Bypassing Kernel Barriers

Fuzzing Linux Kernel in Userspace with LKL

Eugene Rodionov, Xuan Xing

6/26/2025

Google

A conversation happened long time ago…

- My manager:
  - Maybe we should fuzz the Linux HID driver?
  - Any idea how to do it?
- <if I were an experienced kernel hacker>:
  - syzkaller?
- Actual me:
  - clang -fsanitize=fuzzer drivers/hid/hid-core.c kernel_stubs.c …..
- Clang:
  - fatal error: too many errors emitted, stopping now [-ferror-limit=]
- Matt Alexander (former Googler/RedTeamer):
  - Oh, you want to run kernel code in userspace? try LKL!

Agenda

- Introduction
- LKL Basics
- Fuzzing on LKL
- LKL Fuzzers
- Quick Demo
- Conclusion
- Q&A

Introduction

Who are we?

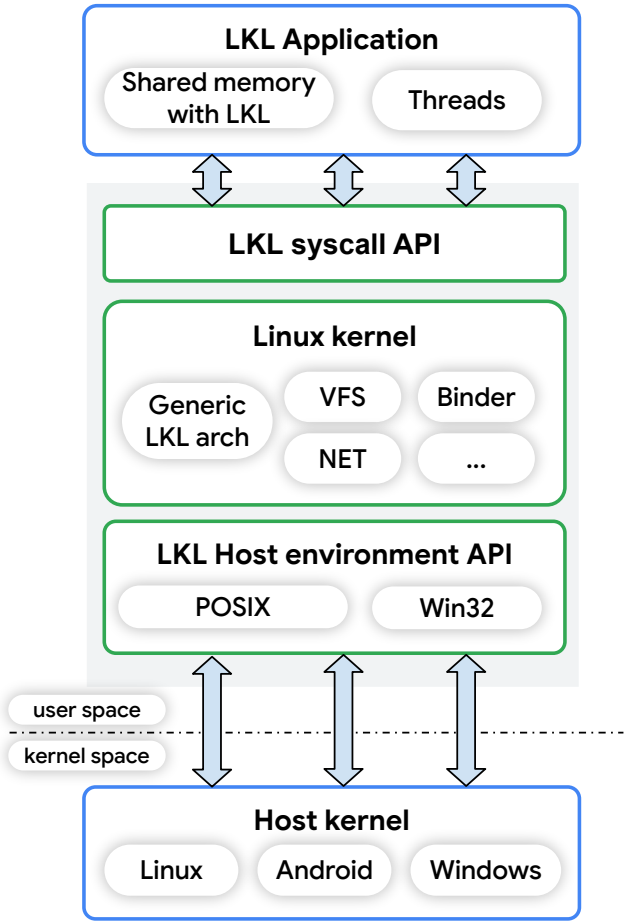
Protect the Android ecosystem through offensive security

- Offensive Security Reviews to verify (break) security assumptions
- Scale through tool development (e.g. continuous fuzzing)
- Develop proof of concepts to demonstrate real-world impact
- Assess the efficacy of security mitigations

But what is LKL?

LKL Overview

- Linux Kernel Library (LKL)[1] builds Linux kernel as a user-space library
  - Implemented as Linux arch-port
  - LKL vs UML
- LKL building blocks
  - Host environment API -- portability layer
  - Linux kernel code
  - LKL syscall API exposed to the user-space application
- Run kernel code without launching a VM
  - Kernel unit testing
  - Fuzzing![2,3]



[1] https://github.com/lkl/linux
[2] Xu et al., Fuzzing File Systems via Two-Dimensional Input Space Exploration
[3] https://github.com/atrosinenko/kbdysch

Using LKL from your C program

```
int ret = lkl_start_kernel(&lkl_host_ops, "mem=50M");

lkl_mount_fs("sysfs");
lkl_mount_fs("proc");
lkl_mount_fs("dev");

int binder_fd = lkl_sys_open("/dev/binder", O_RDWR | O_CLOEXEC, 0);
void *binder_map = lkl_sys_mmap(NULL, BINDER_VM_SIZE,
                    PROT_READ, MAP_PRIVATE, binder_fd, 0);

struct lkl_binder_version version = { 0 };
ret = lkl_sys_ioctl(binder_fd, LKL_BINDER_VERSION, &version);
```

Using LKL from your C program

```
int ret = lkl_start_kernel(&lkl_host_ops, "mem=50M");

lkl_mount_fs("sysfs");
lkl_mount_fs("proc");
lkl_mount_fs("dev");

int binder_fd = lkl_sys_open("/dev/binder", O_RDWR | O_CLOEXEC, 0);
void *binder_map = lkl_sys_mmap(NULL, BINDER_VM_SIZE,
                    PROT_READ, MAP_PRIVATE, binder_fd, 0);

struct lkl_binder_version version = { 0 };
ret = lkl_sys_ioctl(binder_fd, LKL_BINDER_VERSION, &version);
```

