Improving HPC Security with Targeted Syscall Fuzzing

1st Vincent M. Weaver
Electrical and Computer Engineering
University of Maine
Orono, Maine, USA
vincent.weaver@maine.edu

***Abstract*—**
All modern computer systems, including supercomputers, are vulnerable to a wide variety of security exploits. Performance analysis tools are an often overlooked source of vulnerabilities. Performance measurement interfaces can have security issues that lead to information leakage, denial of service attacks, and possibly even full system compromise. Desktop systems can mitigate risk by disabling performance interfaces, but that is not always possible on HPC systems where performance (and thus measurement) is paramount. We investigate various ways of finding security issues in the performance measurement stack. We introduce the perf_fuzzer, a tool that methodically finds bugs in the Linux `perf_event_open()` system call. We also discuss the perf_data_fuzzer which looks for userspace bugs in the `perf` analysis tool. We describe the development of the fuzzing tools, examine the bugs found, and discuss ways to prevent such bugs from occurring in the future.

***Index Terms*—fuzzing, Linux perf, perf_event, HPC security**

I. Introduction

Modern high-performance supercomputers are complex systems. Their design is complicated enough that application performance cannot be easily estimated. To optimize performance a repeated cycle of iterative optimization is undertaken: performance is measured, the code is improved, then the process cycles back again to the measurement step.

Obtaining accurate performance information is always a challenge. To aid in this, modern computer processors provide hardware performance counters which provide low-level access to detailed information gathered during the execution of code. This can provide great insight into the sources of slowdowns and bottlenecks found in modern high-performance computing (HPC) workloads.

The constant drive for fast and efficient code means that users of HPC systems are much more likely to depend on the results of performance counters than the users of desktop or mobile systems. This puts pressure on system administrators to enable the performance counters, despite the potential security issues this exposes. This danger is somewhat mitigated because HPC systems tend to exist in locked down controlled environments. Useful code does get developed and brought in from outside these protected spaces, so it is critical that the operating systems and performance tools running on the untrusted code are robust against security issues.

There are various security concerns exposed by the use of hardware performance counters:

- Detailed timing results can allow information leakage. (See the recent Spectre and Meltdown [1] issues as prominent examples of such attacks).
- Using the counters requires system-level access to low level CPU interfaces such as model specific registers (MSRs). Users with direct access to MSRs can compromise the entire system. Usually this is prevented by having the operating system mediate access to the MSRs, but this depends on this interface being secure.
- Device drivers controlling the counters should be robust against invalid input, but bugs in this area can lead to crashes (denial of service) or even code execution, privilege escalation, and compromise of the machine.

In this paper we describe tools we have developed to automatically find bugs in Linux performance infrastructure so they can proactively be fixed before they can be exploited.

A. Secure Access to the Hardware Counters

Programs wishing to access the performance counters typically cannot do so directly, instead a library or operating system provides an abstracted interface. On x86 compatible systems the counters are implemented as MSRs which by default require kernel-level permissions to access.

On Linux the official method of accessing performance information is by using the `perf_event` interface and the associated `perf_event_open()` system call [2]. The user cannot access low-level hardware directly, rather the operating system is used, and the operating system is responsible for making sure the performance counters are properly managed. This can simplify the tools, but adds overhead to the performance measurements.

Some low-overhead system monitoring tools such as Likwid [3] bypass `perf_event` and access the MSRs directly. This requires privileged access; to talk to the MSRs a driver is needed, often the minimal Linux `/dev/msr` interface that allows user code to talk directly to the hardware. While this has low overhead, it turns out that direct, unrestricted access to the MSRs is not desirable. Arbitrary write access allows dangerous actions, such as redirecting interrupt handlers to your own code which leads to system compromise. Even read

access is considered dangerous, if only because it can leak information useful to hackers, such as the address of operating system structures in memory.

Attempts have been made to harden the user MSR interface by only allowing known safe MSRs (via the msr-safe driver [4]). This work is not included with Linux by default and needs to be separately compiled and installed. Despite these precautions, there has been a push by Linux kernel developers to eliminate any direct user access to MSRs as they believe it is not possible to fully harden such usage.

B. Fuzzing for Security

An important characteristic of an operating system is being able to prevent a hostile user from compromising the system's security. In practice bugs appear in operating system code despite the developer's best efforts to avoid them.

In an attempt to proactively find bugs in the Linux kernel we have written the perf_fuzzer, a tool that methodically probes Linux's `perf_event_open()` system call. We do this by fuzzing, which is a technique of finding bugs by systematically generating invalid or almost invalid input for an interface and seeing if it can cause crashes. Although it sounds unlikely, this is an extremely effective way of finding bugs, especially simple programming mistakes such as poorly validated input or off-by-one errors. Fuzzing can be done at any level of the computing stack, from high-level user programs [5] down to the underlying hardware implementation [6]. In the best case nothing happens, but often the malformed inputs can cause crashes, or worse, lead to exploitable security issues.

