Linux Kernel Fuzzing Project
build 3.0.4

Li Li, David Stuve, Hafed Alghamdi*

Oregon State University
Group 26

Abstract

The main objective of this project is applying fuzz testing against Linux system calls on kernel build 3.0.4. The team will develop a tool that automates the test against a dozen of Linux kernel system calls applies tremendous values against these system calls and write the results to a log file where it can be analyzed.

I. Introduction

Fuzz testing or fuzzing is a software testing technique, often automated or semi-automated, that involves providing invalid, unexpected, or random data to the inputs of a computer program. The program is then monitored for exceptions such as crashes, or failing built-in code assertions or for finding potential memory leaks. Fuzzing is commonly used to test for security problems in software or computer systems.

The field of fuzzing originated with Barton Miller at the University of Wisconsin in 1988. This early work includes not only the use of random unstructured testing, but also a systematic set of tools to evaluate a wide variety of software utilities on a variety of platforms, along with a systematic analysis of the kinds of errors that were exposed by this kind of testing. In addition, they provided public access to their tool source code, test procedures and raw result data.

There are two forms of fuzzing program, mutation-based and generation-based, which can be employed as white-, grey-, or black-box testing.[1] File formats and network protocols are the most common targets of testing, but any type of program input can be fuzzed. Interesting inputs include environment variables, keyboard and mouse events, and sequences of API calls. Even items not normally considered "input" can be fuzzed, such as the contents of databases, shared memory, or the precise interleaving of threads.

For the purpose of security, input that crosses a trust boundary is often the most interesting.[2] For example, it is more important to fuzz code that handles the upload of a file by any user than it is to fuzz the code that parses a configuration file that is accessible only to a privileged user.

II. Linux kernel fuzz testing

Dave Jones, the developer of Trinity, noted that the earliest system call fuzz tester that he had heard of was Tsys, which was created around 1991 for System V Release 4. Another early example was a fuzz tester developed at the University of Wisconsin in the mid-1990s that was run against a variety of kernels, including Linux.

Naive fuzz testers are capable of finding some kernel bugs, but the use of purely random inputs greatly limits their efficacy. To see why this is, we can take the example of the madvise() system call, which allows a process to advise the kernel about how it expects to use a region of memory. This system call has the following prototype:

```
int madvise(void *addr, size_t length, int advice);
```

*A thank you or further information

madvise() places certain constraints on its arguments: addr must be a page-aligned memory address, length must be non-negative, and advice must be one of a limited set of small integer values. When any of these constraints is violated, madvise() fails with the error EINVAL. Many other system calls impose analogous checks on their arguments.

A naive fuzz tester that simply passes random bit patterns to the arguments of madvise() will, almost always, perform uninteresting tests that fail with the (expected) error EINVAL. As well as wasting time, such naive testing reduces the chances of generating a more interesting test input that reveals an unexpected error.

Thus, a few projects started in the mid-2000s with the aim of bringing more sophistication to the fuzz-testing process. One of these projects, Dave's scrashme, was started in 2006. Work on that project languished for a few years, and only picked up momentum starting in late 2010, when Dave began to devote significantly more time to its development. In December 2010, scrashme was renamed Trinity. At around the same time, another quite similar tool, iknowthis, was also developed at Google.

III. Smart fuzz testing

Trinity performs intelligent fuzz testing by incorporating specific knowledge about each system call that is tested. The idea is to reduce the time spent running "useless" tests, thereby reaching deeper into the tested code and increasing the chances of testing a more interesting case that may result in an unexpected error. Thus, for example, rather than passing random values to the advice argument of madvise(), Trinity will pass one of the values expected for that argument.

Likewise, rather than passing random bit patterns to address arguments, Trinity will restrict the bit pattern so that, much of the time, the supplied address is page aligned. However, some system calls that accept address arguments don't require memory aligned addresses. Thus, when generating a random address for testing, Trinity will also favor the creation of "interesting" addresses, for example, an address that is off a page boundary by the value of sizeof(char) or sizeof(long). Addresses such as these are likely candidates for "off by one" errors in the kernel code.

