Fuzzing Host-to-Guest Attack Surface in Android Protected KVM
Agenda

- Introduction into Android pKVM
  - Overview & motivation
  - Attack surface
- Virtio driver stack in pKVM
  - Why fuzzing?
  - Challenges with fuzzing virtio front-end drivers
- Fuzzing virtio front-end drivers with LKL
  - Linux Kernel Library for fuzzing
  - Overview of the developed fuzzers for pVM
- Conclusion & Future work
Terminology

- ABL -- Android bootloader
- AVB -- Android verified boot
- AVF -- Android virtualization framework
- GKI -- Generic kernel image
- LKL -- Linux kernel library
- Microdroid -- a Google-provided mini-Android OS that runs in a pVM
- pKVM -- Protected KVM
- pVM -- Protected virtual machine
- PVMFW -- Protected virtual machine firmware
- SMP -- Symmetric multiprocessing
Who we are

Will Deacon
- Active upstream kernel developer, co-maintaining aarch64 architecture port, locking, atomics, memory model, TLB, SMMU, ...
- Leading the Protected KVM project to enable KVM on Android

Eugene Rodionov
- Android Red Team security engineer
- Focused on finding & exploiting vulnerabilities in low-level software in AOSP and Pixel devices
Android Protected KVM: Overview

- Protected KVM introduces a new security model where the host and the deprivileged guest VMs mutually distrust each other.

- **Mutual distrust:**
  - Protected KVM provides security for the guest VMs even if the host kernel is compromised
  - A malicious guest cannot escape into the host (Android) or cannot compromise another guest VM

- **Deprivileged guests:**
  - Guest VMs don’t need TrustZone privileges and run in non-secure world EL1/EL0

- Protected KVM on Arm64: A Technical Deep Dive by Quentin Perret
- Now You See Me, Now You Don't: Splitting pKVM Into Discrete, Mutually Exclusive Address Spaces by Marc Zyngier
- All Bark and no Bite: vCPU Stall Detection for KVM Guests by Sebastian Ene
- Panel discussion: KVM-based virtualization contributor Q&A by Will Deacon, et al
Building pillars of pKVM security

Host & guest VM software
- VMM and protected VM payload
- Process untrusted input received from the host/guest respectively
- Prioritizing host-to-guest attacks

Hypervisor
- Enforces isolation of the guests between each other and from the host
- Protects pVM bootloader and sealing keys

Attestation & Sealing
- Enable external services to attest the integrity of protected VMs
- Enable per-VM instance secret data

Secure boot & AVB
- Enforces authenticity of the hypervisor and Android kernel
- Provides attestation services and protects sealing keys
Virtio attack surface
### Attacking guests via virtio

#### Host: virtio back-end

<table>
<thead>
<tr>
<th>buffer PA</th>
<th>buffer len</th>
<th>flags</th>
<th>next desc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7fff..800</td>
<td>0x100</td>
<td>0x0</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

#### Guest: virtio front-end

**Virtio used buffer**

- flags
- idx
- (desc idx, length)
- ...

**Virtio avail buffer**

- flags
- idx
- desc idx
- ...

### Host <==> guest shared memory

- Virtio descriptors
- Virtio used buffer
- Virtio avail buffer
### Host: virtio back-end

<table>
<thead>
<tr>
<th>buffer PA</th>
<th>buffer len</th>
<th>flags</th>
<th>next desc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7fff..800</td>
<td>0x100</td>
<td>0x0</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

### Guest: virtio front-end

1. Put request in the virtio queue
Attacking guests via virtio

Host:
virtio back-end

1. Put request in the virtio queue

2. Process request from the queue

3. Put response in the virtio queue

Guest:
virtio front-end

1. Put request in the virtio queue

<table>
<thead>
<tr>
<th>Buffer PA</th>
<th>Buffer len</th>
<th>Flags</th>
<th>Next Desc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7fff..800</td>
<td>0x100</td>
<td>0x0</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Virtio descriptors

Virtio used buffer
- Flags
- Idx
- (Desc Idx, length)
- ...

Virtio avail buffer
- Flags
- Idx
- Desc Idx
- ...

