LF live

MENTORSHIP SERIES

LF live MENTORSHIP SERIES

Fuzzing the Linux Kernel

Andrey Konovalov
Senior Software Engineer, Google

March 2nd 2021

Who am I?

- Andrey Konovalov
- Work on Linux kernel bug detectors, fuzzers, and exploit mitigations
  - KASAN, syzkaller, Memory Tagging
- xairy.github.io
- @andreyknvl

LF Live MENTORSHIP SERIES

My experience with Linux kernel fuzzing

- Network fuzzing via syscalls
  - 3 LPE exploits

- External network fuzzing

- External USB fuzzing
  - 300+ bugs

Disclaimer

- I'm biased:
  - I use syzkaller (a state-of-the-art Linux kernel fuzzer)

- But this is:
  - Not another syzkaller talk
  - A getting-started overview of the Linux kernel fuzzing field
  - A best-effort compilation of ideas, tips, and references

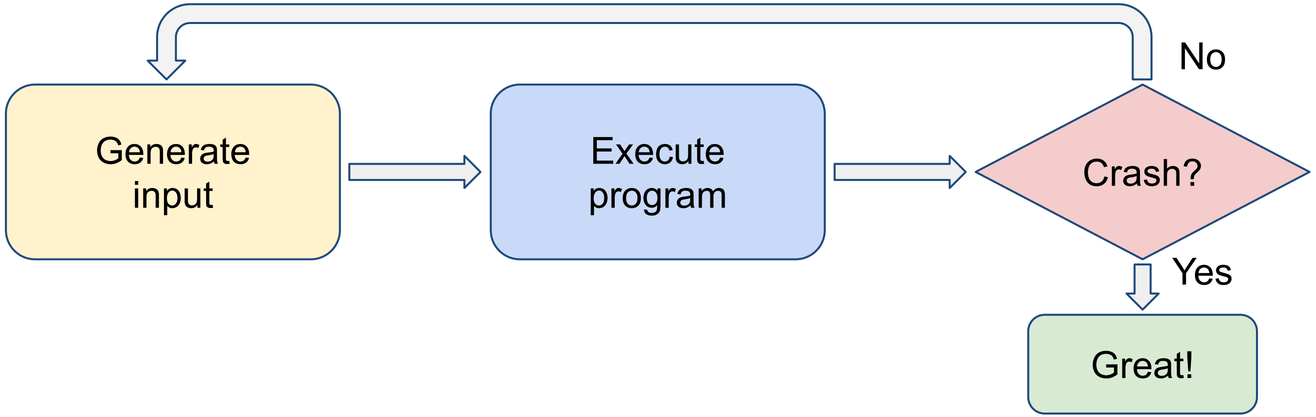
Agenda

- Fuzzing
- Fuzzing the Linux kernel
  - Overview
  - Trinity and syzkaller
  - Approaches
  - Tips
- Collecting coverage with KCOV
- Final notes

Fuzzing

Fuzzing

- Fuzzing — feeding in random inputs until the program crashes



Programs

- Fuzzing — feeding in random inputs until the program crashes

- Programs:
  - Application
  - Library
  - Kernel
  - Firmware
  - ...

Fuzzing

- Fuzzing — feeding in random inputs until the program crashes

- — How do we execute the program?
- — How do we inject inputs?
- — How do we generate inputs?
- — How do we detect crashes (or other kinds of bugs)?
- — How do we automate the process?

Fuzzing

- Fuzzing — feeding in random inputs until the program crashes

- — How do we execute the program?
- — How do we inject inputs?
- — **How do we generate inputs?**
- — How do we detect crashes?
- — How do we automate the process?

Depend on the target (program)

Generating inputs

- Let's say we have an XML file parser
- How do we generate inputs for it when fuzzing?

