointer points, until the compiler meets a NULL in memory. The second parameter is designed as the purpose of open function. That means this parameter indicates that the file that is going to be opened is opened for writing or reading, and so on. The third parameter is for setting the mode of the file. It only works when the sys_open function is used to create a new file. The mode parameter will work as permission of a file with the current umask value. The return value of this system call is an int type value, which indicates the file descriptor of the opened file. This file descriptor pointing to the file that has been opened and mapped into the file system.

When we test this call, we will give several illegal inputs as its parameter. Or it is correct in syntax but meaningless semantically. When we consider the first parameter of sys_open(), which is a pointer pointing to a char value, we could give an int type value to check if the function could find the problem. More than this, we will put a point pointing to char there but points to an address which is meaningless or illegal. In this case, it is semantically meaningless and should cause an error of segmentation fault.

When we consider the second parameter of sys_open(), we find that this function will never be incorrect. It has similar mechanism as the parameter mask of sys_umask. When the data is given to sys_open, the second parameter, which is named as flag inside the function, will do and operation bit by bit. That means the useless bits will be covered. The macros this function uses as flag shadows are like O_WRONLY, O_RDONLY, O_CREAT, O_TRUNC, etc.

The third parameter is explained as the mode of the file if the file is creating by this sys_open function call. This parameter is a data in int type. But when it is used inside the function, it does and operation bit by bit. The useless bits will be covered by current umask value. In a conclusion, it will never fail.

Here is the declaration of sys_open():

int sys_open(const char * filename,int flag,int mode);

When we test this function, we firstly will test the first parameter with incorrect data in syntax. Then we test it with the data that is correct in syntax but meaningless semantically. Then we will test the first parameter which indicates just a string but not a file name. That means the first parameter filename does indicates a file. We will test that if the file path is not correct, what will happen. More than this, we will test the first parameter with a real path but there is no file named as the parameter. The function should check out the problems on the first parameter. Or it means that there are bugs in the system when it deal with the system call sys_open().

3.3.6 sys_creat()

sys_creat() is a system call which is used to create a new file. It has two parameters as input, which are filename and the mode. The first parameter which is char pointer, which is similar to the first parameter of sys_open. And the second the parameter mode is an int type value. This value the similar to the third parameter of the sys_open call. The reason that sys_creat is like sys_open is that the sys_creat function actually calls the sys_open. That means the sys_creat function uses the sys_open directly. The difference is that when

sys_creat calls the sys_open, it gives a parameter of O_CREAT and O_TRUNC as the second parameter of sys_open. And give the first parameter of sys_creat itself to sys_open as the first parameter. And sys_creat passes its second parameter as the third parameter of sys_open.

The recent version of Linux uses some macros for doing and operation when it calculates mode. For example, there are some macros like S_ISUID, S_ISGID, S_ISVTX, etc.

Sys_creat() is declaration as following state:

int sys_creat(const char * pathname, int mode);

When testing this call, for the first parameter we use a similar method like the first parameter of sys_open. And we test the second input value of sys_creat with the similar as the third parameter of sys_open. For example, we would put a pointer pointing to an int type of data as the first parameter of sys_creat. Or we would pass a void pointer to this state and check if the system could throw out an error instead of coring dump. WhatâĂŹs more, we could pass a char value as the second parameter of sys_creat. And actually that parameter will never fail.

3.3.7 sys_getcwd()

sys_getcwd() is a system function that will get the absolute path of current working path in the process. This function has two parameter. They are identified as the buffer which is used to store the absolute path and the maximum size that buffer could allow to store. The first parameter is used to store the path. The second is a unsigned long type of value. If the input for the second parameter is not unsigned long, system or compiler will automatically transfers it to a type of unsigned long.

This function is a little different from the functions we just discussed. The first parameter, which is a pointer pointing to a string is both input and output. That means the buffer will be useful after the function is called. When we test it, we should check the result of the buffer.

The return value is a pointer pointing to the buffer which is storing the path. If it fails to get the current working path, it will return a NULL as the pointer pointed and set errno to ERANGE.

The function is defined as the following format:

char * sys_getcwd (char char *buf, unsigned long size);

When we test it with fuzzing method, we will pass some incorrect data. For example, we will give a void pointer or a pointer pointing to some meaningless address as the first parameter. More than this, we will pass a pointer pointing to a buffer which is less than the path need. When we consider the second input value, we will give a number of size which is less that the path expects. For example, if the path has a length of ten characters. But the size number is five. And in this case, we test sys_getcwd with a buffer longer than ten and another buffer less than ten.

