# LCA: The Trinity fuzz tester

By **Michael Kerrisk**
February 6, 2013

The Linux kernel developers have long been aware of the need for better testing of the kernel. That testing can take many forms, including testing for performance regressions and testing for build and boot regressions. As the term suggests, regression testing is concerned with detecting cases where a new kernel version causes problems in code or features that already existed in previous versions of the kernel. Of course, each new kernel release also adds new features. The Trinity fuzz tester is a tool that aims to improve testing of one class of new (and existing) features: the system call interfaces that the kernel presents to user space.

Insufficient testing of new user-space interfaces is a long-standing issue in kernel development. Historically, it has been quite common that significant bugs are found in new interfaces only a considerable time after those interfaces appear in a stable kernel—examples include `epoll_ctl()`, `kill()`, `signalfd()`, and `utimensat()`. The problem is that, typically, a new interface is tested by only one person (the developer of the feature) or at most a handful of people who have a close interest in the interface. A common problem that occurs when developers write their own tests is a bias toward tests which confirm that expected inputs produce expected results. Often, of course, bugs are found when software is used in unexpected ways that test little-used code paths.

Fuzz testing is a technique that aims to reverse this testing bias. The general idea is to provide unexpected inputs to the software being tested, in the form of random (or semi-random) values. Fuzz testing has two obvious benefits. First, employing unexpected inputs mean that rarely used code paths are tested. Second, the generation of random inputs and the tests themselves can be fully automated, so that a large number of tests can be quickly performed.

### History

Fuzz testing has a history that stretches back to at least the 1980s, when fuzz testers were used to test command-line utilities. The history of system call fuzz testing is nearly as long. During his talk at linux.conf.au 2013 [ogv video, mp4 video], 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 [postscript] developed at the University of Wisconsin in the mid-1990s that was run against a variety of kernels, including Linux.

Tsys was an example of a "naïve" fuzz tester: it simply generated random bit patterns, placed them in appropriate registers, and then executed a system call. About a decade later, the kg_crashme tool was developed to perform fuzz testing on Linux. Like Tsys, kg_crashme was a naïve fuzz tester.

Naïve 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);
```

`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 naïve 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 naïve 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.

### Intelligent 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 errors—for 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. By now, it might start to become clear that you don't want to run Trinity on a system that has the only copy of your family photo albums.

In addition to annotations, each system call can optionally have a `sanitise` routine (Dave's code employs the British spelling) 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.

**Running trinity**

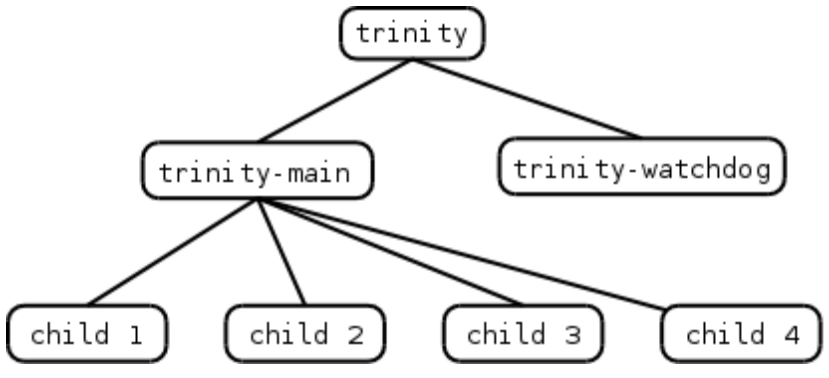
The Trinity home page has links to the Git repository as well as to the latest stable release (Trinity 1.1, which was released in January 2013). Compilation from source is straightforward; then Trinity can be invoked with a command line as simple as:

```
$ ./trinity
```

With no arguments, the program repeatedly tests randomly chosen system calls. It is also possible to test selected system calls using one or more instances of the `-c` command-line option. This can be especially useful when testing new system calls. Thus, for example, one could test just the `madvise()` system call using the following command:

```
$ ./trinity -c madvise
```

In order to perform its work, the `trinity` program creates a number of processes, as shown in the following diagram:




The `main` process performs various initializations (e.g., opening the file descriptors and creating the memory mappings used for testing) and then kicks off a number (default: four) of child processes that perform the system call tests. A shared memory region (created by the initial `trinity` process) is used to record various pieces of global information, such as open file descriptor numbers, total system calls performed, and number of system calls that succeeded and failed. The shared memory region also records various information about

each of the child processes, including the PID, and the system call number and arguments for the system call that is currently being executed as well as the system call that was previously executed.

The `watchdog` process ensures that the test system is still working correctly. It checks that the children are progressing (they may be blocked in a system call), and kills them if they are not; when the `main` process detects that one of its children has terminated (because the `watchdog` killed it, or for some other reason), it starts a new child process to replace it. The `watchdog` also monitors the integrity of the memory region that is shared between the processes, in case some operation performed by one of the children has corrupted the region.

Each of the child processes writes to a separate log file, recording the system calls that it performs and the return values of those system calls. The file is synced just before each system call is performed, so that if the system panics, it should be possible to determine the cause of the panic by looking at the last recorded system call in each of the log files. The log file contains lines such as the following, which show the PID of child process, a sequential test number, and the system call arguments and result:

```
[17913] [0] mmap(addr=0, len=4096, prot=4, flags=0x40031, fd=-1, off=0) = -1 (Invalid argument)
[17913] [1] mmap(addr=0, len=4096, prot=1, flags=0x25821, fd=-1, off=0x80000000) = -541937664
[17913] [2] madvise(start=0x7f59dff7b000, len_in=3505, advice=10) = 0
...
[17913] [6] mmap(addr=0, len=4096, prot=12, flags=0x20031, fd=-1, off=0) = -1 (Permission denied)
...
[17913] [21] mmap(addr=0, len=4096, prot=8, flags=0x5001, fd=181, off=0) = -1 (No such device)
```

Trinity can be used in a number of ways. One possibility is simply to leave it running until it triggers a kernel panic and then look at the child logs and the system log in order to discover the cause of the panic. Dave has sometimes left systems running for hours or days in order to discover such failures. New system calls can be exercised using the `-c` command-line option described above. Another possible use is to discover unexpected (or undocumented) failure modes of existing system calls: suitable scripting on the log files can be used to obtain summaries of the various failures of a particular system call.

Yet another way of using the `trinity` program is with the `-V` (victim files) option. This option takes a directory argument: the program will randomly open files in that directory and pass the resulting file descriptors to system calls. This can be useful for discovering failure modes in a particular filesystem type. For example, specifying an NFS mount point as the directory argument would exercise NFS. The `-V` flag can also be used to perform a limited kind of testing of *user-space* programs. During his linux.conf.au presentation, Dave demonstrated the use of the following command:

```
$ ./trinity -V /bin -c execve
```

This command has the effect of executing random programs in `/bin` with random string arguments. Looking at the system log revealed a large number of programs that crashed with a segmentation fault when given unexpected arguments.

**Results**

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.

**Limitations and future work**

Although Trinity is already quite an effective tool for finding bugs, there is scope for a lot more work to make it even better. An ongoing task is to add support for new system calls and new system call flags as they are added to the kernel. Only about ten percent of system calls currently have `sanitise` routines. Probably many other system calls could do with `sanitise` routines so that tests would get deeper into the code of those system calls without triggering the same common and expected errors. Trinity supports many network protocols, but that support could be further improved and there are other networking protocols for which support could be added.

Some system calls are annotated with an `AVOID_SYSCALL` flag, which tells Trinity to avoid testing that system call. (The `--list` option causes Trinity to display a list of the system calls that it knows about, and indicates those system calls that are annotated with `AVOID_SYSCALL`.) In some cases, a system call is avoided because it is uninteresting to test—for example, system calls such as `fork()` have no arguments to fuzz and `exit()` would simply terminate the testing process. Some other system calls would interfere with the operation of Trinity itself—examples include `close()`, which would randomly close test file descriptors used by child processes, and `nanosleep()`, which might put a child process to sleep for a long time.

However, there are other system calls such as `ptrace()` and `munmap()` that are currently marked with `AVOID_SYSCALL`, but which probably could be candidates for testing by adding more intelligence to Trinity. For example, `munmap()` is avoided because it can easily unmap mappings that are needed for the child to execute. However, if Trinity added some bookkeeping code that recorded better information about the test mappings that it creates, then (only) those mappings could be supplied in tests of `munmap()`, without interfering with other mappings needed by the child processes.

Currently, Trinity randomly invokes system calls. Real programs demonstrate common patterns for making system calls—for example, opening, reading, and closing a file. Dave would like to add test support for these sorts of commonly occurring patterns.

An area where Trinity currently provides poor coverage is the multiplexing `ioctl()` system call, "the worst interface known to man". The problem is that `ioctl()` is really a mass of system calls masquerading as a single API. The first argument is a file descriptor referring to a device or another file type, the second argument is a request type that depends on the type of file or device referred to by the first argument, and the data type of the third argument depends on the request type. To achieve good test support for `ioctl()` would require annotating each of the request types to ensure that it is associated with the right type of file descriptor and the right data type for the third argument. There is an almost limitless supply of work here, since there are hundreds of request types; thus, in the first instance, this work would probably be limited to supporting a subset of more interesting request types.

There are a number of other improvements that Dave would like to see in Trinity; the source code tarball contains a lengthy `TODO` file. Among these improvements are better support for "destructors" in the system call handling code, so that Trinity does not leak memory, and support for invoking (some) system calls as root. More generally, Trinity's ability to find further kernel bugs is virtually limitless: it simply requires adding ever more intelligence to each of its tests.

---



**LCA: The Trinity fuzz tester**
Posted Feb 6, 2013 22:06 UTC (Wed) by **mmeehan** (subscriber, #72524) [Link]

It'd be really interesting (and likely equally challenging) to use the Trinity annotations and sanitise code to perform static analysis on userspace programs. With all the effort that must go into generating a formal definitiion around system calls there must be other ways of exposing/using that data usefully.

> **Code annotations for static analysis in user-space**
> Posted Feb 7, 2013 1:43 UTC (Thu) by **michaelr** (guest, #73025) [Link]
>
> Annotating source code to do static analysis is not new to practical user-space programs. Microsoft has long used and advocated its Static Annotation Language (http://msdn.microsoft.com/en-us/library/ms182032.aspx), which captures many of the same concepts as Trinity's annotation system.
>
> SAL annotates the function prototypes directly; the nearest equivalent to the `madvise()` Trinity annotation above would be something like:
>
> ```
> int madvise(
>   _In_reads_bytes_(length) void *addr,
>   _In_ size_t length,
>   _In_ _In_range_(MADV_NORMAL, MADV_UNMERGEABLE) int advice
> );
> ```
>
> The `_In_reads_bytes_(length)` tag on `addr` captures the Trinity annotations for both `addr` and `length`. I've assumed that `MADV_NORMAL` is the numerically lowest advice value and `MADV_UNMERGEABLE` is the highest, as SAL doesn't deal well with enumeration constants that aren't defined in an enum type.
>
> Microsoft requires use of SAL in all its C and C++ code as part of its Security Development Lifecycle (http://msdn.microsoft.com/en-us/library/windows/desktop/cc307398.aspx), and advocates its uses by third-party application and driver developers on Windows. I don't know whether they use it for fuzzing guidance as Trinity does, but I do know that they have at least two static analyzers that understand it (one of which ships with recent versions of their compiler) and they feel it to have helped tremendously in writing more secure code.

**LCA: The Trinity fuzz tester**
Posted Feb 7, 2013 15:34 UTC (Thu) by **ortalo** (subscriber, #4654) [Link]

As a duty to someone who welcomed me to computer science research, my personal (first) reference on such subject is:
http://scholar.google.fr/scholar?hl=fr&q=thevenod+fos...

I am not sure she would be interested in studying the linux kernel, but... you never know.

**LCA: The Trinity fuzz tester**
Posted Feb 11, 2013 22:21 UTC (Mon) by **fintler** (guest, #81368) [Link]

Although it's nowhere near as polished, I ended up creating something similar a few years back for xnu (for mac os x, based on sysfuzz). https://github.com/fintler/xnufuzz

**LCA: The Trinity fuzz tester**
Posted Feb 24, 2013 20:43 UTC (Sun) by **rwmj** (subscriber, #5474) [Link]

I had a better idea for improving fuzz-testing. You use a genetic algorithm to "evolve" the fuzzed parameters, with the cost function being how much kernel code is executed. Conveniently systemtap lets you precisely measure how much code has been executed within a system call by putting a systemtap tap on every line of code (usually limited to the specific kernel module under test).

More here:
http://rwmj.wordpress.com/2010/11/22/half-baked-ideas-fee...

I actually implemented a fair bit of this.

> **LCA: The Trinity fuzz tester**
> Posted Feb 25, 2013 12:53 UTC (Mon) by **spender** (subscriber, #23067) [Link]
>
> Sounds exactly like security research published in 2006:
>
> https://www.blackhat.com/presentations/bh-usa-06/BH-US-06...
>
> -Brad
>
> > **LCA: The Trinity fuzz tester**
> > Posted Feb 25, 2013 14:35 UTC (Mon) by **rwmj** (subscriber, #5474) [Link]
> >
> > Yup, it looks like I hit on the same idea that these researchers found in 2006. The fuzz tester that is the subject of this article could do a lot better.