To date our fuzzer has found seventeen major bugs in the Linux kernel. Most are denial of service (DoS) bugs that can crash a system, but at least one is a local root exploit. Fixes for all of these bugs have been contributed back upstream to the main Linux kernel source tree. Testing continues to find new bugs, although they are becoming more obscure and harder to isolate and fix.

In addition to our kernel work, we also are working on the perf_data_fuzzer which probes the widely used userspace `perf` analysis tool.

C. Does this Apply to HPC?

One might ask, in a highly regulated supercomputing environment, who cares about security issues? Unlike typical desktop, mobile, or server computing environments in HPC users are tightly screened and machines tend to be isolated from the open internet.

In these cases security is still a concern. Bugs that can crash a HPC system (either accidentally or maliciously) can lead to long, expensive downtimes. Sensitive jobs are often run on these systems, so information leakage or full compromise can lead to data or code being obtained by non-authorized users. Not all code has been fully audited and users are not always careful about where they source code from. Finally, do not underestimate the ingenuity of outside actors being able to bring down sensitive computing machinery via obscure bugs: a prime example of this is the Stuxnet attack [7] which managed to infect air-gapped systems.

II. Related Work

Related work in this area is split between supercomputing security research and general fuzzing research.

A. Supercomputer Security

Previous supercomputer security research has often been concerned with issues other than the performance measurement interface. Usually the concerns are things such as proper user authentication, stolen computing time, and the possibility of leaking information over the network [8]–[10].

Malin and van Heule [11] describe using continuous monitoring and software inventory control to manage the security of large clusters. Costin [12] looks at HPC tools that can lead to security issues, but in this case it was infrastructure monitoring tools, not performance ones.

B. Fuzzing

The use of random inputs when testing computer systems has a long history, although at times it has been considered less effective than more formal testing methods [13]. The term "fuzzing" was first coined by Miller in 1988 as part of a class project determining why line noise over a "fuzzy" modem connection would crash many UNIX utilities. This research was extended by Miller et al. [5] to investigate the causes of the crashes on a wide range of UNIX systems. While they focus on userspace utilities rather than kernel interfaces, many of the bugs they find (including NULL pointer dereferences and lack of bounds checking on arrays) are the same as those found by us with perf_fuzzer 25 years later.

Miller et al. revisited tool fuzzing in 1995 [14] and found that they could still crash over 40% of common system utilities on UNIX and Linux systems. Forrester and Miller extended the work to look at Windows NT [15] and Miller, Cooksey and Moore looked at Mac OSX [16] userspace programs and found similar userspace error rates to those on UNIX. Most of these investigations look at userspace utilities; our work concentrates on operating system kernel interfaces.

Operating systems have many potential interfaces exposed to users that can harbor bugs. Carrette's CrashMe [17] program attempts to crash the operating system by fuzzing the instruction stream. Unlike our work, this does not target system calls directly, but the entire operating system and underlying hardware in the face of random processor instructions. Medonça and Neves [18] fuzz at the device driver level by externally sending malicious inputs to wireless networking hardware. Cadar et al. [19] use an analysis tool that examines executables and generates inputs based on program flow; they apply this to finding crashing bugs in the Linux filesystem code with malicious filesystem images. Another interface open for bugs in modern systems is the virtual machine interface [20], [21].

Koopman et al. [22] look at the robustness of five different operating systems (Mach, HP-UX, QNX, LynxOS and Stratus FTX) by injecting random data at the operating system interface, focusing on seven commonly used system calls: `read()`, `write()`, `open()`, `close()`, `fstat()`, `stat()`, and `select()`. On four of the five systems bugs

severe enough to require a restart were found. Our work is similar to this, but involves focusing on a single system call on the Linux operating system.

Existing Linux system call fuzzers such as Jones' Trinity [23] and Ormandy's iknowthis [24] test the majority of available system calls with varied parameters. They currently do not focus on one system call, and only have limited support for using system call dependency information to chain together related system calls the way that perf_fuzzer can.

Vyukov's syzkaller [25] is currently one of the more popular fuzzers actively being used in Linux development. It depends on custom written templates that describe the system calls, but adds coverage-based support that instruments the kernel to automatically detect when a fuzzed access has caused issues. While this is much more powerful than plain random testing, it still misses some of the bugs found by `perf_fuzzer`, as we use detailed knowledge of what makes a valid sequence of perf_event related system calls, something that is hard for a fully automated fuzzer to work out experimentally.

III. Security issues in HPC Tools

HPC developers typically do not interact with the Linux kernel directly, but rather they use various high-level tools which collect performance data and provide data analysis. These tools may talk directly to the Linux perf_event interface, or they might interface via a lower level library (such as PAPI [26]) which provides an abstraction to the underlying operating system. Each layer of abstraction makes it easier to code, but the downside it that each level can be the home to security issues.

Programs being run in an HPC environment are usually not known for security issues. They tend to be long-running simulations doing large amounts of floating point math and only rarely take in user input or communicate with the outside world. One should not be complacent though, as any process on Linux has the power to communicate with the kernel and do things like start network connections. A clever hacker that finds a bug in an otherwise run-of-the-mill simulation could potentially cause a lot of trouble if they can convince the program to operate on untrusted input.

