UMaine VMW Group Tech Report UMAINE-VMW-TR-FUZZER-UPDATE-2016-11, November 2016

perf_fuzzer November 2016 Status Update

Vincent M. Weaver
University of Maine
Electrical and Computer Engineering
vincent.weaver@maine.edu

Abstract

The perf_fuzzer utility attempts to find bugs in the Linux kernel by creating targeted, random, inputs to various interfaces relating to the `perf_event_open()` system call. These random, yet almost correct, inputs are designed to find corner-cases in the operating system interface that have been overlooked and may lead to warnings, errors, crashes, or worse. The fuzzer has found numerous bugs over the years, many of which have since been fixed in the upstream Linux kernel. Despite these fixes, the fuzzer is still capable of crashing Linux.

Most of the easy-to-fix bugs were fixed years ago, and newly introduced trivial bugs are often also found by other more generic fuzzers (such as trinity and syzkaller). The bugs found by perf_fuzzer tend to be obscure, hard-to-reproduce concurrency issues deep inside the kernel.

This document is an update on the current status of the fuzzer, as well as an overview of the bugs being found, and attempts to fix them.

A brief overview of this paper was given at the 2016 Linux Plumbers Conference Fuzzing Microconference.

1 Introduction

The perf_fuzzer [11, 10] utility creates targeted, random, inputs designed to exercise the interface surrounding the Linux `perf_event_open()` system call [9].

More generic Linux system call fuzzers exist, such as trinity [5] and syzkaller [8]. These typically try to exercise all possible system calls at once more or less randomly. In contrast, the `perf_fuzzer` has knowledge of other system calls that provide inputs to the fuzzer, as well as system calls known to operate on file descriptors returned by the call. This includes complex interactions with `mmap()`, `ioctl()`, `poll()`, `prctl()`, `read()`, and `signal()`. It is unlikely that a generic fuzzer would set up valid interactions among all of these system calls by random chance.

2 Motivation

The primary motivation for the design of the fuzzer was to reassure sysadmins of high-performance computing (HPC) systems that it was safe to allow users full access to the `perf_event_open()` syscall (via the `perf_event_open()` interface. Sysadmins tend to be paranoid, so by default they would want to completely disable access if there were any threat at all (this is difficult currently as perf cannot be disabled in any meaningful way by default, although some distributions apply custom patches to allow it to be disabled).

There is the `perf_event_paranoid` interface, and by default it has a restrictive setting (“2”) enabling only user-only events. Detailed performance results, especially system-wide per-cpu results, work best with the more permissive (“0”). How can we show this is safe?

Most users of the `perf_event` interface, including the kernel developers, mostly use the bundled `perf` tool. Any interfaces not used by `perf` are essentially untested. I work with the PAPI [6] performance library which uses the perf_event_open interface in other ways, and in fact for a while the PAPI validation tests would crash the kernel.

This incident, along with a root exploit found by trinity (due to perf_event code I had contributed) led me to develop the perf_fuzzer in order to more thoroughly test the interface.

Unfortunately the fuzzer has had the opposite result that I intended; instead of giving confidence that perf_event is safe, it has led for a push to disable the interface altogether [3].

3 Finding Bugs

The perf_fuzzer has found numerous bugs over the years, many of which have been fixed in the upstream Linux kernel. Despite years of effort, it is still possible to crash the Linux kernel with the fuzzer.

Most easy-to-fix bugs were fixed years ago. There are often a number of bugs found the first time perf_fuzzer is found on a new architecture, but after that it levels off, and newly introduced bugs are found quickly as many groups use perf_fuzzer as a test before submitting patches.

What is left to be found are long, insidious lurking bugs that can take days to week to find, and are often more or less impossible to replicate. I try to report them, and kernel developers (especially Peter Zijlstra) go through great trouble trying to reproduce and fix them. It's a losing battle though for a variety of reasons.

