Week 12: Lecture A

Kernel Fuzzing

Monday, April 1, 2024

How are semester projects going?

Smoothly?

Obstacles?

The Next Few Weeks

Part 4: New Frontiers in Fuzzing

Monday Meeting	Wednesday Meeting
Apr. 01 **Fuzzing OS Kernels** ▶ Readings:	Apr. 03 **LLM-guided Fuzzing** ▶ Readings:
Apr. 08 **Fuzzing Compilers** (guest lecture by John Regehr) ▶ Readings:	Apr. 10 **Fuzzing Hardware** ▶ Readings:
Apr. 15 **Fuzzing Multi-language Software** ▶ Readings:	Apr. 17 **Final Presentations I**
Apr. 22 **Final Presentations II**	Apr. 24 **No Class (Reading Day)**

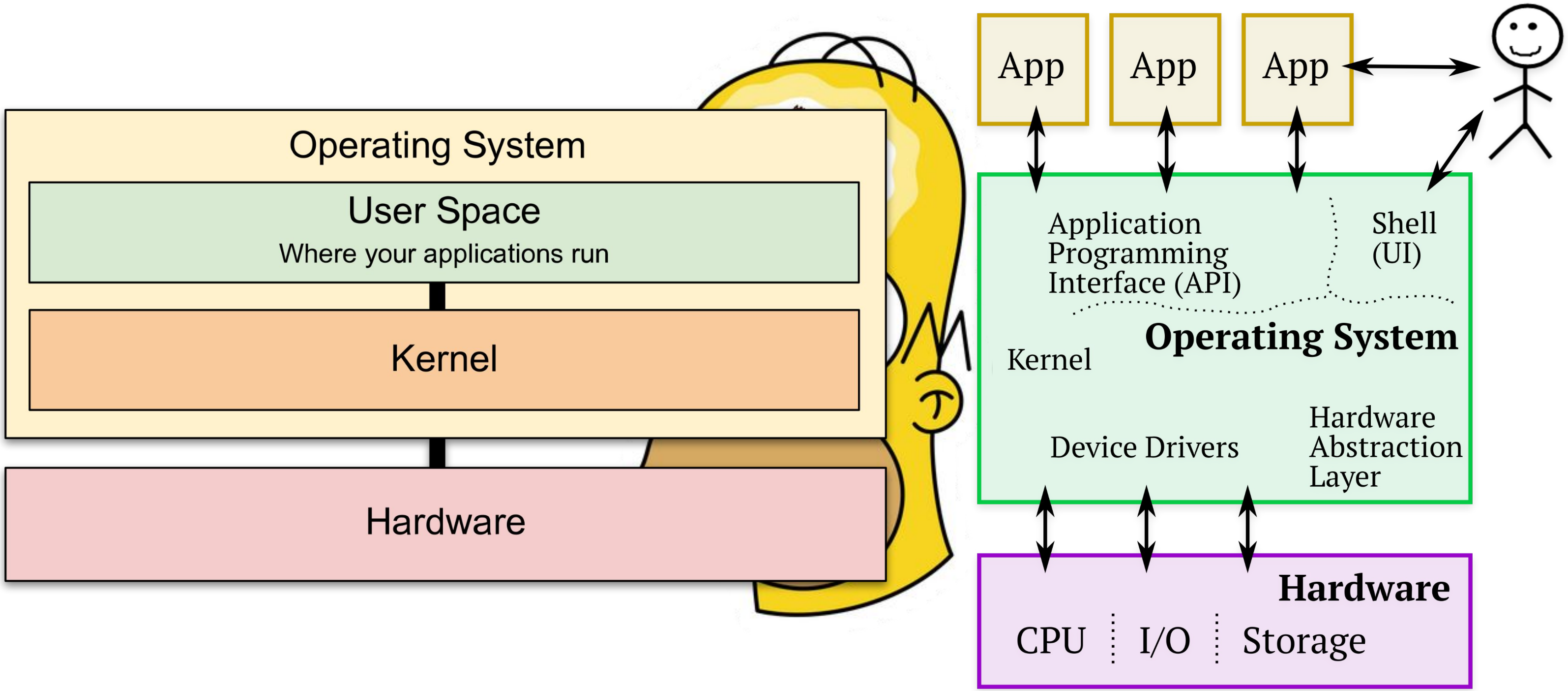
Recap: Project Schedule

- **Apr. 17th & 22nd:** final presentations
  - ~~15–20~~ **5-minute** slide deck and discussion
  - What you did, and why, and what results
- We have 26 teams…
  - So, 13 teams per two days
  - **5 minute presentation each**
  - One-minute audience Q&A
  - Keep the details tight!
- What's most important:
  - High-level technique
  - Challenges and workarounds
  - Key results (bugs found, other successes, etc.)

