Fuzzing Test Of System Calls in Linux Kernel

Kai Shi, Wen-yin Wang, Xu Zheng
Oregon State University
Group 36

Abstract

The main purpose of this document is using fuzzing test tool to test the Linux kernel system calls. The object Linux kernel is 3.14.24 version. sys_open(), sys_read(), sys_write(), sys_lseek(), sys_close(), sys_chdir() will be tested by fuzzing test tool. Also, basic information about fuzzing test will be provided, and we will explain the reason why we design fuzzing testing tool in this method. Last, testing results and analysis will be post at the end of document.

1 Executive Summary

This document shows the process and results of a fuzzing test tool. We write our own fuzzing test according to the trinity test tool. This fuzzing test tool can test 6 system calls in Linux 3.14.24 version: sys_open(), sys_read(), sys_write(), sys_lseek(), sys_close(), sys_chdir().

Purpose: To give an example of fuzzing test tool to test six Linux kernel system call.

Introduction: In this part, basic information and history of fuzzing is introduced. Also, the main developments of fuzzing test from college to company is significant in fuzzing history. These information will be provided in the introduction part.

Trinity Fuzzing Test Tool: This part introduce some basic information about trinity fuzzing test tool. It gives the most traditional method of trinity fuzzing test tool.

Our Fuzzing Test Tool: In this part, we introduce our design and the method we use. It gives details of our fuzzing test tool code. We explain why we use this method in our fuzzing test tool and provides some main issues we meet.

Linux Modules: Because our aim is testing Linux kernel system calls, we need to put our codes in the Linux kernel. Building modules to test system calls is a wise way to solve this problem. There are two ways to build modules: living in the source tree, and living externally. It is difficult to use the first way because we cannot decide where to place our module and it will cause lower efficiency of system.

System Calls And Analysis: This part provides details of each six system call. Parameters and expressions are given in this part. We also discuss about how to test these system calls, some bugs and solutions.

Results: Testing results are shown in Results part. Analysis of results are provided also. A table will be post to show the final testing results of our fuzzing test tool.

Conclusion: In this part, we will briefly review the entire document that we have written. We will give the result of our fuzzing test. At the end of this document, we will summarize what we have learned during this project. We believe that fuzzing will have a big development in the future.

2 Introduction

Fuzz testing or Fuzzing is a software testing technique used to detect bugs and security loopholes in software, operating systems or networks [1]. It is a Black Box testing, which consists in detecting bugs by inputting a lot of random data to the system in an attempt to make it crash.

Barton Miller at the University of Wisconsin developed fuzz testing in 1988. From 1988 to 2015, series of studies performed by Professor Miller and his associates. They have developed a variety of fuzzers and use them to evaluate the reliability of applications on various UNIX operating systems. It is difficult to determine precisely when fuzzing first became used in the black-hat community, as the extent and speed of uptake of fuzzing techniques is not well known. It is clear that two important steps in the spread were Aitel's 2002 paper and the release of SPIKE, and the availability of PROTOS at the same time. By 2005 it had been transitioned from a little known niche technique to one that was being widely discussed and adopted [2].

There are two main fuzzing approaches: mutation based and generation based [3]. Mutation based approach is also known as a "dumb" fuzzer, because it is all about mutating the existing input values. In contrast to mutation, generation based approach is an intelligent fuzzer. It is about generating the input based on the format or specification. Although mutation based approach is easier to implement than generation, generation is better because it has better code coverage and code paths [3].

In the fuzzing test, we use random data to fuzz a program and see which part of the program will be attacked. The key of the fuzzing test is that the inputs fuzzed are illogical. It means that the fuzzing test will put random data as much as possible to the target program instead of predicting which part of the program will be attacked. In this way, we can find bunches of bugs which will not be found with logical inputs. Fuzz testing usually can find the most serious faults.

Although fuzzing is simple, it offers a high benefit-to-cost ratio. It will reveal defects with high efficiency. And it is also easy to design and setup. That is why fuzzing is used widely as an effective testing technique.