A more likely source of security issues can be found in the performance measurement area, as these tools interact with both the operating system and the user in complex ways that are harder to audit. If you can trick a user into running a performance analysis tool on untrusted input (say a previously recorded `perf.data` file of unknown origin) and this input was specifically constructed to trigger a buffer overrun, this could take control of the tool execution. This could lead to dangerous code being executed, such as erasing files or opening illicit network connections. While this is bad, in theory the danger should be limited in scope to whatever the user could do with their account.

This crafting of invalid input that can corrupt or crash the Linux `perf` tool is not hypothetical. In this paper we discuss the perf_data_fuzzer, a tool we have written that has found exploitable bugs. Our findings are not unique, various other similar bugs have been found with the tool over the years.

To truly cause trouble, one needs to be able to escalate beyond user code and disrupt the operating system. One way of doing this is by finding bugs triggered by the system call interface. This is the goat of our perf_fuzzer utility, to test how robust the Linux `perf_event_open()` syscall is when probed by untrusted input.

IV. Motivation for the perf_fuzzer

We undertook the design of the perf_fuzzer after ongoing frustration from finding bugs in the `perf_event_open()` system call. We use the PAPI [26] performance library, which is widely used in the HPC community. PAPI tends to exercise a different subset of functionality than the more commonly used `perf` command-line utility distributed with the Linux kernel source. Since most kernel developers restrict their perf_event usage to `perf`, any functionality not exercised by that tool can break without being noticed. Work on PAPI has turned up numerous kernel bugs, as seen in Table I. These issues were all found by programs trying to exercise normal, expected functionality of the interface. It is easy to add reactionary tests that test for these bugs after they are known, such as our `perf_event_test` [27] test suite. However this does not help in finding new, unknown bugs introduced during the fast-paced Linux kernel development process.

The existing Trinity fuzzer added support for `perf_event_open()` soon after the system call was introduced. Trinity initially had limited support for the call, making it extremely unlikely that valid or near-valid events would be generated. We contributed slightly better support in November 2011 as an ongoing part of research into the interface. Not much came of this until April 2013 when Rantala [28] found a bug using Trinity where the 64-bit `attr.config` value was being copied to a 32-bit integer before being sanity checked. This bug meant that the high 32-bits could be controlled by the user, and eventually it was discovered that this could be exploited by a local user to get root privileges (CVE-2013-2094). More worrisome, the kernel code change that introduced this bug happened in 2010 and was possibly being exploited soon after, but it took 3 years for the bug to be found and fixed. This incident is what sparked perf_fuzzer development.

V. The perf_fuzzer

The Linux perf_event performance monitoring subsystem has a complex interface that is not completely exercised by a naïve fuzzer. A full description of the interface can be found in the `perf_event_open.2` manpage [2]. The `perf_event_open()` interface is complex enough that it has the longest manual page of any system call, longer even than the elaborate `ptrace()` system call.

The prototype for the system call looks like this:

```
int perf_event_open(struct perf_event_attr *attr,
                    pid_t pid, int cpu, int group_fd,
                    unsigned long flags);
```

It takes five input arguments:

- `attr` is a complicated structure describing the event to be created with 40+ inter-related fields,
- `pid` specifies which process id to monitor (0 indicating current, -1 indicating all),
- `cpu` specifies which CPU core to monitor (-1 indicating all),
- `group_fd` allows an event to join a group leader, creating a group of events that can be read simultaneously,
- and `flags` allows setting various optional event flags.

There are two common ways of using perf_event: one is monitoring a program belonging to a user (anyone can do this by default), the other is system-wide measurement (which generally requires root permissions to avoid leaking sensitive information between users).

Opening an event with `perf_event_open()` is only a small part of the perf_event experience. Many bugs that are found do not happen solely at open, but also depend on interactions with other calls. Various other kernel interfaces interact with perf_event:

- `prctl()` (process control) can be used to start and stop all events in a process,
- `ioctl()` is used to start, stop, and otherwise get information about events,
- `read()` returns the current values of counters and some additional information,
- `mmap()` can map pages that provide event info as well as a circular ring buffer where the kernel places sampled event information,
- `poll()` can wait for overflow or buffer-full signals,
- and, various files under `/proc` and `/sys` provide extra event information and configuration settings.

The perf_event implementation involves low-level code scattered throughout the kernel, making the interface complex to debug. Hardware events are generally programmed by writing to CPU model specific registers (MSRs on x86). Hardware events can overflow, triggering non-maskable (NMI) interrupts. Software events (counts of kernel maintained values such as context-switches and interrupt counts) require placing perf_event code in time critical kernel functions. The perf_event interface has also grown to include the hardware breakpoint interface and has major connections to the ftrace system tracing interface. In addition support has been added to support running Berkeley Packet Filter (BPF) programs in the kernel in conjunction with events, further increasing the potential sources of bugs.