- Determinism – most remaining bugs are not deterministic. Even with best attempts to guarantee playing back the same syscall order cannot always reproduce.
- Bugs last past end of program. Either through corrupting data structures, use-after free, or weird RCU issues. So initial trigger might be long gone.
- It takes long enough to generate a crash (days) that logging and replaying is not a valid solution.
- Turning on various kernel debug features like lockdep, etc. SLAB poison helps a bit. Nothing else really has turned up much (low hanging fruit already gone).
- ftrace – It is often recommended to enable ftrace and gather traces. This takes enormous amounts of time and disk space and has rarely aided in finding bugs.

Because it has not been possible to completely eliminate bugs, kernel bisection is usually not possible to try to track down the causes of bugs. This has also hindered further development of perf_fuzzer; it works so well now I have had less reason to enhance the fuzzer.

4 Current Bug Status

I try to keep a full list of bugs I find here:
`http://web.eece.maine.edu/~vweaver/projects/perf_events/fuzzer/bugs_found.html`

It is hard to keep up to date, as due to the random nature of the fuzzer it can be unclear what kernel commit, if any, might have fixed a particular bug.

For this report I have run the fuzzer a number of times on a number of machines and reported the time it took to crash the machine and what WARNing and errors happened along the way.

Table 1: Systems fuzzed in this paper.

Architecture	name	type
x86_64	a10	AMD a10
x86_64	core2	Intel Core2
x86_64	p4	Intel Pentium 4
x86_64	skylake	Intel Skylake
x86_64	haswell	Intel Haswell
arm	pi2	Raspberry Pi 2
sparc	spacr64	Sun Ultrasparc

4.1 Experimental Setup

I have a few machines that I consistently run the fuzzer on. I try for a wide variety of machines, but this is limited because I cannot use the systems for general work if there's a chance the fuzzer might crash it. There's also other issues such as noise (the fuzzer can trigger fans), getting debug messages (usually you need to have a serial console to another machine), and the ability to reboot the machine if it crashes. (The haswell machine has a bad habit of crashing with the Ethernet interrupt stuck on and will take down the entire Ethernet switch when it goes).

Ideally the systems will be running the latest linus-git kernel; otherwise you end up chasing known and fixed bugs. Also the kernel devels aren't excited about chasing bugs in old kernels. This is not much of an issue on x86 hardware, but can be difficult for architectures such as ARM. I currently have at least 17 different ARM boards [1] but I have only fuzzed on Raspberry Pi and Pandaboard boards because the rest of the systems would take too much time to get an upstream kernel running on.

The systems actually fuzzed in this report are shown in Figure 1.

I fuzz with the current version of the perf_fuzzer using the somewhat arbitrarily named helper script `./fast_repro99.sh`. Instead of letting the fuzzer run indefinitely (which would lead to impossibly large log files if trying to reproduce) the script only runs for a few tens of thousands of syscalls before restarting. In theory this would make reproducing things faster, although with the low reproducibility of current bugs this might not be a useful distinction.

These results are from the 4.9-rc0 kernel just before the 4.9-rc1 release. I did not test 4.9-rc1 proper because some sort of change broke the boot on my machines and bisecting was inconclusive.

The Linux kernel has a mechanism for limiting the perf_event interface, the `/proc/sys/kernel/perf_event_paranoid` file. Values of -1 mean unrestricted, 0 means allow per-cpu system wide data, 1 means allow both kernel and user measurements, and 2 means only allow restricted userspace measurements.

Table 2: Fuzzer results at paranoid level 2 (user only).

machine	warnings	time to crash	kernel
p4	1	7m49s	4.9-rc0
core2	1	n/a (7days+)	4.9-rc0
haswell	1	3d9h26m	4.9-rc0
skylake	1	7d8h37m	4.9-rc0
a10	1	2d	4.9-rc0
sparc	0	30s	3.2
pi2	?	n/a	4.8?