3 Trinity Fuzzing Testing

Trinity is an intelligent Linux System call fuzz tester. Instead of passing purely random values as arguments for the system call, Trinity will pass one of the values expected based on the format. In this way, Trinity reduces the time spent running useless tests, thereby increasing the chances of testing a more interesting case that may result an unexpected error [4].

In order to perform this kind of intelligent fuzzing testing, Trinity must have some understanding of the arguments for each system call. This is accomplished by defining structures that annotate each system call. On startup, Trinity creates a list of file descriptions, by opening pipes, scanning sysfs, procfs, /dev, and creates a bunch of sockets using random network protocols. Then when a syscall need an fd, it gets passes one of these at random [5].

Trinity has been proved that it is successful at finding bugs. According to Dave Jones, the developer of Trinity, he finds more than

150 bugs using Trinity in 2012. And other people who were using Trinity found many more bugs. Interestingly, Trinity has found bugs not only in system call code, but also in many other parts of the kernel include the networking stack, virtual memory code, and drivers. In addition, Trinity has discovered a number of pieces of kernel code that had poor test coverage or indeed no testing at all. The oldest bug that Trinity has so far found dated back to 1996.

4 Our Fuzzing Testing

After we had understood Trinity, we designed our own fuzzing testing tool. As we know, fuzzing means giving random data as input values to systems. First of all, we need to specify system calls that we used in our fuzzing test, so that we can determine what kinds of inputs we should generate. Based on what we learned in CS344, we decided to fuzz six system calls which used commonly: chdir(), open(), read(), lseek(), write(), close().

When we were implementing our program, we separated our codes into four sections as below:

1. Main section: it contains Linux kernel header files, definitions, prototypes, declarations, initial and exit function of module. Also, we define a structure type, fd_info, to store all of data that we used in a single entry.

2. Initial section: it contains initialization of all data that we used in our program. We initialize structure and global data in this section.

3. Fuzzing section: it contains our fuzzing codes. We implement two fuzzing functions: fuzz() and fuzz_string(). fuzz() uses a Linux function, which is get_random_bytes(), to return a random number. fuzz_string() uses fuzz() to get a random number and use this number to get a random alphabet, then return the random string.

4. Function section: it contains our testing functions of six system calls, sys_chdir(), sys_open(), sys_read(), sys_lseek(), sys_write(), sys_close(), respectively. In addition, we implement a function, info_print(), to print the results.

We implement and test our program in our own directory. After we confirmed that our code is flawless, we rewrite it into an external Linux module. And we will describe more details in the following part.

5 Linux Modules

What we plan to do is building an external Linux module, which named fuzzing module. This module includes codes to test 6 system calls automatically. And then we load this fuzzing module into the Linux kernel.

Linux modules are pieces of code that can be loaded and unloaded into the kernel upon demand. We can add the module source to the kernel source, either as a patch or merging your code into the official tree. Alternatively,you can maintain and build your module source outside the source tree[6].

Therefore, there are two ways to build modules:

- Living in the source tree
- Living externally

It is difficult to decide where to place fuzzing module in the kernel source tree, so we choose to use the external module.

We plan to test 6 system calls in our module. We create a Makefile in source directory with this single line: obj-m := fishing.o. This compiles fuzzing.c into fuzzing.ko. We need to instruct make on how to find the kernel source files and base Makefile.This is also easy:

make -C /kernel/source/location SUBDIRS= $PWD modules.

Last, we can install module into /lib/modules/version/kernel/. The following build command is used to install compiled modules into the correct location:

make modules_install.

6 System Calls and Analysis

1. **sys_open():**

   ```
   int sys_open(const char *pathname,
            int flags,int mode);
   ```

   sys_open() opens a file in a given pathname and returns the file descriptor. A call to open() creates a new open file description, an entry in the system-wide table of open files. This entry records the file offset and the file status flags (modifiable via the fcntl(2) F_SETFL operation). A file descriptor is a reference to one of these entries; this reference is unaffected if pathname is subsequently be removed or modified to refer to a different file.

   In sys_open(), we generate a random string as filename and store it in fd_name. Then, we call open() to open by this filename. and we store the result in fd_no and fd_open_flag. Through the result, we can observe whether the return value is correct or not.