A. The perf_fuzzer Design

Jones introduced the Trinity fuzzer [23], [29], first as scrashme in 2006, and then renamed Trinity in 2010. Trinity does a remarkable job of finding bugs, but it currently runs system calls mostly independently, and so takes a long time to discover certain bugs. An interface like perf_event often has bugs that involve various system calls interacting in a complex set of ways that are hard to describe with the current Trinity infrastructure.

The perf_fuzzer re-uses the `perf_event_open()` fuzzing routines provided by Trinity. Sharing code between the two projects avoids duplicated work and ensures that any improvements in one project are included in the other.

At startup the perf_fuzzer parses the command line. It seeds the random number generator, either based on the time, or else via a value passed by the user (to enable re-running with same initial start conditions). This value is also printed and written to disk to ease reproduction of a run. The process id is logged so that during replay any invocations using the previous process id are re-mapped to the current one. Various structures are initialized, including calling the Trinity `syscall_perf_event_open.init()` routine and creation of a Trinity-compatible "page_rand".

Next the signal handlers are initialized. These can be a source of errors as the more widely used `perf` utility does not use signal handlers (it uses `poll()` to detect overflows). The perf_fuzzer sets up counter overflows to trigger SIGRT signals (as PAPI does) because they queue and avoid losing signals when a system is busy. Eventually the queues can fill and the kernel handles this by sending SIGIO; we set up handlers for both SIGRT and SIGIO. The SIGRT handler disables the event causing the signal, reads event values and then restarts the event. If the SIGIO handler is triggered it means we are stuck in a tight overflow storm and not making forward progress, so it attempts to close the event causing the issues (this is difficult, especially if the event was created in another thread before forking). An additional SIGQUIT handler is set up that will dump the current open event state so a user can monitor the current status of the fuzzing.

The main perf_fuzzer event loop is then entered, which loops forever randomly selecting one of the following tasks:

- `perf_event_open()` a random event,
- `close()` a random event,
- `ioctl()` a random event,
- `prctl()` the process,
- `read()` a random event,
- `write()` a random event,
- access a random file,
- `fork()` the process,
- `poll()` an event,
- corrupt the mmap page, or
- run a million instructions.

B. Reproducibility

One highly desirable trait of a fuzzer is that it has reproducible results: given the same random seed the same exact values are generated by the fuzzer. This can greatly ease debugging of problems, and is useful for creating regression tests to verify if a particular bug has been fixed.

The perf_fuzzer has been carefully written to be as reproducible as possible, although full determinism is not always possible when measuring performance events because outside factors (such as hardware interrupts, kernel interactions, and other system activity) can vary from run to run. Event avail-

ability can vary between kernel versions and processor types, further reducing the possibility of deterministic results.

To ease reproducibility, a header is generated which includes enough information to recreate a fuzzing run. This makes it easy to include this state into bug reports and allows more easily recreating test conditions that cause failures. The header includes the version of perf_fuzzer, the Linux version and architecture, and the processor type. Also included is the random number seed, which allows replicating the random number generation exactly. Some kernel settings are also saved, such as the `/proc/sys/kernel/perf_event_max_sample_rate` value controls the maximum event sample rate. If this value differs from the original run then some events may fail because they set the sample rate too high. This is a particularly tricky value, as the kernel will automatically adjust this downward (outside of user control) if it thinks interrupts are happening too quickly. Another kernel value is `/proc/sys/kernel/perf_event_paranoid`. This allows the system administrator to allow access to some events (such as system-wide events) that are disabled by default for normal users for security reasons. If this value differs from the default then some events that would normally fail will instead open without error.

C. Logging and Replay

perf_fuzzer has a logging mode that can be enabled. An ASCII text file is generated: for each action a letter indicating the action type is printed followed by a list of the parameters needed to replay the action.

Logs quickly get large and the entire file contents can be important. Bugs are often not simply caused by the last `perf_event_open()` call, but by a long chain of related actions scattered throughout the log. Determining the last action that causes a lockup can be difficult as crashes can happen quickly enough that key values are not logged to disk. Even running `sync()` before logging is not always enough to capture the value (and that slows the fuzzing process). The behavior of the fuzzer is usually deterministic enough that multiple runs with the same random seed usually get to the same place, so a special trigger can be inserted in the code to pause just before the last problem causing action.

D. Fuzzing Environment

The perf_fuzzer can cause crashes so severe that the kernel has no time to log the error. Sometimes the lockups are so bad they can crash the ethernet card too which can take down the local network. The best way to ensure kernel crashes are logged is to have another dedicated monitoring system connected to the fuzzed system via a serial cable.

To catch bugs early, the fuzzer can be run on pre-release versions of Linux. In addition, there are kernel debugging options such as KASAN and lockdep that allow catching some classes of bugs immediately rather than allowing bugs to corrupt kernel state in ways only detectable much later.