- Idea #1: just generate random data

Random inputs

```
if (input[0] == '<')
    if (input[1] == 'x')
        if (input[2] == 'm')
            if (input[3] == 'l')
                // Need to reach at least here.
```

- Parser expects the file to start with "`<xml`" header
- Fuzzer needs **~2^32 guesses** to get past the header check

Better inputs

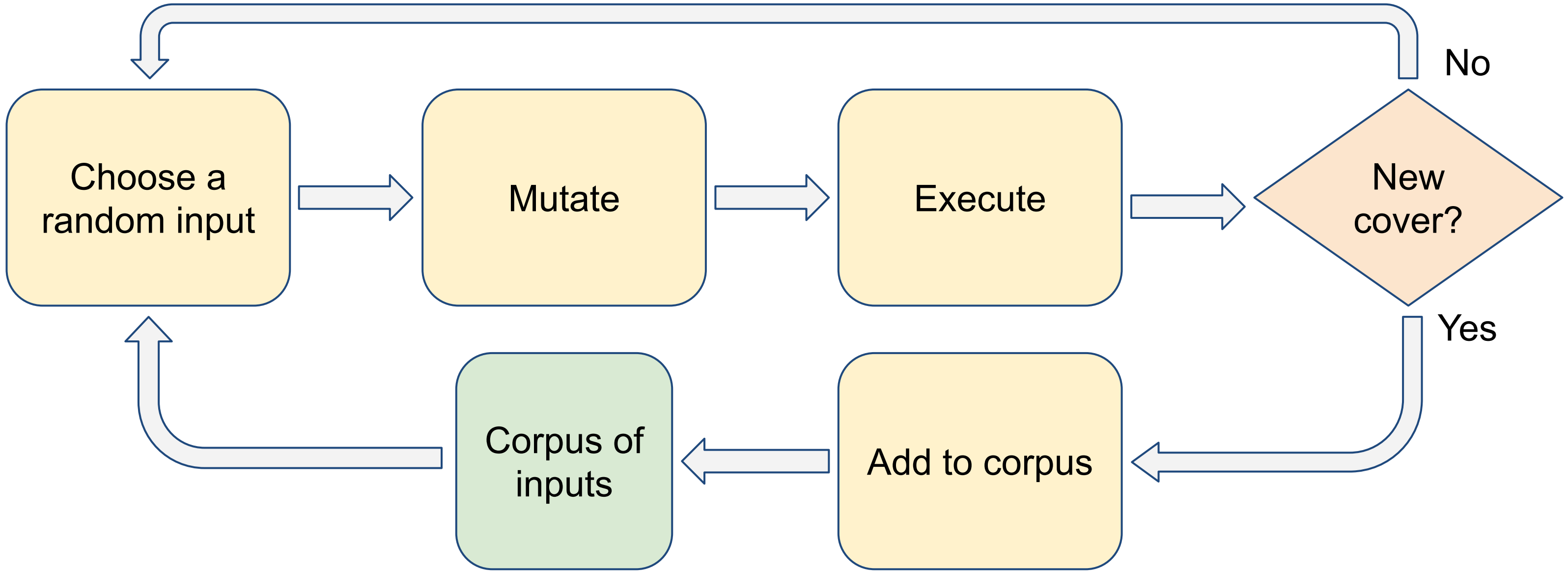
- Random binary data works poorly as inputs
- So what should we do?
- Generate better inputs
- How?
  a. Structured inputs (a.k.a. structure-aware fuzzing)
  b. Guided generation (e.g. coverage-guided fuzzing)
  c. Collecting a corpus of sample inputs and mutating them

MENTORSHIP SERIES

Structured inputs

```
XML_GRAMMAR = {
    "<start>": ["<xml-tree>"],
    "<xml-tree>": ["<text>", "<xml-open-tag><xml-tree><xml-close-tag>",
                   "<xml-openclose-tag>", "<xml-tree><xml-tree>"],
    "<xml-open-tag>":      ["<<id>>", "<<id> <xml-attribute>>"],
    "<xml-openclose-tag>": ["<<id>/>", "<<id> <xml-attribute>/>"],
    "<xml-close-tag>":     ["</<id>>"],
    "<xml-attribute>" :    ["<id>=<id>", "<xml-attribute> <xml-attribute>"],
    "<id>":                ["<letter>", "<id><letter>"],
    "<text>" :             ["<text><letter_space>","<letter_space>"],
    "<letter>":            srange(string.ascii_letters + string.digits +"\""+"'"+"."),
    "<letter_space>":      srange(string.ascii_letters + string.digits +"\""+"'"+" "+"\t"),
}
```

