A trio of fuzzers

By **Jake Edge**
November 9, 2016

Linux Plumbers Conference

In the testing and fuzzing microconference at the 2016 Linux Plumbers Conference, three separate kernel fuzzing projects were presented and discussed. It didn't take long to see that there are a few different opportunities for the projects to collaborate even though they have different focuses and operational modes. Some of that collaboration got started the next day in a session on creating a formal specification of the Linux user-space APIs.

Fuzzing is a testing technique aimed at finding bugs by providing random inputs to programs and APIs. Often, the bugs found are exploitable security vulnerabilities, so there is a strong push to apply fuzzing to the kernel.

syzkaller

First up was Dmitry Vyukov, who was presenting about syzkaller, which is a coverage-guided fuzzer. The idea is to take a corpus of "interesting inputs" and to record the code paths that are executed for those inputs. The inputs are then mutated in various ways, and if a new code path is taken, that input gets added to the corpus. This has been done before in user space, but he wanted to apply it to the kernel.

One problem is in how to get the coverage information out of the kernel. Over the last year or so, GCC has gained the ability to insert a function call into every basic block in the code. The syzkaller team added the `CONFIG_KCOV` option to the kernel to build it using that GCC mode and to provide a way for user space to retrieve the coverage information from debugfs.

Vyukov spent a bit of time explaining the system-call descriptions that are used to guide the mutations of the inputs. It is mostly about argument and return types for the function, but goes beyond simply the C types of those elements. For example, it can describe things like the map file descriptors that are created by the `bpf()` system call (when using the `MAP_CREATE` flag), which are placed into a structure that gets passed to other `bpf()` operations.

The system-call descriptions allow syzkaller to generate and mutate programs that make system calls. The algorithm used is to start with an empty corpus; each iteration either generates a new program or chooses one from the corpus to mutate. The program is executed and the coverage is collected; if new parts of the kernel code have been reached, the program is added into the corpus.

There is a threaded execution mode to handle blocking system calls. For example, a `read()` on a pipe will block until a `write()` is done on the other end of the pipe, so each system call is done in its own thread. There is also a "collider" that tries to simultaneously run consecutive system calls in the program, which finds "a lot of data races", he said. It is currently implemented in a basic form. There is an open question how to make it better and how hard to collide the calls.

Another area that is a work in progress is a way to provide "external stimulus" to the programs. For example, a program may need to have some network packets delivered to it. The idea is to have hooks in the kernel to inject data in the right places, but it is not completely clear where those hooks should be. It needs to be done synchronously in the thread so that the code coverage information can be collected. It also needs to be reproducible so that problems can be tracked down. Part of that is that he needs to ensure that the kernel is not keeping state information between calls, such as dynamic cookie values.

He is also looking at doing smarter program mutation; the program space is "enormous". Right now, there is some basic prioritization, so that a program that works with TCP sockets will add more calls that use sockets. But he thinks there is more work to be done; using the knowledge of what the program is doing, there should be a way to mutate the programs better. He doesn't have an exact solution, but hopes to get there.

But a small group of people cannot describe all of the system calls, he said. Sometimes a good description requires domain knowledge for the system call and underlying subsystem. Also, new system calls, flags, and fields in structures are constantly being added, so the syzkaller team cannot keep up. The previous day he had gotten together with the BPF folks and made the description of that system call better.

Currently, those descriptions are part of syzkaller itself, but it would be nice to have them in the kernel tree, Vyukov said. There are two proposed locations for those descriptions: `include/uapi` or `Documentation`. Sasha Levin suggested that a machine-readable description of the system calls could be used by other tools to, for example, validate system calls at runtime. And Thomas Gleixner objected that "random descriptions" as documentation is not what is needed; a formal specification of the user-space API should be created instead. There is a group in Germany working on safety certifications for Linux that would be willing to help with that effort, he said, as would some kernel developers. That idea was generally well-received and would come up again in the microconference.

Each program that is generated for testing is self-contained, Vyukov said. They run in one process with separate directories for each. The programs can become arbitrarily long if needed; if there is a resource that the program needs to use, it must create it as there are no facilities for passing things into the program.

In less than a year, though, syzkaller has found more than 300 runtime failures. Many of those were security issues, he said, so reports from the syzkaller team should not be ignored.

Gleixner suggested that syzkaller exercise the signal code in the kernel because it is a code path that has not had much coverage over the years. If one thread is blocked in a system call and another thread sends it a signal that will get into code paths that are rarely executed. Vyukov said that syzkaller has support for some of the signal-related system calls, but not to do what Gleixner proposed. It should be possible to do so, though.

Trinity

Dave Jones then stepped forward to talk about the Trinity system-call fuzzer. Jones gave more details on how Trinity functions in a talk at linux.conf.au in 2013.

Over the last year, he has been trying to get the child process that is making the system calls to run longer before it fails from bugs in Trinity itself. It used to be on the order of 50 system calls before a crash, but is now around 5000 calls. Longer-running processes meant that he needed to add garbage collection for the `mmap()` regions so that the processes do not get a visit from the out-of-memory (OOM) killer.

