# Fuzzing perf_events



**August 5, 2015**
This article was contributed by Vince Weaver

You might be surprised to learn that starting with Linux 2.6.31 (in 2009) it has been rather easy to crash the Linux kernel. This date marks the introduction of the perf_event subsystem. It is likely that perf_event is not any more prone to errors than any other large kernel subsystem, but it has the distinction of being subjected to intense testing from the perf_fuzzer tool, which methodically probes the interface for bugs. Given enough time, the tool will trigger a crash (and often fill the system logs with warnings along the way).

For the past two years, an effort has been underway to fix every bug found by the perf_fuzzer. Progress has been made, thanks to the perf_event maintainers (especially Peter Zijlstra), who have been extremely patient and helpful in fixing the errors reported. The tool now takes hours to days before a crash occurs; previously crashes would happen within seconds. Most of the low-hanging fruit has been found, with the remaining problems tending toward complex race conditions that are hard to replicate, isolate, and fix.

## Fuzzer background

A fuzzer is a tool that tests an interface by generating almost (but not quite) valid inputs with the intention of finding corner-cases that trigger unexpected behavior. The term dates back to 1988 when Barton Miller noticed that Unix utilities would die horribly when fed unexpected characters by his noisy ("fuzzy") modem connection. Since then, much work has been done on fuzzers, mostly by security researchers looking for exploitable bugs in programs.

The most common use case is to fuzz user-space programs, but work has been done on fuzzing operating system kernels. Various Linux kernel fuzzers are available (such as Tavis Ormandy's iknowthis) but by far the most well-known is Dave Jones's Trinity (see this conference report and this discussion of memory-management fuzzing for more information on Trinity). There are likely other kernel fuzzers out there; undoubtedly security researchers have developed internal ones that are not publicly available. What separates the perf_fuzzer from other projects is that it investigates not the entire kernel interface, but only that of a *single* system call.

## Motivation

So why write a perf_event-specific fuzzer? I have spent much time in the performance-analysis area of High Performance Computing, specifically the maintenance of the cross-platform PAPI (Performance API) performance measurement library. The sudden appearance of the perf_event interface in 2009 (of which I have been diplomatically reported as being "sometimes critical") led to major code changes in the PAPI project. As an outside project, PAPI has faced many compatibility issues and ABI breakage due to kernel changes, issues the kernel developers tend not to notice due to the primary consumer of the interface (the `perf` tool) being shipped in sync with the kernel tree.

The need to keep up with the high rate of changes in the perf_event interface led me to undertake many activities designed to understand the interface, including writing the perf_event_open(2) man page as well as creating a series of validation tests. While these tests continue to find bugs, they tend to be reactionary—added after the fact (and often too late to be fixed before the new behavior has become ABI).

As Trinity rose in visibility, I sent some basic patches to improve perf_event support, enough that it had a much higher chance of generating valid events. Those patches were there for a few years until suddenly they became famous in 2013 due to the release of a dangerous kernel exploit. After this wakeup call, I wondered what other issues might be lurking in the interface. This led to more improvements contributed to Trinity, followed by the development of a separate, system-call-specific fuzzer.

## The perf_event interface

The perf_event interface is complex, mirroring the complicated nature of the underlying hardware performance interfaces. The primary entry point is the `perf_event_open()` system call, which takes five arguments. The first argument is a structure with over forty different parameters that interact in complex ways. The system call returns one file descriptor per event opened; these file descriptors can be manipulated by a number of other system calls:

- `prctl()` — can enable or disable all active events in a process
- `ioctl()` — is used to control events, including starting, stopping, and adjusting existing events
- `read()` — can be used to read event values
- `mmap()` — can be used to set up a primary and an auxiliary ring buffer capable of buffering event results
- `poll()` — can be used to wait until buffers need to be read

In addition to system calls, there are other ways to interact with the infrastructure. Some CPUs support reading event values from user space without entering the kernel (on x86 this is the `rdpmc` instruction). Events can be configured to send a signal on overflow after a pre-determined count or if an `mmap()` buffer is full. There are also a large number of files under the `/sys` and `/proc` filesystems that affect the interface.

