Coverage-guided kernel fuzzing with syzkaller

March 2, 2016
This article was contributed by David Drysdale

If your software deals with untrusted user input, it's a good idea to run a fuzzer against the program. For the Linux kernel, the most effective fuzzer of recent years has been Dave Jones's Trinity system call tester. But there's a new system call fuzzer in town, Dmitry Vyukov's syzkaller, and early results from it look promising — over 150 bugs uncovered in the mainline kernel (plus several dozen in Google's internal kernels) in a few months of operation.

Fuzzing in user space

The basic idea of fuzzing — feeding huge numbers of random inputs into a program and watching for crashes — has been around for a long time, but a naive implementation that just blindly emits random data is too inefficient to find all but the most shallow bugs. One technique for finding deeper bugs is to use a "template-based" fuzzer, which generates input variations from built-in knowledge about the possible/valid patterns (i.e. templates) for the program under test — information that needs to be manually created for each particular target (or class of targets).

However, more recently, "coverage-guided" fuzzers have appeared, notably Michał Zalewski's american fuzzy lop (which LWN covered back in September) and Clang's LibFuzzer, which operate without target-specific templates. Instead, these fuzzers work with an instrumented build of the binary under test, so that code coverage information is exposed. The fuzzer tries to maximize the amount of code covered (building an ever-expanding corpus of test inputs along the way), by mutating existing inputs and saving anything that hits new code.

As well as detecting out-and-out crashes, fuzzers also work well in combination with tools that expose latent bugs, such as Clang's sanitizers — compiler options that add instrumentation to the generated code so that incorrect behavior generates an error at run-time:

- AddressSanitizer (ASAN), which detects memory access errors.
- ThreadSanitizer (TSAN), which detects data races between different threads.
- MemorySanitizer (MSAN), which detects uninitialized reads: code whose behavior relies on memory contents that have not been initialized to a specific value.
- UndefinedBehaviorSanitizer (UBSAN), which detects the use of various features of C/C++ that are explicitly listed as resulting in undefined behavior.

(Most, but not all, of the sanitizers have been ported from Clang to GCC; however, it remains the case that the most useful tools appear first, or even exclusively, for Clang/LLVM — another reason to hope for the complete success of the LLVMLinux project.)

Fuzzing the kernel

The Linux kernel is certainly a piece of software that is exposed to untrusted user input, so it is an important target for fuzzing. The kernel is also sufficiently high-profile that it has been worth writing specific, template-based fuzzers for different areas of it, such as the filesystem or the perf_event subsystem. For the system call interface in general, the Trinity fuzz tester is the main tool that is currently used. It fuzzes the kernel in an intelligent way that is driven by per-system call templates.

In recent months, Vyukov and a team from Google have brought coverage-guided fuzz testing to the kernel with syzkaller, which uses a hybrid approach. As with Trinity, syzkaller relies on templates that indicate the argument domains for each system call, but it also uses feedback from code coverage information to guide the fuzzing.

The need for instrumentation does make syzkaller more complicated to set up than Trinity. To start with, the compiler option to generate the needed coverage data has only recently been added to GCC (as `-fsanitize-coverage=trace-pc`), so the kernel needs to be built with a fresh-from-tip version of GCC.

It is worth noting that Jones has considered feedback-guided fuzzing for Trinity in the past, but found the coverage tools that were available at the time to be too slow. The Google team behind syzkaller is primarily made up of compiler developers rather than kernel developers, so they may have an easier job of upgrading the tools to match the task in hand.