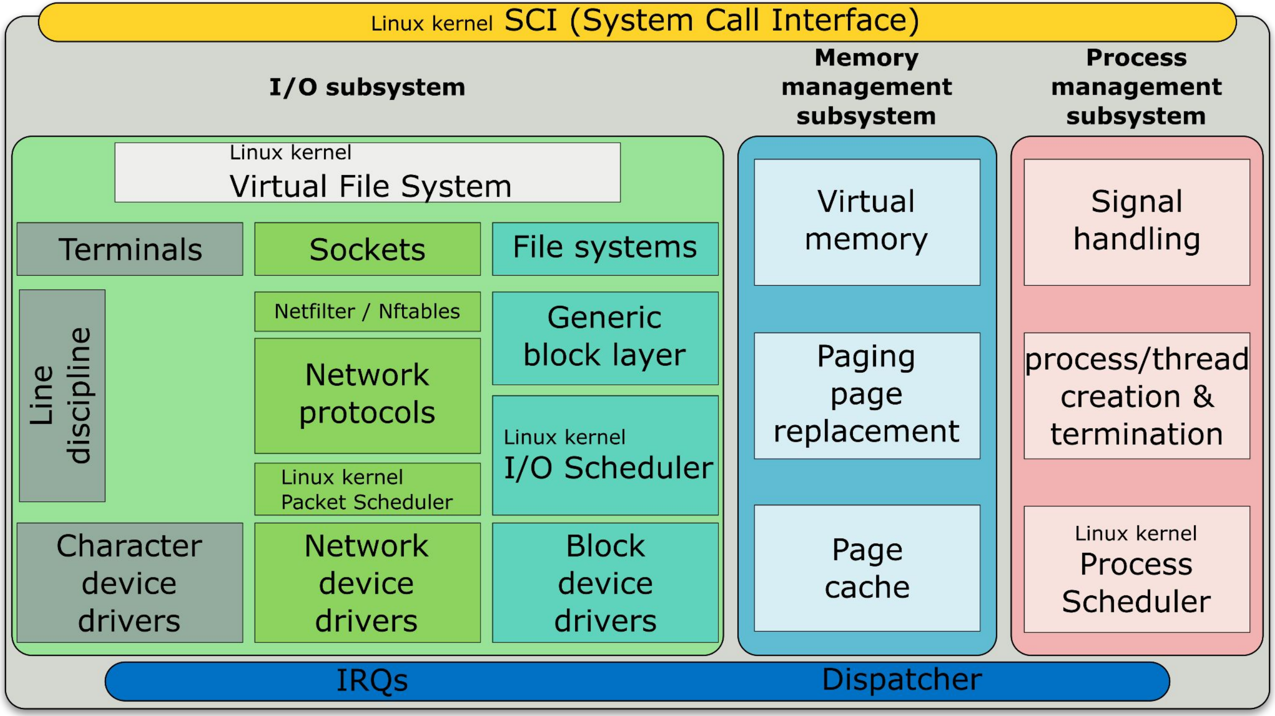
Questions?

Kernels

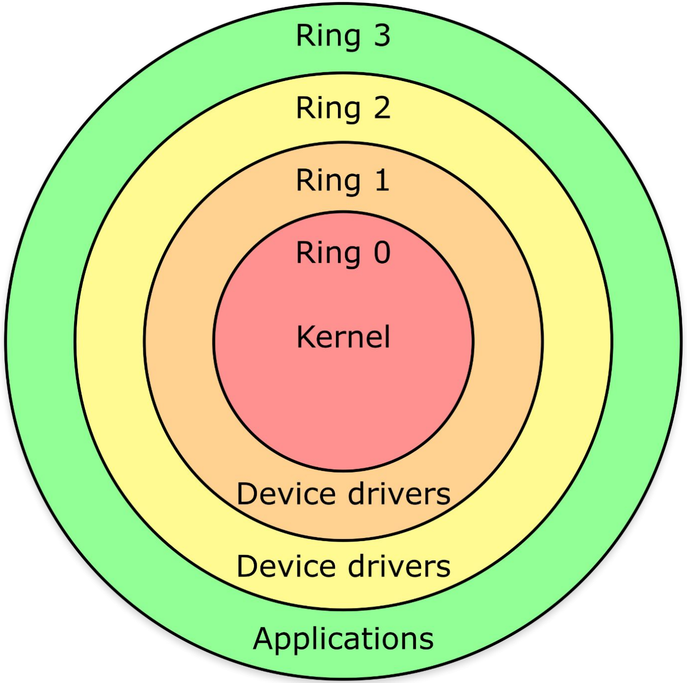
What are kernels?

