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**

<table>
<thead>
<tr>
<th>Monday Meeting</th>
<th>Wednesday Meeting</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Apr. 01</strong></td>
<td><strong>Apr. 03</strong></td>
</tr>
<tr>
<td><strong>Fuzzing OS Kernels</strong></td>
<td><strong>LLM-guided Fuzzing</strong></td>
</tr>
<tr>
<td>► Readings:</td>
<td>► Readings:</td>
</tr>
<tr>
<td><strong>Apr. 08</strong></td>
<td><strong>Apr. 10</strong></td>
</tr>
<tr>
<td><strong>Fuzzing Compilers</strong> (guest lecture by John Regehr)</td>
<td><strong>Fuzzing Hardware</strong></td>
</tr>
<tr>
<td>► Readings:</td>
<td>► Readings:</td>
</tr>
<tr>
<td><strong>Apr. 15</strong></td>
<td><strong>Apr. 17</strong></td>
</tr>
<tr>
<td><strong>Fuzzing Multi-language Software</strong></td>
<td><strong>Final Presentations I</strong></td>
</tr>
<tr>
<td>► Readings:</td>
<td></td>
</tr>
<tr>
<td><strong>Apr. 22</strong></td>
<td><strong>Apr. 24</strong></td>
</tr>
<tr>
<td><strong>Final Presentations II</strong></td>
<td><strong>No Class (Reading Day)</strong></td>
</tr>
</tbody>
</table>
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?

Ring 3
Ring 2
Ring 1
Ring 0
Kernel
Device drivers
Device drivers
Applications

Least privileged

Most privileged
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

```c
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

- Problems?
  - ???

```c
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

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

```c
while (1){
    syscall(rand(), rand(), rand());
    syscall(rand_fd(), rand_addr());
}
```
- Joint effort by Google and the Linux kernel dev team
- Continuous kernel fuzzing and crash reporting
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

```c
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

```c
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**

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

**SyzKaller-generated conforming test case**

```c
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
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.

Source: syzkaller: adventures in continuous coverage-guided kernel fuzzing
Does it work?

KASAN: OOB write in watch_queue_set_filter

```c
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 fffff880a9b310dc 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 9007:
    kcalloc 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.c:188 [inline]
    exit_to_usermode_loop+0x316/0x380 arch/x86/entry/common.c:164
}
```

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

Stefan Nagy
SyzBot: Real-time “Interface” to SyzKaller

https://syzkaller.appspot.com/upstream
# SyzKaller’s Trade-Offs

<table>
<thead>
<tr>
<th></th>
<th>Physical device</th>
<th>VM / Emulator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fuzzing surface</td>
<td>Native (includes device drivers)</td>
<td>Only what the VM supports</td>
</tr>
<tr>
<td>Management</td>
<td>Hard, hardware gets bricked</td>
<td>Easy</td>
</tr>
<tr>
<td>(restarting, debugging,</td>
<td></td>
<td></td>
</tr>
<tr>
<td>getting kernel logs)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scalability</td>
<td>Buy more devices</td>
<td>Spawn more VMs</td>
</tr>
</tbody>
</table>
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?