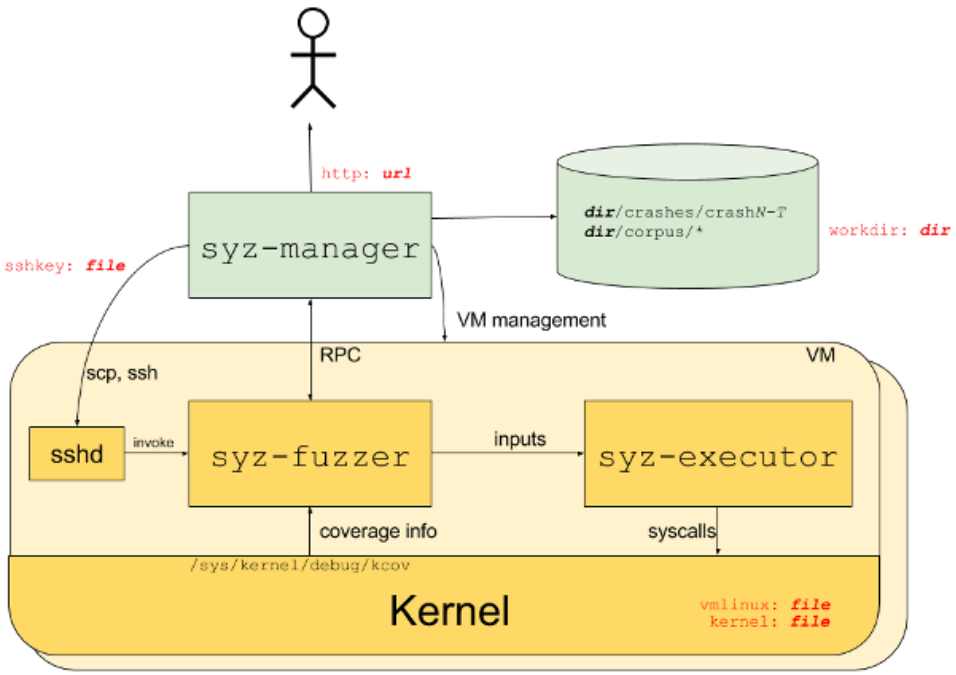
Another complication is that the coverage data needs to be tracked on a per-task basis and exported from the kernel to the outside world (via a debugfs entry at `/sys/kernel/debug/kcov`). The kernel patch to do this, and to invoke the relevant compiler options (all under `CONFIG_KCOV`), is currently under discussion but looks likely to be merged soon.

As mentioned above, the most effective bug-hunting occurs when the system call fuzzing is combined with tools that make latent bugs more visible. The kernel version of AddressSanitizer, KASAN, is the most straightforward of the sanitizers to enable (it is already included in the kernel as the `CONFIG_KASAN` build option), and it's also helpful to turn on various kernel debug features that expose incorrect use of internal kernel APIs, such as:

- `CONFIG_PROVE_LOCKING` to catch potential deadlocks.
- `CONFIG_PROVE_RCU` to catch potential bugs in RCU-using code.
- `CONFIG_DEBUG_ATOMIC_SLEEP` to find code that calls potentially-sleeping functions in an atomic section.

Using these options means that errors get emitted for bugs that might otherwise pass unnoticed ninety-nine times out of a hundred (but which are correspondingly harder to find and fix on the hundredth roll of the dice).

With these preliminaries in place, syzkaller can then be run over a set of QEMU virtual machines running the instrumented kernel under test. The structure of the various syzkaller processes is described by the diagram below, which was taken from the project's documentation (and where red text indicates configuration entries).



The results

To see the results of syzkaller in action, we attempt to reproduce a null-dereference bug in System V shared-memory processing that was first reported in October 2015. We speed up the process by narrowing the range of system calls tested to just those mentioned in that email thread, via the `enable_syscalls` parameter in syzkaller's configuration file. We also make sure our test kernel is built with full namespace support; this allows the fuzzer to run its tests in individual sandboxes that do not interfere with each other (using the `dropprivs` configuration flag). This is particularly useful when dealing with an interprocess resource like shared memory.

While the fuzzer is running, it provides a minimal web server to allow the user to see progress. The main status page displays fuzzing statistics and a list of the tested system calls; each of the latter provides links to further pages:

- A corpus page showing the sequences of system calls that have been run that include the given system call. For example, the page for `remap_file_pages()` might include "shmget-shmat-remap_file_pages" as a summary of particular sequence of system calls that has been run by the fuzzer.
- A coverage page that shows which parts of the kernel source code were hit (provided that the kernel was configured with `CONFIG_DEBUG_INFO` and `addr2line` is in the `PATH`), either during the processing of a specific corpus input or during all corpus inputs that include the given system call.
- A priority page that shows the biases used when randomly generating other system calls to run in combination with the given system call. These priorities are partly based on compatible argument types (for example, syzkaller

is more likely to combine two system calls that both take socket file descriptor arguments), and partly based on the frequency with which particular pairs of system calls appear in the current corpus (indicating that the pair has been effective in hitting new code in the past).

After running for a while, syzkaller generates a report file that includes a kernel oops; this file includes a log of the sequences of system calls that were being run, together with the log output for a null pointer dereference. Feeding the main fault address from the oops output into the `addr2line` tool reveals that the problem is in `shm_lock()`, which is being called from `shm_open()` as part of processing a `remap_file_pages()` system call.

However, we still have to narrow down the precise sequence that causes the problem, as the report file includes 204 distinct sequences of system calls. The `syz-repro` tool helps with this process; starting from the configuration file and the crash report file, it first narrows down to the particular sequence that triggers the crash — usually one of the few immediately preceding the log output. Next, it repeatedly attempts to *minimize* that particular sequence of system calls, by generating simpler versions of the sequence and checking that they still induce a crash.