Using LKL from your C program

```
int ret = lkl_start_kernel(&lkl_host_ops, "mem=50M");

lkl_mount_fs("sysfs");
lkl_mount_fs("proc");
lkl_mount_fs("dev");

int binder_fd = lkl_sys_open("/dev/binder", O_RDWR | O_CLOEXEC, 0);
void *binder_map = lkl_sys_mmap(NULL, BINDER_VM_SIZE,
                    PROT_READ, MAP_PRIVATE, binder_fd, 0);

struct lkl_binder_version version = { 0 };
ret = lkl_sys_ioctl(binder_fd, LKL_BINDER_VERSION, &version);
```

Using LKL from your C program

```
int ret = lkl_start_kernel(&lkl_host_ops, "mem=50M");

lkl_mount_fs("sysfs");
lkl_mount_fs("proc");
lkl_mount_fs("dev");

int binder_fd = lkl_sys_open("/dev/binder", O_RDWR | O_CLOEXEC, 0);
void *binder_map = lkl_sys_mmap(NULL, BINDER_VM_SIZE,
                    PROT_READ, MAP_PRIVATE, binder_fd, 0);

struct lkl_binder_version version = { 0 };
ret = lkl_sys_ioctl(binder_fd, LKL_BINDER_VERSION, &version);
```

LKL Quirks

- MMU implementation
  - Inspired by MMU implementation in UML
  - Implements 3-level page tables & flat memory model
  - Uses the host's `mmap` to map/unmap LKL's physical pages at provided virtual addresses
- Emulates multiple processes
  - Implements custom `new_thread_group_leader` syscall to modify `tgid`

Applications

- Built-in applications
  - fs2tar: converting file system image to a tar archive
  - cptofs: copies files to/from a file system image
  - lklfuse: mounting filesystem images in userspace, without root privileges
- Kernel fuzzing
  - kBdysch: A collection of kernel fuzzers in userspace
  - Janus: Fuzzing file system using lkl
  - Sample fuzzers: Sample fuzzers developed by Android Red Team (HID, binder)
- Other interesting ideas
  - LKL.js: running Linux kernel on JavaScript directly
  - User Space TCP: a full TCP stack in userspace for better security

Enabling Fuzzing on LKL

Anatomy of LKL fuzzer

LKL enables fuzzing Linux kernel code in user-space

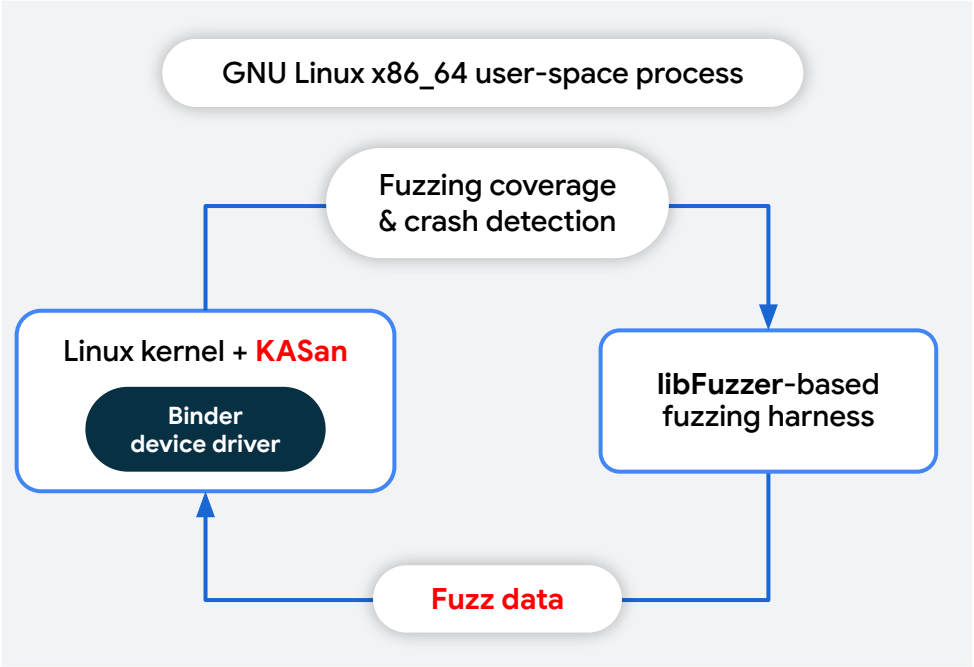
- Use in-process fuzzing engine, such as **libFuzzer**

Advantages

- High fuzzing performance on x86_64
- Ease of custom modifications
  - e.g. mocking hardware, custom scheduler(?)

Limitations

- No SMP in LKL
- x86_64 vs aarch64 -- potential false positives, false negatives



LKL fuzzing limitations

- No SMP support
  - Not easy to test certain concurrency scenarios
- Challenges with coverage build
  - Large binary size
  - Taking too long to build
- Only support x86-64 arch
  - Limited AArch/AArch64 support
  - Drivers specific to non-x86-64 architecture can’t be fuzzed
- Need to mock lower level interfaces
  - Mocking hardware support
  - Using virtio backend
- Not in linux main branch (yet)
  - Requiring effort to be upstreamed