2. **sys_read():**

   ```
   ssize_t  read(int fd,
                 void *buf,
                 size_t count);
   ```

   read() attempts to read up to count bytes from file descriptor fd into the buffer buf. Fd is the file descriptors of the data need to be read. Data will be put into the buffer. Count shows the number of characters the read function reads.

   If count is zero, read() may detect the errors described below. In the absence of any errors, or if read() does not check for errors, a read() with a count of 0 returns zero and has no other effects.

   If count is greater than SSIZE_MAX, the result is unspecified.

   On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number.

   In sys_read(), we read from the file that we open in previous test and store the result in fd_read_flag. Through the result, we can observe whether the return value is correct or not.

3. **sys_write():**

   ```
   ssize_t sys_write(unsigned int fd,
                     const charâĹŮ buf ,
                     size\_t count );
   ```

   write() writes up to count bytes from the buffer buf pointed to the file referred to by the file descriptor fd.

   The difference between read() and write() is that count in read() can be given but the count in write() must be the number of characters should be written to the file. Otherwise, it will cause corruption.

   In sys_write(), we write into the file

that we open in previous test and store the result in fd_write_flag. Through the result, we can observe whether the return value is correct or not.

4. **sys_lseek():**

```
lseek (int handle, off_t offset,
       int fromwhere);
```

When using the UNIX system calls to read the file content, system read data from which position of file is completely decided by the file pointer.

Every open file has a read/write position, and when opening a file, the read position is the beginning of the file. When the system calls read () or write (), the position of reading or writing will increase. lseek () is used to control the position of reading and writing.

Upon successful completion, the resulting offset, as measured in bytes from the beginning of the file, shall be returned. Otherwise, (off_t)-1 shall be returned, errno shall be set to indicate the error, and the file offset shall remain unchanged[8].

In sys_lseek(), we search in the file that we already opened and store the result in fd_lseek_flag. Through the result, we can observe whether the return value is correct or not.

5. **sys_close():**

```
int close(int fd);
```

close() closes a file descriptor, so that it no longer refers to any file and may be reused. Close() returns zero on success. On error, -1 is returned, and errno is set appropriately.

In sys_close(), we close the file that we opened in previous function and store the result in fd_close_flag. Through the result, we can observe whether the return value is correct or not.

6. **sys_chdir():**

```
int sys_chdir(const char * filename);
```

chdir() is a system call which could change the current working path of a process. Parameter is the pointer to the directory. Each process has a current working directory, and if the process call chdir() to change the directory, it applies only to child process. It will not affect parent 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.

In sys_chdir(), we generate a random string as pathname and store it in fd_dir. Then we use this pathname to change our current path and store the result in fd_chdir_flag. Through the result, we can observe whether the return value is correct or not.

7 Results

We focus on the exceptions of our fuzzing test. For each system call, we evaluated the return value. If success, we assign 1 to the flag. If failed, we assign 0 to the flag. If exception happened, we assign -1 to the flag. After loading the module into the Linux kernel, we ran our fuzzing test tool, it printed the testing information in the screen. We analyzed those testing information, and summarized our testing results in the following table:

Table 1: *Fuzz Testing Results*

System Calls	Passing Rate	Comments
sys_open()	Low	See Comment #1
sys_read()	Medium	See Comment #2
sys_write()	Medium	See Comment #3
sys_lseek()	high	See Comment #4
sys_close()	high	See Comment #5
sys_close()	Low	See Comment #6

We do another test aimed at the following system calls: read(), write(), close(). Because if open() failed, it is meaningless to continue to test these system calls.

Comment#1: Because the filename in our fuzzing test is generated randomly, open() function will fail if the filename does not exist. Only the random filename exists, the open() function test will succeed. The rate of generating existed filenames is very low, therefore, the passing rate of open() in our fuzzing test is low.

Comment#2: We evaluate the passing rate of read() in the premise of the success of open(). The result shows the passing rate is medium. Because only the fd permission includes O_RDONLY or O_RDWR, read() system call will succeed.