VI. The `perf_data_fuzzer`

We have made an additional tool, the perf_data_fuzzer, which attempts to fuzz the `perf.data` file created by the `perf record` command. We realized the need for this while writing a tool to parse this file, and we noticed how easy it is to cause `perf` to segfault when fed invalid files. This tool found a number of bugs, detailed in Table V, some of which are exploitable. Running `perf report` on an untrusted `perf.data` file (say one sent to you in e-mail, or one that you found in a user directory) could take over the program and make it do arbitrary things. Combined with the exploits found with our other fuzzer, this could allow completely taking over a system simply by running `perf` as a regular user.

VII. Fuzzer Results

Table II summarizes the major perf_event bugs that have been found (and subsequently fixed) by Trinity and perf_fuzzer from April 2013 through August 2022. Over twenty major bugs have been found, which is more than those found by more traditional methods over the preceding four years as shown in Table I.

A. Critical Bugs Found

perf_fuzzer triggers a wide variety of bugs; not all of them are dangerous or security issues. What follows is a summary of the types of issues we have found thus far.

1) Crash / Hang / Panic / Denial of Service: The most annoying type of bug found is one that completely crashes the computer. Tracking down this type of bug is difficult as logging and debugging information are often lost.

These bugs have security implications; at the very least they are "Denial of Service" (DoS) attacks. Even in cases where the operating system does not crash outright, often the system will be left in an unusable or fragile state that needs rebooting. These bugs can often be triggered by a regular user to make the system unavailable. Despite this, reports of this nature are treated with fairly low urgency by the perf_event developers unless a small triggering case can be created.

An example of this type of bug is the "perf/ftrace wrong permissions check" bug fixed in the 3.13 kernel. The ftrace infrastructure allows creating perf_event events that trigger at various predefined code locations in the kernel. The fuzzer created an event that caused an overflow on every function entry; if set up to overflow, then the overflow handler will trigger this event, which can recursively cause another overflow which triggers another event, etc., causing the kernel to get trapped in an endless loop. The machine will become unresponsive at this point, although the watchdog might eventually kick in and display a "kernel is stuck" message.

2) Local Root Exploit: Sometimes a bug that only looks like a crash or panic can turn out to have far greater security implications. If a bug lets user-supplied values get written into unexpected parts of kernel memory, eventually a clever user will be able to figure out how to use this to escalate their privileges and obtain root access.

TABLE I
LINUX KERNEL PERF_EVENT SECURITY BUGS FROM 2009-2013 FOUND WITHOUT FUZZERS.

Type	CVE	Fixed (version/git commit)		Description
root exploit	CVE-2009-3234	2.6.32	b3e62e35058fc744	buffer overflow
crash	CVE-2010-4169	2.6.37	63bfd7384b119409	improper mmap hook
crash	-	2.6.39	ab711fe08297de14	task context scheduling
memleak	-	2.6.39	38b435b16c36b0d8	inherited events leak memory
crash	CVE-2011-2521	2.6.39	fc66c5210ec2539e	x86 msr registers wrong
DoS	CVE-2011-4611	2.6.39	0837e3242c73566f	ppc cause unexpected interrupt
crash	CVE-2011-2918	3.1	a8b0ca17b80e92fa	software event overflow
crash	-	3.5	9c5da09d266ca9b3	cgroup reference counting
crash	CVE-2013-2146	3.9	f1923820c447e986	offcore mask allows writing reserved bit
crash	-	3.9	1d9d8639c063caf6	pebs/bts state after suspend/resume

TABLE II
LINUX PERF_EVENT SECURITY BUGS FOUND BY FUZZERS STARTING FROM APRIL 2013.
(T=TRINITY, P=PERF_FUZZER, H=HONGGFUZZ [30], S=SYZKALLER [25])

