phd Positive Hack Days ORIGIN

Fuzzing the Linux kernel

Andrey Konovalov, xairy.io

May 20th 2021

Who am I?

- Andrey Konovalov

- Work on Linux kernel bug detectors, fuzzers, and exploit mitigations
  - KASAN, syzkaller, Memory Tagging

- xairy.io
- @andreyknvl

My experience with Linux kernel fuzzing

- Network fuzzing via syscalls
  - 3 LPE exploits
- External network fuzzing
- External USB fuzzing
  - 300+ bugs

Agenda

- Fuzzing
- Fuzzing the Linux kernel
    - Legacy
    - Foundation
    - Charged
- Approaches
- Tips
- Final note

Concepts, from simplest to most involved

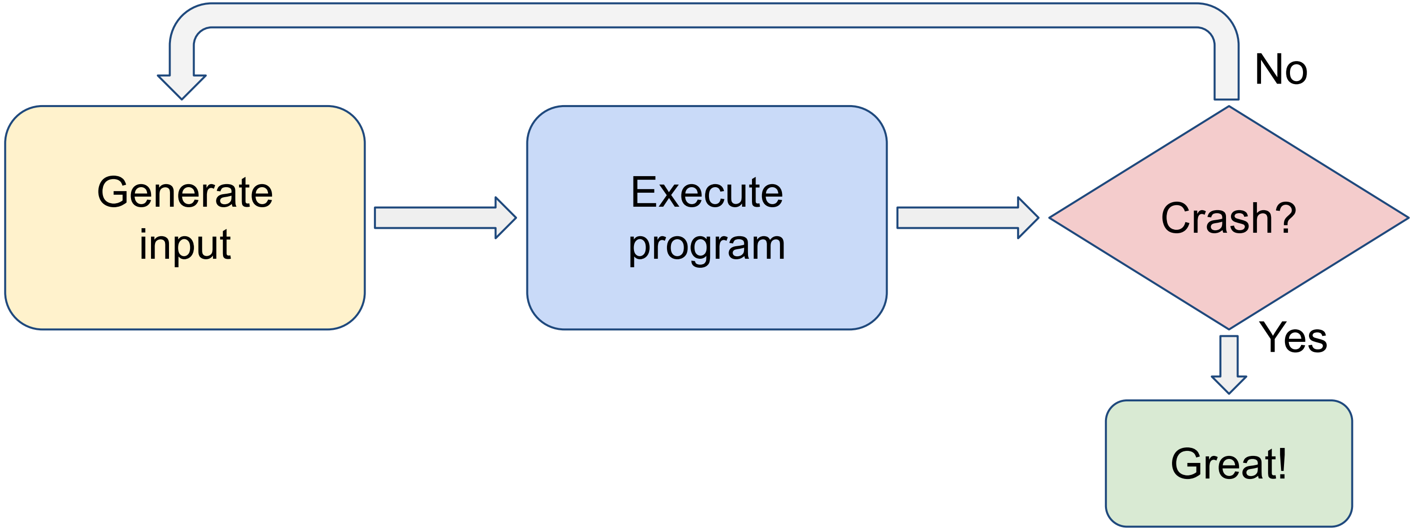
ФСТЭК

phd Positive Hack Days ORIGIN

Fuzzing

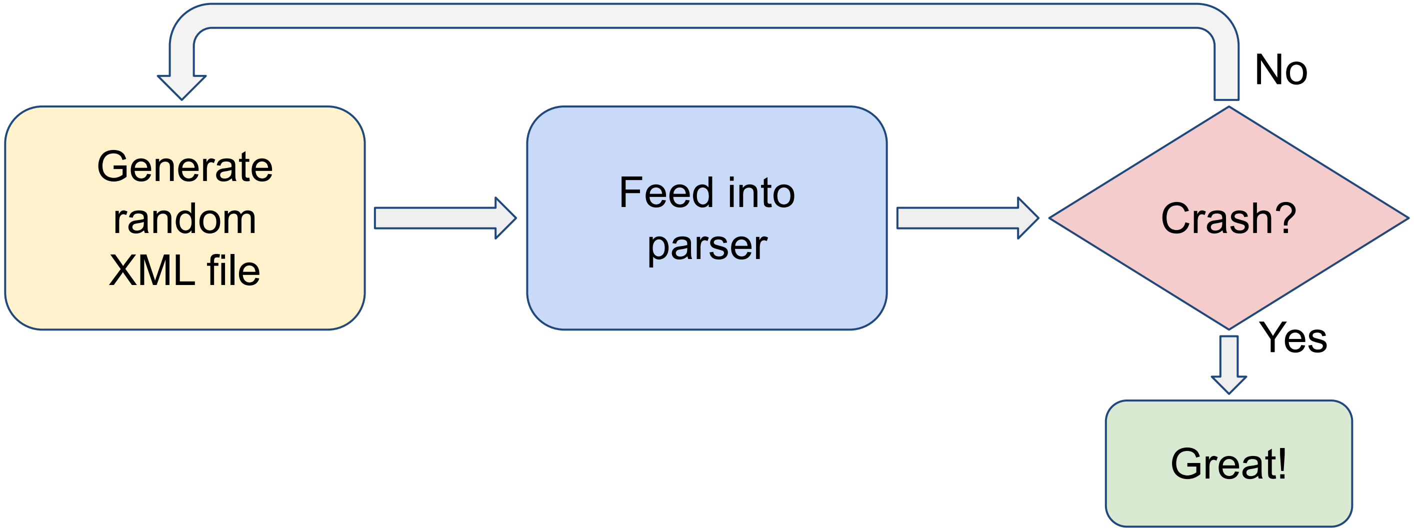
Fuzzing

- Fuzzing — feeding in random inputs until the program crashes



Fuzzing an XML parser

- Fuzzing — feeding in random XML files until the parser 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?
- — What are inputs?
- — How do we inject inputs?
- — How do we generate inputs?
- — How do we detect bugs?
- — How do we automate the process?

Kernel fuzzing

- Fuzzing — feeding in random inputs until **the kernel** crashes

- — How do we run **the kernel**?
- — What are inputs?
- — How do we inject inputs?
- — How do we generate inputs?
- — How do we detect bugs?
- — How do we automate the process?

phd Positive Hack Days ORIGIN

Fuzzing the Linux kernel: Legacy

Running the kernel

- Fuzzing — feeding in random inputs until the kernel crashes

- — **How do we run the kernel?**
- — What are inputs?
- — How do we inject inputs?
- — How do we generate inputs?
- — How do we detect bugs?
- — How do we automate the process?