Comment#3: We evaluate the passing rate of write() in the premise of the success of open(). The result shows the passing rate is medium. Because only the fd permission includes O_WRONLY or O_RDWR, write() system call will succeed.

Comment#4: When testing lseek(), we assume every fd is valid. This system call read from the beginning, current position, and the end of the testing files, and the passing rate is high.

Comment#5: We evaluate the passing rate of close() in the premise of the success of open(). We give some particular file names, and these file name exist. Final result is close() can pass all the time. This result shows close() system call has a strong robustness.

Comment#6: Because the pathname of our fuzzing test is generated randomly, chdir() function will fail if the pathname does not exist. Only the random pathname exists, the chdir() function test will succeed. The rate of generating existed pathnames is very low, therefore, the passing rate of chdir() in our fuzzing test is low.

Through the table and comments above, we found that sys_open() and sys_chdir() are hard to success with the random input values, because random input values are hard to match filename and pathname exactly. However, the other four system calls are more likely success after we open the file correctly. Only control unit, such as permission and mode, could cause failure of system calls. After 5000 times testing, we did not find any exception in our program. So, we can conclude that Linux kernel is stable.

8 Conclusions

Fuzzing is a testing technique that discovered software vulnerability. Because more and more programmers are currently focusing on the developing software security, fuzzing is more widely used as a code testing technique. From coming out the concept of Fuzzing to the extensive using, fuzzing only experienced 20 years. Due to the effectiveness and the convenience of its test, fuzzing now has became a milestone in the history of software testing tools.

First, we have a deep understanding about fuzzing test. According to the results of the fuzzing, we did not find any vulnerability in the Linux system. These six Linux kernel system calls that we tested did not generate system damage, shutdown or blocking. So we conclude that this Linux 3.14.24 has robustness with the testing of the fuzzing test tool which we wrote.

All in all, in this project we have an overview of the fuzzing test. We have already known the concept of fuzzing, brief history of fuzzing. Also, we knew that fuzzing as a software-testing tool has became a necessary application vulnerabilities detecting technique. This project reviewed all the previous assignments we have done and linked all the techniques together. For this assignment, we have a new understanding for Linux kernel programming. We believe that fuzzing, as the vulnerability and security testing tool will have a bright future.

References

[1] P. Garg. (2012, January) Fuzzing mutation vs. generation. [Online]. Available: http://resources.infosecinstitute.com/fuzzing-mutation-vs-generation/

[2] D. Grove and D. Gerhardy, "An empirical study of the reliability of unix utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990.

[3] D. Jared, "The evolving art of fuzzing," pp. 2–3, June 2006.

[4] J. Knudsen, "Practical considerations of fuzzing: Generating insight into areas of risk," *Horizons*, 2013.

[5] R. McNally and K. Yiu, "Fuzzing: The state of the art," *DSTO Defence Science and Technology Organisation*, 2012.

