Testing Linux System Calls
With Fuzzy and Semi-Random Data

Hafed Alghamdi, David Stuve, Li Li [*]

Oregon State University
Group 26

Abstract

Software fuzz testing is testing software with random inputs, and a short history is given. The design of the fuzz testing tool Fuzzy is described, and is applied against a dozen popular system calls in Linux 3.0.4 kernel and Debian 6.5. A fairly high rate of crashes were found with fuzz testing, indicating that there are defects and interactions in the Linux kernel that can lead to crashes or vulnerabilities.
Keywords: Linux Kernel, Testing, Fuzzing, Kernel 3.0.4

I. Introduction

Software quality is one of the principal goals in the field of computer science and software engineering. Testing is one means to achieving the goal of software quality, and its traditional form has been in the construction of test cases. Test cases are written against the specification alone (white box) or against the specification and actual code (black box) and are written until a certain percentage of code coverage is achieved. Code coverage is the percentage of instructions and paths through the targeted software that are executed when running the test cases. Software is then considered to be of sufficient quality when enough test cases pass. The downside to this approach is due to the highly manual and thus expensive nature of writing tests, and so code coverage is almost always less than ideal. There is considerable room for improvement in how code is tested.

Fuzzing or fuzz testing is a clever attempt to solve the high cost of manually created test cases. The idea behind it is that code can be also be exercised by randomly generated inputs, and with a sufficiently large number of random inputs one can approach or even surpass the code coverage of enumerated test cases. And as with any probablistic problem solving method one needs time and horsepower, and there are no guarantees of complete coverage. Another advantage fuzzing has over traditional test case generation is in overcoming testing bias - test cases written by humans cover expected inputs, fuzzing can also explore the space of unexpected inputs.

II. A Brief History of Fuzz Testing

Fuzz testing seems to have its origins with Barton Miller's group at the University of Wisconsin[4]. They created a tool named fuzz to generate random characters which were fed into Unix utilities, generating an impressive 25-35% crash rate. Miller's group revisited fuzz testing in 1995[5] and found Unix variants had improved, but still had high crash rates from simple fuzz testing.

[*]Random order.

Fuzz testing increased in sophistication over the years, evolving from naively random inputs to intelligent fuzz testing where valid structures, strings, or bit streams are created and then randomly permuted to create semi-valid inputs[3]. In 2006 Dave Jones created **scrashme** which later became Trinity in 2010, and it used this more intelligent fuzz testing. Other similar tools such as Google's **iknowthis** were developed in the same time frame.

Fuzzing has gained popularity in the field of security research, and is now considered one of the most effective ways to identify software vulnerabilities[1]. A hacker or security researcher can use fuzzing to identify and target defects in software, looking for potential exploits. The cybersecurity division of the Software Engineering Institute, CERT, has a widely used tool named **BFF** (for Basic Fuzzing Framework). **BFF** is an impressively advanced tool which applies machine learning and evolutionary techniques to fine tune its random parameter generation to increase the efficiency in which it finds defects and potential exploits[2].

III. Introducing Fuzzy: A Small Fuzz Tester

Our team set out to fuzz test a dozen system calls using on Centos 6.5 and the Linux 3.0.4 kernel. "Fuzzy" seemed like an appropriate name for our small fuzz tester, and we designed it to be similar to the general purpose fuzzing framework found in the tool Trinity. We decided early on to fuzz system calls that are accessible via the *syscall()* function, with the thought that the library functions that often wrap them may be either error checking or reparing questionable inputs and thus shielding the kernel from our attempts to fuzz it. A system call failure was defined as either hanging or crashing - valid error return codes were considered to be a success. And after reviewing the Trinity blog for descriptions of historic weaknesses in the Linux kernel we decided to focus on a multithreaded fuzzing of system calls that worked with files and filesystems and file descriptors.