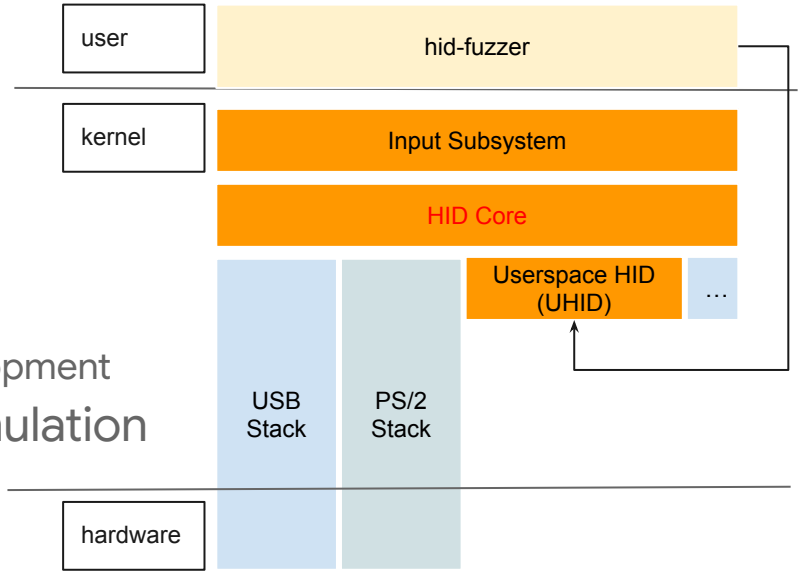
LKL Fuzzers

LKL fuzzers

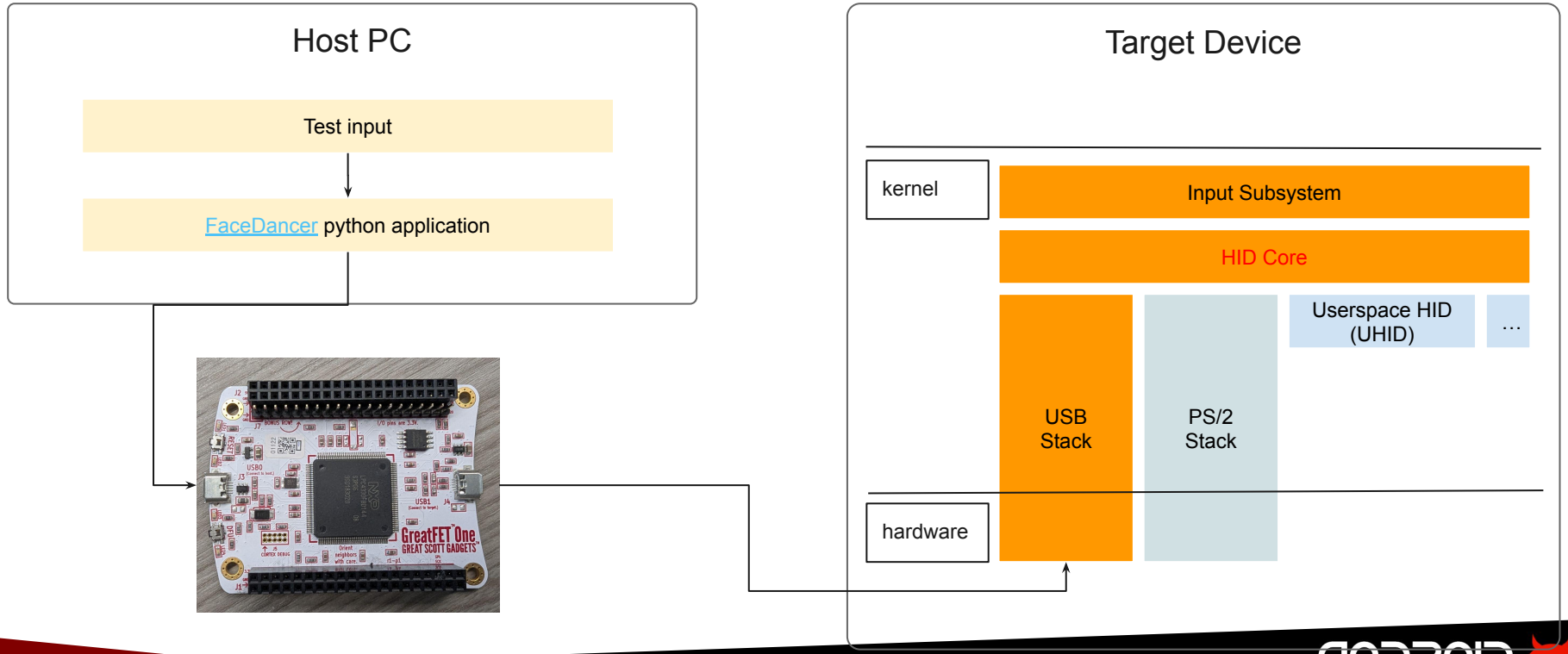
- Human Interface Device (HID) fuzzer
  - A sample LKL fuzzer targeting HID driver
- Binder fuzzer
  - Targeting Android Binder driver
- Virtio driver fuzzing
  - Used by Android pKVM with special attacking scenario (malicious host against guest VM)
  - Virtio based device drivers

Human Interface Device (HID) fuzzing

- Target:
  - Linux HID system
  - Complex spec (hid1_11.pdf, hut1_5.pdf)
- Attacking scenario:
  - Physical access (juice jacking)
- Details
  - Inspired by kBdysch fuzzing project
  - A proof of concept for LKL fuzzing support development
- Issues discovered and verified with USB simulation
  - Two bulletin class issues (CVE-2020-0465)
  - 6 stability issues



Human Interface Device (HID) fuzzing (cont.)