Running the kernel

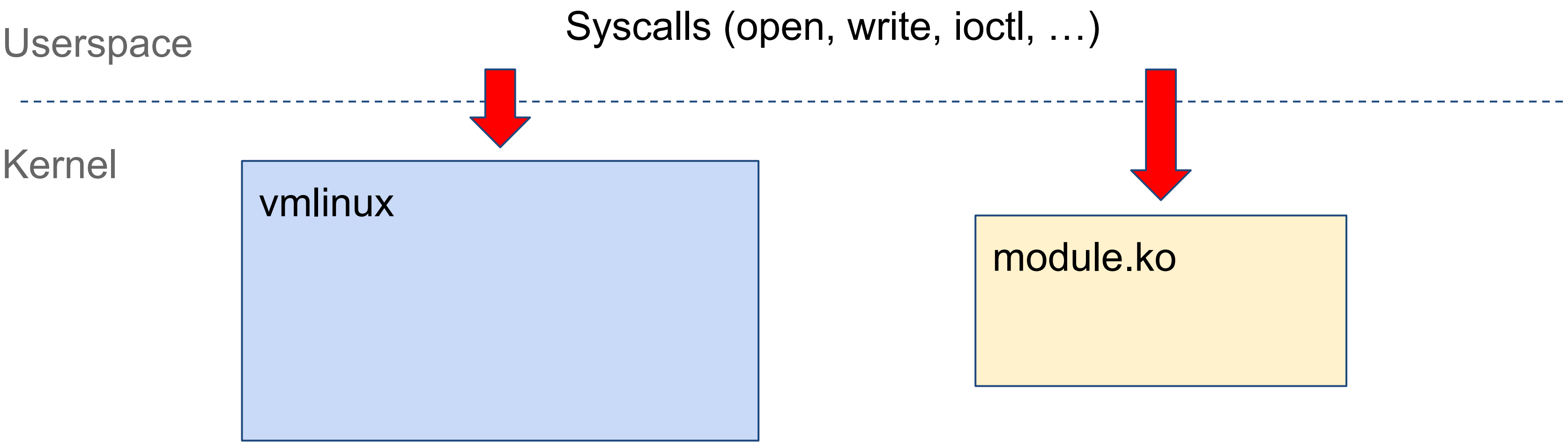
	Physical device	VM (e.g. QEMU)
Fuzzing surface	Native (includes device drivers)	Only what the VM supports
Management (restarting, debugging, getting kernel logs)	Hard; hardware gets bricked	Easy
Scalability	Buy more devices	Spawn more VMs

Kernel inputs

- Fuzzing — feeding in random inputs until the kernel crashes

- — How do we run the kernel? QEMU or physical device
- — **What are inputs?**
- — How do we inject inputs?
- — How do we generate inputs?
- — How do we detect bugs?
- — How do we automate?

Kernel inputs



Legacy approach

- Fuzzing — feeding in random inputs until the kernel crashes

—	How do we run the kernel?	QEMU or physical device
—	What are inputs?	Syscalls
—	**How do we inject inputs?**	Execute a binary
—	How do we generate inputs?	
—	How do we detect bugs?	
—	How do we automate?	

Works everywhere!

Generating inputs

- Fuzzing — feeding in random inputs until the kernel crashes

— How do we run the kernel?	QEMU or physical device
— What are inputs?	Syscalls
— How do we inject inputs?	Execute a binary
— **How do we generate inputs?**	
— How do we detect bugs?	
— How do we automate?	

Generating inputs for userspace apps

- In case of 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, duh

- How?
  1. Structured inputs (a.k.a. structure-aware fuzzing)
  2. [Discussed later]
  3. [Discussed later]