3.4 Random in kernel space

In kernel space, we cannot use rand() function which we usually used in user space. But Linux kernel provide us a significant function which name is get_random_bytes() that can generate random numbers for every byte. Besides of random numbers, we also need a function that can generate random strings. Unfortunately, we found that we need to write such function by ourselves.

The algorithm for the random number function which name is rand_num(int start, int end, int flag) is that use get_random_bytes() to get a random number and then use mod operation to get a number locates in a specific range. We set a temporary variable i, and pass it as the first argument to the get_random_bytes()

. And give positive 2 to the second argument of the get_random_bytes(). These two parameters will lead they get_random_bytes() function generate a random number with 2 bytes, which is -32767 to 32767. This random number will be stored in variable i. Then variable i will be used to mod the difference between a start number and a end number, these two numbers will be passed in our rand_num() function as the first and second parameter. For the third parameter, we use this parameter for this function to identify whether this function need to keep the negative numbers. Because the variable i maybe a negative number, so the result could be a negative number. But in particular situation, we just need positive numbers in program, so the third parameter is necessary to distinguish whether we need negative numbers or not.

The algorithm for the random string function which name is rand_string(char *s, int size, int flag) is that randomly pick one of character from an array named text and do the same operation several times to produce a random string. There are 54 characters in the text array. They are alphabets which contains 52 characters, "/"and ".". Two of special characters are used to constitute legal directory and legal filename. The third parameter is used to confirm how many slashes are needed in this random string. If it is zero, that means this string is a pure random string.

4 Result

After load the fuzzing module in a running kernel, a series of result will be printed out on the screen. These are the fuzzing test results. All the calls of sys_umask are successful, that is because the parameters that give to sys_umask will be logically anded with a o777 to prevent sys_umask from dirty data. All the calls of sys_chdir are failed with errno 2, which represents no such file or directory.For the calls of sys_mkdir, those of which directory is exist will always success no matter what mode it received. This situation is the same as sys_umask, because kernel will use the mode parameter logically and with another number to prevent itself directly receive a dirty data. All of the calls of sys_rmdir are failed with errno 2, because kernel cannot find a legal path. Most the calls of sys_open are failed, but serveral of them failed with errno 22 and the rest of them failed with errno 2, 22 means that this is an invalid argument. A few all calls of sys_open are success because the mode can be logically anded with another number, follows the same principle with sys_umask or sys_mkdir. All of the calls of sys_creat are successful because the same reason with sys_umask. For sys_getcwd, it will always success on condition that the char pointer is not NULL. And it will always failed with errno 14, which represents bad memory address if the char pointer is NULL.

Table 1: *Fuzz Testing Results*

System Call	Fail Condition
sys_umask	Always Success (No crashes)
sys_chdir	Always Fail (ERRNO 2)
sys_mkdir	Always Success (No crashes)
sys_rmdir	Always Fail (ERRNO 2)
sys_open	If flags or mode are error (ERRNO 22) or filename doesn't exits (ERROR 2)
sys_creat	Always Success (No crashes)
sys_getcwd	If the first parameter is NULL (ERRNO 14)

ERRNO 2: No such file or directory
ERRNO 14: Bad address
ERRNO 22: Invalid arguments

5 Conclusion

Fuzzing, a software vulnerability test, has became an increasingly important software testing technology. The process of fuzzing test is to input random set of parameters to the objective software. Since these random data and code do not need to meed the context of the logic of target software, software testing become simple. All in all, fuzzing is an important tool for verifying the robust of a process. The study of fuzzing has a thriving a prospect.

5.1 Conclusion of our fuzzing

In this assignment, we use a fuzzing tool which is written by ourselves testing six different kinds of system calls. We find that through randomly generating input parameters for seven system calls, Linux 3.14.24 didnâĂŹt appear crash or blocking circumstance. More than just this, the kernel also will give several clear log information or return value to what error has generated so that we can use those information to debug our code. Hence, the version of Linux 3.14.24 has a strong robustness.

5.2 What we learnt from this assignment

For this assignment, our group members have learn the concept of the fuzzing test. We already learned a brief history of the origin of fuzzing and how did we start our own fuzzing. We wrote our own fuzzing as an external Linux kernel module and send random parameters to the tested system calls. Through analyzing on the results, Detecting the loopholes in the Linux system. This project allows us a deep understanding of the operating mechanism of fuzzing test and the importance of fuzzing for testing the vulnerabilities in software. Fuzzing is an indispensable tool for programmers to detect systemsâĂŹ reliability and safety.