Exploitability of HID bugs

- We didn't attempt to exploit identified issues in HID driver
- However, ...

https://i.blackhat.com/EU-21/Thursday/EU-21-Bogaard-Geist-Achieving-Linux-Kernel-Code-Execution-Through-A-Malicious-USB-Device.pdf

DEMO: USB HID Driver issue discovery

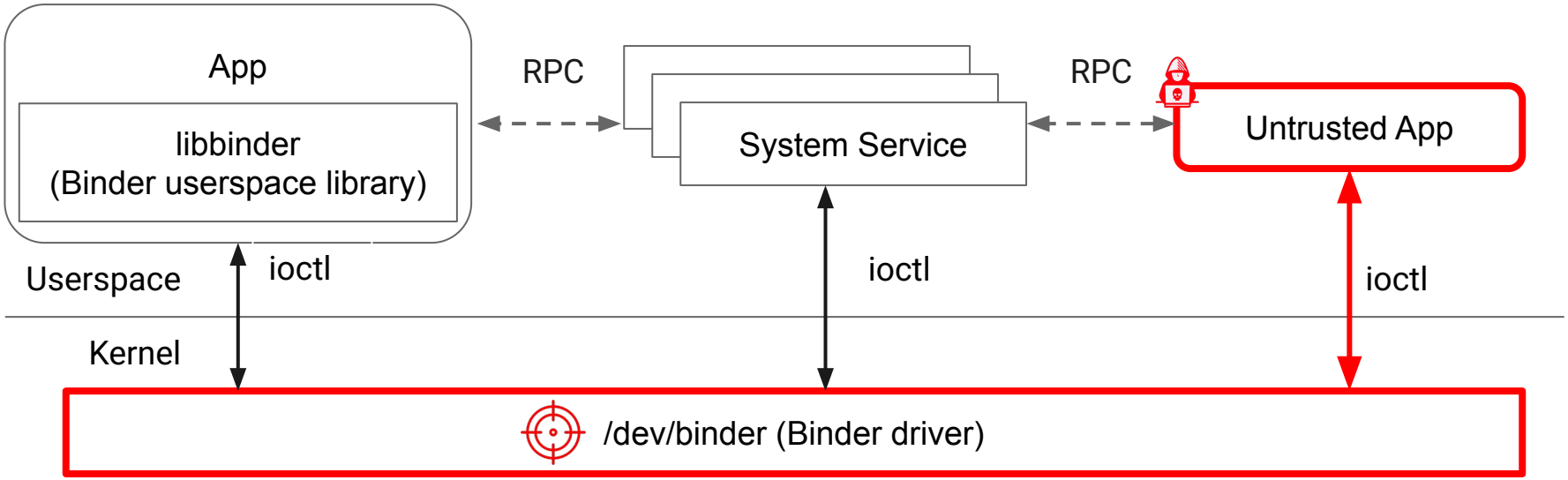
Android Binder Driver Fuzzing

Google

What is Binder?

- Primary inter-process communication (IPC) channel on Android
- Support passing file descriptors, objects containing pointers, etc.
- Composed of a userspace library (libbinder) and a kernel driver (/dev/binder)
- Provide Remote Procedure Call (RPC) framework for Java and C/C++

Binder Threat Model

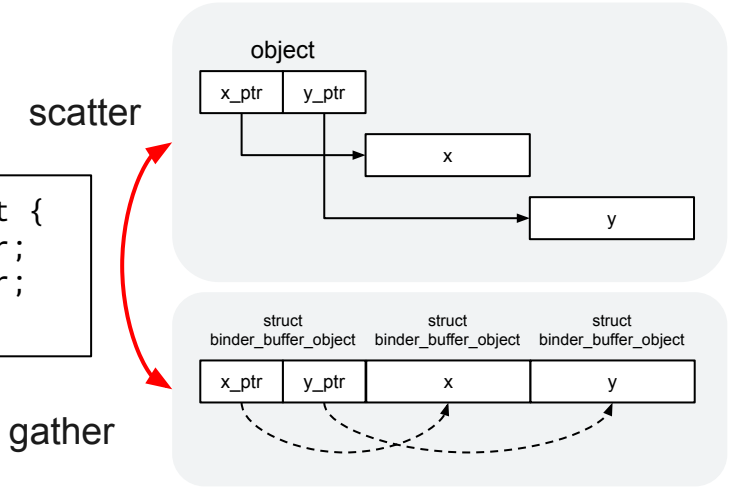


Binder Fuzzing Challenges

Data dependencies

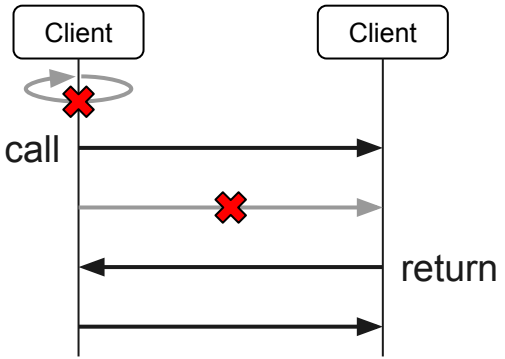
- Binder commands
- Scatter-gather data structures (BINDER_TYPE_PTR)

```
struct object {
  char *x_ptr;
  char *y_ptr;
};
```