Which	Type	CVE	Fixed in Linux		Description
T	root exploit	CVE-2013-2094	3.9	8176cced706b5e5d	32/64 bit cast
P	crash	-	3.10	9bb5d40cd93c9dd4	mmap accounting hole
P	crash	-	3.10	26cb63ad11e04047	mmap double free
P	panic	-	3.11	d9f966357b14e356	ARM array out of bounds
P	root exploit	CVE-2013-4254	3.11	c95eb3184ea1a3a2	ARM event validation
P	panic	-	3.11	868f6fea8fa63f09	ARM64 array out of bounds
P	panic	-	3.11	ee7538a008a45050	ARM64 event validation
P	panic	-	3.13	6e22f8f2e8d81dca	alpha array out-of-bounds
P/T	crash	CVE-2013-2930	3.13	12ae030d54ef2507	perf/ftrace wrong permissions check
P	crash	-	3.14	0ac09f9f8cd1fb02	pagefault ftrace cr2 corruption
P	crash	-	3.15	46ce0fe97a6be753	race when removing event
P	crash	-	3.15	ffb4ef21ac4308c2	function cannot handle NULL return
P	reboot	-	3.17	3577af70a2ce4853	race in perf_remove_from_context()
P	crash	-	3.19	98b008dff8452653	misplaced parenthesis in rapl_scale()
P	crash	-	3.19	c3c87e770458aa00	fix the grouping condition
P	crash	-	3.19	a83fe28e2e453924	Fix put_event() ctx lock
P	crash	-	3.19	af91568e762d0493	IVB-EP uncore assign events
P	crash	-	4.0	d525211f9d1be8b5	Fix perf_callchain() hang
H	memleak	-	4.0	a83fe28e2e453924	fix put_event() ctx leak
P	crash	-	4.1	8fff105e13041e49	arm64/arm reject groups spanning PMUs
P	crash	-	4.1	15c1247953e8a452	snb_uncore_imc_event_start crash
P	crash	-	4.2	57ffc5ca679f499f	Fix AUX buffer refcounting
P	panic	-	4.5	fb822e6076d97269	powerpc: Oops destroying hw_breakpoint event
P	crash	-	4.8	0b8f1e2e26bfc6b9	crash in perf_cgroup_attach
P	crash	-	4.9	7fbe6ac02485504b	vmalloc stack unwinder crash
P(?)	exploit	CVE-2017-6001	4.10	321027c1fe77f892	perf_event_open() vs. move_group race
S	bug	-	4.11	e552a8389aa409e2	Fix use-after-free in perf_release()
P	crash	-	4.15	99a9dc98ba52267c	BTS causes crash with KPTI meltdown fixes
P	crash	-	4.20	472de49fdc53365c	BTS crash, uninitialized ptr
S	crash	-	5.3	1cf8dfe8a661f046	Race between close() and fork()
P	panic	-	5.5	242bff7fc515d8e5	i915 null pointer dereference
P	crash	CVE-2021-28971	5.12	d88d05a9e0b6d935	NULL pointer dereference with PEBS on haswell

The initial vulnerability that prompted the design of perf_fuzzer was such an exploit. An improperly checked config value for a software event allowed a user to arbitrarily increment any memory location. It was possible to use this to redirect the undefined instruction interrupt vector to point to user-supplied code, which then can carry out the privilege escalation (Edge [31] describes this in more detail).

A different bug found by perf_fuzzer is the "ARM event validity" bug. The `validate_event()` function took a group leader of an event and called `armpmu->get_event_idx()`. If the group leader was not an `armpmu` type, then the function pointer was located past the end of the struct and would have whatever arbitrary value happened to be beyond it in memory. If you were unlucky, this arbitrary value was a valid user address. For a short window of time in the 3.11-rc cycle this value pointed to a value initialized to INT_MIN, which is a valid user mappable address of `0x80000000`. We wrote code that mapped proper exploit code there and managed to escalate our privileges to root. Luckily this bug was found and fixed before it made it into a released kernel.

3) Warnings: Throughout the Linux kernel code are "warnings": debug macros of the type WARN_ON used as asserts. These catch corner cases the author of the code thinks are

TABLE III
LINUX PERF_EVENT WARNING AND BUG ASSERTIONS FOUND BY FUZZERS (T=TRINITY, P=PERF_FUZZER, Z=TRINITY RUN BY 0-DAY TESTER)

Which	Type	Fixed in Linux		Description
P	WARNING	3.11	734df5ab549ca44f	WARNING: at kernel/events/core.c:2122
P	WARNING	3.14	26e61e8939b1fe87	WARNING at arch/x86/kernel/cpu/perf_event.c:1076
T,Z	BUG	3.17-next	caught early	BUG: unable to handle kernel NULL pointer
P	WARNING	3.19	9fc81d87420d0d3f	WARNING: Can't find any breakpoint slot
P	BUG	3.19	af91568e762d0493	BUG: uncore_assign_events()
T	WARNING	4.0	2fde4f94e0a95312	WARNING: add_event_to_ctx()
P	WARNING	4.1	2cf30dc180cea808	WARNING: trace_events_filter.c replace_preds
P	WARNING	4.2	b4875bbe7e68f139	WARNING: trace_events_filter.c replace_preds
P	WARNING	4.2	93472aff802fd7b6	WARNING: Fix active_events imbalance
P	BUG	4.9	c499336cea8bbe15	BUG: KASAN: slab-out-of-bounds
P	WARNING	4.9	e96271f3ed7e702f	WARNING: KASAN global-out-of-bounds in match_token
P	WARNING	4.17	9e5b127d6f334681	WARNING: armv8: Fix perf_output_read_group()
P	WARNING	4.19	7ccc4fe5ff9e3a13	WARNING: powerpc: sched_task function thread-imc
P	WARNING	4.19	6cbc304f2f360f25	WARNING: unwind errors with PEBS entries

TABLE IV
LINUX PERF_EVENT CORRECTNESS BUGS FOUND WHILE USING FUZZER. (T=TRINITY, P=PERF_FUZZER)

Which	Type	Fixed in Linux		Description
P	Aliasing	3.13	0022cedd4a7d8a87	ftrace config value 64-bit but only lower 32 checked
P	Correctness	3.15	0819b2e30ccb93ed	sample_period unsigned cast to signed
P	Correctness	3.16	643fd0b9f5dc40fe	flags value 64-bit but only lower 32 checked
P	Correctness	4.11	1572e45a924f254d	Fix perf_cpu_time_max_percent check
P	Wrong Resource	4.15	1289e0e29857e606	RAPL readings using wrong MSRs
P	Wrong Return	5.5	da9ec3d3dd0f1240	perf_event_open() returns 0 on failure
P	MSR error	5.12	2dc0572f2cef8742	unchecked MSR error from KVM event
P	MSR error	5.19	b0380e13502adf7d	unchecked MSR access error on HSW

TABLE V
RESULTS OF PERF_DATA_FUZZER FUZZING OF PERF INTERACTION WITH PERF.DATA FILES.