There is a proposal to add a level of "3" to totally disable events, but the perf_event developers have fought this (though various distributions are maintaining the patch anyway) [3]. It is actually currently not possible to disable perf_event, either at compile time or at run time, which is seen as a security issue.

I fuzz at all the various levels in both an attempt to see the safety the levels, as well as to narrow down the cause of some of the bugs.

5 Fuzzing Results

The results are split up by paranoid level.

There are a few common warnings that show up at all levels. The first is the warning when the NMI interrupt took too much time leading to throttling:

```
perf: interrupt took too long (3152 > 3135),
lowering kernel.perf_event_max_sample_rate
to 63250
```

Another is a WARNING in the breakpoint code which is reproducible and might be an actual bug but no one has bothered to chase it down. `WARNING: CPU: 0 PID: 24577 at arch/x86/kernel/hw_breakpoint.c:121 Can't find any breakpoint slot`

5.1 Fuzzing Paranoid Level 2

Paranoid level 2 should be the safest level, and you would not expect to be able to crash at this level. We can and do, as shown in the summary in Table 2.

- Sparc64 locks up quickly with `BUG: Bad page state in process kworker/0:2 pfn:0db6e` my kernel is so old (3.2) that this might have been fixed since. It is hard to get newer kernels on this machine.
- ARM: the raspberry Pi2 seems to not find any bugs. It has a very simple and straightforward interface though and the developers are known to run the fuzzer themeselves so maybe it is clean. It's a lot slower so it would take longer to hit the same number of syscalls tested.
- The Pentium 4 warned first with `WARNING: CPU: 0 PID: 456 at arch/x86/events/core.c:1248 x86_pmu_start+0xae/0x100` and then crashes pretty quickly with `BAD LUCK: lost 60330898 message(s) from NMI context!` It's arguable whether anyone cares about Pentium 4 anymore.
- The Core2 machine generates many warnings but manages to stay up for over a week. It hits the breakpoint warning. Also generates many `Uhhuh. NMI received for unknown reason 2d on CPU 1. Do you have a strange power saving mode enabled? Dazed and confused, but trying to continue.` messages.
- The Haswell machine has a few warnings and manages to crash spectacularly. No message, only `[293191.486147] ------------[ cut here ]------------` in the log.
- The skylake machine managed to hang too. It first hit the breakpoint warning. `INFO: rcu_sched self-detected stall on CPU 3-...: (5249 ticks this GP) idle=54b/140000000000001/0 softirq=42134142/42134142 fqs=0 (t=5250 jiffies g=60776826 c=60776825 q=84) rcu_sched kthread starved for 5250 jiffies! g60776826 c60776825 f0x0 RCU_GP_WAIT_FQS(3) ->state=0x1` Seems to be the dreaded `perf_cgroup_attach+0x70/0x70` issue.
- The a10 machine took longer but eventually crashed, and before that was in a state where it wasn't quite crashed but processes would hang for arbitrary amounts of time.

  ```
  WARNING: CPU: 2 PID: 29960 at
  arch/x86/kernel/hw_breakpoint.c:121
  Can't find any breakpoint slot
  ```

  Crashed pretty hard but somehow the serial console locked up so it was not possible to get a full recording of the initial error. Did get: `NMI watchdog: BUG: soft lockup - CPU#0 stuck for 22s! [perf_fuzzer:9306]`

5.2 Fuzzing Paranoid Level 1

Paranoid level 1 also enables kernel events. Crashing tends to happen more quickly here, as seen in Table 3.