Coverage-guided generation



Guided generation

- Types of signal
  - Code coverage (thus, coverage-guided fuzzing)
  - Memory state
  - ...
- Can combine with structured inputs approach and mutate accordingly
  - Inserting/removing tags in case of XML

Collecting corpus

- Collecting a set of sample input
  - XML files in case of XML
- Mutating them and feeding into the program

- Can combine with the previous two approaches
  - Need to parse samples to mutate with structure awareness

Understanding fuzzing better

- Write a fuzzer from scratch
  - Build simple fuzzer by Michal Melewski
  - Fuzzing Like A Caveman by h0mbre

Fuzzing the Linux kernel

Fuzzing

- Fuzzing — feeding in random inputs until the program crashes

- — How do we execute the program?
- — How do we inject inputs?
- — How do we generate inputs?
- — How do we detect crashes (or other kinds of bugs)?
- — How do we automate the process?

Kernel fuzzing

- Fuzzing — feeding in random inputs until the kernel crashes

- 🟦 — How do we run the kernel?
- 🟧 — How do we inject inputs?
- 🟨 — How do we generate inputs?
- 🟪 — How do we detect crashes (or other kinds of bugs)?
- 🟩 — How do we automate the process?

Kernel fuzzing

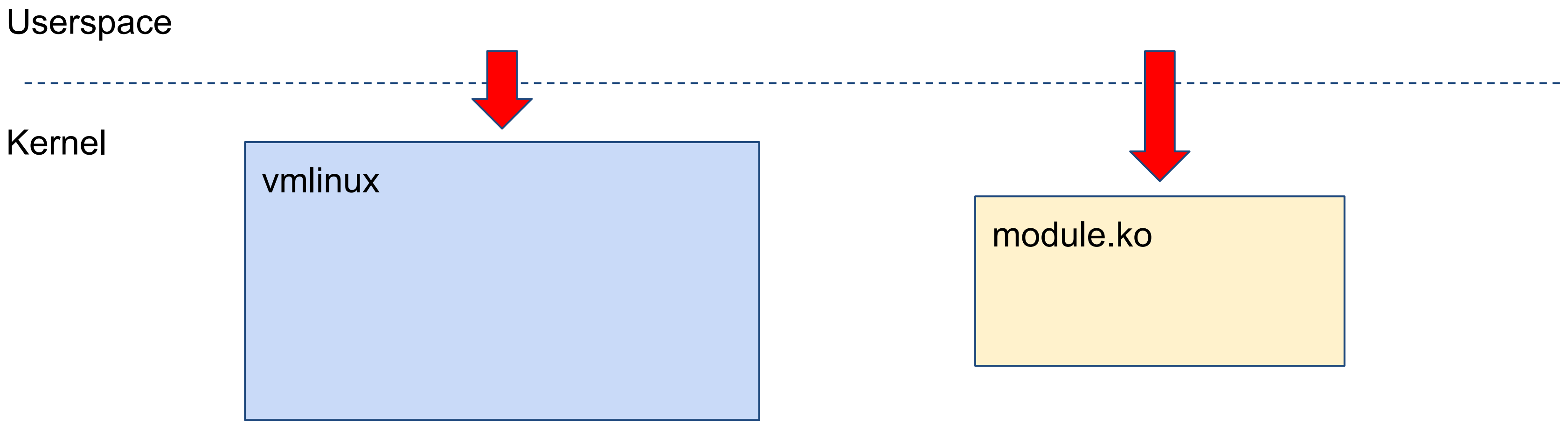
- Fuzzing — feeding in random inputs until the kernel crashes

- — How do we run the kernel?
- **— How do we inject inputs?**
- **— How do we generate inputs?**
- — How do we detect crashes (or other kinds of bugs)?
- — How do we automate the process?