Table 1: *Files in the Fuzzy Tarball*

Filename	Description
README	Instructions on how to build and run Fuzzy
makefile.sh	Script to build Fuzzy, create log and PID directories
stop_clean.sh	Script to delete zombies processes, PID files, and log files
gentree.sh	Script to create dummy file tree for testing
fuzzer.c	Main loop and create manage child processes
fuzzer.h	Constants for process types
sandbox.c	Generate random arguments, and make syscalls
sandbox.h	File pool and fuzzing constants
syscall_def.h	Syscall description definitions
syscall_def.c	Lookup syscall descriptions

Much like Trinity, Fuzzy relies on a detailed description of the valid arguments for each system call that it tests. Fuzzy's flexible method for describing arguments is an array of structs of type *scall_desc* (for system call description) defined in *syscall_def.h*.

```
typedef struct
```

```
{
    int    scid;             // int number of syscall
    char   name[20];         // name of the system call
    int    argnum;           // number of arguments
    int    arg_type[4][20];  // for each arg: list of possible valid argument types
} scall_desc;
```

Each argument description can have up to 20 tags defining what argments are valid for it, and they are logically ORed together. All of the tags are defined in *syscall_def.h*. An example system call description (for *sys_creat()*) which has two arguments follows, where argument one is a path, and argument two is a mode for the created path:

```
SYS_creat, "SYS_creat", 2,          // 2 arguments to sys_creat
{
  // arg 1
  { FUZ_ARG_PATH_FILE_EXIST,        // path to a file that exists OR
    FUZ_ARG_PATH_FILE_NONEXIST,     // path to a file that doesn't OR
    FUZ_ARG_PATH_DIR_EXIST,         // path to a directory that exists
    FUZ_ARG_END },                  // end of description of arg 1
  // arg 2
  { FUZ_ARG_OPEN_MODE,              // only mode (an int) allowed
    FUZ_ARG_END },                  // end of description of arg 2
  // arg 3
  { FUZ_ARG_END }                   // no argument # 3
}
```

With the detailed description of a system call's arguments, one of the tags is selected at random for each argument. (Because they are logically ORed together only one per argument is necessary.) The function *sandbox_syscall_fuzarg()* in *sandbox.c* is then invoked to randomly produce the semi-valid that corresponds to the tag. A good example of semi-random input generation is creating the mode for *sys_lseek()* - where a valid seek mode is selected at random, or a completely random integer may be selected a certain percentage of the time:

```
case FUZ_ARG_LSEEK_MODE:
  if (rand()%2)
    i = SEEK_SET;          // random decide - this valid value
  else if (rand()%2)
    i = SEEK_CUR;          // or a different valid value
  else if (rand()%2)
    i = SEEK_END;          // or another valid value
  else
    i = rand();            // or a completely random value!
  ((int*)sandbox[argno])[0] = i;
  fprintf(log_stream, "arg #%d = %d\n", argno, i);
  return 0;
```

Fuzzy knows more than just integer values - it also can fill out structures randomly as well. In this example, a semi-valid *struct timespec* is needed, and it is created by filling in a *timespec* structure with random values for *tv_sec* and *tv_nsec*. Seconds is limited to something reasonable, less than 3 in this case. Similar techniques are used to generate other structures, buffers, and strings - the proper size but populated with random values.

```
case FUZ_ARG_TIMESPEC:
```

```
t = (struct timespec*)(intptr_t)sandbox[argno][0];
t->tv_sec = rand()%3;    // limit to a reasonable # of seconds
t->tv_nsec = ((long)rand() << 16) + rand();
fprintf(log_stream, "arg #%d = (%ld sec, %ld nanosec)\n", argno, t->tv_sec, t->tv_nsec);
return 0;
```

One of the more sophisticated behaviors that Fuzzy does is creating a sandbox subdirectory, and then maintaining a file descriptor pool of closed files and files opened for reading or writing. Testing processes then can use those files, and the pool is updated as files are used. Testing is thus more effective because file descriptors can be abused by attempting to access closed files, or files that have recently been used for another purpose.

At a high level view Fuzzy has three types of processes. The main process spawns all other processes, including any worker processes which later crash. The watchdog process wakes up regularly and monitors the worker processes and logs the current state. And worker processes do the actual system calls with semivalid inputs. Worker processes have several different behaviors in how they make system calls. The round robin behavior just calls all tested system calls in sequence and repeats one or ten times. The random worker behavior chooses a random system call and calls it one or ten times, regardless of results. The strategy behind multiple worker behaviors was not just to make random system calls, but random sequences of system calls in hopes of finding problematic combinations that reveal deeper bugs than just crashing on a single bad input.