In addition, many system calls that expect a memory address require that address to point to memory that is actually mapped. If there is no mapping at the given address, then these system calls fail (the typical error is ENOMEM or EFAULT). Of course, in the large address space available on modern 64-bit architectures, most of the address space is unmapped, so that even if a fuzz tester always generated page-aligned addresses, most of the resulting tests would be wasted on producing the same uninteresting error. Thus, when supplying a memory address to a system call, Trinity will favor addresses for existing mappings. Again, in the interests of triggering unexpected errors, Trinity will pass the addresses of "interesting" mappings, for example, the address of a page containing all zeros or all ones, or the starting address at which the kernel is mapped.

In order to bring intelligence to its tests, Trinity must have some understanding of the arguments for each system call. This is accomplished by defining structures that annotate each system call. For example, the annotation file for madvise() includes the following lines:

```
struct syscall syscall_madvise = {
    .name = "madvise",
    .num_args = 3,
    .arg1name = "start",
    .arg1type = ARG_NON_NULL_ADDRESS,
    .arg2name = "len_in",
    .arg2type = ARG_LEN,
    .arg3name = "advice",
    .arg3type = ARG_OP,
    .arg3list = {
        .num = 12,
        .values = { MADV_NORMAL, MADV_RANDOM,
                MADV_SEQUENTIAL, MADV_WILLNEED,
                MADV_DONTNEED, MADV_REMOVE,
                MADV_DONTFORK, MADV_DOFORK,
                MADV_MERGEABLE, MADV_UNMERGEABLE,
                MADV_HUGEPAGE, MADV_NOHUGEPAGE },
    }
    ....
};
```

This annotation describes the names and types of each of the three arguments

that the system call accepts. For example, the first argument is annotated as ARG_NON_NULL_ADDRESS, meaning that Trinity should provide an intelligently selected, semi-random, nonzero address for this argument. The last argument is annotated as ARG_OP, meaning that Trinity should randomly select one of the values in the corresponding list (the MADV_* values above).

The second madvise() argument is annotated ARG_LEN, meaning that it is the length of a memory buffer. Again, rather than passing purely random values to such arguments, Trinity attempts to generate "interesting" numbers that are more likely to trigger errorsâĂŤfor example, a value whose least significant bits are 0xfff might find an off-by-one error in the logic of some system call.

Trinity also understands a range of other annotations, including ARG_RANDOM_INT, ARG_ADDRESS (an address that can be zero), ARG_PID (a process ID), ARG_LIST (for bit masks composed by logically ORing values randomly selected from a specified list), ARG_PATHNAME, and ARG_IOV (a struct iovec of the kind passed to system calls such as readv()). In each case, Trinity uses the annotation to generate a better-than-random test value that is more likely to trigger an unexpected error. Another interesting annotation is ARG_FD, which causes Trinity to pass an open file descriptor to the tested system call. For this purpose, Trinity opens a variety of file descriptors, including descriptors for pipes, network sockets, and files in locations such as /dev, /proc, and /sys. The open file descriptors are randomly passed to system calls that expect descriptors.

In addition to annotations, each system call can optionally have a sanitise routine that performs further fine-tuning of the arguments for the system call. The sanitise routine can be used to construct arguments that require special values (e.g., structures) or to correctly initialize the values in arguments that are interdependent. It can also be used to ensure that an argument has a value that won't cause an expected error. For example, the sanitise routine for the madvise() system call is as follows:

```
static void sanitise_madvise(int childno)
{
  shm->a2[childno] = rand() % page_size;
}
```

This ensures that the second (length) argument given to madvise() will be no larger than the page size, preventing the ENOMEM error that would commonly result when a large length value causes madvise() to touch an unmapped area of memory. Obviously, this means that the tests will never exercise the case where madvise() is applied to regions larger than one page. This particular sanitize routine could be improved by sometimes allowing length values that are larger than the page size.