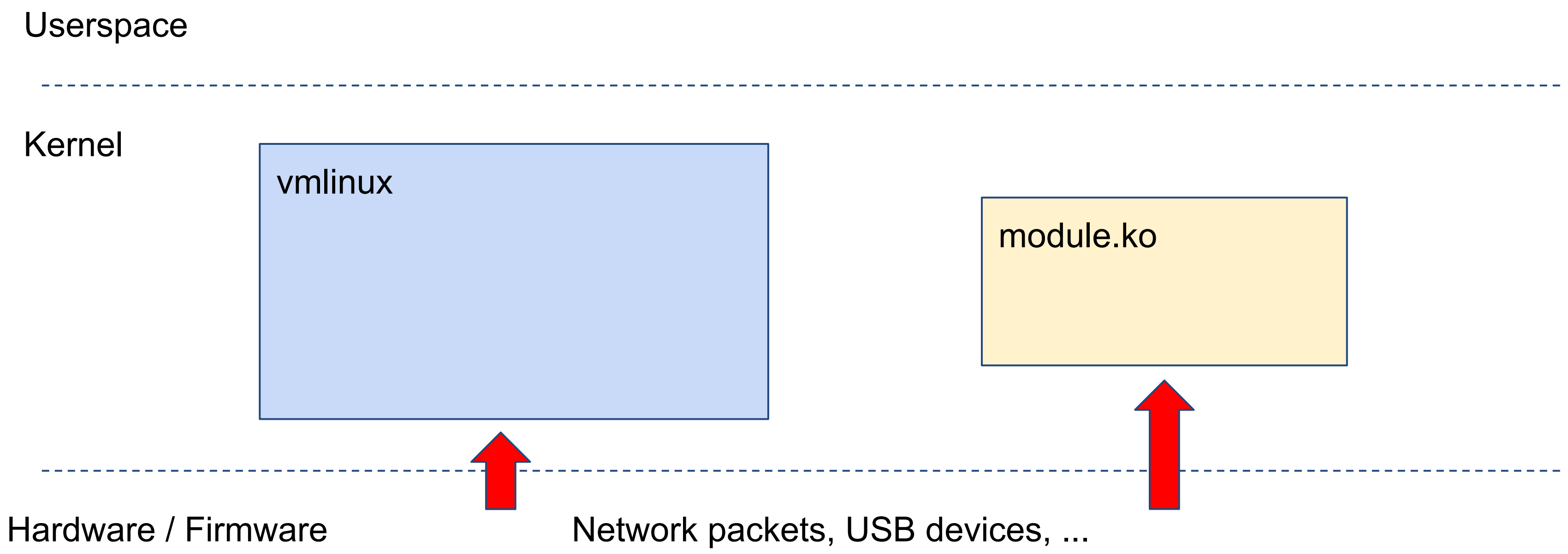
Kernel inputs

- What inputs does the kernel have?

Kernel inputs: syscalls



Kernel inputs: external



Injecting inputs

- Syscalls
  - Execute a binary
- External
  - Either from userspace or through hypervisor/emulator
  - Userspace
    - Network: `/dev/tun`; USB: `/dev/raw-gadget` + Dummy UDC/HCD
  - Hypervisor/emulator:
    - USB: QEMU + usbredir (vUSBf)

Generating inputs

- Dumb fuzzer generates random blobs
- Smarter fuzzer generates structured blobs

- But the kernel doesn't accept blobs as inputs
  - (Except when limiting fuzzing surface to e.g. a single syscall)

Input structure: syscalls

- Most syscalls are used as an API
  - A sequence of calls
  - Arguments are structured
  - Return values / output fields of structures are used in subsequent calls

```
int fd = open("/dev/something", …);
ioctl(fd, &{0x10, ...});
close(fd);
```

Input structure: syscalls

- Most syscalls are used as an API
  - A sequence of calls
  - Arguments are structured
  - Return values / output fields of structures are used in subsequent calls
- => API-aware fuzzing
  - Inputs are API call sequences
  - Generated and mutated accordingly

Input structure: other syscalls

- Not all syscalls work as straightforward API
- Or accept simple structures as arguments

- clone, sigaction
  - API with callbacks?