A key feature that he has added in the last year is to alter the access patterns for the `mmap()` regions. Now a test run can access the first page, last page, every other page, or a random page (as it had done previously) in the region. That, he said, turned up a lot of bugs.

He has also been improving the BPF support in Trinity. In the last six months, he has added better BPF generation. The plan is to attach random BPF programs to tracepoints and "see what falls out". There have also been stability and debuggability improvements made.

For a new system call, Trinity can find trivial bugs in the implementation pretty quickly. The others that it finds are those that take a long time to run and are hard to reproduce. In addition, there is no way to get a real handle on what caused the crash. He does dump the last system calls made in all of the threads, but that may not really help. He is going to add network logging which would allow Trinity to store more state before the crash.

Mathieu Desnoyers asked if Trinity could use the tracing facilities in the kernel, such as the ftrace function tracer or LTTng, to gather its state. Jones said that was a good idea that he wanted to explore. Steve Rostedt suggested that choosing only certain functions to trace could be one way to reduce the size of the trace. The ftrace ring buffer will be part of what gets dumped in a kernel crash dump, so that could be used to extract the tracepoints that were hit before a crash.

At Facebook, where Jones works, he has machines running Trinity constantly. He would like to make it more parallelizable across machines so that crashes that take 24 hours to reproduce could be done in an hour on, say, 50 machines. He wants to find a way to reduce the randomness in Trinity when trying to reproduce problems so that crashes happen more predictably and can hopefully be diagnosed more quickly.

Jones noted that he wanted to talk to Desnoyers and Rostedt about tracing, as well as Vyukov about the system-call descriptions that syzkaller uses. Vyukov said that it might make sense for Trinity to use the syzkaller descriptions, and Jones agreed. Right now, it is a lot manual work to maintain that information, which is something Jones hates doing. He also often misses new system calls and flags that get added. There is a lot of value in having two fuzzers that work differently sharing infrastructure, he said.

The conversation turned back to the idea of a formal system-call specification. Mike Frysinger noted that user space could use that information in a variety of ways; `strace` could use it, for example. Jones said it is surprising that "we haven't gotten there already", but it appears that a critical mass is forming around the idea at this point.

Someone asked if the macros that define system calls in the kernel could be extended with more metadata to automatically generate the specification. Vyukov said there is a lot more that is needed than could be put into the macros; for example, valid flag values and crypto algorithms would need to part of the specification. Levin said that it was important to write the specification rather than get the information automatically; the kernel should "chase the specification", instead of the other way around.

Jones concluded by saying that he plans to get back to the networking code in Trinity. He hasn't really been finding networking bugs over the last year and would like to change that. He wants to talk with Vyukov about his ideas for injecting network traffic into processes. There is a huge opportunity to collaborate on that, he said.

perf_fuzzer

The last fuzzer presented in the track is a special-purpose one that simply fuzzes a single system call: `perf_event_open()`. Vince Weaver spoke about his perf_fuzzer tool, which he has also written about for LWN. Weaver is a professor at the University of Maine and said that he had wanted to give his presentation using a Raspberry Pi with a custom operating system that he wrote, but had recently broken the frame buffer support, so he went without slides (as had Jones before him).

He set out to fuzz perf because he wanted to show that it is safe for use in high-performance computing (HPC). Users of HPC systems hate crashes, but want to use the performance counters to monitor their workloads. `perf_event_open()` has different paranoia levels that can be set via a sysctl parameter (`kernel.perf_event_paranoid`). The default is 2, which does not allow regular users access to some of the more interesting perf features. He set out to show that it would be safe to reduce that value to 0, so that users could access those features.

Unfortunately, his results have shown the opposite. A value of 0 is not safe at this point, which has led to calls to further restrict `perf_event_open()`. That is not at all what he wanted to see, he said.

`perf_event_open()` has a complicated interface that involves a structure with lots of entries that can interact in different ways. The man page is not that great, which he can say since he wrote it. `perf_event_open()` also interacts with other system calls.

At this point, all of the low-hanging fruit has been caught in `perf_event_open()`. Other fuzzers, such as Trinity and syzkaller don't find any problems, but perf_fuzzer still does. The perf subsystem has "tentacles all over the kernel", it is entangled with the proc filesystem, sysfs,

`fork()`, BPF, tracepoints, and more. It also generates a lot of non-maskable interrupts (NMIs). Beyond that are the hardware interactions that make it hard to have deterministic results. Things like cache misses are not deterministic at the hardware level, for example. Also, performance counters cannot be virtualized.

Over the previous two weeks, he had run a bunch of tests with perf_fuzzer, which was rather tedious to do, Weaver said. He wanted to run with a recent kernel from Git, which is easy to do on x86, but not so easy for ARM. He has difficulty getting Git kernels to run on his boards.