Results are gathered from log files. Each worker logs its intended action before it makes the system call (including arguments), so that its log file will reflect the last system call it made before it crashed.

IV. System Calls We Tested

1. **execve**() - Execute program

   ```
   int execve(const char *filename,
              char *const argv[],
              char *const envp[]);
   ```

   Executes the program pointed to by *filename*. *filename* must be either a binary executable, or a script starting with a line of the form #!. **execve**() does not return on success, and the text, data, bss, and stack of the calling process are overwritten by that of the program loaded.

2. **sys_chdir**() - Change working directory

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

   chdir() changes the current working directory of the calling process to the directory specified in *path*.

   On success, zero is returned. On error, –1 is returned, and *errno* is set appropriately.

3. **sys_chmod**() - Change file permissions

   ```
   int sys_chmod(const char* filename,
                 mode_t mode);
   ```

Change the permissions of a file. On success, zero is returned. On error, –1 is returned, and *errno* is set appropriately.

4. **sys_close**() - Close a file descriptor

   ```
   int sys_close(unsigned int fd);
   ```

   Closes a file descriptor, so that it no longer refers to any file and may be reused. Returns zero on success. On error, –1 is returned, and *errno* is set appropriately.

5. **sys_creat**() - Create a file or device

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

   creat() is equivalent to *open()* with flags equal to **O_CREAT** | **O_WRONLY** | **O_TRUNC**.

6. **sys_lchown**() - Change file ownership

   ```
   int sys_lchown(const char* filename,
                  uid_t user,
                  gid_t group);
   ```

   These system calls change the owner and group of a file. On success, zero is returned. On error, –1 is returned, and *errno* is set appropriately.

7. **sys_link**() - Make a new name for a file

   ```
   int sys_link(const char* oldname,
                const char* newname);
   ```

   link() creates a new link (also known as a hard link) to an existing file. If *newpath* exists it will *not* be overwritten. On success, zero is returned. On error, –1 is returned, and *errno* is set appropriately.

8. **sys_lseek**() - Set read/write file offset

   ```
   off_t sys_lseek(unsigned int fd,
                   off_t offset,
                   unsigned int origin);
   ```

   Upon successful completion, **lseek**() returns the resulting offset location as measured in bytes from the beginning of the file. On error, the value *(off_t) –1* is returned and *errno* is set to indicate the error.

9. **sys_mknod**() - Create a directory or special or ordinary file

   ```
   int sys_mknod(const char* filename,
                 int mode,
                 dev_t dev);
   ```

   Creates a filesystem node (file, device special file or named pipe) named *pathname*, with attributes specified by *mode* and *dev*.

   Returns zero on success, or –1 if an error occurred (in which case, *errno* is set appropriately).

10. **sys_open**() - Open and possibly create a file or device

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

Given a *pathname* for a file, returns a file descriptor, a small, nonnegative integer for use in subsequent system calls (**read**(2), **write**(2), **lseek**(2), **fcntl**(2), etc.). The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process.

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 (modf the following *access modes*: **O_RDONLY**, **O_WRONLY**, or **O_RDWR**. These request opening the file read-only, write-only, or read/write, respectively.

11. **sys_read**() - Read from a file descriptor

```
ssize_t sys_read(unsigned int fd,
                 char* buf,
                 size_t count);
```

Attempts to read up to *count* bytes from file descriptor *fd* into the buffer starting at *buf*. On error, –1 is returned, and *errno* is set appropriately.

12. **sys_time**() - Get time in seconds

```
int sys_time(int* tloc);
```

time() returns the time as the number of seconds since the Epoch, 1970-01-01 00:00:00 +0000 (UTC). If *t* is non-NULL, the return value is also stored in the memory pointed to by *t*.

On success, the value of time in seconds since the Epoch is returned. On error, *((time_t) –1)* is returned, and *errno* is set appropriately.