- eBPF, KVM (also netfilter?)
  - Need to generate valid code
  - Script-aware fuzzing? (Something like fuzzilli?)

Input structure: external

- Network packets
  - Might seem like blobs
  - More like API due to TCP SYN/ACK numbers, SCTP cookies, etc.

- USB (also FUSE?) is weird
  - Host-driven communication
  - The fuzzer is responding to API calls
  - Not knowing which call will be next

Generating inputs

- Input "structures":
  - API
  - API with callbacks
  - Scripts
  - USB-like stuff

- Different from typical random/structured blobs
- A fuzzer should generate and mutate inputs accordingly

Code coverage

- Compiler instrumentation
  - KCOV
  - Other hacks piggy-backing on top of GCC/Clang
- Emulator
  - TriforceAFL via QEMU
  - Unicorefuzz via Unicorn
- Hardware tracing features
  - kAFL via Intel PT

Kernel fuzzing

- Fuzzing — feeding in random inputs until the kernel crashes

- — **How do we run the kernel?**
- — How do we inject inputs?
- — How do we generate inputs?
- — **How do we detect crashes (or other kinds of bugs)?**
- — **How do we automate the process?**

Running the kernel

	Physical device	VM / Emulator
Fuzzing surface	Native (includes device drivers)	Only what the VM supports
Management (restarting, debugging, getting kernel logs)	Hard, hardware gets bricked	Easy
Scalability	Buy more devices	Spawn more VMs

Detecting bugs

- Dmitry Vyukov gave a talk about this last week
- Mentorship Session: Dynamic Program Analysis for Fun and Profit [slides]

- TL;DR: Use dynamic bug detectors
  – KASAN, KMSAN, KCSAN, …

- Write your own detectors
  – Checks for logical bugs, asserts, etc.

Automation

- Monitoring kernel log for crashes
- Restarting crashed VMs
- Deduplicating crashes
- Generating reproducers
- Reporting bugs / tracking fixes

- (All that other fun stuff syzkaller/syzbot do)

Trinity and syzkaller

Trinity

- Trinity (and similar fuzzers) in essence:

```
while (true)
    syscall(rand(), rand_fd(), rand_struct_of_proper_type());
```

- Infinite stream of syscalls
- API-aware
- No guidance

syzkaller vs Trinity

- syzkaller ~= Trinity +
  - notion of a test case (including isolation) +
  - coverage-guidance (using KCOV) +
  - language to describe API/structures (syzlang) +
  - automation (scalability, reproducers, dashboards, syzbot)

- syzkaller: goes deeper, finds more bugs, easier to extend
- Trinity: finds less bugs, easier to deploy as a drop-in binary

LF Live MENTORSHIP SERIES

Fuzzing approaches

Approaches

- Building kernel code as userspace app and fuzzing that
- Reusing a userspace fuzzer (AFL, libFuzzer, …)
- Using syzkaller
- Writing a fuzzer from scratch

Building in userspace

- Works for code that is separable from the rest of the kernel
- No need to bother with emulators/hypervisors

- github.com/iovisor/bpf-fuzzer
- Kernel Fuzzing in Userspace (fuzzing ASN.1) by Eric Sesterhenn

Reusing a userspace fuzzer

- Take a userspace fuzzer (AFL, libFuzzer, …)
- Interact with the kernel instead of calling into e.g. a userspace library
- Need to plug kernel coverage into the fuzzer