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"),
}
```

Generating kernel inputs

- Can generate structured blobs
- But the kernel does not accept blobs as inputs
    - (Except when limiting fuzzing surface to e.g. a single syscall)

Example of a kernel input

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

Example of a kernel input

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

- A sequence of calls

Example of a kernel input

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

- A sequence of calls
- Arguments are structured

Example of a kernel input

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

- A sequence of calls
- Arguments are structured
- Return values (output fields of structures) used in subsequent calls

Example of a kernel input

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

- A sequence of calls
- Arguments are structured
- Return values (output fields of structures) used in subsequent calls

- Syscalls are used as an API

API-aware fuzzing

- Fuzzer knows about API calls and their arguments
  - Need to describe APIs manually for the kernel
    - (No way to generate them automatically)

- Fuzzer remembers and then uses return/output values
  - Example: keep a list of opened file descriptors of each type

Legacy approach

- Fuzzing — feeding in random inputs until the kernel crashes

— How do we run the kernel?	QEMU or physical device
— What are inputs?	Syscalls
— How do we inject inputs?	Execute a binary
— How do we generate inputs?	API-awareness
— How do we detect bugs?	Kernel panics
— How do we automate?	`while (true) syscall(…)`

This is Trinity!

Fuzzing the Linux kernel: Foundation

Foundational approach

- Fuzzing — feeding in random inputs until the kernel crashes

- — How do we run the kernel?	QEMU or physical device
- — What are inputs?	Syscalls
- — How do we inject inputs?	Execute binary
- — How do we generate inputs?	
- — How do we detect bugs?	
- — How do we automate?	

Generating inputs

- Fuzzing — feeding in random inputs until the kernel crashes

—	How do we run the kernel?	QEMU or physical device
—	What are inputs?	Syscalls
—	How do we inject inputs?	Execute binary
—	**How do we generate inputs?**	
—	How do we detect bugs?	
—	How do we automate?	

Better inputs

- Random binary data works poorly as inputs
- So what should we do?
- Generate better inputs, duh