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

Appendix 1: Source Code

```
//      CS544
//      Group 36
//      Kai Shi
//      Wen-Yin Wang
//      Xu Zheng
//      Final paper
//      Fuzzing test

#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/unistd.h>
#include <linux/random.h>
#include <linux/string.h>
#include <linux/errno.h>
#include <linux/fcntl.h>
#include <linux/sysfs.h>
#include <linux/moduleloader.h>
#include <linux/file.h>
#include <linux/fs.h>
#include <linux/dirent.h>

#define NUM 5000
unsigned long **sys_call_table;

int (*getcwd)(char *buf, size_t size);
int (*chdir)(char *path);
int (*open)(char *filename, int flag, int mode);
int (*read)(int fd, char *buf, int size);
int (*lseek)(int fd, off_t offset, int whence);
int (*write)(int fd, char *buf, int size);
int (*close)(int fd);

char fd_addr_db[10][50];
int fd_pm_db[3];
int fd_offset_db[3];
char alphabet_db[40];

struct fd_info{
        char fd_dir[50];
        char fd_name[50];
        char buff[1024];
        int fd_pm;
        int fd_no;
        int fd_offset;
        int fd_chdir_flag;
        int fd_open_flag;
        int fd_read_flag;
        int fd_lseek_flag;
        int fd_write_flag;
        int fd_close_flag;
}fd_info_t;

struct fd_info info[NUM];
```

```
void fuzzing_test(void);
void info_init(struct fd_info* info);
void fuzz_init(void);
int fuzz(int n);
void fuzz_string(char* s, int n);
void sys_chdir(struct fd_info* info);
void sys_open(struct fd_info* info);
void sys_read(struct fd_info* info);
void sys_write(struct fd_info* info);
void sys_lseek(struct fd_info* info);
void sys_close(struct fd_info* info);
void info_print(struct fd_info* info);

static unsigned long **find_sys_call_table(void)
{
        unsigned long int offset = PAGE_OFFSET;
        unsigned long **sct;
        while (offset < ULLONG_MAX) {
                sct = (unsigned long **)offset;
                if (sct[__NR_close] == (unsigned long *) sys_close) {
                        printk(KERN_INFO "Interceptor: Found syscall table at
                            address: 0x%02lX\n",
                        (unsigned long) sct);
                        return sct;
                }
        offset += sizeof(void *);
        }
        return NULL;
}

static int __init fuzzer_init(void)
{
        printk(KERN_ALERT "Testing fuzz.\n");
        sys_call_table = find_sys_call_table(); //find system call table
        fuzz_init();    //initial data
        fuzzing_test(); //do fuzzing test
        return 0;
}

static void __exit fuzzer_exit(void)
{
        printk(KERN_ALERT "End fuzz.\n");
}

void fuzzing_test(void)
{
        int i;
        for(i = 0; i < NUM; i++){
                info_init(&info[i]);
                sys_chdir(&info[i]);
                sys_open(&info[i]);
                sys_read(&info[i]);
                sys_lseek(&info[i]);
                sys_write(&info[i]);
                sys_close(&info[i]);
                info_print(&info[i]);
        }
}

void info_init(struct fd_info* info)
{
        sprintf(info -> fd_name, " ");
        info -> fd_pm = fd_pm_db[fuzz(3)];
        info -> fd_no = 0;
        info -> fd_offset = fd_offset_db[fuzz(3)];
        info -> fd_chdir_flag = 0;
        info -> fd_open_flag = 0;
        info -> fd_read_flag = 0;
        info -> fd_lseek_flag = 0;
        info -> fd_write_flag = 0;
        info -> fd_close_flag = 0;
}

void fuzz_init(void)
{
        fd_pm_db[0] = O_RDWR;
        fd_pm_db[1] = O_RDONLY;
        fd_pm_db[2] = O_WRONLY;
        fd_offset_db[0] = SEEK_SET;
        fd_offset_db[1] = SEEK_CUR;
        fd_offset_db[2] = SEEK_END;
        sprintf(alphabet_db, "_-.0123456789abcdefghijklmnopqrstuvwxyz/");
}

int fuzz(int n)
{
        int i;
        get_random_bytes(&i, 1);
        if(i < 0){
                abs(i);
        }
        return i % n;
}

void fuzz_string(char* s, int n)
{
        int i;
        char new_s[n];
        for(i = 0; i < n; i++){
                new_s[i] = alphabet_db[fuzz(40)];
        }
        sprintf(s, "%s", new_s);
}

void sys_chdir (struct fd_info* info)
{
        //test chdir
        //results: Success: 1, Fail: 0, Except: -1
        int i;
        char cwd[100] = {'\0'};
        char new_dir[20] = {'\0'};
        chdir = (void *)sys_call_table[__NR_chdir];
        getcwd = (void *)sys_call_table[__NR_getcwd];