Type	Fixed in Linux		Description
Crash	5.3	7622236ceb167aa3	f_header.attr_size ==0 causes a floating point exception
Hang	5.3	57fc032ad643ffd0	parsing perf.data leads to being stuck in infinite loop
Buffer Overflow	?	Not Fixed Yet	buffer overflow in perf_header__read_build_ids

invalid but unlikely.

Fuzzers often trigger these messages. Sometimes the problem reported is real and can be fixed, sometimes it is a false positive and just silenced. It is still important to report these although such problems rarely cause crashes. A list of warnings triggered by perf_fuzzer can be seen in Table III.

B. Other Bugs Found

There are perf_event bugs in the kernel that are not obviously security bugs, but just problems with the interface. Fuzzers are not designed to catch these bugs but these can be noticed while tracking down other more serious issues. Table IV shows various bugs of this type that were found and fixed.

C. Bugs Avoided

Now that the perf_fuzzer tool has become known in the kernel development community, it has started being used to catch bugs in patches before they are applied to the kernel tree. For example, the ARM perf_event developers encourage usage of perf_fuzzer during new patch submission [32].

VIII. FUTURE WORK

After years of work the number of bugs found by perf_fuzzer has tapered off. This means there are more opportunities to improve the fuzzer to exercise other parts of the kernel. In addition the `perf` tool that is tightly-coupled to the perf_event interface turns out to also have many bugs and it is not heavily fuzzed. We have started work on remedying that, especially as a lot of trouble can be caused if you can convince a sysadmin to run the `perf` tool on a `perf.data` file that is crafted to take advantage of parsing bugs.

There are many future directions that can be explored:

- Testing more exotic ways of generating file descriptors, such as events being passed across an opened socket,
- Setting up breakpoints inside of perf_event data structures,
- Testing the cgroup (container) support. The perf_event interface supports special cgroup events, but the perf_fuzzer does not explicitly test this,
- More advanced coverage of multithreaded code. The current *fork()* fuzzing code is simplistic and does not test multiple children or errors caused by `exec()` of a new process

- More intelligent raw hardware event choices. Currently the fuzzer picks raw hardware events completely at random. There are libraries that provide valid raw event values, such as libpfm4 [33], that can be used to create more likely to be valid CPU events.
- Fuzzing the Berkeley Packet Filter (BPF) interface which can be used to enhance event collection
- Being sure to exercise some of the more obscure performance features found on Intel processors, such as Intel Processor Trace, PEBS, and others

Another goal is widening the test coverage. Most of the fuzzing has been done on x86 systems (Core2, Haswell, Skylake, Ryzen) as well as a few ARM systems. We have also done limited fuzzing on Power, but we are aware of other groups who have also tested on that architecture.

We hope to test other architectures, especially non-Intel systems, and server systems that have more advanced performance units with features such as Uncore, Offcore, and energy events. The fuzzer can also be used to test emulated systems (such as qemu) or the interfaces inside of virtual machines.

IX. Conclusion

The perf_fuzzer tool is a unique system-call specific fuzzing tool that has found over twenty critical bugs in the Linux kernel. These bugs found are over and above those found by more generic fuzzers, showing that targeted domain knowledge can find bugs that more generic fuzzers miss.

Our work has made the Linux performance interface more robust, and allows system administrators more confidence in allowing users to use standard performance analysis tools with less fear of crashes or system compromise.

X. Availability

The perf_fuzzer toolsuite is free software and is available for download:
`https://github.com/deater/perf_event_tests`

References

[1] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *ArXiv e-prints*, Jan. 2018.

[2] V. Weaver, "perf_event_open manual page," in *Linux Programmer's Manual*, M. Kerrisk, Ed., Dec. 2013.

[3] J. Treibig, G. Hager, and G. Wellein, "LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments," in *Proc. of the First International Workshop on Parallel Software Tools and Tool Infrastructures*, Sep. 2010.

[4] K. Shoga, B. Rountree, M. Schulz, and J. Shafer, "Whitelisting MSRs with msr-safe," in *Proc. 3rd Workshop on Extreme-scale Programming Tools*, Nov. 2014.

[5] B. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Communications of the ACM*, vol. 33, no. 12, 1990.

[6] D. Wood, G. Gibson, and R. Katz, "Verifying a multiprocessor cache controller using random case generation," University of California, Berkeley, Tech. Rep. UCB/CSD-89-490, 1989.