References

[1] J. Demott, "The evolving art of fuzzing," *DEF CON*, 2006.

[2] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *ACM Sigplan Notices*, vol. 43, no. 6. ACM, 2008, pp. 206–215.

[3] R. Love, *Linux kernel development*. Pearson Education, 2010.

[4] P. A. Michael Sutton, Adam Green, *FUZZING: Brute Force Vulnerability Discovery*, 1st ed. Wesley Professional, July 2007.

[5] N. Rathaus and G. Evron, *Open source fuzzing tools*. Syngress, 2011.

[6] A. Takanen, "Fuzzing: the past, the present and the future," *SSTIC'09*, 2009.

Appendix I: C Source code – fuzzing.c

```c
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/unistd.h>
#include <linux/random.h>
#include <linux/fs.h>
#include <linux/mm.h>
#include <asm/uaccess.h>

//fuzzing loop times
#define _NUM 50

//Store the 0xc17882c0 in this pointer as the sys_call_table
void **sys_call_table;
mm_segment_t old_fs;

void fuzzing_test_umask(void);
void fuzzing_test_chdir(void);
void fuzzing_test_mkdir(void);
void fuzzing_test_rmdir(void);
void fuzzing_test_open(void);
void fuzzing_test_create(void);
void fuzzing_test_getcwd(void);
int rand_num(int s, int e, int flag);
void rand_string(char *des, int size, int flag);

//Define syscalls
asmlinkage int (*umask)(int);
asmlinkage int (*chdir)(char *);
asmlinkage int (*mkdir)(char *, int);
asmlinkage int (*rmdir)(char *);
asmlinkage int (*open)(char *, int, int);
asmlinkage int (*create)(char *, int);
asmlinkage int (*getcwd)(char *, int);

static int fuzzing_init(void)
{
        printk(KERN_ALERT "####################Fuzzing is running
            !####################\nPlease wait a moment.\n\n\n");

        //Get sys_call_table from c17882c0 which is live in System.map
        sys_call_table = (void *)0xc17882c0;

        old_fs = get_fs();
        set_fs(KERNEL_DS);

        //Call every fuzzing functon
        fuzzing_test_umask();
        fuzzing_test_chdir();
        fuzzing_test_mkdir();
        fuzzing_test_rmdir();
        fuzzing_test_open();
        fuzzing_test_create();
        fuzzing_test_getcwd();

        return 0;
}
```

```
static void fuzzing_exit(void)
{
        set_fs(old_fs);
        printk(KERN_ALERT "####################Fuzzing has done
            !####################\n");
}

void fuzzing_test_umask(void)
{
        int i = 0;

        //Get umask from sys_call_table
        umask = (void *)sys_call_table[__NR_umask];

        printk(KERN_ALERT "--------------------Start fuzzing test: sys_umask
            .--------------------\n");

        //Negative numbers
        printk(KERN_ALERT "Test with negative numbers\n");
        for(i = 0; i < _NUM; i++) {
                int m = rand_num(-32000, -1, 0);
                int v = 0;
                v = umask(m);
                printk(KERN_ALERT "%d's m is %d\t v is %d.\n", i, m, v);
        }

        //The numbers which bigger than o777
        printk(KERN_ALERT "\nTest with numbers bigger than o777\n");
        for(i = 0; i < _NUM; i++) {
                int m = rand_num(512, 32000, 1);
                int v = 0;
                v = umask(m);
                printk(KERN_ALERT "%d's m is %d\t v is %d.\n", i, m, v);
        }

        printk(KERN_ALERT "--------------------Done fuzzing test: sys_umask
            .--------------------\n\n\n");
}

void fuzzing_test_chdir(void)
{
        int i = 0;

        //Get chdir from sys_call_table
        chdir = (void *)sys_call_table[__NR_chdir];

        printk(KERN_ALERT "--------------------Start fuzzing test: sys_chdir
            .--------------------\n");

        //Pure random string
        printk(KERN_ALERT "Test with pure random string\n");
        for(i = 0; i < _NUM; i++) {
                char dir[50] = {'\0'};
                int v = 0;
                rand_string(dir, 50, 0);
                v = chdir(dir);
                printk(KERN_ALERT "%d's dir is: %s\nsys_chdir return value is %d\
                    n", i, dir, v);
        }

        //Generate string looks like a legal directory string
```

```
        printk(KERN_ALERT "\nTest with a string looks like a legal directory\n");
        for(i = 0; i < _NUM; i++) {
                char dir[50] = {'\0'};
                int v = 0;
                rand_string(dir, 50, rand_num(1,7,1));
                v = chdir(dir);
                printk(KERN_ALERT "%d's dir is: %s\nsys_chdir return value is %d\
                    n", i, dir, v);
        }

        printk(KERN_ALERT "----------------------Done fuzzint test: sys_chdir