State dependencies

- Synchronous IPC
  - Cannot send transaction to oneself
  - Multiple outstanding transactions not allowed



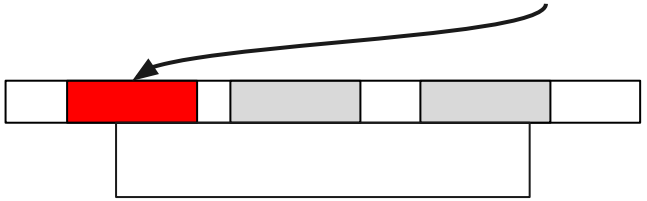
Binder Fuzzing Challenges

State dependencies

- Some inputs depend on previous IOCTL calls
  - e.g. Transaction buffers (BC_FREE_BUFFER)

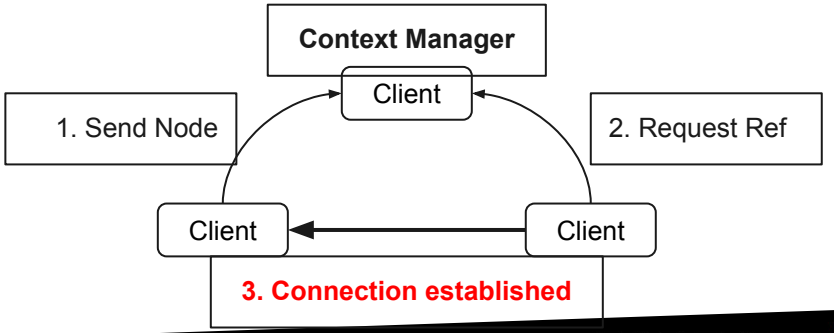
```
ioctl(binder_fd, BINDER_WRITE_READ, x) // 1.

// y = x->read_buffer->...->buffer
ioctl(binder_fd, BC_FREE_BUFFER, y )   // 2.
```



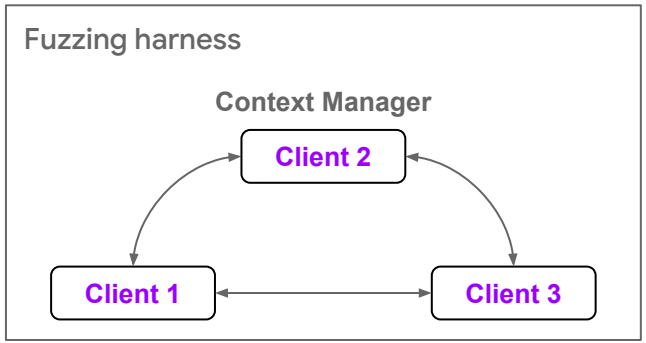
Multi-process coordination

- All communication requires a **Context Manager**
- Node & Ref setup is required to establish a connection



LKL Binder Fuzzing Harness

- Simulate IPC interactions between multiple clients
  - 3 **clients** (1 Context Manager)
  - **IOCTL calls** and **data**



Fuzz data

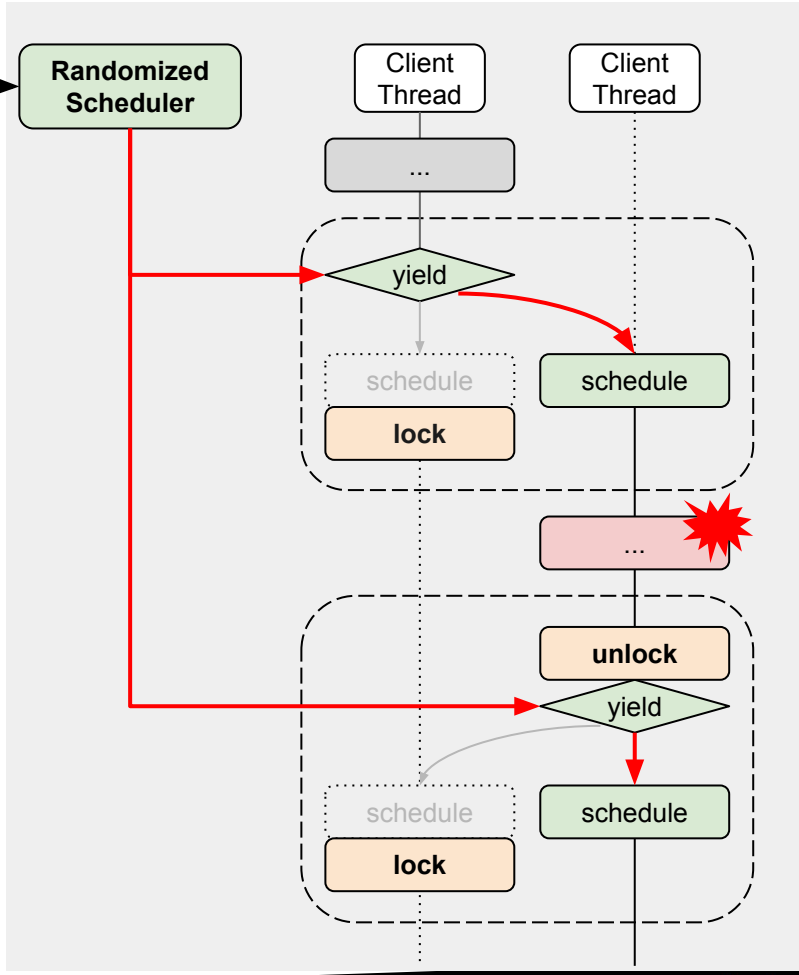
```
client_1 {
  binder_write {
    binder_commands {
      transaction {
        binder_objects { binder { ptr: 0xbeef } }
      }
    }
  }
}
client_2 {
  binder_read { ... }
  binder_write {
    binder_commands { free_buffer { ... } }
  }
}
client_3 { ... }
client_2 { ... }
client_3 { ... }
client_1 { ... }
```