Trinity has been rather successful at finding bugs. Dave reports that he has himself found more than 150 bugs in 2012, and many more were found by other people who were using Trinity. Trinity usually finds bugs in new code quite quickly. It tends to find the same bugs repeatedly, so that in order to find other bugs, it is probably necessary to fix the already discovered bugs first.

Interestingly, Trinity has found bugs not just in system call code. Bugs have been discovered in many other parts of the kernel, including the networking stack, virtual memory code, and drivers. Trinity has found many error-path memory leaks and cases where system call error paths failed to release kernel locks. In addition, it 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.

IV. Planning

The team decided to follow a modern plan to apply fuzz testing best practice to seek the maximum benefits from this assignment. So, the following plan is summarizing the project phases and milestones:

1. Check Trinity Blog and find unclean system calls discovered in 3.0 Linux kernel

or later versions and choose 2 out of them in addition to 10 clean system calls.

2. Define a failure criteria.

3. Develop our fuzzing tool based on defined failure criteria

4. Write the team final report in Latex in parallel for about 10 pages to meet the project deadline

V. System Calls

1. Read from a file descriptor

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

read() attempts to read up to *count* bytes from file descriptor *fd* into the buffer starting at *buf*.

On files that support seeking, the read operation commences at the current file offset, and the file offset is incremented by the number of bytes read. If the current file offset is at or past the end of file, no bytes are read, and **read**() returns zero.

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. It is not an error if this number is smaller than the number of bytes requested; this may happen for example because fewer bytes are actually available right now (maybe because we were close to end-of-file, or because we are reading from a pipe, or from a terminal), or because **read**() was interrupted by a signal. On error, –1 is returned, and *errno* is set appropriately. In this case it is left unspecified whether the file position (if any) changes.

2. Write to a file descriptor

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

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

The number of bytes written may be less than *count* if, for example, there is insufficient space on the underlying physical medium, or the **RLIMIT_FSIZE** resource limit is encountered (see **setrlimit**(2)), or the call was interrupted by a signal handler after having written less than *count* bytes. (See also **pipe**(7).)

For a seekable file (i.e., one to which **lseek**(2) may be applied, for example, a regular file) writing takes place at the current file offset, and the file offset is incremented by the number of bytes actually written. If the file was **open**(2)ed with **O_APPEND**, the file offset is first set to the end of the file before writing. The adjustment of the file offset and the write operation are performed as an atomic step.

POSIX requires that a **read**(2) which can be proved to occur after a **write**() has returned returns the new data. Note that not all filesystems are POSIX conforming.

On success, the number of bytes written is returned (zero indicates nothing was written). On error, –1 is returned, and *errno* is set appropriately.

If *count* is zero and *fd* refers to a regular file, then **write**() may return a failure status if one of the errors below is detected. If no errors are detected, 0 will be returned without causing any other effect. If *count* is zero and *fd* refers to a file other than a regular file, the results are not specified.

3. Open and possibly create a file or device

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

Given a *pathname* for a file, **open**() 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.

By default, the new file descriptor is set to remain open across an **execve**(2) (i.e., the **FD_CLOEXEC** file descriptor flag described in **fcntl**(2) is initially disabled; the **O_CLOEXEC** flag, described below, can be used to change this default). The file offset is set to the beginning of the file (see **lseek**(2)).

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 removed or modified to refer to a different file. The new open file description is initially not shared with any other process, but sharing may arise via **fork**(2).

The argument *flags* must include one of the following *access modes*: **O_RDONLY**, **O_WRONLY**, or **O_RDWR**. These request opening the file read-only, write-only, or read/write, respectively.

4. Close a file descriptor

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

close() closes a file descriptor, so that it no longer refers to any file and may be reused. Any record locks (see **fcntl**(2)) held on the file it was associated with, and owned by the process, are removed (regardless of the file descriptor that was used to obtain the lock).