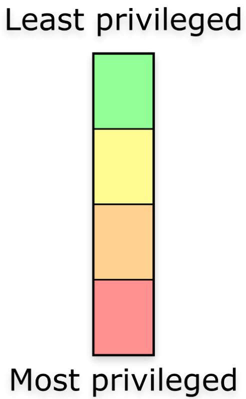


What does a kernel even do?



Why fuzz kernels?

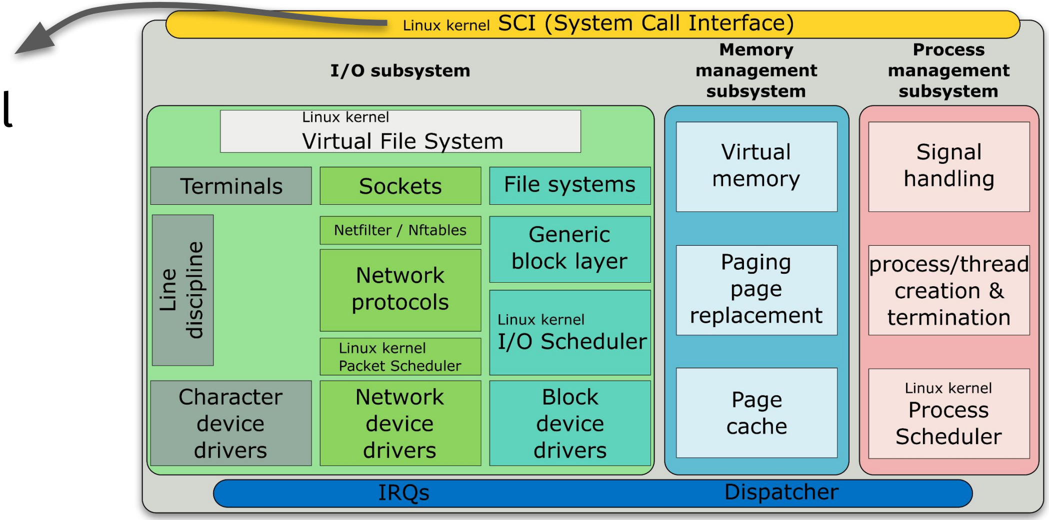




Fuzzing Kernels

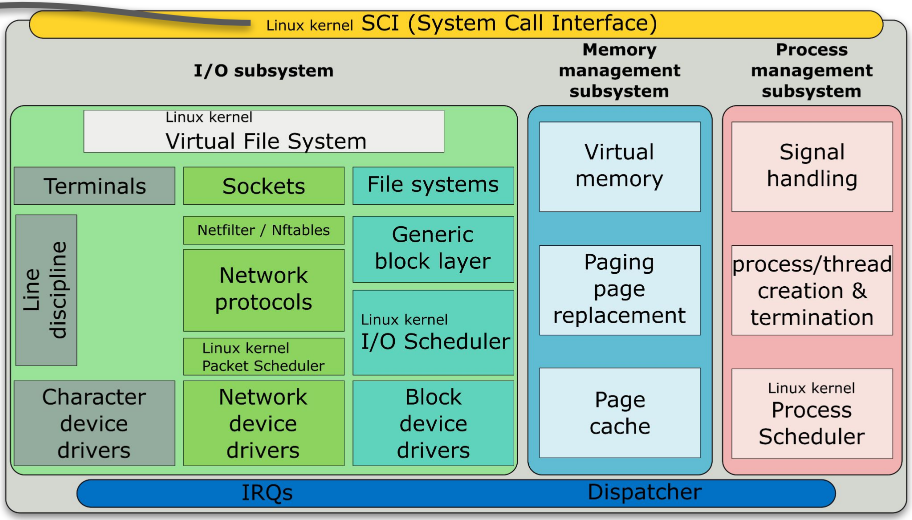
How can we even fuzz a kernel?

- **System calls** = the “interface” for sending data to the kernel



How can we even fuzz a kernel?

- **System calls** = the "interface" for sending data to the kernel
- App fuzzers generate testcases containing **random bytes of data**
- Kernel fuzzers generate programs containing **random system calls**
  - Random syscall sequences
  - Random syscall arguments



Kernel Fuzzing Challenges

Kernel Fuzzing Challenges

- Feedback:
  - Must instrument or emulate entire kernel… slow!
  - Sanitizers require total rewriting to support kernels

Kernel Fuzzing Challenges

- Feedback:
  - Must instrument or emulate entire kernel... slow!
  - Sanitizers require total rewriting to support kernels

- Execution:
  - Way more code being executed than applications
  - Running on bare metal = unrecoverable crashes
  - Running in a VM is better, but sacrifices performance