LKL Randomized Scheduler

Deterministically simulate thread interleaving based on fuzz data[1]

Insert yield points before/after synchronization primitives

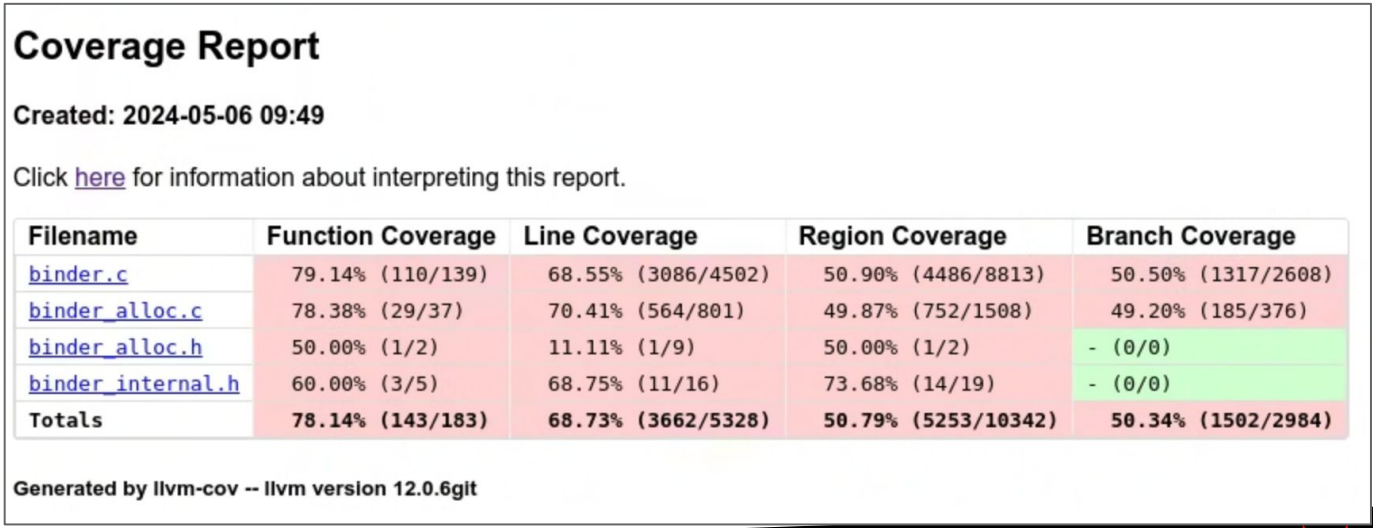
- spin_lock, spin_unlock
- mutex_lock, mutex_unlock



[1] Williamson, N., Catch Me If You Can: Deterministic Discovery of Race Conditions with Fuzzing. Black Hat USA, (2022).

LKL Binder Fuzzing Results

- Achieved 68% line coverage
- Discovered CVE-2023-20938 & CVE-2023-21255



Coverage Report

Created: 2024-05-06 09:49

Click here for information about interpreting this report.

Filename	Function Coverage	Line Coverage	Region Coverage	Branch Coverage
binder.c	79.14% (110/139)	68.55% (3086/4502)	50.90% (4486/8813)	50.50% (1317/2608)
binder_alloc.c	78.38% (29/37)	70.41% (564/801)	49.87% (752/1508)	49.20% (185/376)
binder_alloc.h	50.00% (1/2)	11.11% (1/9)	50.00% (1/2)	- (0/0)
binder_internal.h	60.00% (3/5)	68.75% (11/16)	73.68% (14/19)	- (0/0)
Totals	**78.14% (143/183)**	**68.73% (3662/5328)**	**50.79% (5253/10342)**	**50.34% (1502/2984)**