Table 3: Fuzzer results at paranoid level 1 (user+kernel).

machine	warnings	time to crash	kernel
core2	2	1d15h20m	4.9-rc0
haswell	0	21h25m	4.9-rc0
skylake	?	n/a (5d+)	4.9-rc0
a10	?	2h15m	4.9-rc0

- Sparc64 – didn't bother testing, only old kernel available so uninteresting
- ARM – didn't have time to test (pi2 board used in another project)
- Pentium 4 – didn't bother testing, it crashed so quickly and is not common anymore
- Core2 – hit the breakpoint warning. Hit this: `WARNING: CPU: 1 PID: 19400 at kernel/events/ring_buffer.c:546 __rb_free_aux+0x40/0xe8 WARNING: CPU: 0 PID: 18824 at kernel/events/core.c:4998 perf_mmap_close+0x151/0x252`. Eventually locked up with no message.
- Haswell Got this warning: `WARNING: CPU: 0 PID: 19729 at kernel/events/core.c:4998 perf_mmap_close+0x386/0x390` Then kept chugging past at least one stall: `INFO: rcu_sched detected stalls on CPUs/tasks: 2-...: (0 ticks this GP) idle=8ed/140000000000000/0 softirq=1096112/1096112 fqs=225 (detected by 6, t=5559 jiffies, g=1304654, c=1304653, q=181)`

  Remarkably recovered, but then `INFO: rcu_sched detected stalls on CPUs/tasks: 4-...: (1 GPs behind) idle=23b/140000000000000/2 softirq=5456919` with hard crash at `NMI watchdog: Watchdog detected hard LOCKUP on cpu 5` with not much debug info.
- skylake – First this warning: `WARNING: CPU: 5 PID: 25876 at kernel/events/core.c:4998 perf_mmap_close+0x2dd/0x2f0` Then lots of stalls `INFO: rcu_sched detected stalls on CPUs/tasks: 1-...: (1 GPs behind) idle=e53/140000000000001/2 softirq=17327005/17327006 fqs=1672 (detected by 0, t=5286 jiffies, g=24835965, c=24835964, q=246)` Including a big one that shut off the timekeeping source: `clocksource: timekeeping watchdog on CPU1: Marking clocksource 'tsc' as unstable because the skew is too large` But in the end didn't crash.
- a10 – hung quickly `BAD LUCK: lost 42 message(s) from NMI context! WARNING: CPU: 0 PID: 21338 at arch/x86/mm/fault.c:435 vmalloc_fault+0x58/0x1f0 BUG: stack guard page was hit at ffffc90008500000 (stack is ffffc900084fc000..ffffc900084fffff) kernel stack overflow (page fault): 0000 [#1] SMP` This was reported to linux-kernel and a fix committed.

Table 4: Fuzzer results at paranoid level 0 (system-wide/uncore).

machine	warnings	time to crash	kernel
core2	3	21h19m	4.9-rc0
haswell	3	8h58m	4.9-rc0
skylake	0	4h50m	4.9-rc0
a10	1	7h55m	4.9-rc0

5.3 Fuzzing Paranoid Level 0

Paranoid level 0 enables per-CPU events, including things like uncore, offcore, and RAPL. It exercises a lot more of the unusual CPU event sources.. Crashing tends to happen even more quickly here, as seen in Table 4.

- Core2 `WARNING: CPU: 0 PID: 24818 at kernel/events/ring_buffer.c:546 __rb_free_aux+0x40/0xe8` Breakpoint warning. `WARNING: CPU: 0 PID: 23863 at net/sched/sch_generic.c:316 dev_watchdog+0xde/0x139` Locked up no message?
- Haswell

  `WARNING: CPU: 2 PID: 24400 at kernel/events/ring_buffer.c:546 __rb_free_aux+0x107/0x120 WARNING: CPU: 2 PID: 24400 at kernel/events/core.c:4998 perf_mmap_close+0x386/0x390` The following only happens on Haswell and only in uncore mode, an interrupt register gets stuck. This is likely a hardware problem. `WARNING: CPU: 3 PID: 0 at arch/x86/events/intel/core.c:2093 intel_pmu_handle_irq+0x2ca/0x460 core: clearing PMU state on CPU#3` We get slab corruption. I suspect this is due to use-after free and not a bad DIMM, but who knows.