Much of the trickiness of the perf_event interface is due to the way it interacts with deep kernel internals. There are software events that count performance-critical operations such as context switches. The overflow interrupts are handled as non-maskable interrupts (NMIs) on many systems, which are often difficult to set up and tricky to debug. All user breakpoint support is handled by perf_event, and there are also interfaces for creating ftrace events and attaching eBPF filters. The wide scope of the perf_event code allows more chances for code to go wrong.

For the full details on the interface, you can read the manual page. In a testament to the complexity involved, this man page is the longest system-call man page, even longer than the one for `ptrace()`.

### How the fuzzer works

The perf_fuzzer program is just an infinite loop that randomly picks a piece of the interface to test each time through. Inputs are also picked randomly; while in theory using completely random inputs would eventually explore the entire search space, in practice this would take quite a long time. In order to more easily trigger bugs, inside knowledge of the interface is used to generate valid and mostly-valid event parameters in addition to purely random ones.

The following actions are chosen from at random:

- Open a random event — this code is shared in common with Trinity. The fuzzer repeats until a valid event is created.
- `mmap()` an event — attempts are made to attach regular and auxiliary `mmap()` buffers to an event.
- `sigaction()` an event — set up an overflow signal handler.
- Close an event.
- Call `ioctl()` on an event — this includes starting, stopping, refreshing, changing the period, attaching an ftrace filter, and attaching an eBPF program.
- `prctl()` the process — this will enable or disable all events.
- Read from an event.
- Access a `/proc` or `/sys` file.
- Fork the process — currently only one child process is allowed at a time to avoid fork bombs and file descriptor exhaustion.
- Poll an event.
- Doing nothing (to allow some variations in timing).

The difference between this tool and Trinity is that the latter simply attempts to open an event. In the rare occasion it generates a valid event, all further actions on the file descriptor are completely random. perf_fuzzer uses knowledge of the interface to pick the system calls known to be relevant and targets them, allowing bugs to be found much faster than a completely random search would.

### Using the fuzzer

Running the fuzzer is straightforward. First check out the git tree:

```
git clone http://www.github.com/deater/perf_event_tests.git
```

Enter the fuzzer directory, run `make`, and the `perf_fuzzer` binary will be generated. You can simply run that binary, but it typically will error out at some point due to some as-of-yet unexplained weirdness with signal handling. A better option is to run the included `fast_repro99.sh` script, which will repeatedly run the fuzzer for short amounts of time.

Before running, you probably want to configure your system to make error reporting easy. This usually involves setting up a serial console, as bugs are often hard crashes that cannot be captured easily otherwise. Once you have started the fuzzing it's just a matter of waiting. The fuzzer will print regular updates on its fuzzing status, such as:

```
Iteration 10000
    Open attempts: 144277  Successful: 920  Currently open: 47
          EPERM : 24
          ENOENT : 783
          E2BIG : 12577
          EBADF : 12118
          EACCES : 588
          EBUSY : 6
          EINVAL : 116933
          EOPNOTSUPP : 328
          Type (Hardware 200/19733)(software 333/20972)\
               (tracepoint 67/20782)(Cache 60/18015)\
               (cpu 210/20976)(breakpoint 22/20669)\
               (power 0/2625)(intel_bts 28/2456)\
               (uncore_imc 0/2733)(#9 0/17)(#10 0/9)\
               (#11 0/8)(#12 0/10)(#13 0/6)(#14 0/6)\
               (>14 0/15260)
Close:  873/873 Successful
Read:   744/844 Successful
Write:  0/887 Successful
Ioctl:  328/873 Successful: (ENABLE 76/76)(DISABLE 82/82)\
                            (REFRESH 6/104)(RESET 70/70)\
                            (PERIOD 5/84)(SET_OUTPUT 7/95)\
                            (SET_FILTER 2/108)(ID 80/80)\
                            (SET_BPF 0/84)(#9 0/0)(#10 0/0)\
                            (#11 0/0)(#12 0/0)(#13 0/0)\
                            (#14 0/0)(>14 0/90)
Mmap:   434/1063 Successful: (MMAP 434/1063)(TRASH 95/146)\
                             (READ 21/127)(UNMAP 428/1046)\
                             (AUX 0/251)(AUX_READ 3/14)
Prctl:  880/880 Successful
Fork:   450/450 Successful
Poll:   820/897 Successful
Access: 452/915 Successful
```

```
Overflows: 0
SIGIOs due to RT signal queue full: 0
```

If you do find a bug, please report it to the linux-kernel mailing list. It's helpful if you can isolate it first. At startup, the tool will output the command line (including the random number seed) to use to try to repeat the behavior. There are a large number of sources of non-determinism in performance-measurement code, but the perf_fuzzer tries to avoid as many of these as possible. It is often (but not always) possible to regenerate a bug by re-running the fuzzer with the same command line.