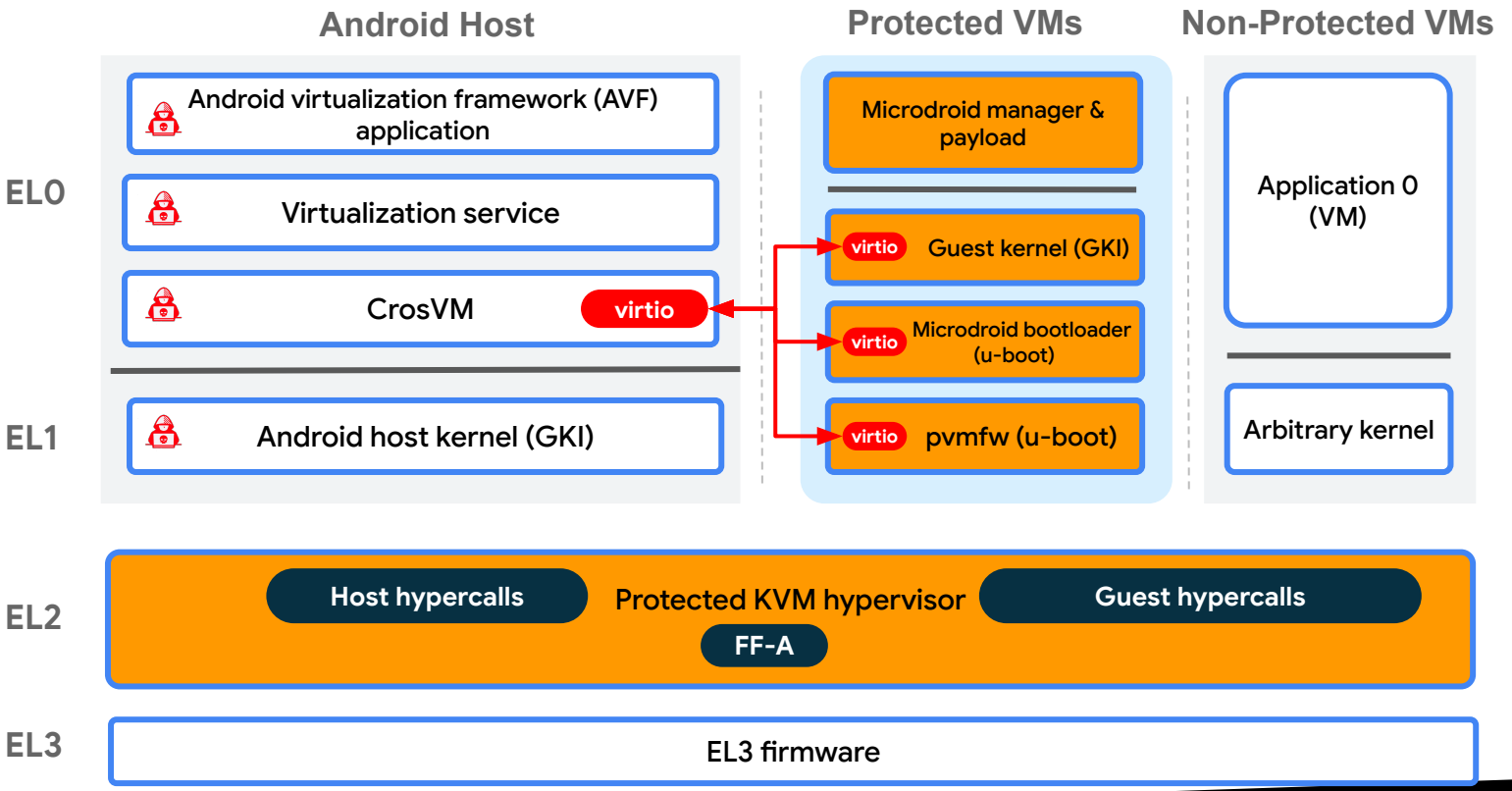
Generated by llvm-cov -- llvm version 12.0.6git