If *fd* is the last file descriptor referring to the underlying open file description (see **open**(2)), the resources associated with the open file description are freed; if the descriptor was the last reference to a file which has been removed using **unlink**(2) the file is deleted.

close() returns zero on success. On error, –1 is returned, and *errno* is set appropriately.

5. 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. 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.

This new name may be used exactly as the old one for any operation; both names refer to the same file (and so have the same permissions and ownership) and it is impossible to tell which name was the "original".

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

7. Execute program

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

execve() executes the program pointed to by *filename*. *filename* must be either a binary executable, or a script starting with a line of the form:

#!

argv is an array of argument strings passed to the new program. By convention, the first of these strings should contain the filename associated with the file being executed. *envp* is an array of strings, conventionally of the form

key=value, which are passed as environment to the new program. Both *argv* and *envp* must be terminated by a NULL pointer. The argument vector and environment can be accessed by the called program's main function, when it is defined as:

```
int main(int argc, char *argv[],
         char *envp[]);
```

On kernel 2.6.23 and later, most architectures support a size limit derived from the soft **RLIMIT_STACK** resource limit (see **getrlimit**(2)) that is in force at the time of the **execve**() call. (Architectures with no memory management unit are excepted: they maintain the limit that was in effect before kernel 2.6.23.) This change allows programs to have a much larger argument and/or environment list.

For these architectures, the total size is limited to 1/4 of the allowed stack size. (Imposing the 1/4-limit ensures that the new program always has some stack space.) Since Linux 2.6.25, the kernel places a floor of 32 pages on this size limit, so that, even when **RLIMIT_STACK** is set very low, applications are guaranteed to have at least as much argument and environment space as was provided by Linux 2.6.23 and earlier. (This guarantee was not provided in Linux 2.6.23 and 2.6.24.) Additionally, the limit per string is 32 pages (the kernel constant **MAX_ARG_STRLEN**), and the maximum number of strings is 0x7FFFFFFF.

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.

8. 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*.

fchdir() is identical to **chdir**(); the only difference is that the directory is given as an open file descriptor.

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

9. 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.

10. Create a directory or special or ordinary file

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

The system call **mknod**() creates a filesystem node (file, device special file or named pipe) named *pathname*, with attributes specified by *mode* and *dev*.

The *mode* argument specifies both the permissions to use and the type of node to be created. It should be a combination (using bitwise OR) of one of the file types listed below and the permissions for the new node.

The permissions are modified by the process's *umask* in the usual way: the permissions of the created node are *(mode & ~umask)*.

The file type must be one of **S_IFREG**, **S_IFCHR**, **S_IFBLK**, **S_IFIFO** or **S_IFSOCK** to specify a regular file (which will be created empty), character special file, block special file, FIFO (named pipe), or UNIX domain socket, respectively. (Zero file type is equivalent to type **S_IFREG**.)

If the file type is **S_IFCHR** or **S_IFBLK** then *dev* specifies the major and minor

numbers of the newly created device special file (**makedev**(3) may be useful to build the value for *dev*); otherwise it is ignored.

If *pathname* already exists, or is a symbolic link, this call fails with an **EEXIST** error.

The newly created node will be owned by the effective user ID of the process. If the directory containing the node has the set-group-ID bit set, or if the filesystem is mounted with BSD group semantics, the new node will inherit the group ownership from its parent directory; otherwise it will be owned by the effective group ID of the process.

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

11. Change permissions of a file

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

These system calls change the permissions of a file. They differ only in how the file is specified.

On NFS filesystems, restricting the permissions will immediately influence already open files, because the access control is done on the server, but open files are maintained by the client. Widening the permissions may be delayed for other clients if attribute caching is enabled on them.

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

12. Change ownership of a file

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

These system calls change the owner and group of a file.

Only a privileged process (Linux: one with the **CAP_CHOWN** capability) may change the owner of a file. The owner of a file may change the group of the file to any group of which that owner is a member. A privileged process (Linux: with **CAP_CHOWN**) may change the group arbitrarily.