The fuzzer does support dumping system call traces, which can be fed into provided tools that both replay the traces as well as generate sample test code for repeating the problem. In the early days of the fuzzer, these tools were incredibly valuable. However, most bugs found these days tend to be obscure race conditions that are difficult to capture in this way, as the act of logging to disk disrupts timing enough to not trigger the bug.

### Bugs found

A large number of issues have been turned up by the fuzzer: some are simply warnings, some cause denial of service (i.e. crashes), and at least one has been a root exploit.

All of these bugs can be problematic, as currently it is not possible to completely disable the perf_event interface, either at run time or compile time. Currently perf_event support must be built into the kernel, there is no possibility of making it a loadable module or disabling it during the build process. There have been some efforts to change this, but it is complicated by the fact that hardware breakpoint support depends on the presence of perf_event. In addition, while it is possible to disable some of the perf_event interface at runtime via `/proc/sys/kernel/perf_event_paranoid`, even at the most paranoid level it is still possible to create user-space measurement events.

A full list of all bugs found can be found on the project's website. Included below is a list of crashing bugs that have been found and *fixed* since the introduction of the tool; many more are still under investigation.

| Type | CVE | Fixed in Linux | | Description |
|---|---|---|---|---|
| crash | - | 3.10 | 9bb5d40cd93c9dd4 | `mmap()` accounting hole |
| crash | - | 3.10 | 26cb63ad11e04047 | `mmap()` double free |
| panic | - | 3.11 | d9f966357b14e356 | ARM array out of bounds |
| root exploit | CVE-2013-4254 | 3.11 | c95eb3184ea1a3a2 | ARM event validation |
| panic | - | 3.11 | 868f6fea8fa63f09 | ARM64 array out of bounds |
| panic | - | 3.11 | ee7538a008a45050 | ARM64 event validation |
| panic | - | 3.13 | 6e22f8f2e8d81dca | Alpha array out-of-bounds |
| crash | CVE-2013-2930 | 3.13 | 12ae030d54ef2507 | perf/ftrace wrong permissions check |
| crash | - | 3.14 | 0ac09f9f8cd1fb02 | page fault ftrace cr2 corruption |
| crash | - | 3.15 | 46ce0fe97a6be753 | race when removing event |
| crash | - | 3.15 | ffb4ef21ac4308c2 | function cannot handle NULL return |
| reboot | - | 3.17 | 3577af70a2ce4853 | race in `perf_remove_from_context()` |
| crash | - | 3.19 | 98b008dff8452653 | misplaced parenthesis in `rapl_scale()` |
| crash | - | 3.19 | c3c87e770458aa00 | fix the grouping condition |
| crash | - | 3.19 | a83fe28e2e453924 | Fix `put_event()` ctx lock |
| crash | - | 3.19 | af91568e762d0493 | IVB-EP uncore assign events |
| crash | - | 4.0 | d525211f9d1be8b5 | Fix `perf_callchain()` hang |
| crash | - | 4.1 | 8fff105e13041e49 | arm64/arm reject groups spanning PMUs |
| crash | - | 4.1 | 15c1247953e8a452 | `snb_uncore_imc_event_start()` crash |
| crash | - | 4.2 | 57ffc5ca679f499f | Fix AUX buffer refcounting |

### Sample bug

An example bug found by the fuzzer was the one that eventually lead to CVE-2013-4254. A routine fuzzing on a 32-bit ARM Pandaboard triggered a kernel panic. In this particular bug, the ARM `validate_event()` function called `armpmu->get_event_idx()` on the group leader for an event. However, if the group leader was not an `armpmu` type, but a PMU type with a shorter structure size (such as a software event), then the function pointer being called was whatever arbitrary value happened to be at that memory offset, which was past the end of the structure. Normally, this would just cause a panic and crash but, if you were unlucky, this arbitrary value was a valid user address. For a short window of time in the 3.11-rc development cycle, this value pointed to a location initialized to `INT_MIN` which is a valid user mappable address of `0x80000000`. If a user mapped exploit code there, this bug could lead to privilege escalation, and some code was put together that did just that. Luckily, this bug was found and fixed before it made it into any released kernel.