        getcwd(cwd, sizeof(cwd));
        fuzz_string(new_dir, 20);
        sprintf(info -> fd_dir, "%s", new_dir);
        if((i = chdir(new_dir)) == -1){
                info -> fd_chdir_flag = 0;
        }
        else if(i == 0){
                info -> fd_chdir_flag = 1;
        }
        else{
                info -> fd_chdir_flag = -1;
        }
        printk(KERN_ALERT "random pathname: %s, chdir_flag: %d\n", info -> fd_dir
            , info -> fd_chdir_flag);
}

void sys_open(struct fd_info* info)
{
        //test open
        //results: Success: 1, Fail: 0, Except: -1
        char new_name[10] = {'\0'};
        open = (void *)sys_call_table[__NR_open];
        fuzz_string(new_name, 10);
        sprintf(info -> fd_name, "%s", new_name);
        if((info -> fd_no = open(info -> fd_name, info -> fd_pm, 0644)) == -1){
                info -> fd_open_flag = 0;
        }
        else if(info -> fd_no >= 0){
                info -> fd_open_flag = 1;
        }
        else{
                info -> fd_open_flag = -1;
        }
        printk(KERN_ALERT "random filename: %s, open_flag: %d\n", info -> fd_name
            , info -> fd_open_flag);
}

void sys_read(struct fd_info* info)
{
        //test read
        //results: Success: 1, Fail: 0, Except: -1
        int i;
        read = (void *)sys_call_table[__NR_read];
        if((i = read(info -> fd_no, info -> buff, sizeof(info -> buff))) == -1){
                info -> fd_read_flag = 0;
        }
        else if(i >= 0){
                info -> fd_read_flag = 1;
        }
        else{
                info -> fd_read_flag = -1;
        }
        printk(KERN_ALERT "file number: %d, read_flag: %d\n", info -> fd_no, info
                -> fd_read_flag);
}

void sys_lseek (struct fd_info* info)
{
        //test lseek
        //results: Success: 1, Fail: 0, Except: -1
        int i;
        lseek = (void *)sys_call_table[__NR_lseek];
        if((i = lseek(info -> fd_no, 0, info -> fd_offset)) == -1){
                info -> fd_lseek_flag = 0;
        }
        else if(i >= 0){
                info -> fd_lseek_flag = 1;
        }
        else{
                info -> fd_lseek_flag = -1;
        }
        printk(KERN_ALERT "file number: %d, lseek_flag: %d\n", info -> fd_no,
            info -> fd_lseek_flag);
}

void sys_write(struct fd_info* info)
{
        //test write
        //results: Success: 1, Fail: 0, Except: -1
        int i;
        write = (void *)sys_call_table[__NR_write];
        if((i = write(info -> fd_no, info -> buff, strlen(info -> buff))) == -1){
                info -> fd_write_flag = 0;
        }
        else if(i >= 0){
                info -> fd_write_flag = 1;
        }
        else{
                info -> fd_write_flag = -1;
        }
        printk(KERN_ALERT "file number: %d, write_flag: %d\n", info -> fd_no,
            info -> fd_write_flag);
}

void sys_close(struct fd_info* info)
{
        //test close()
        //results: Success: 1, Fail: 0, Except: -1
        int i;
        close = (void *)sys_call_table[__NR_close];
        if((i = close(info -> fd_no)) == -1){
                info -> fd_close_flag = 0;
        }
        else if(i == 0){
                info -> fd_close_flag = 1;
        }
        else{
                info -> fd_close_flag = -1;
        }
        printk(KERN_ALERT "file number: %d, close_flag: %d\n", info -> fd_no,
            info -> fd_close_flag);
}

void info_print (struct fd_info* info)
{
        printk(KERN_ALERT "\nFile information:\n");
        printk(KERN_ALERT "\tFile dir: %s\n\tFile name: %s\n\tFile permission: %d
            \n\tFile descriptor no: %d\n\tFile offset: %d\n", info -> fd_dir,
            info -> fd_name, info -> fd_pm, info -> fd_no, info -> fd_offset);
        printk(KERN_ALERT "\tFile chdir flag: %d\n\tFile open flag: %d\n\tFile
            read flag: %d\n\tFile sleek flag: %d\n\tFile write flag: %d\n\tFile
            close flag: %d\n", info -> fd_chdir_flag, info -> fd_open_flag, info
            -> fd_read_flag, info -> fd_lseek_flag, info -> fd_write_flag, info
            -> fd_close_flag);
        printk(KERN_ALERT "End file.\n");
}

module_init(fuzzer_init);
module_exit(fuzzer_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Group 36");
MODULE_DESCRIPTION("FUZZER MODULE");
```