Table 5: Fuzzer results at paranoid level -1 (tracepoints too).

machine	warnings	time to crash	kernel
core2	0	14m	4.9-rc0
haswell	?	didn't run	4.9-rc0
skylake	?	34m	4.9-rc0
a10	?	lost log	4.9-rc0

`Slab corruption (Tainted: G W ): task_struct start=ffff8800bb7e6500, len=5760 010: 6b 6b 6b 6b 6b 6b 6b 6b 6a 6b 6b 6b 6b 6b 6b 6b kkkkkkkkjkkkkkkk Single bit error detected. Probably bad RAM.` Crash hard (nothing in log)

- skylake `INFO: rcu_sched self-detected stall on CPU NMI watchdog: Watchdog detected hard LOCKUP on cpu 7 INFO: rcu_sched detected stalls on CPUs/tasks:` Unresponsive.

- a10 `WARNING: CPU: 3 PID: 8399 at arch/x86/mm/fault.c:435 vmalloc_fault+0x58/0x1f0 BUG: stack guard page was hit at ffffc90001720000 (stack is ffffc9000171c000..ffffc9000171ffff)`

5.4 Fuzzing Paranoid Level -1

I usually avoid doing this, as this enables ftrace/tracepoints and this triggers "don't-do-that-then" type bugs, such as inserting an interrupt tracepoint inside of the interrupt handler which then overflows causing recursive interrupts. You can get some pretty spectacular stack traces, and crashes happen quickly as seen in Table 5.

- core2

  `NMI watchdog: BUG: soft lockup - CPU#1 stuck for 22s!` then hard lockup.

- haswell – did not have a chance to run this

- skylake

  `NMI watchdog: Watchdog detected hard LOCKUP on cpu 3` Crashed.

- a10 – this system always seems to hit this bug, odd the others don't. `BAD LUCK: lost 43 message(s) from NMI context! WARNING: CPU: 2 PID: 26737 at arch/x86/mm/fault.c:435`

Table 6: Fuzzer as root (lots of /proc).

machine	warnings	time to crash	kernel
core2	?	1h13m	4.9-rc0

5.5 Fuzzing as Root

Fuzzing as root has all the fun of -1, but also the perf_fuzzer does some wacky thigns to various /proc files which can be pretty verbose. This is generally not reccommended, but Peter Z has fixed many of the issues so in general the things preventing this (such as setting impossibly fast interrupt rates) are no longer a problem. See Table 6. Only ran for core2.

- core2

  `INFO: rcu_sched self-detected stall on CPU NMI watchdog: Watchdog detected hard LOCKUP on cpu 0`

6 Open Questions

Various questions for discussion.

6.1 Can we design a tool to auto-generate the equivalent of perf_fuzzer?

You can use the kernel header files, the manpage, maybe even reverse engineer the kernel ABI to figure out what parameters to send a syscall. How can you train it though on what combinations of syscalls go together? Trace perf? Analyze the kernel code somehow?

6.2 Can we enhance kernel to make it possible to debug the weird perf_fuzzer generated deadlocks?

This is a major stumbling block for me. Recording and capturing errors after a fuzzing run can take up to an hour. Trying to find/reproduce the bug can take days, with little reward or hope the bug will actually be found.

6.3 Can we make anyone care about fuzzing?

As an academic I have had trouble getting funding or publications out of this work. The work is considered "not novel", "iterative", or a "solved problem". Yet the reviewers often stress I am doing important work and continue doing it, even without support. I end up feeling like Clifford Stoll [7] who ran into similar problems when chasing a hacker.

6.4 Can we make people contribute back?

Lots of people run fuzzers. However as open source projects go, they get little feedback. Dave Jones also ran into this problem with Trinity [2].