### Current and future work

Work on the fuzzer has been progressing. Now that bugs are harder to trigger there has been time available to enhance the functionality rather than spending the (sometimes considerable) time needed to push a bug through to a fix in the kernel tree.

Recent work includes adding support for testing new features (such as the introduction of the AUX ring buffer for high-volume events) as well as a push to fully test all of the perf_event `ioctl()` calls. Support for `ioctl()` is more complex than it sounds, as fuzzing it involved creating an ftrace filter generator (and the ftrace filter language is lacking documentation), as well as some method of fuzzing the new eBPF language. The eBPF fuzzing is currently stalled; even though there is existing code for generating eBPF programs in Trinity, it turns out you need to be root to actually attach such programs, and running perf_fuzzer as root is not advised. It might be worthwhile to harden the fuzzer so that it can run as root; part of the problem is there are various "expected" ways you can crash your machine as root. This includes setting the overflow threshold so low that the machine gets stuck in an interrupt storm, as well as problems where the ftrace function tracer can get stuck recursively trying to trace interrupt handler functions.

While I plan to continue to work on this project as long as it finds bugs, it is important to note that it is a volunteer effort with no support and few outside contributors. This can be frustrating at times, as noted by Dave Jones's recent comments regarding the Trinity fuzzer. I have encountered problems similar to his, where finding and fixing kernel bugs is appreciated, but the work is not considered important enough to get employer support. This work is also not considered groundbreaking enough to easily lead to publications or grants through traditional venues, which is needed by academics like myself. With the demise of the Ottawa Linux Symposium, there are also no longer any peer-reviewed conferences with proceedings for Linux, which are critical for academics trying to publish Linux-related work in a way that matters to their home institutions.

One area with room for improvement is the amount of testing that is done using perf_fuzzer. The current primary testing environment is a horribly abused Haswell desktop. Tests are also run on Intel Core 2, Pentium 4, and AMD Jaguar machines when convenient. It would be nice to test on other architectures; there is probably low-hanging fruit for the untested ones. Power is notable as an architecture with elaborate perf_event support but, as of yet, has not been tested. I have many ARM machines and try to test on those when possible. This is complicated because, when fuzzing, you want to run an up-to-the-minute mainline kernel (to avoid hitting older, known bugs), but it can be extremely time-consuming if not impossible to run mainline kernels on arbitrary ARM development boards.

One final area of future work would be to extend this to other system calls. There are many others out there that are currently not well-tested. Even if a custom tool is not designed, it would be worthwhile to extend Trinity with the lessons learned from perf_fuzzer. While fuzzing is not the only route to ensuring a secure kernel environment, it is definitely a powerful tool in the Linux security toolbox.

---

(Log in to post comments)

**Fuzzing perf_events**
Posted Aug 5, 2015 12:52 UTC (Wed) by **wildea01** (subscriber, #71011) [Link]

> Power is notable as an architecture with elaborate perf_event support
> but, as of yet, has not been tested.

I tested ppc64 a while back (3.11) and it fell over on my G5:

http://www.willdeacon.ukfsn.org/bitbucket/oopsen/ppc64_oo...

That's one of the "problems" with fuzzing: it's a great way to crash or exploit the system, but the bug reports can be really hard work to nail down.

**Fuzzing perf_events**
Posted Aug 5, 2015 14:21 UTC (Wed) by **deater** (subscriber, #11746) [Link]

> That's one of the "problems" with fuzzing: it's a great way to crash or
> exploit the system, but the bug reports can be really hard work to nail down.

Yes, I think my fuzzer work has been 1% writing code, 4% running the fuzzer, and 95% pushing on the bugs until the fixes make it into the kernel tree.

I really would like to get a modern Power machine to run tests on locally (as it's usually impolite to fuzz on someone else's server unless they are really nice and don't mind rebooting all the time). However Power hardware is a bit difficult to source and the price tends to be high enough to trigger extra paperwork from the purchasing department so I haven't gotten around to it yet. Maybe I should try to resurrect the old G4 laptop that was having severe power-connector issues.