- Works fine for fuzzing blob-like inputs: filesystem images, netlink, etc.
- But other kernel inputs aren't blobs => Need custom generators/mutators
- SockPuppet: A Walkthrough of a Kernel Exploit for iOS 12.4 by Ned Williamson
  - (Turning structure-aware fuzzing into API-aware with libprotobuf-mutator)

Using syzkaller

- See syzkaller talks for usage
- Good at fuzzing API-based interfaces out-of-the-box

- Tip #1: Don't just fuzz mainline with the default config
  - Add new descriptions
  - Tighten attack surface: fuzz a small number of related syscalls
  - Fuzz distro kernels

syzkaller is extensible

- Tip #2: Build your fuzzer on top of syzkaller
  - Coverage-Guided USB Fuzzing with Syzkaller [slides] by Andrey Konovalov
  - KVM: dev_kvm.txt, common_kvm_amd64.h, ifuzz

- Tip #3: Use syzkaller as a framework
  - Only use crash parsing code
  - Only use VM management code
  - ...

Writing a fuzzer from scratch

- Might be beneficial for targeted fuzzing
- E.g. when the interface is not API-based

- For inspiration:
  - Writing the world's worst Android fuzzer, and then improving it by Brandon Falk
  - Fuzzing for eBPF JIT bugs in the Linux kernel by Simon Scannell
  - Fuzzing the Linux kernel (x86) entry code by Vegard Nossum

Fuzzing tips

Read the code

- Understand the code you're fuzzing
  - What kind of inputs it expects
  - Which part you are trying to target

- Write a fuzzer based on that
  - Writing fuzzer based on specs/docs does not work well

Is my fuzzer good?

- Check code coverage, make sure you cover the targeted layer
- Inject bugs (`WARN_ON()/BUG_ON()`) and check that fuzzer finds them
- Revert fixes for bugs/CVEs and check that fuzzer finds them

Fast vs smart

- Fast fuzzer
  - More execs/sec
- Smart fuzzer
  - Better input generation
  - Relevant guidance signal

- Focus on smart in the first place
  - Formal investigation would be interesting; related paper and discussion

LF Live MENTORSHIP SERIES

Collecting coverage with KCOV

KCOV overview

- A tool for collecting code coverage from the Linux kernel
- Available upstream, enabled with `CONFIG_KCOV`
- Based on compiler instrumentation => need to rebuild the kernel
- Collects coverage from:
  - User threads (i.e. kernel code that handles syscalls)
  - Background thread and softirqs (with kernel code annotations)

Instrumentation

- GCC/Clang pass that inserts a function call into every basic block

```
if (...) {
    ...
}
```

```
__sanitizer_cov_trace_pc();  // 1
if (...) {
     __sanitizer_cov_trace_pc();  // 2
    ...
}
__sanitizer_cov_trace_pc();  // 3
```

Userspace interface

- Kernel DebugFS extension that exposes coverage per-thread

```
int fd = open("/sys/kernel/debug/kcov", ...);
unsigned long *cover = mmap(NULL, ..., fd, 0);
ioctl(fd, KCOV_ENABLE, ...);
// Now, coverage from the current kernel thread is collected into cover.
// Each __sanitizer_cov_trace_pc() call saves its PC.
```

Relevant coverage

- KCOV collects coverage from the current user thread (by default)
  - This is deliberate: ignoring irrelevant code executed in background

- Problem: an input might trigger relevant background code
  - Syscall handler passing work to a global worker thread
  - Opening a devices spawns a thread that handles it

Background coverage

- Solution: annotating relevant kernel code

```
void background_thread() {
      kcov_remote_start(UNIQUE_ID);  // Start collecting coverage associated with UNIQUE_ID.
      ...
      kcov_remote_stop();  // Stop collecting coverage.
}
```

- But how to pass `UNIQUE_ID` from userspace?