Security researchers run and use the fuzzers, but don't report bugs, or else report them and get bug bounties and such without even letting us know. I have had this problem with perf_fuzzer, where possibly CVE-2016-0805 was found with the fuzzer but it has not been possible to get the author of the talk to confirm or deny this [12].

7 Future Work

There are various pieces of future work I'd like to complete if I have time. The main thing is to get better support for some large interfaces that interact with perf_event_open, namely cgroups, eBPF, and ftrace.

eBPF is an interesting one, as it is very powerful, but at the same time is a full programmable language and fuzzing it would require fuzzing a whole new architecture.

8 Conclusion

The perf_fuzzer continues to find bugs in the kernel and seems as if it will for the foreseeable future, if only because the bugs it finds are nearly impossible to properly isolate.

Properly fuzzing and reporting the bugs is boring and tedious. It is not much fun, and so the work suffers especially if you are doing it in your rare free time. However fuzzers are a valuable tool for finding bugs, and thus we need to power through and keep developing them. If somehow this process could be more automated (both in generating the fuzzers and in finding/reporting bugs) it would be amazing.

9 After the Conference

The Fuzzing and Testing mini-workshop at the 2016 Linux Plumbers Conference was rushed, but it did lead to useful discussion and outcomes. LWN posted a good summary of the fuzzer discussion [4].

There is now a push for documenting the kernel syscall ABIs in a way that can lead to better automated fuzzing. I put forth the claim that the perf_event interface is complicated enough to defy description, but that was not easily believed. PeterZ backed me up on the fact that perf_fuzzer was still causing crashes whlie the other fuzzers were not.

Still, it's difficult enough to *intentionally* create something complex like an Intel PT tracing AUX buffer event, let alone stumble upon how to do it with random fuzzing.

Vyukov, the syzkaller author, strongly suggested that we fuzzer users compile with the newish KASAN kernel address sanatizer in order to better find bugs. This was a great suggestion, as once I got my kernels up to date (the MODVERSION breakage introduced in 4.9-rc1 was nearly impossible to track down) I started fuzzing again. And promptly found a number of bugs via KASAN, including a few in the uncore that I think were the ones we've been chasing for a very long time.

At least four bugs have been found and fixed using `perf_fuzzer` and KASAN.

References

[1] CLOUTIER, M., PARADIS, C., AND WEAVER, V. A raspberry pi cluster instrumented for fine-grained power measurement. *Electronics 5*, 4 (2016), 61.

[2] CORBET, J. Jones: Future development of trinity. *Linux Weekly News* (July 2015).

[3] EDGE, J. Disallowing perf_event_open(). *Linux Weekly News* (Aug. 2016).

[4] EDGE, J. A trio of fuzzers. *Linux Weekly News* (Nov. 2016).

[5] JONES, D. Trinity: A Linux system call fuzzer. `http://codemonkey.org.uk/projects/trinity/`.

[6] MUCCI, P. J., BROWNE, S., DEANE, C., AND HO, G. PAPI: A portable interface to hardware performance counters. In *Proc. Department of Defense HPCMP User Group Conference* (June 1999).

[7] STOLL, C. *The cuckoo's egg: tracking a spy through the maze of computer espionage*, 1st ed. Doubleday, 1989.

[8] VYUKOV, D. syzkaller: a distributed, unsupervised, coverage-guided Linux syscall fuzzer. `https://github.com/google/syzkaller`.

[9] WEAVER, V. perf_event_open manual page. In *Linux Programmer's Manual*, M. Kerrisk, Ed. Dec. 2013.

[10] WEAVER, V. Fuzzing perf_events. *Linux Weekly News* (Aug. 2015).

[11] WEAVER, V., AND JONES, D. perf_fuzzer: Targeted fuzzing of the perf_event_open() system call. Tech. Rep. UMAINE-VMW-TR-PERF-FUZZER, University of Maine, Aug. 2015.

[12] WU, W. PERF: From profiling to kernel exploiting. In *HITBSecConf2016*.