**Fuzzing perf_events**
Posted Aug 5, 2015 19:20 UTC (Wed) by **gebi** (subscriber, #59940) [Link]

With open power at least purchase cost is not the big issue anymore.
Maybe this can help you: http://www.tyan.com/solutions/tyan_openpower_system.html

**Fuzzing perf_events**
Posted Aug 5, 2015 20:38 UTC (Wed) by **kleptog** (subscriber, #1183) [Link]

> I really would like to get a modern Power machine to run tests on locally (as it's usually impolite to fuzz on someone else's server unless they are really nice and don't mind rebooting all the time).

Isn't it possible to running the fuzzing under something like User Mode Linux, which would turn the machine crashing into some segfault which could be trapped by a debugger? Or some kind of virtualisation? Or are the bugs mostly race conditions that aren't accurately replicated this way?

**Fuzzing perf_events**
Posted Aug 6, 2015 9:48 UTC (Thu) by **MarkRutland** (subscriber, #74197) [Link]

You could certainly catch some bugs that way (those due to core code and/or SW PMU drivers), but you wouldn't be able to hit bugs which rely on HW PMU drivers.

Of the 20 bugs listed in the article, at least 8 require a HW PMU driver to be active to trigger.

**Fuzzing perf_events**
Posted Aug 5, 2015 23:57 UTC (Wed) by **PaulMcKenney** (subscriber, #9624) [Link]

Or you could request use of the OpenPOWER systems at Oregon State University's Open Source Lab:
http://osuosl.org/services/powerdev/request_hosting

**Fuzzing perf_events**
Posted Aug 6, 2015 13:08 UTC (Thu) by **deater** (subscriber, #11746) [Link]

> Or you could request use of the OpenPOWER systems at Oregon State
> University's Open Source Lab

Do testdrives like that allow installing custom git kernels, rebooting if there's a crash, and capturing serial console output?

Maybe the answer is yes, especially with how Power machines seem to always have a VM-like layer between the hardware and the OS. Otherwise though fuzzing tends to be a bad idea on systems you don't have local access or root permissions.

**Fuzzing perf_events**
Posted Aug 8, 2015 16:22 UTC (Sat) by **PaulMcKenney** (subscriber, #9624) [Link]

I suggest filling out the form, stating clearly what you need to do, and see what their response is.

**Fuzzing perf_events**
Posted Aug 6, 2015 13:15 UTC (Thu) by **tdaitx** (subscriber, #56964) [Link]

And there is the Mini Cloud at Unicamp: http://openpower.ic.unicamp.br/minicloud/index.html

"The MiniCloud provides free access to Power® virtual machines. It allows easy access to an environment that can be configured for development, testing or migration of applications to Power. The virtual machines of MiniCloud run on PowerKVM™, which supports running a large number of virtual machines on a single scale-out Linux server."

**Fuzzing perf_events**
Posted Aug 7, 2015 16:23 UTC (Fri) by **flussence** (subscriber, #85566) [Link]

As a non-kernel-hacker, am I likely to have any software that uses perf_events? If not, I'll look forward to the day when I can yank it out of my kconfig...

**Fuzzing perf_events**
Posted Aug 7, 2015 16:44 UTC (Fri) by **spender** (subscriber, #23067) [Link]

No. We've been disabling it for unprivileged users since 2013 in grsecurity and haven't received a single complaint yet.

-Brad

**Fuzzing perf_events**
Posted Aug 11, 2015 4:02 UTC (Tue) by **graydon** (guest, #5009) [Link]

I am not a kernel hacker and I use perf about a dozen times a day. It's the best profiler I've ever worked with. But YMMV.

**Fuzzing perf_events**
Posted Aug 11, 2015 11:23 UTC (Tue) by **andresfreund** (subscriber, #69562) [Link]

Yea, agreed. It's far from perfect, but it's pretty unbeatable in how quickly you can get a fairly accurate picture of performance problems.

My biggest annoyance is that call graphs are only usable if you a) recompile software using frame pointers b) use dwarf mode which generates huge data files and isn't available on many existing systems c) use lbr mode which requires a new cpu, kernel and perf version... I hate whoever made the decision to default gcc to not using frame pointers in newer gcc versions. I kinda see the point on x86-32 which is really register starved, but it's not that bad on 64bits and it makes debugging and profiling so much harder.
Second to that comes the far too generic and thus hard to search name ;)