Global and local background threads

- Global background threads
  - Spawned from init code during boot

- Local background threads
  - Spawned from syscall handlers
  - Attached to a user-owned instance of a device

Global threads

- No easy way to pass `UNIQUE_ID` from userspace
- Use predefined `UNIQUE_ID`

Global threads

```
void hub_event() {  // Handles USB devices, executed in global background thread, one thread per USB bus.
      kcov_remote_start(kcov_remote_handle(KCOV_SUBSYSTEM_USB, bus_num));  // Start collecting coverage
      ...                                                                  // into a dedicated buffer.
      kcov_remote_stop();  // Copy collected coverage to the connected KCOV device.
}
```

```
int fd = open("/sys/kernel/debug/kcov", ...);
unsigned long *cover = mmap(NULL, ..., fd, 0);
ioctl(fd, KCOV_REMOTE_ENABLE, {..., handles = {kcov_remote_handle(KCOV_SUBSYSTEM_USB, bus_num)}, ...});
// Now, coverage from global background kernel thread is collected into cover.
```

Local threads

- Can pass `UNIQUE_ID` from userspace

Local threads

```
long vhost_dev_set_owner(struct vhost_dev *dev) {  // Called when opening /dev/vhost.
      dev->kcov_handle = kcov_common_handle();  // current->kcov_handle
      worker = kthread_create(vhost_worker, dev, "vhost-%d", current->pid);
}
```

```
int fd = open("/sys/kernel/debug/kcov", ...);
unsigned long *cover = mmap(NULL, ..., fd, 0);
ioctl(fd, KCOV_REMOTE_ENABLE, {..., common_handle = getpid(), ...});  // current->kcov_handle = PID
// Now, coverage from local background kernel threads is collected into cover.
```

LF Live MENTORSHIP SERIES

Local threads

```
long vhost_dev_set_owner(struct vhost_dev *dev) {  // Called when opening /dev/vhost.
      dev->kcov_handle = kcov_common_handle();
      worker = kthread_create(vhost_worker, dev, "vhost-%d", current->pid);
}
```

```
static int vhost_worker(struct vhost_dev *dev) {
                  kcov_remote_start_common(dev->kcov_handle);
                  work->fn(work);
                  kcov_remote_stop();
}
```

Multiprocess fuzzing

- When fuzzing from multiple processes in one VM
- Global threads
  - Need a dedicated thread per each fuzzing process
  - USB: each fuzzing process gets its own USB bus
- Local threads
  - Just use a unique `common_handle` for each process (process number)

Final notes

Note #1

- Developing fuzzers is engineering
  - You have to be good at writing code (besides reading it for review)

Note #2

- Good fuzzers find too many bugs
  - Not all of them dangerous (fuzzing became the new static analysis :)
  - And not all of them get fixed :(

- Distilling the bugs that matter?
  - Automatically detecting bugs that are exploitable?

Linux kernel fuzzing materials

- Articles/papers:
  - github.com/xairy/linux-kernel-exploitation#vulnerability-discovery
  - wcventure.github.io/FuzzingPaper/#kernel-fuzzing
  - syzkaller docs: research ; syzkaller docs: talks

- People to follow
  - @dvyukov, @gamozolabs, whoever else's work was linked in this talk

- Telegram channel with links on Linux kernel security: t.me/linkersec

LF live MENTORSHIP SERIES

Thank you for joining us today!

We hope it will be helpful in your journey to learning more about effective and productive participation in open source projects. We will leave you with a few additional resources for your continued learning:

- The LF Mentoring Program is designed to help new developers with necessary skills and resources to experiment, learn and contribute effectively to open source communities.
- Outreachy remote internships program supports diversity in open source and free software
- Linux Foundation Training offers a wide range of free courses, webinars, tutorials and publications to help you explore the open source technology landscape.
- Linux Foundation Events also provide educational content across a range of skill levels and topics, as well as the chance to meet others in the community, to collaborate, exchange ideas, expand job opportunities and more. You can find all events at events.linuxfoundation.org.