            .----------------------\n\n\n");
}

void fuzzing_test_mkdir(void)
{
        int i = 0;
        char *testdir = "mkdir_test";

        //Get mkdir from sys_call_table
        mkdir = (void *)sys_call_table[__NR_mkdir];

        //Generate testdir and change into it
        umask(0);
        mkdir(testdir, 0644);
        chdir(testdir);

        printk(KERN_ALERT "----------------------Start fuzzing test: sys_mkdir
            .----------------------\n");

        //Pure random string
        printk(KERN_ALERT "Test with pure random string\n");
        for(i = 0; i < _NUM; i++) {
                char dir[50] = {'\0'};
                int v = 0;
                int mode = rand_num(-32000,-1,0);
                rand_string(dir, 50, 0);
                v = mkdir(dir,mode);
                printk(KERN_ALERT "%d's dir is: %s, mode is: %d\nsys_mkdir return
                     value is %d\n", i, dir, mode, v);
        }

        //Generate string looks like a legal directory string
        printk(KERN_ALERT "\nTest with a string looks like a legal directory\n");
        for(i = 0; i < _NUM; i++) {
                char dir[50] = {'\0'};
                int v = 0;
                int mode = rand_num(1,32000,1);
                rand_string(dir, 50, rand_num(1,7,1));
                v = mkdir(dir,mode);
                printk(KERN_ALERT "%d's dir is: %s, mode is: %d\nsys_mkdir return
                     value is %d\n", i, dir, mode, v);
        }

        chdir("..");

        printk(KERN_ALERT "----------------------Done fuzzint test: sys_mkdir
            .----------------------\n\n\n");
}

void fuzzing_test_rmdir(void)
```

```
{
        int i = 0;

        //Get rmdir from sys_call_table
        rmdir = (void *)sys_call_table[__NR_rmdir];

        printk(KERN_ALERT "---------------------Start fuzzing test: sys_rmdir
            .--------------------\n");

        //Pure random string
        printk(KERN_ALERT "Test with pure random string\n");
        for(i = 0; i < _NUM; i++) {
                char dir[50] = {'\0'};
                int v = 0;
                rand_string(dir, 50, 0);
                v = rmdir(dir);
                printk(KERN_ALERT "%d's dir is: %s\nsys_rmdir return value is %d\
                    n", i, dir, v);
        }

        //Generate string looks like a legal directory string
        printk(KERN_ALERT "\nTest with a string looks like a legal directory\n");
        for(i = 0; i < _NUM; i++) {
                char dir[50] = {'\0'};
                int v = 0;
                rand_string(dir, 50, rand_num(1,7,1));
                v = rmdir(dir);
                printk(KERN_ALERT "%d's dir is: %s\nsys_rmdir return value is %d\
                    n", i, dir, v);
        }

        printk(KERN_ALERT "---------------------Done fuzzint test: sys_rmdir
            .--------------------\n\n\n");
}

void fuzzing_test_open(void)
{
        int i = 0;
        char *testdir = "open_test";

        //Get open from sys_call_table
        open = (void *)sys_call_table[__NR_open];

        //Generate testdir and change into it
        umask(0);
        mkdir(testdir, 0644);
        chdir(testdir);

        printk(KERN_ALERT "---------------------Start fuzzing test: sys_open
            .--------------------\n");

        //Pure random string, negative flags and negative mode
        printk(KERN_ALERT "Test with random filename, negative flags and negative
             mode\n");
        for(i = 0; i < _NUM; i++) {
                char filename[50] = {'\0'};
                int flags = rand_num(-32000,-1,0);
                int mode = rand_num(-32000,-1,0);
                int v = 0;
                rand_string(filename, 50, 0);
                v = open(filename, flags, mode);
```

```
                printk(KERN_ALERT "%d's filename is: %s\tflags is: %d\tmode is: %
                    d\nsys_open return value is %d\n", i, filename, flags, mode,
                    v);
        }

        //Pure random string, flags and mode look like legal
        printk(KERN_ALERT "\nTest with random filename, flags and mode look like
            normal number but semantically incorrect\n");
        for(i = 0; i < _NUM; i++) {
                char filename[50] = {'\0'};
                int flags = rand_num(1,32000,0);
                int mode = rand_num(1,32000,0);
                int v = 0;
                rand_string(filename, 50, 0);