Kernel Fuzzing Challenges

- Feedback:
  - Must instrument or emulate entire kernel... slow!
  - Sanitizers require total rewriting to support kernels
- Execution:
  - Way more code being executed than applications
  - Running on bare metal = unrecoverable crashes
  - Running in a VM is better, but sacrifices performance
- “Weird” stuff:
  - Other processes, threads, interrupts, non-determinism
  - Unreproducible crashes (largely caused by the above)

Early Kernel Fuzzers

- Basic test case structure:
  - Totally random parameters
  - If known, use correct types

```
while (1){
    syscall(rand(), rand(), rand());
    syscall(rand_fd(), rand_addr());
}
```

Early Kernel Fuzzers

- Basic test case structure:
  - Totally random parameters
  - If known, use correct types

```
while (1){
    syscall(rand(), rand(), rand());
    syscall(rand_fd(), rand_addr());
}
```

- **Problems?**
  - **???**

Early Kernel Fuzzers

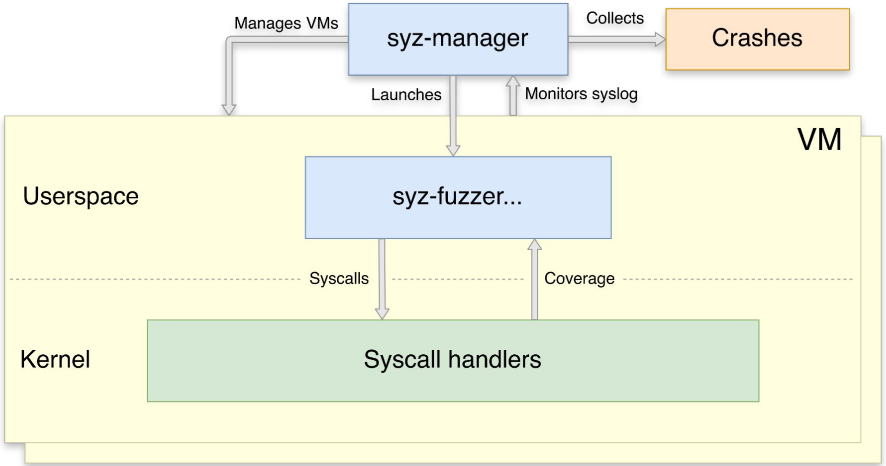
- Basic test case structure:
  - Totally random parameters
  - If known, use correct types

- **Problems?**
  - Incorrect ordering
  - Little/no dataflow
  - No PoC reproducers
  - **Finds shallow bugs**

```
while (1){
    syscall(rand(), rand(), rand());
    syscall(rand_fd(), rand_addr());
}
```

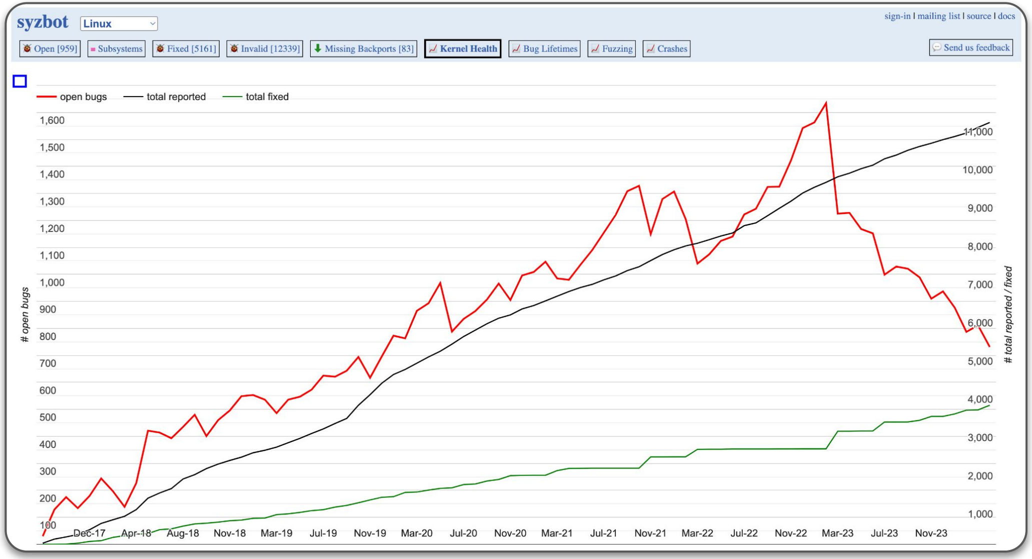
SyzKaller

- Joint effort by Google and the Linux kernel dev team
- Continuous kernel fuzzing and crash reporting