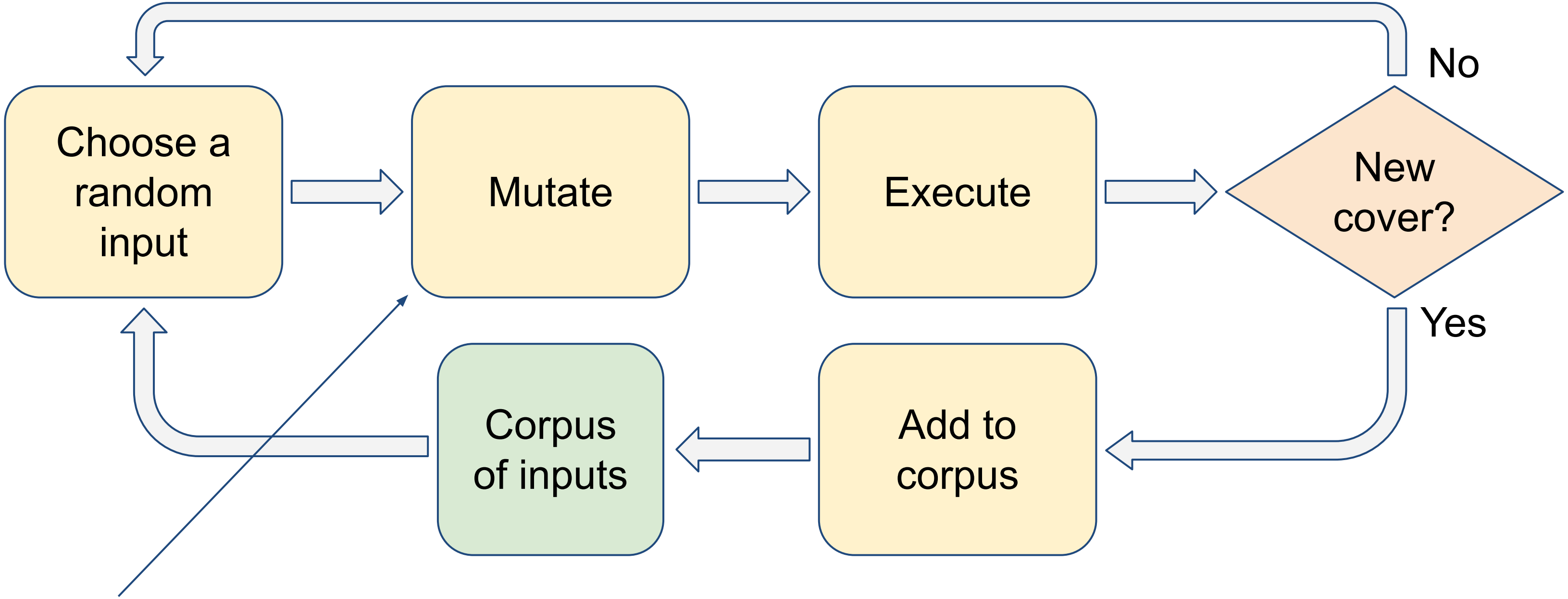
- How?
  1. Structured inputs (a.k.a. structure-aware fuzzing)
  2. [Discussed later]
  3. [Discussed later]

Better inputs

- Random binary data works poorly as inputs
- So what should we do?
- Generate better inputs, duh

- How?
  1. Structured inputs (a.k.a. structure-aware fuzzing)
  2. Coverage-guided generation (a.k.a coverage-guided fuzzing)
  3. [Discussed later]

Coverage-guided generation



Mutate according to the structure (e.g. insert/remove XML tags)

Applying to the kernel

- Need a notion of a test case
  - Unlike infinite stream of calls the legacy approach had
  - => Generate (and mutate) finite API-call sequences

- Need a way to collect relevant code coverage
  - Relevant — only from code that handles syscalls
  - => Use KCOV
    - Based on compiler instrumentation
    - Collects coverage from the current task context

Running the kernel

- Fuzzing — feeding in random inputs until the kernel crashes

— How do we run the kernel?	QEMU or physical device
— What are inputs?	Syscalls
— How do we inject inputs?	Execute binary
— How do we generate inputs?	API-awareness + KCOV
— **How do we detect bugs?**	
— How do we automate?	

Detecting kernel bugs

- Kernel panic is not a good indicator
    - Some bugs are not panics (e.g. info-leaks)
    - Other bugs do not panic immediately (e.g. memory corruptions)

- Use dynamic bug detectors
    - Dynamic — finds bugs that happen during execution

Dynamic bug detectors for the kernel

- Most notable: KASAN — detects memory corruptions
  - Slab/stack/global out-of-bounds, use-after-frees, etc.

- There are many more: UBSAN, KMSAN, KCSAN, ...
- Dynamic Program Analysis for Fun and Profit [slides]
  by Dmitry Vyukov

- Note: detectors not tied to fuzzer, can use with Trinity as well

Running the kernel

- Fuzzing — feeding in random inputs until the kernel crashes

— How do we run the kernel?	QEMU or physical device
— What are inputs?	Syscalls
— How do we inject inputs?	Execute binary
— How do we generate inputs?	API-awareness + KCOV
— How do we detect bugs?	KASAN and others
— **How do we automate?**	

Automation

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

- How? Write code!

Foundational approach

- Fuzzing — feeding in random inputs until the kernel crashes

—	How do we run the kernel?	QEMU or physical device
—	What are inputs?	Syscalls