[7] N. Falliere, L. Murchu, and E. Chien, *W32.Stuxnet Dossier*, Symantec, Feb. 2011.

[8] M. Bishop, "UNIX security in a supercomputing environment," in *Proc. ACM/IEEE Conference on Supercomputing*, Nov. 1989.

[9] M. Pourzandi, D. Gordon, W. Yurcik, and G. Koenig, "Clusters and security: Distributed security for distributed systems," in *Proc. IEEE International Symposium on Cluster Computing and the Grid*, May 2005.

[10] G. Markowsky and L. Markowsky, "Survey of supercomputer cluster security issues," in *Proc. International Conference on Security and Management*, Jun. 2007.

[11] A. Malin and G. Van Heule, "Continuous monitoring and cyber security for high performance computing," in *Proc. 1st Workshop on Changing Landscapes in HPC Security*, Jun. 2013, pp. 9–14.

[12] A. Costin, "All your cluster-grids are belong to us: Monitoring the (in)security of infrastructure monitoring systems," in *Proc. IEEE Conference on Communication and Network Security*, Sep. 2015.

[13] G. Myers, *The Art of Software Testing*. New York: Wiley, 1979.

[14] B. Miller, D. Koski, C. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl, "Fuzz revisited: A re-examination of the reliability of UNIX utilities and services," University of Wisconsin-Madison, Tech. Rep. 1268, Apr. 1995.

[15] J. Forrester and B. Miller, "An empirical study of the robustness of Windows NT applications using random testing," in *4th USENIX Windows Systems Symposium*, Aug. 2000.

[16] B. Miller, G. Cooksey, and F. Moore, "An empirical study of the robustness of MacOS applications using random testing," in *Proc. of the 1st International Workshop on Random Testing*, Jul. 2006.

[17] G. Carette, "CRASHME: A system robustness exerciser," http://crashme.codeplex.com/, 1991. [Online]. Available: http://crashme.codeplex.com/

[18] M. Medonça and N. Neves, "Fuzzing wi-fi drivers to locate security vulnerabilities," in *Proc. 7th European Dependable Computing Conference*, May 2008, pp. 110–119.

[19] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler, "EXE: Automatically generating inputs of death," in *Proc. of the 13th ACM conference on Computer and communications security*, Nov. 2006, pp. 322–335.

[20] A. Gauthier, C. Mazin, J. Iguchi-Cartigny, and J.-L. Lanet, "Enhancing fuzzing technique for OKL4 syscall testing," in *Proc. 6th Annual on Conference Availability, Reliability and Security*, Aug. 2011, pp. 728–733.

[21] L. Martignoni, R. Paleari, G. Roglia, and D. Bruschi, "Testing system virtual machines," in *Proc. of the 19th international symposium on Software testing and analysis*, Jul. 2010, pp. 171–182.

[22] P. Koopman, J. Sung, C. Dingman, D. Siewiorek, and T. Marz, "Comparing operating systems using robustness benchmarks," in *Proc. of the 16th Symposium on Reliable Distributed Systems*, Oct. 1997, pp. 72–79.

[23] D. Jones, "Trinity: A Linux system call fuzzer," http://codemonkey.org.uk/projects/trinity/. [Online]. Available: http://codemonkey.org.uk/projects/trinity/

[24] T. Ormandy, "iknowthis: i know this, it's UNIX," http://code.google.com/p/iknowthis/. [Online]. Available: http://code.google.com/p/iknowthis/

[25] D. Vyukov, "syzkaller: a distributed, unsupervised, coverage-guided Linux syscall fuzzer," https://github.com/google/syzkaller. [Online]. Available: https://github.com/google/syzkaller

[26] D. Terpstra, H. Jagode, H. You, and J. Dongarra, "Collecting performance data with PAPI-C," in *3rd Parallel Tools Workshop*, 2009, pp. 157–173.

[27] V. Weaver, "perf_event validation tests," http://web.eece.maine.edu/ vweaver/projects/perf_events/validation/, 2014.

[28] T. Rantala, "[PATCH] perf: treat attr.config as u64 in perf_swevent_init()," http://marc.info/?l=linux-kernel&m=136588264507457.

[29] M. Kerrisk, "LCA: The Trinity fuzz tester," *Linux Weekly News*, Feb. 2013. [Online]. Available: http://lwn.net/Articles/536173/

[30] R. Swiecki and F. Gröbert, "honggfuzz," https://github.com/google/honggfuzz/. [Online]. Available: https://github.com/google/honggfuzz/

[31] J. Edge, "An unexpected perf feature," *Linux Weekly News*, May 2013. [Online]. Available: http://lwn.net/Articles/550901/

[32] W. Deacon, "Re: [PATCH v2 5/7] ARM: perf_event: Fully support Krait CPU PMU events," https://lkml.org/lkml/2014/1/21/323.

[33] S. Eranian, "Perfmon2: a flexible performance monitoring interface for Linux," in *Proc. 2006 Ottawa Linux Symposium*, Jul. 2006, pp. 269–288.