                v = open(filename, flags, mode);
                printk(KERN_ALERT "%d's filename is: %s\tflags is: %d\tmode is: %
                    d\nsys_open return value is %d\n", i, filename, flags, mode,
                    v);
        }

        chdir("..");

        printk(KERN_ALERT "---------------------Done fuzzint test: sys_open
            .---------------------\n\n\n");
}

void fuzzing_test_create(void)
{
        int i = 0;
        char *testdir = "creat_test";

        //Generate testdir and change into it
        umask(0);
        mkdir(testdir, 0644);
        chdir(testdir);

        //Get create from sys_call_table
        create = (void *)sys_call_table[__NR_creat];

        printk(KERN_ALERT "---------------------Start fuzzing test: sys_create
            .---------------------\n");

        //Pure random string, negative mode
        printk(KERN_ALERT "Test with random filename and negative mode\n");
        for(i = 0; i < _NUM; i++) {
                char filename[50] = {'\0'};
                int mode = rand_num(-32000, -1, 0);
                int v = 0;
                rand_string(filename, 50, 0);
                v = create(filename, mode);
                printk(KERN_ALERT "%d's filename is: %s\tmode is: %d\nsys_creat
                    return value is %d\n", i, filename, mode, v);
        }

        //Pure random string, mode looks like legal
        printk(KERN_ALERT "\nTest with random filename and some maybe legal mode\
            n");
        for(i = 0; i < _NUM; i++) {
                char filename[50] = {'\0'};
                int mode = rand_num(1, 32000, 1);
                int v = 0;
```

```
                rand_string(filename, 50, 0);
                printk(KERN_ALERT "%d's filename is: %s \t mode is %d\nsys_creat
                    return value is %d\n", i, filename, mode, v);
        }

        chdir("..");

        printk(KERN_ALERT "---------------------Done fuzzint test: sys_create
            .---------------------\n\n\n");
}

void fuzzing_test_getcwd(void)
{
        int i = 0;

        //Get chdir from sys_call_table
        getcwd = (void *)sys_call_table[__NR_getcwd];

        printk(KERN_ALERT "---------------------Start fuzzing test: sys_getcwd
            .---------------------\n");

        //Pure random string, negative size
        printk(KERN_ALERT "Test with random string and negative size\n");
        for(i = 0; i < _NUM; i++) {
                char dir[50] = {'\0'};
                int size = rand_num(-32000,-1,0);
                int v = 0;
                rand_string(dir, 50, 0);
                v = getcwd(dir, size);
                printk(KERN_ALERT "%d's dir is %s\tsize is %d\nsys_getcwd return
                    value is %d\n", i, dir, size, v);
        }

        //NULL pointer, positive size
        printk(KERN_ALERT "\nTest with NULL pointer and positive size\n");
        for(i = 0; i < _NUM; i++) {
                char *dir = NULL;
                int size = rand_num(1,32767,1);
                int v = getcwd(dir, size);
                printk(KERN_ALERT "%d's dir is %s\tsize is %d\nsys_getcwd return
                    value is %d\n", i, dir, size, v);
        }

        printk(KERN_ALERT "---------------------Done fuzzint test: sys_getcwd
            .---------------------\n\n\n");
}

int rand_num(int s, int e, int flag)
{
        int result = 0, i = 0;

        get_random_bytes(&i, 2);
        result = (i%(e-s+1))+s;

        if(flag) {
                if(result < 0) {
                        result = 0 - result;
                }
        }

        return result;
```

```
}

void rand_string(char *des, int size, int flag)
{
        int i = 0;

        static const char text[] = "_.
            abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ";

        for(i = 0; i < size; i++) {
                des[i] = text[rand_num(0,54,1)];
        }
        des[i] = '\0';

        for(i = 0; i < flag; i++) {
                des[rand_num(0,size-1,1)] = '/';
        }
}

module_init(fuzzing_init);
module_exit(fuzzing_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("YUNFAN LI");
MODULE_DESCRIPTION("HELLO WORLD TEST MODULE");
```

Appendix II: Makefile

```
#Yunfan Li, Zhicheng Fu, Xiang Li
#CS544-001
#Final Project: Fuzzing
obj-m := fuzzing.o
KERNELDIR = /scratch/spring2015/cs444-group35/linux/

all:
        make -C $(KERNELDIR) M=$(PWD) modules
        rm -f *~

clean:
        make -C $(KERNELDIR) M=$(PWD) clean
```