—	How do we inject inputs?	Execute binary
—	How do we generate inputs?	API-awareness + KCOV
—	How do we detect bugs?	KASAN and others
—	How do we automate?	All that mentioned fancy stuff

This is syzkaller! (in its base)

PHD Positive Hack Days ORIGIN

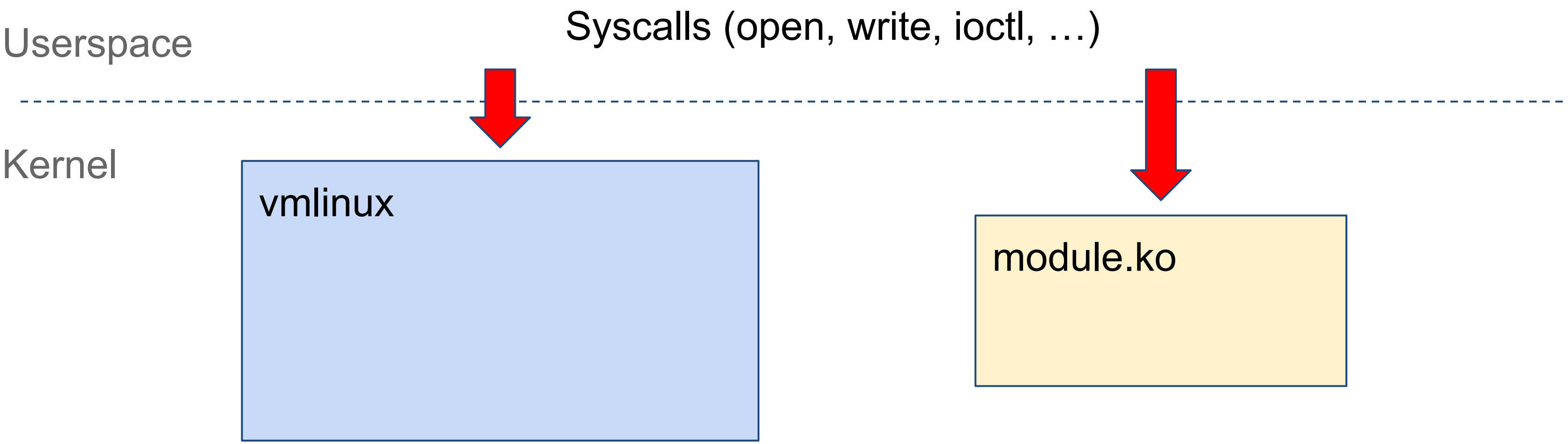
Fuzzing the Linux kernel: Charged

Running kernel code in userspace

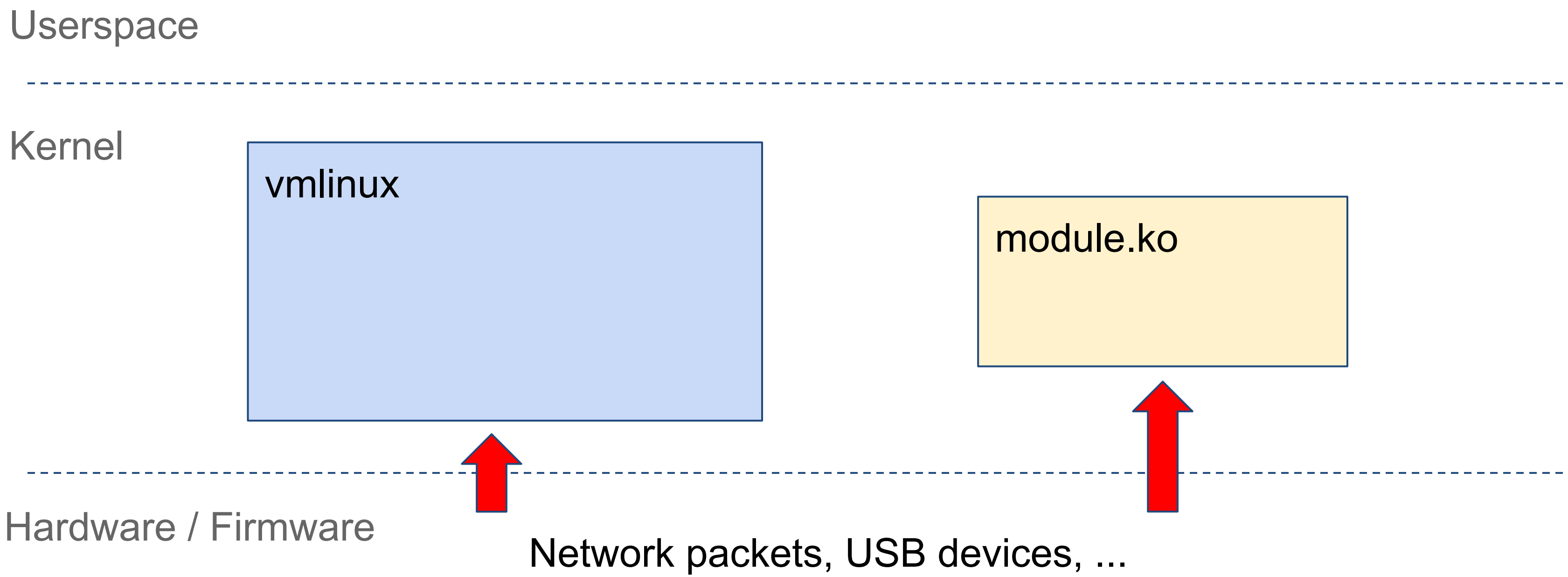
- Works for code that is separable from the rest of the kernel
- No need to bother with emulators/hypervisors
- Downside: hard to maintain and scale

- github.com/iovisor/bpf-fuzzer
- Kernel Fuzzing in Userspace (fuzzing ASN.1) by Eric Sesterhenn
- Designing sockfuzzer, a network syscall fuzzer for XNU by Ned Williamson
  - (Turning structure-aware fuzzing into API-aware with libprotobuf-mutator)

Kernel inputs: syscalls



Kernel inputs: external



Userspace

Kernel

vmlinux

module.ko

Hardware / Firmware

Network packets, USB devices, ...

Injecting external inputs

- Unlike syscalls, can not simply execute a binary
- Inject either from userspace or through hypervisor

- Userspace
  - Network: `/dev/tun`
  - USB: `/dev/raw-gadget` + Dummy UDC/HCD

- Hypervisor/emulator
  - USB: QEMU + usbredir (vUSBf)

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

Collecting code coverage

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

Relevant code coverage

- Collecting coverage for the current task works in many cases

- But relevant code might be executed in a different context
  - Example: syscall uses workers to process input
  - Example: USB control packets are processed in global threads

- KCOV supports collecting coverage from background threads and interrupts via custom annotations

Beyond code coverage

- Code coverage is not the only relevant guidance signal
  - Memory state
  - Object state
  - ...

Better inputs

- Random binary data works poorly as inputs
- So what should we do?
- Generate better inputs, duh

- How?
  1. Structured inputs (a.k.a. structure-aware fuzzing)
  2. Coverage-guided generation (a.k.a coverage-guided fuzzing)
  3. [Discussed later]

Collecting a corpus of samples

- Random binary data works poorly as inputs
- So what should we do?
- Generate better inputs, duh

- How?
  1. Structured inputs (a.k.a. structure-aware fuzzing)
  2. Coverage-guided generation (a.k.a coverage-guided fuzzing)
  3. Collect a corpus of sample inputs and mutate them