If the *owner* or *group* is specified as –1, then that ID is not changed.

When the owner or group of an executable file are changed by an unprivileged user the **S_ISUID** and **S_ISGID** mode bits are cleared. POSIX does not specify whether this also should happen when root does the **chown**(); the Linux behavior depends on the kernel version. In case of a non-group-executable file (i.e., one for which the **S_IXGRP** bit is not set) the **S_ISGID** bit indicates mandatory locking, and is not cleared by a **chown**().

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

13. Reposition read/write file offset

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

The **lseek**() function allows the file offset to be set beyond the end of the file (but this does not change the size of the file). If data is later written at this point, subsequent reads of the data in the gap (a "hole") return null bytes (\0) until data is actually written into the gap. Since version 3.1, Linux supports the following additional values for *whence*:

SEEK_DATA Adjust the file offset to the next location in the file greater than or equal to *offset* containing data. If *offset* points to data, then the file offset is set to *offset*.

SEEK_HOLE Adjust the file offset to the next hole in the file greater than or equal to *offset*. If *offset* points into the middle of a hole, then the file offset is set to *offset*. If there is no hole past *offset*, then the file offset is adjusted to the end of the file (i.e., there is an implicit hole at the end of any file).

In both of the above cases, **lseek**() fails if *offset* points past the end of the file.

These operations allow applications to map holes in a sparsely allocated file. This can be useful for applications such as file backup tools, which can save space when creating backups and preserve holes, if they have a mechanism for discovering holes.

For the purposes of these operations, a hole is a sequence of zeros that (normally) has not been allocated in the underlying file storage. However, a filesystem is not obliged to report holes, so these operations are not a guaranteed mechanism for mapping the storage space actually allocated to a file. (Furthermore, a sequence of zeros that actually has been written to the underlying storage may not be reported as a hole.) In the simplest implementation, a filesystem can support the operations by making **SEEK_HOLE** always return the offset of the end of the file, and making **SEEK_DATA** always return *offset* (i.e., even if the location referred to by *offset* is a hole, it can be considered to consist of data that is a sequence of zeros).

The **_GNU_SOURCE** feature test macro must be defined in order to obtain the definitions of **SEEK_DATA** and **SEEK_HOLE** from *<unistd.h>*.

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.

14. High-resolution sleep() call

```
long sys_nanosleep(struct timespec *rqtp,
                   struct timespec *rmtp);
```

nanosleep() suspends the execution of the calling thread until either at least the time specified in **req* has elapsed, or the delivery of a signal that triggers the invocation of a handler in the calling thread or that terminates the process.

If the call is interrupted by a signal handler, **nanosleep**() returns –1, sets *errno* to **EINTR**, and writes the remaining time into the structure pointed to by *rem* unless *rem* is NULL. The value of **rem* can then be used to call **nanosleep**() again and complete the specified pause (but see NOTES).

The structure *timespec* is used to specify intervals of time with nanosecond precision. It is defined as follows:

```
struct timespec {
    time_t tv_sec;  /* seconds */
    long   tv_nsec; /* nanoseconds */
};
```

The value of the nanoseconds field must be in the range 0 to 999999999.

Compared to **sleep**(3) and **usleep**(3), **nanosleep**() has the following advantages: it provides a higher resolution for specifying the sleep interval; POSIX.1 explicitly specifies that it does not interact with signals; and it makes the task of resuming a sleep that has been interrupted by a signal handler easier.

On successfully sleeping for the requested interval, **nanosleep**() returns 0. If the call is interrupted by a signal handler or encounters an error, then it returns –1, with *errno* set to indicate the error.

15. Change access and/or modification times of an inode

```
int sys_utime(char* filename,
              struct utimbuf* times);
```

The **utime**() system call changes the access and modification times of the inode specified by *filename* to the *actime* and *modtime* fields of *times* respectively.

If *times* is NULL, then the access and modification times of the file are set to the current time.