DEMO: Test case for CVE-2023-20938

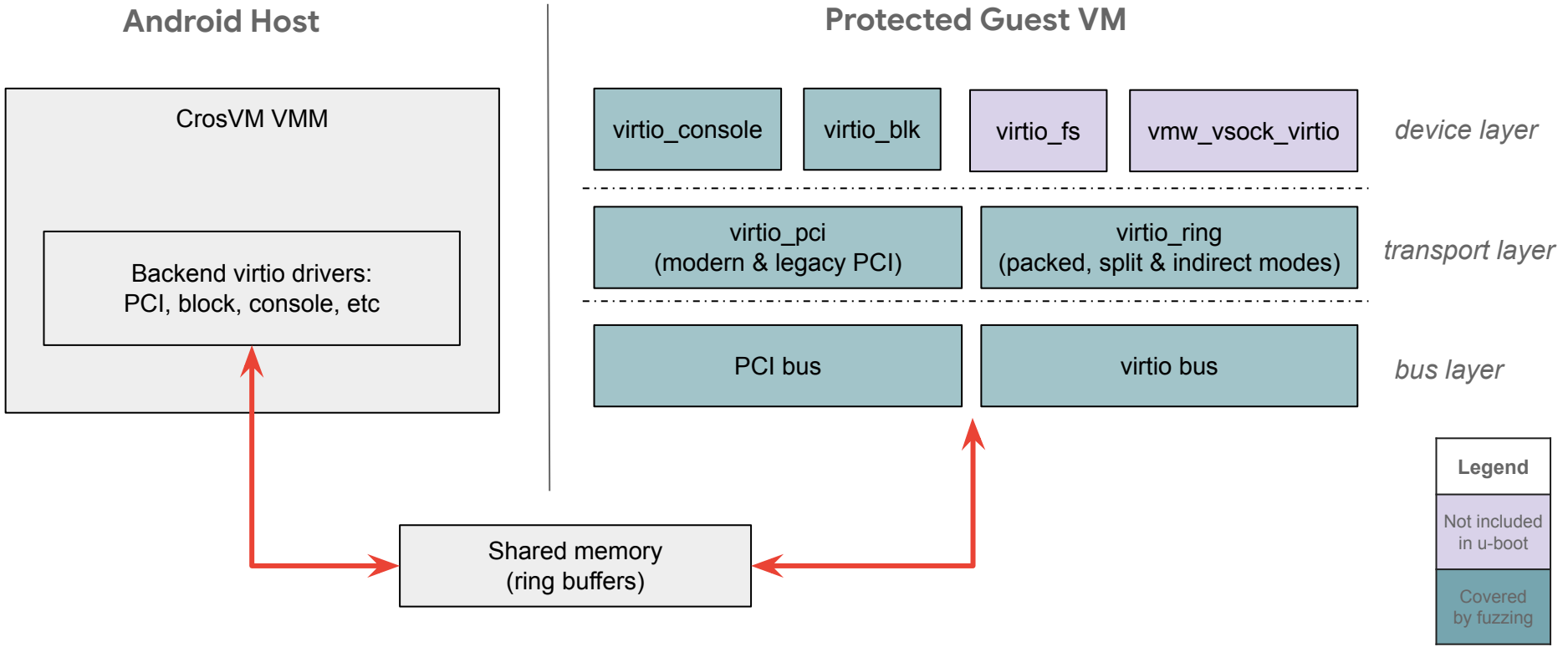
Virtio fuzzing for Android pKVM

Google

Android Protected KVM Attack Surface



Protected VM virtio attack surface



Virtio Front-end Fuzzers

Kernel under test:

- Android13-5.10

virtio_ring:

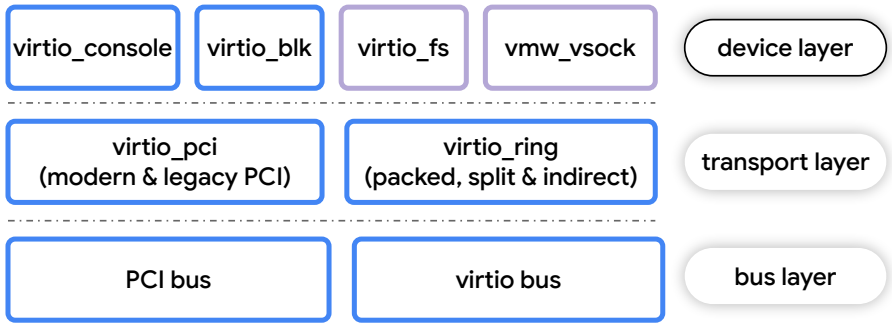
- fuzzes ring-buffer processing functionality
- handles both split & packed mode

virtio_pci:

- fuzzes PCI configuration space
- LKL arch-specific implementation of PCI bus
- mock-out PCI MMIO in the fuzzer harness

virtio_blk:

- mutates the virtio_blk configuration block



Virtio_blk fuzzer finding

```
int block_read_full_page(struct page *page, get_block_t *get_block)
{
     struct buffer_head *bh, *head, *arr[MAX_BUF_PER_PAGE];

     ...
     do {
          if (buffer_uptodate(bh))
               continue;

          if (!buffer_mapped(bh)) {
               int err = 0;

               ...
          }

          arr[nr++] = bh;    <====== OOB write on stack

     } while (i++, iblock++, (bh = bh->b_this_page) != head);
     ...
}
```

Virtio_blk fuzzer finding

- With the block size **0xe5e5e5e5**:
  - `inode->i_blkbits == 32`
  - `1 << READ_ONCE(inode->i_blkbits)` is undefined behavior in C
  - `1 << READ_ONCE(inode->i_blkbits) == 1` on x86 architecture

```
static struct buffer_head *create_page_buffers(struct page *page, ...)
{
      BUG_ON(!PageLocked(page));
      if (!page_has_buffers(page))
            create_empty_buffers(page, 1 << READ_ONCE(inode->i_blkbits), b_state);
      return page_buffers(page);
}
```

Conclusions

Key takeaways

- LKL can be an effective way to fuzz certain kernel scenarios
  - Scenarios requiring complex setup (e.g. hardware-to-kernel attack surface)
  - Dependencies to some userspace libraries
  - Visual code coverage and debugging can be very useful
  - It's good to have alternatives

- The fact that you don't know something might be your advantage
  - Discovering new approaches from naive ideas
  - Learning without predefined mindset

Future work

- ~~Enable MMU~~ (https://github.com/lkl/linux/pull/551)
- ~~Upstream Android Binder fuzzer~~ (https://github.com/lkl/linux/pull/564)

Creating more fuzzers and profit!

Acknowledgement

- Octavian Purdila and LKL maintainers
- https://github.com/atrosinenko/kbdysch
- Android pKVM team
- Android Binder team

Resources

- libFuzzer – a library for coverage-guided fuzz testing
- Achieving Kernel Code Execution through Malicious USB device
- Android Virtualization Framework and pKVM security
- Emulating USB device with Python
- Attacking Android Binder: Analysis and Exploitation of CVE-2023-20938
- Binder Internals

Questions?