SyzKaller

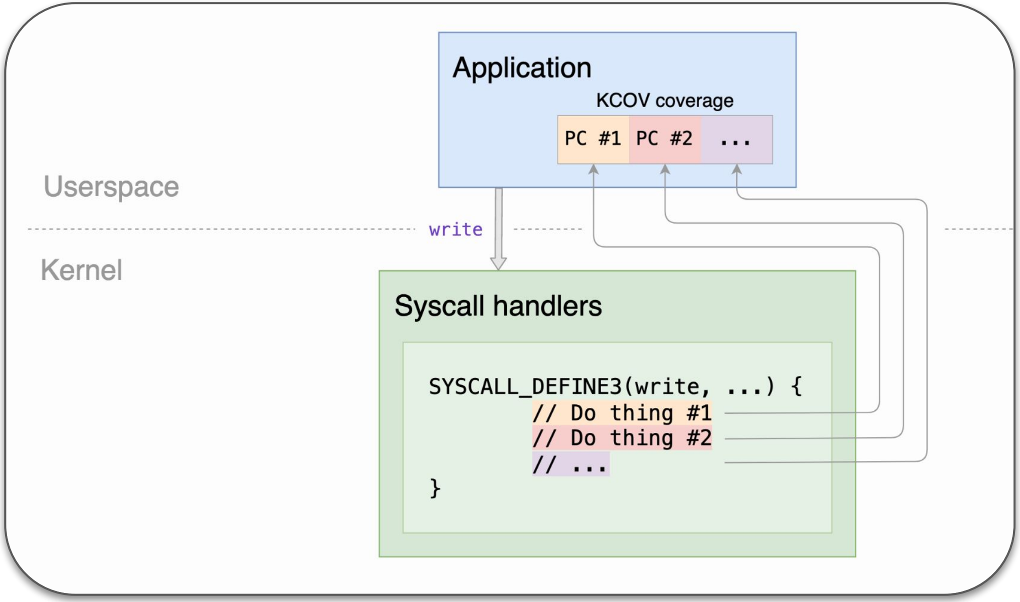
- Joint effort by Google and the Linux kernel dev team
- Continuous kernel fuzzing and crash reporting
- By far the most successful kernel fuzzing effort ever



SyzKaller's Code Coverage: KCov

- Compiler instrumentation
  - Basic block level callbacks
  - Runtime lib to record coverage

- Exposes coverage via interface `/sys/kernel/debug/kcov`
  - User-mode fuzzing process reads
  - Orchestration via Syz-Manager that operates outside of the VM



SyzLang: SyzKaller's Description Language

- **Key idea:** bring structure-aware mutation to kernel fuzzing

```
open (file ptr[in, filename], flags flags[open_flags]) fd
read (fd fd, buf ptr[out, array[int8]], count bytessize[buf])
close (fd fd)

open_flags = O_RDONLY, O_WRONLY, O_RDWR, O_APPEND
```

Source: *syzkaller: adventures in continuous coverage-guided kernel fuzzing*

SyzLang: SyzKaller's Description Language

- **Key idea:** bring structure-aware mutation to kernel fuzzing
  - Syscall names and args
  - Flow between syscalls

```
open (file ptr[in, filename], flags flags[open_flags]) fd
read (fd fd, buf ptr[out, array[int8]], count bytessize[buf])
close (fd fd)

open_flags = O_RDONLY, O_WRONLY, O_RDWR, O_APPEND
```

Source: *syzkaller: adventures in continuous coverage-guided kernel fuzzing*

SyzLang: SyzKaller's Description Language

- Given a SyzLang description, SyzKaller will **fill-in the data**

SyzLang description for struct foo

```
foo {
    f1 int32
    f2_len len[f2, int16]
    f3_len len[f3, int8]
    f2 array[int8]
    f3 array[bar]
}
```

SyzKaller-generated conforming test case

```
0x12345678,              // f1 (4 bytes)
0x002,                   // f2_len (2 bytes)
0x03,                    // f3_len (1 byte)
[0x0a, 0x0b],            // f2 (2*1 bytes)
[{...},{...},{...}]      // f3 (3*sizeof(bar) bytes)
```

Source: *syzkaller: adventures in continuous coverage-guided kernel fuzzing*

SyzLang: SyzKaller's Description Language

- Customizable to **any syscall**
  - E.g., to fuzz a new device driver, just need to model its `ioctl()` syscall handler via SyzLang
- Generally written by hand
  - Requires a lot of expertise
- Emerging work on automation
  - Trace mining, static analysis, LLMs

```
syz_usb_connect$hid(                                                # connects a USB-HID device
        speed flags[usb_device_speed],                              # device speed
        dev_len len[dev],                                           # device descriptor's length
        dev ptr[in, usb_device_descriptor_hid],                     # USB-HID device descriptor
        descs ptr[in, vusb_connect_descriptors]                     # USB descriptors requested
                                                                    #   during enumeration
) fd_usb_hid (timeout[3000], prog_timeout[3000])

syz_usb_control_io$hid(fd fd_usb_hid,
                       descs ptr[in, vusb_descriptors_hid],
                       resps ptr[in, vusb_responses_hid]) (timeout[300])
```