Changing timestamps is permitted when: either the process has appropriate privileges, or the effective user ID equals the user ID of the file, or *times* is NULL and the process has write permission for the file.

The *utimbuf* structure is:

```
struct utimbuf {
    time_t actime;  /* access time */
    time_t modtime; /* modification time */
};
```

The **utime**() system call allows specification of timestamps with a resolution of 1 second.

The **utimes**() system call is similar, but the *times* argument refers to an array rather than a structure. The elements of this array are *timeval* structures, which allow a precision of 1 microsecond for specifying timestamps. The *timeval* structure is:

```
struct timeval {
    long tv_sec;  /* seconds */
    long tv_usec; /* microseconds */
};
```

times[0] specifies the new access time, and *times*[1] specifies the new modification time. If *times* is NULL, then analogously to **utime**(), the access and modification times of the file are set to the current time. On success, zero is returned. On error, –1 is returned, and *errno* is set appropriately.

VI. Results

During extensive fuzz testing of Linux kernel 3.0.4 (CentOS system 5.6.) thanks to a application's log system (each fuzz worker process has own logfile) we found a number of system calls crashing fuzzing subprocesses. No hangs detected.

We have analyzed a large number of cases and find some patterns, leading to errors.

We are publishing analysis results below:

Table 1: *Fuzz Testing Results*

Rate	System Call	Failure Type	Pattern
High	lseek()	Panic	Note #2
High	lchown()	Panic	Note #3
High	chmod()	Panic	Note #3
Mid	execve()	Panic	Note #5
Mid	creat()	Panic	Note #4
Mid	link()	Panic	Note #6
Mid	mknod()	Panic	Note #7
Mid	open()	Panic	Note #9
Low	chdir()	Panic	Note #8
Low	time()	Panic	Note #10

Notes:

1. Almost every system call (besides specified in table) receiving file descriptor as first argument it's possible to crash by passing file descriptor just closed with *close()* system call.

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

3. Random data used as security attributes (when required uid, gid, permission bitmask, etc) will crash calling process with high probability.

4. For some reason if try to create file with existing name (file or dir) same time passing random data to mode_t bitmask creation mode crash is very probable.

5. *sys_execve()* possible to crash with garbage file instead of executable or script or even with just passing NULL pointers to all arguments.

6. Using existing file/directory as arguments for *link()* will crash.

7. Cambination of existing file/directory as first argument and random mode_t (see Note #3) will crash mknod().

8. *chdir()* with existing file as an input will crash only in batch, after a number of another fuzz tested system calls (which didn't crash). This is complex case and is a subject for kernel debugging but we already located a scope of the problem.

9. *open()* with existing directory as an input will crash sometimes only in batch, 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.

10. *time()* as NULL pointer as a buffer argument will crash sometimes only in batch, after a number of another fuzz tested system calls (which didn't crash). This is complex case and is a subject for kernel debugging but we already located a scope of the problem.

VII. Discussion

I. Generalization of the results

Some patterns consists of number of arguments (which may implictly be dependant or influence each another). Or as we just seen even more difficult: previous fuzzed calls may "poison" some internal kernel structures and some another next system call may crash, hovewer it will not crash if call it alone in a fuzz sequense after system reboot. Ofter it's difficult or impossible to discover a crash pattern without deep kernel source code analysis.

However, fuzzer's log files have all information required to pass to problematic system call during kernel debugging for easy bug reproduction.

Our results points to the scope of a potential kernel code problem, not to a problem itself. To publish kernel bug report or to debug kernel code ourselves in future it's important to know exact system call arguments caused crash. Best is to try find a pattern(-s) in system call arguments if it's possible use large number of crash statistics.

II. Conclusions

Using random (or better, enginered pattern) data proved to be a simple and effective way to test robustness of many aspects of operating systems. The tool developed during this project can be used for easy testing of any modern UNIX/Linux kernel. The developers of these kernels would benefit from using this tool as a way to improve the reliability of their systems. Every new release of an operating system should be subjected to a few days of kernel fuzz testing.