When only using the facilities available with a paranoid value of 2, a Pentium 4 will crash in eight ~~hours~~ minutes, while a Core 2 will run for a week without any problems. Others, such as Haswell, will crash in two or three days. A Raspberry Pi will run for more than a week without problems, but that may be due to how slow the processor is.

Most of the bugs are either RCU or NMI stalls and it is not clear what the underlying problem actually is. Sometimes the systems lock up with nothing on the serial console. Many of the problems are not reproducible either.

For a paranoid value of 1, the Core 2 will crash in one day, while a Haswell will crash in two hours. At a paranoid level of 0, the Core 2 takes 21 hours to crash, and the Haswell will crash in one or two hours. He has run into something he suspects is a hardware bug on Haswell as well. A paranoid value of -1 (no restrictions) is "not recommended", he said. So, after years of work finding and fixing bugs in `perf_event_open()`, you can still crash a machine by running perf_fuzzer long enough. Those crashes will be hard to debug, which is frustrating.

Weaver likes the idea of a more formalized specification for system calls and other user-space APIs. It would be nice to have a way to find out about new things added to the interfaces and it is great to see more people get interested in these problems. He has tried getting funding and in having his research published without much success. In addition, perf_fuzzer is being used to find bugs and collect bug bounties, but he rarely gets any code contributions from others—or even hears from them at all.

For a long time, work on improving perf_fuzzer has been stalled because it was so easy to crash machines with it. There simply has been no impetus to add more features. He would like to add support for control groups and BPF to the fuzzer, possibly by reusing the Trinity BPF fuzzing code. He finished by noting that he has a paper [PDF] on his web site that goes into some detail on how perf_fuzzer operates.

Vyukov wondered if Weaver had tried using the KernelAddressSanitizer (KASAN) while fuzzing with perf_fuzzer. Weaver said that he had not done so, but there was no real reason for that, so it is probably worth trying. Vyukov also asked what the advantage was for a specialized fuzzer and perf developer Peter Zijlstra quickly spoke up to say that perf_fuzzer is still finding bugs, while syzkaller can't crash perf any longer.

Part of the difference may be that syzkaller runs in a virtual machine, normally, while perf_fuzzer runs on bare metal. Weaver also pointed out that `perf_event_open()` has specialized needs for things like its `mmap()` region that Trinity and syzkaller may not be creating in the right way because they lack the specialized knowledge of the interface that perf_fuzzer has. Jones suggested that both Trinity and perf_fuzzer might benefit from some coverage guidance to help in reproducing bugs when they are encountered.

That's where the conversation wound down, but it is clear there are several areas of collaboration that could, and seemingly will, be pursued. The system-call (or user-space API) specification gained quite a bit of traction in the microconference; we will have to see where that ends up. Meanwhile, these three fuzzers (and perhaps others that might crop up) can benefit from each others' work—which should hopefully lead to more kernel bugs being found and fixed.

[Thanks to LWN subscribers for supporting my travel to Santa Fe for LPC.]

(Log in to post comments)

A trio of fuzzers
Posted Nov 10, 2016 1:29 UTC (Thu) by **roc** (subscriber, #30627) [Link]

> Also, performance counters cannot be virtualized.

They certainly can. E.g. KVM with cpu="host-passthrough" gives virtualized access to the x86 PMU just fine on modern CPUs.

A trio of fuzzers
Posted Nov 10, 2016 1:30 UTC (Thu) by **roc** (subscriber, #30627) [Link]

BTW it works fine with VMWare too. Xen HVM nominally has support but it needs work.

A trio of fuzzers
Posted Nov 10, 2016 3:13 UTC (Thu) by **deater** (subscriber, #11746) [Link]

I am pretty sure that while kvm supports a handful of virtualized events, it it no way provides direct access to the bare-metal performance monitoring unit MSRs. Nor does it allow access to the much more complicated PMUs (uncore, BTS, processor trace) that are responsible for many of the current perf_fuzzer issues.

I'd be glad to be proved wrong, but I don't see anything in arch/x86/kvm/pmu_intel.c

A trio of fuzzers
Posted Nov 10, 2016 20:03 UTC (Thu) by **roc** (subscriber, #30627) [Link]

OK, thanks for clarifying that. But the blanket statement "performance counters cannot be virtualized" was incorrect.

A trio of fuzzers
Posted Nov 10, 2016 6:09 UTC (Thu) by **deater** (subscriber, #11746) [Link]

A minor point, but Pentium 4 crashed in 8 minutes, not 8 hours.

It's hard to get people excited about fixing Pentium 4 issues these days, partly because it's mostly obsolete, but also because the Pentium 4 performance monitoring unit is so weird compared to other Intel PMUs.

Using KASAN
Posted Nov 14, 2016 22:14 UTC (Mon) by **deater** (subscriber, #11746) [Link]

I configured KASAN as suggested and indeed perf_fuzzer triggers an issue within minutes. It's possibly even the annoying memory corruption bug that's been eluding us for months. Here's hoping this will let us finally fix it.