SyzKaller's Mutation

- Inserting / removing syscalls
- Changing syscall args:
  - Resizing arrays / buffers
  - Changing union options
  - Flags
  - Len / bytesize
  - Filename
  - Pointers
- The usual AFL-style mutators:
  - Bit / byte flips, insert / remove bytes, etc.

```
r0 = socket$can_j1939(AUTO, AUTO, AUTO)
ioctl$ifreq_SIOCGIFINDEX_vcan(r0, AUTO, &AUTO={'vxcan0\x00', <r1=>0x0})
bind$can_j1939(r0, &AUTO={AUTO, r1, 0x0, {0x0, 0x0, 0x0, 0x0}, 0x0}, AUTO)
r2 = socket$can_j1939(AUTO, AUTO, AUTO)
ioctl$ifreq_SIOCGIFINDEX_vcan(r2, AUTO, &AUTO={'vxcan1\x00', <r3=>0x0})
bind$can_j1939(r2, &AUTO={AUTO, r3, 0x0, {0x0, 0x0, 0x0, 0x0}, 0x0}, AUTO)
connect$can_j1939(r2, &AUTO={AUTO, r3, 0x0, {0x0, 0x0, 0x0, 0x0}, 0x0}, AUTO)
sendmsg$can_j1939(r2, &AUTO={0x0, 0x0, &AUTO={&AUTO='data', AUTO},
                             0x1, 0x0, 0x0, 0x0}, 0x0)
recvmsg$can_j1939(r0, &AUTO={0x0, 0x0, &AUTO=[{&AUTO='----', AUTO}],
                             0x1, 0x0, 0x0, 0x0}, 0x0)
```

Source: *syzkaller: adventures in continuous coverage-guided kernel fuzzing*

Does it work?

KASAN: OOB write in watch_queue_set_filter

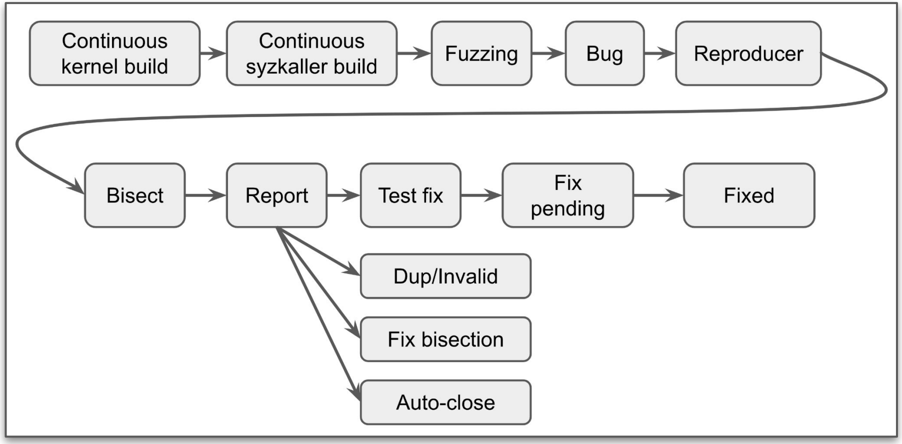
```
int main() {
  mmap(0x20000000, 0x1000000, 3, 0x32, -1, 0);
  intptr_t res = 0;
  res = open("/dev/watch_queue", 0, 0);
  if (res != -1)
    r[0] = res;
  *(uint32_t*)0x20000240 = 1;
  *(uint32_t*)0x20000244 = 0;
  *(uint32_t*)0x20000248 = 0x300;
  *(uint32_t*)0x2000024c = 0;
  *(uint32_t*)0x20000250 = 0;
  *(uint32_t*)0x20000254 = 0;
  *(uint32_t*)0x20000258 = 0;
  *(uint32_t*)0x2000025c = 0;
  *(uint32_t*)0x20000260 = 0;
  *(uint32_t*)0x20000264 = 0;
  *(uint32_t*)0x20000268 = 0;
  *(uint32_t*)0x2000026c = 0;
  *(uint32_t*)0x20000270 = 0;
  ioctl(r[0], 0x5761, 0x20000240);
}
```