In our example, after a few iterations of `syz-repro`, a fairly short sequence of system calls pops out:

```
mmap(&(0x7f0000000000)=nil, (0x2000), 0x3, 0x32, \
     0xffffffffffffffff, 0x0)
r0 = shmget(0x5, (0x2000), 0x200, &(0x7f0000b03000)=nil)
shmat(r0, &(0x7f0000b03000)=nil, 0x6000)
shmctl(r0, 0x3, &(0x7f0000000000+0xe4b)={ \
       0x3, <r1=>0xffffffffffffffff, 0x0, 0xffffffffffffffff, \
       0xffffffffffffffff, 0x1, 0xfa, 0x3, 0xee, 0x10000, 0x6520, \
       0x5, 0xffffffffffffffff, 0x0, 0x0})
shmctl(r0, 0xe, &(0x7f0000000000+0x28f)={ \
       0x1000, <r2=>0xffffffffffffffff, \
       <r3=>0xffffffffffffffff, 0x0, <r4=>0x0, 0x7, \
       0x100000000, 0x5, 0x6, 0x0, 0x2, 0x4, <r5=>0x0, \
       0xffffffffffffffff, 0xef0})
shmctl(r0, 0xc, &(0x7f0000002000-0x50)={ \
       0x80, r1, r4, r2, r3, 0x7, 0x10000, 0x5, 0xff, 0x80000000, \
       0x9, 0x3, r5, 0xffffffffffffffff, 0x2})
shmctl(r0, 0x0, &(0x7f0000001000-0x50)={ \
       0x1, 0x0, 0x0, 0xffffffffffffffff, 0x0, 0x1, 0x5, 0x5059, \
       0x3, 0x6301, 0x8001, 0xfffffffffffffffd, 0xffffffffffffffff, \
       0x0, 0x6})
remap_file_pages(&(0x7f0000b03000)=nil, (0x2000), 0x0, 0x7, \
                 0x21dd964cfba54855)
```

To confirm that this is a reproducible bug scenario, we feed this system call script into syzkaller's `syz-prog2c` utility, which generates a 100-line program that reproduces the problem on the test kernel.

At this point, a bit of human intervention helps to reduce the size of the program further. Looking at the `shmctl()` invocations, we notice that the first two calls are for `IPC_INFO` and `SHM_INFO`, both of which read values from the kernel rather than modifying anything. Next, we might also suspect that `SHM_UNLOCK` is a no-op, as nothing has been locked. After removing those calls and their data setup, we are left with an extremely short program that does indeed reproduce our null dereference (at least for now — a fix is on its way):

```
#include <unistd.h>
#include <sys/syscall.h>
#include <string.h>

long r[5];

int main()
{
    memset(r, -1, sizeof(r));
    r[0] = syscall(SYS_mmap, 0x20000000ul, 0x2000ul, 0x3ul, 0x32ul,
                   0xffffffffffffffffful, 0x0ul);
    r[1] = syscall(SYS_shmget, 0x5ul, 0x2000ul, 0x200ul, 0x20b03000ul, 0, 0);
    r[2] = syscall(SYS_shmat, r[1], 0x20b03000ul, 0x6000ul, 0, 0, 0);
    r[3] = syscall(SYS_shmctl, r[1], 0x0ul, 0x20000fb0ul, 0, 0, 0);
    r[4] = syscall(SYS_remap_file_pages, 0x20b03000ul, 0x2000ul,
                   0x0ul, 0x7ul, 0x21dd964cfba54855ul, 0);
    return 0;
}
```

Unfortunately, not all problems are as straightforward to reproduce and isolate as this one. Bugs may only be triggered by interactions between multiple test programs (when the `procs` configuration option is greater than one) if persistent or global resources are involved. More commonly, bugs may only be triggered by interactions between different threads in the same program; the fuzzing process deliberately executes system calls in parallel across multiple threads — which increases the chances of finding bugs at the cost of making it harder to narrow down the reproduction scenario. (Building the kernel with KTSAN enabled is particularly helpful for finding multithreaded problems, as it makes latent data races explicitly visible.)

To help with reproduction, syzkaller includes a tool (`syz-execprog`) for re-running a crash script under various options. The `-threaded` option governs whether the system call script is run across multiple threads, and (if it is) the `-collide`

option forces the threads to explicitly execute system calls in parallel. To catch heisenbugs, the `-repeat` option also allows the script to be re-run arbitrarily many times.