     - Moonshine uses `strace`

Detecting more bugs

- Modify existing bug detectors
  - KASAN annotations for custom allocators (mempool)
  - Add info-leak checks for KMSAN for external buses (for USB)

- Write your own bug-detectors
  - Simple `BUG_ON()` assertions
  - More intricate checks for logical bugs

Fuzzing approaches

Fuzzing approaches

- Reusing a userspace fuzzer
- Using syzkaller
- Writing a fuzzer from scratch

Reusing a userspace fuzzer

- Take a userspace fuzzer (AFL, libFuzzer, …)
- Interact with the kernel instead of calling into a userspace library
- Or run kernel code in userspace

- Works fine for fuzzing blob-like inputs: filesystem images, netlink, etc.
- Other kinds of inputs => Need custom generators/mutators
- Need to plug kernel coverage into the fuzzer for coverage-guidance

- Designing sockfuzzer, a network syscall fuzzer for XNU by Ned Williamson

Using syzkaller

- See syzkaller talks for usage
- Good for fuzzing API-based interfaces out-of-the-box
- Custom language to describe API/structures (syzlang)

- Tip #1: Do not 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 me
  - 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

- Great way to learn
- Might be beneficial for targeted fuzzing
- Or if 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

PHD Positive Hack Days ORIGIN

Fuzzing tips

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

Read the code

- Understand the code you are 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

Fast vs smart

- Fast fuzzer
  - More execs/sec
- Smart fuzzer
  - Better input generation, more relevant guidance signal, etc.

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

phd Positive Hack Days ORIGIN

Final note

Writing fuzzers is engineering

- Based on engineering skills
  - Designing systems
  - Writing code
  - Testing and debugging
  - Benchmarking

- => You need basic programming skills to get started
- => You need decent engineering skills to excel

Linux kernel fuzzing materials

- Articles and papers:
  - github.com/xairy/linux-kernel-exploitation#finding-bugs
  - wcventure.github.io/FuzzingPaper/#kernel-fuzzing
  - syzkaller docs: research
  - syzkaller docs: talks

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

Thank you for your attention!

phd Positive Hack Days ORIGIN

Andrey Konovalov, xairy.io

May 20th 2021