V. Results

We ran Fuzzy on the Linux kernel 3.0.4 (CentOS system 6.5) extensively, and analyzed the logs of each of its child processes to see which system calls were last made before crashes. No hangs were detected. After tabulating which system calls were problematic, we found the following system calls and their relative crash frequency in all of Fuzzy's runs. We also noted some patterns that are discussed in the 'Notes' below:

Table 2: *Fuzz Testing Results*

Failure Frequency	Failure Type	System Call Tested	Observations
High	Crash	sys_chmod()	Note #1
High	Crash	sys_lchown()	Note #1
High	Crash	sys_lseek()	Note #2
Medium	Crash	sys_execve()	Note #3
Medium	Crash	sys_creat()	Note #4
Medium	Crash	sys_link()	Note #5
Medium	Crash	sys_mknod()	Note #6
Medium	Crash	sys_open()	Note #7
Low	Crash	sys_chdir()	Note #8
Low	Crash	sys_time()	Note #9
–	None	sys_close()	–
–	None	sys_read()	–

Notes:

Almost every system call (besides *sys_close()* and *sys_read()*) taking a file descriptor as its first argument often crashed when passing a file descriptor just closed with the *sys_close()* system call.

3. *sys_execve()* would crash with files made from random strings instead of executable or valid scripts and even with just passing NULL pointers to all arguments.

8. *sys_chdir()* with an existing file as an input would crash only after a number of another fuzz tested system calls (which didn't crash). This is complex case and is a subject for kernel debugging - these rare crashes are a hint as to the scope of a deeper problem.

1. Random data used as security attributes (as required uid, gid, permission bitmask, etc) would crash processes calling *sys_chmod()* and *sys_lchown()* with high probability.

4. Trying to create a file with an existing name (file or dir) while at the same time passing random data to mode_t bitmask creation mode crash is very probable.

5. Surpisingly, using an existing file/directory as arguments for *sys_link()* will crash the child process.

2. Some calls will crash after batch of subsequent call. For example, *sys_lseek()* will crash when few times calling it with large negative offsets relative to current file position (possible internal *long int* offset calculation overflow).

6. A combination of an existing file/directory as first argument and a random mode_t (see Note #3) will crash sys_mknod().

7. *sys_open()* with an existing directory as an input will crash rarely, after a number of another fuzz tested system calls (which didn't crash). This is complex case and is a subjec for kernel debugging but we already located a scope of the problem.

9. *sys_time()* with a NULL pointer as a buffer argument will crash rarely, after a number of another fuzz tested system calls (which didn't crash). This is another complex use case that deserves more research.

VI. Discussion

I. Generalization of the results

According to our criteria that crashing or hanging rather than returning an error code equates to a bug, Fuzzy seems to have detected bugs in the 3.0.4 Linux kernel. Some of the system calls we tested were pretty easy to suspect of having bugs, especially those where we found a high frequency of crashes. There were however, indications that more subtle bugs were present in the kernel, with system calls that only seemed to crash as part of a larger batch of system calls, while appearing highly reliable on their own. We consider these rare crashes to be indicative of more subtle interaction or side effect defects in the Linux kernel. With Fuzzy's logging system a debugging and code review expedition may find those defects. An interesting future effort could be a longer term run of Fuzzy with a statistical analysis of the log files in an effort to more effectively find evidence of system call interactions causing crashes.

II. Conclusions

While fuzzing may never replace formal testing with use cases, tools such as Fuzzy should have a place in the software quality toolkit. Subjecting a body of code to random and semi-valid inputs can overcome testing bias and improve both its reliability and security. Fuzz testing tools can get far more sophisticated and efficient at finding defects than what we implemented in Fuzzy, but even simple tools such as Fuzzy have value according to our results. All that is required is time to engineer semi-valid input patterns and CPU time to run those patterns against the software to be tested.

References

[1] Sofia Bekrar, Chaouki Bekrar, Roland Groz, and Laurent Mounier. Finding software vulnerabilities by smart fuzzing. In *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, pages 427–430. IEEE, 2011.

[2] Allen D Householder and Jonathan M Foote. Probability-based parameter selection for black-box fuzz testing. 2012.

[3] Michael Kerrisk. Lca: The trinity fuzz tester. 2013.

[4] Barton P Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[5] Barton Paul Miller, David Koski, Cjin Pheow Lee, Vivekananda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. Fuzz revisited: A re-examination of the reliability of unix utilities and services. 1995.