Although these tools don't guarantee a simple reproduction scenario, they seem to be effective in practice — the majority of the syzkaller-generated bug reports have included a short reproducer program, greatly simplifying the process of finding and fixing the underlying bug. The corpus of test inputs can be a helpful resource for quick regression testing of new kernel versions.

What's next

The syzkaller project is under active development, so things are moving fast. As mentioned above, the necessary patches for GCC have gone upstream and should appear in the next version; the concomitant kernel patch is being discussed. Once both are available by default, running syzkaller will only be slightly more inconvenient than running Trinity.

Because syzkaller is a hybrid of a template-based and a coverage-guided fuzzer, it does work best when provided with information about the usage patterns of system calls. To that end, the syzkaller developers are keen to work with kernel developers so that support for particular kernel subsystems can be reviewed and extended (which may well involve making the system call template mechanisms more sophisticated). They would also like to extend architecture support beyond the current somewhat x86_64-specific situation, and would like to further automate the process of extracting a reproducer program (and minimizing the size of that program).

But overall, syzkaller appears to be a worthy addition to the battery of kernel test tools, and its successes reinforce the idea that fuzzing should be considered a best practice for any software project that takes user input.

Coverage-guided kernel fuzzing with syzkaller
Posted Mar 2, 2016 6:16 UTC (Wed) by **pr1268** (subscriber, #24648) [Link]

Silly dyslexia—for a moment there I thought Dmitry's new fuzzing tool was called *skywalker*. ;-)

Many thanks to Mr. Vyukov for creating this tool, and also to Mr. Drysdale for the article.

> **Coverage-guided kernel fuzzing with syzkaller**
> Posted Mar 4, 2016 2:08 UTC (Fri) by **jtc** (guest, #6246) [Link]
>
> "'Silly dyslexia'—for a moment there I thought Dmitry's new fuzzing tool was called skywalker. ;-)"
>
> I think I'd call it "fuzzy dyzlexia"!
>
> > **Coverage-guided kernel fuzzing with syzkaller**
> > Posted Mar 4, 2016 10:11 UTC (Fri) by **vonbrand** (guest, #4458) [Link]
> >
> > They are fuzzying our dyslexia with the name ;-)

Coverage-guided kernel fuzzing with syzkaller
Posted Mar 2, 2016 7:57 UTC (Wed) by **SimonKagstrom** (subscriber, #49801) [Link]

Impressive work, and thanks for the well-written article!

Somewhat related, I've also written a code coverage tool called kcov:

https://github.com/SimonKagstrom/kcov

which as of now collects code coverage for userspace programs using breakpoints. However, the 'k' in the name was meant to be 'kernel', and the idea was to use kprobes and debugfs to setup and report breakpoints in the kernel. I never got that working reliably though, so the kernel part has never really been activated. I suppose I should write a backend for CONFIG_KCOV now though :-)

> **Coverage-guided kernel fuzzing with syzkaller**
> Posted Mar 3, 2016 14:19 UTC (Thu) by **gerdesj** (subscriber, #5446) [Link]
>
> "Impressive work, and thanks for the well-written article!"
>
> Seconded and given that the author makes a statement like this: "Unfortunately, not all problems are as straightforward to reproduce and isolate as this one. " he clearly knows how to communicate effectively to those with a rather lesser knowledge of these things.
>
> Great stuff.

Coverage-guided kernel fuzzing with syzkaller
Posted Mar 2, 2016 17:20 UTC (Wed) by **deater** (subscriber, #11746) [Link]

It's nice to see the kernel fixes getting in, especially the perf_event ones.

This tool turned up many of the same remaining open perf_fuzzer-found bugs, but either because the reports are better or else because the maintainers got prodded more, bug-fixing got kicked into high gear.

Coverage-guided kernel fuzzing with syzkaller
Posted Mar 2, 2016 23:16 UTC (Wed) by **PaXTeam** (guest, #24616) [Link]

> To start with, the compiler option to generate the needed coverage data has only recently been added to GCC
> (as -fsanitize-coverage=trace-pc), so the kernel needs to be built with a fresh-from-tip version of GCC.

the gcc-side instrumentation is very simple and could be done from a plugin, thus extending support all the way back to gcc 4.5.

CII Best practices – please comment!
Posted Mar 3, 2016 16:46 UTC (Thu) by **david.a.wheeler** (subscriber, #72896) [Link]

Thanks for the reference to the Core Infrastructure Initiative (CII) Best Practices (in the link to dynamic_analysis)!! We would *love* for more people to review our draft criteria, and then comment on the issue tracker or mailing list. The more specific, the better.