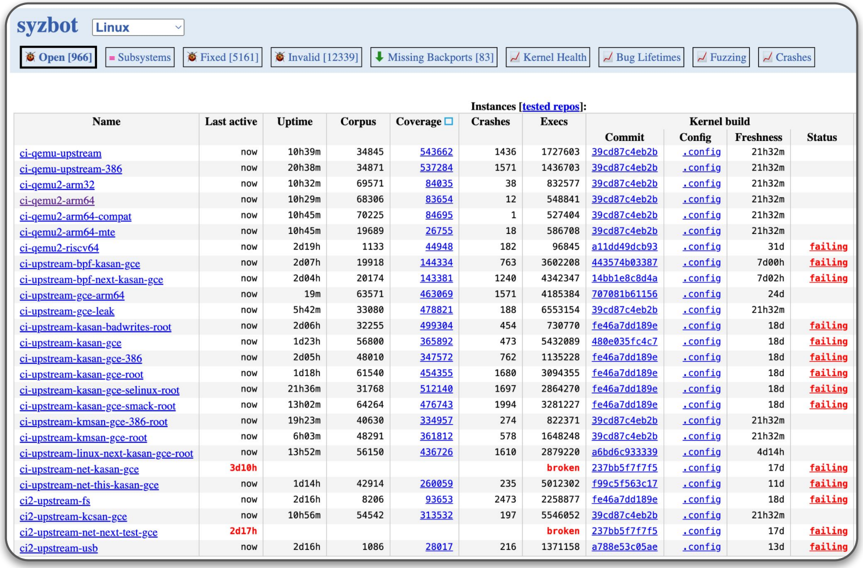
```
BUG: KASAN: slab-out-of-bounds in watch_queue_set_filter
Write of size 4 at addr ffff8880a9b31ddc by task syz-executor545/9

Call Trace:
 __asan_report_store4_noabort+0x17/0x20 generic_report.c:139
 watch_queue_set_filter drivers/misc/watch_queue.c:516 [inline]
 watch_queue_ioctl+0x15ed/0x16e0 drivers/misc/watch_queue.c:555
 do_vfs_ioctl+0x977/0x14e0 fs/ioctl.c:732
 ksys_ioctl+0xab/0xd0 fs/ioctl.c:749

Allocated by task 9097:
 kzalloc include/linux/slab.h:670 [inline]
 watch_queue_ioctl+0xf57/0x16e0 drivers/misc/watch_queue.c:555
 do_vfs_ioctl+0x977/0x14e0 fs/ioctl.c:732
 ksys_ioctl+0xab/0xd0 fs/ioctl.c:749

Freed by task 8821:
 kfree+0x10a/0x2c0 mm/slab.c:3757
 single_release+0x95/0xc0 fs/seq_file.c:609
 __fput+0x2ff/0x890 fs/file_table.c:280
 ____fput+0x16/0x20 fs/file_table.c:313
 task_work_run+0x145/0x1c0 kernel/task_work.c:113
 tracehook_notify_resume include/linux/tracehook.h:188 [inline]
 exit_to_usermode_loop+0x316/0x380 arch/x86/entry/common.c:164
```

Source: *syzkaller: adventures in continuous coverage-guided kernel fuzzing*

SyzBot: Real-time “Interface” to SyzKaller

https://syzkaller.appspot.com/upstream




syzbot Linux

Open [966] | Subsystems | Fixed [5161] | Invalid [12339] | Missing Backports [83] | Kernel Health | Bug Lifetimes | Fuzzing | Crashes

Instances [tested repos]:

Name	Last active	Uptime	Corpus	Coverage	Crashes	Execs	Kernel build			
							Commit	Config	Freshness	Status
ci-qemu-upstream	now	10h39m	34845	543662	1436	1727603	39cd87c4eb2b	.config	21h32m	
ci-qemu-upstream-386	now	20h38m	34871	537284	1571	1436703	39cd87c4eb2b	.config	21h32m	
ci-qemu2-arm32	now	10h32m	69571	84035	38	832577	39cd87c4eb2b	.config	21h32m	
ci-qemu2-arm64	now	10h29m	68306	83654	12	548841	39cd87c4eb2b	.config	21h32m	
ci-qemu2-arm64-compat	now	10h45m	70225	84695	1	527404	39cd87c4eb2b	.config	21h32m	
ci-qemu2-arm64-mte	now	10h45m	19689	26755	18	586708	39cd87c4eb2b	.config	21h32m	
ci-qemu2-riscv64	now	2d19h	1133	44948	182	96845	a11dd49dcb93	.config	31d	failing
ci-upstream-bpf-kasan-gce	now	2d07h	19918	144334	763	3602208	443574b03387	.config	7d00h	failing
ci-upstream-bpf-next-kasan-gce	now	2d04h	20174	143381	1240	4342347	14bb1e8c8d4a	.config	7d02h	failing
ci-upstream-gce-arm64	now	19m	63571	463069	1571	4185384	707081b61156	.config	24d	
ci-upstream-gce-leak	now	5h42m	33080	478821	188	6553154	39cd87c4eb2b	.config	21h32m	
ci-upstream-kasan-badwrites-root	now	2d06h	32255	499304	454	730770	fe46a7dd189e	.config	18d	failing
ci-upstream-kasan-gce	now	1d23h	56800	365892	473	5432089	480e035fc4c7	.config	18d	failing
ci-upstream-kasan-gce-386	now	2d05h	48010	347572	762	1135228	fe46a7dd189e	.config	18d	failing
ci-upstream-kasan-gce-root	now	1d18h	61540	454355	1680	3094355	fe46a7dd189e	.config	18d	failing
ci-upstream-kasan-gce-selinux-root	now	21h36m	31768	512140	1697	2864270	fe46a7dd189e	.config	18d	failing
ci-upstream-kasan-gce-smack-root	now	13h02m	64264	476743	1994	3281227	fe46a7dd189e	.config	18d	failing
ci-upstream-kmsan-gce-386-root	now	19h23m	40630	334957	274	822371	39cd87c4eb2b	.config	21h32m	
ci-upstream-kmsan-gce-root	now	6h03m	48291	361812	578	1648248	39cd87c4eb2b	.config	21h32m	
ci-upstream-linux-next-kasan-gce-root	now	13h52m	56150	436726	1610	2879220	a6bd6c933339	.config	4d14h	
ci-upstream-net-kasan-gce	3d10h					broken	237bb5f7f7f5	.config	17d	failing
ci-upstream-net-this-kasan-gce	now	1d14h	42914	260059	235	5012302	f99c5f563c17	.config	11d	failing
ci2-upstream-fs	now	2d16h	8206	93653	2473	2258877	fe46a7dd189e	.config	18d	failing
ci2-upstream-kcsan-gce	now	10h56m	54542	313532	197	5546052	39cd87c4eb2b	.config	21h32m	
ci2-upstream-net-next-test-gce	2d17h					broken	237bb5f7f7f5	.config	17d	failing
ci2-upstream-usb	now	2d16h	1086	28017	216	1371158	a788e53c05ae	.config	13d	failing

SyzKaller's Trade-Offs

	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

Device Drivers

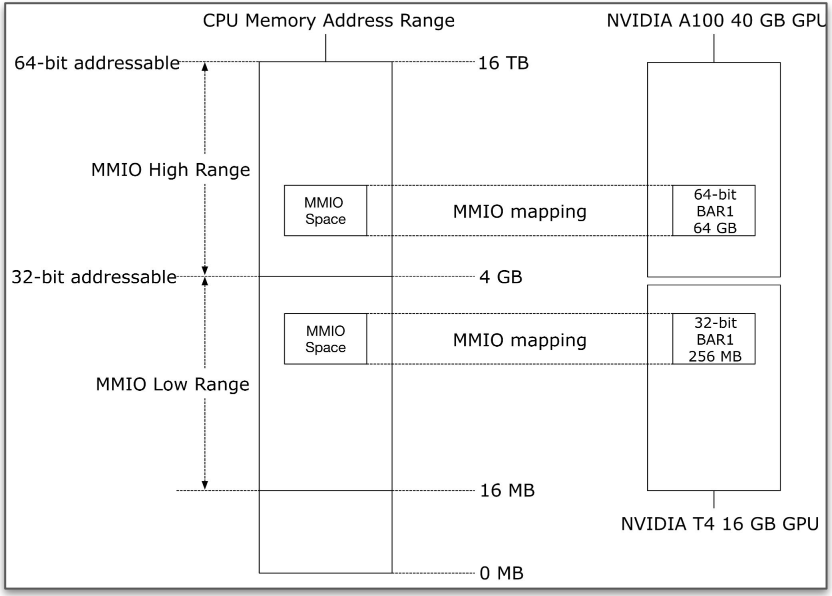
- Largest **attack surface** of the kernel... why?

Device Drivers

- Largest **attack surface** of the kernel… why?
  - Device drivers are run as kernel code
  - It’s all **third-party** code!
- Possible input vectors:
  - **???**

Device Drivers

- Largest **attack surface** of the kernel… why?
  - Device drivers are run as kernel code
  - It's all **third-party** code!
- Possible input vectors:
  - From **user-space**: `ioctl()` syscall
  - From **hardware**: MMIO, DMA, PortIO
  - These require different techniques!
- Fuzzing challenges:
  - Identifying size/bounds of MMIO/DMA
  - Structure of the data they expect, etc.



Questions?