Host <=> guest shared memory buffer
Attacking guests via virtio

Host: virtio back-end
1. Process request from the queue
2. Put response in the virtio queue

Guest: virtio front-end
1. Put request in the virtio queue
4. Process response from the virtio queue

Host <=> guest shared memory

Virtio descriptors

<table>
<thead>
<tr>
<th>buffer PA</th>
<th>buffer len</th>
<th>flags</th>
<th>next desc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7fff..800</td>
<td>0x100</td>
<td>0x0</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Virtio used buffer

| flags |
| idx |
| (desc idx, length) |
| ... |

Virtio avail buffer

| flags |
| idx |
| desc idx |
| ... |
Attacking guests via virtio

**Host:**
- virtio back-end
  2. Process request from the queue
  3. Put response in the virtio queue

**Guest:**
- virtio front-end
  1. Put request in the virtio queue
  4. Process response from the virtio queue
Protected VM virtio attack surface

Android Host

CrosVM VMM

Backend virtio drivers: PCI, block, console, etc

Protected Guest VM

virtio_console
virtio_blk
virtio_fs
vmw_vsock

device layer

virtio_pci
(modern & legacy PCI)

virtio_ring
(packed, split & indirect)

PCl bus

virtio bus

transport layer

bus layer

Legend

Not included
in u-boot

Shared memory
(ring buffers)
Host-to-guest attack vector **isn’t new** for Linux mainline.\(^1\)

However, this attack vector **is new for** Android and pKVM, in particular.

Virtio implementation in u-boot **wasn’t hardened** against malicious host.\(^2\)

---

\(^1\) Hardening virtio. [https://lwn.net/Articles/865216](https://lwn.net/Articles/865216)

\(^2\) virtio: [Harden and test vring](https://lwn.net/Articles/865216) patch series
Fuzzing virtio front-end drivers in the Linux kernel
Why fuzzing virtio drivers?

● One of the most effective ways to find stability and security issues in C/C++ code

● Fuzzing provides continuous security

● Fuzzer harness could be potentially reused across GKI/u-boot
  ○ as long as the same fuzzing engine is used

● Not too many security tools for Linux/Android kernel to choose from:
  ○ syzkaller\textsuperscript{1} & syzbot\textsuperscript{2} is a ‘de-facto standard’ fuzzing tools for kernel

\textsuperscript{1} https://github.com/google/syzkaller
\textsuperscript{2} https://syzkaller.appspot.com/upstream
Virtio fuzzing: challenges

https://github.com/google/syzkaller/blob/master/docs/internals.md
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](https://github.com/lkl/linux)

\(^2\) Xu et al., Fuzzing File Systems via Two-Dimensional Input Space Exploration

\(^3\) [https://github.com/atrosinenko/kbdysch](https://github.com/atrosinenko/kbdysch)
Using LKL from your C program

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

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

dev_t dev = makedev(MISC_MAJOR, UHID_MINOR);
int mknod_result = lkl_sys_mknodat(AT_FDCWD, "/dev/uhid",
    S_IFCHR | S_IRUSR | S_IWUSR, dev);

int fd = lkl_sys_open("/dev/uhid", O_RDWR | O_CLOEXEC, 0);
```
Using LKL from your C program

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

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

dev_t dev = makedev(MISC_MAJOR, UHID_MINOR);
int mknod_result = lkl_sys_mknodat(AT_FDCWD, "/dev/uhid",
    S_IFCHR | S_IRUSR | S_IWUSR, dev);

int fd = lkl_sys_open("/dev/uhid", O_RDWR | O_CLOEXEC, 0);
```
Using LKL from your C program

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

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

dev_t dev = makedev(MISC_MAJOR, UHID_MINOR);
int mknod_result = lkl_sys_mknodat(AT_FDCWD, "/dev/uhid",
    S_IFCHR | S_IRUSR | S_IWUSR, dev);

int fd = lkl_sys_open("/dev/uhid", O_RDWR | O_CLOEXEC, 0);
```
Using LKL from your C program

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

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

dev_t dev = makedev(MISC_MAJOR, UHID_MINOR);
int mknod_result = lkl_sys_mknodat(AT_FDCWD, "/dev/uhid",
    S_IFCHR | S_IRUSR | S_IWUSR, dev);

int fd = lkl_sys_open("/dev/uhid", O_RDWR | O_CLOEXEC, 0);
```
Anatomy of LKL fuzzier

LKL enables fuzzing Linux kernel code in user-space

- use in-process fuzzing engine, such as libFuzzer

Advantages:

- high fuzzing performance on x86_64 cores
- lightweight fuzzers (no need to run VMs)
- easy debugging & crash reproducing (i.e. gdb)
- hardware emulation (e.g. PCI)

Limitations:

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

GNU Linux x86_64 user-space process

Fuzzing coverage & crash detection

Linux kernel + KASan + virtio front-end drivers

libFuzzer-based fuzzer harness

virtio shared memory (ring buffers)

mutated data
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
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;
            ... if (buffer_uptodate(bh))
                continue;
        }

        arr[nr++] = bh;
    } 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_pageBuffers(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);
}
Fuzzing virtio driver stack in u-boot

- Both pvmfw (1st stage) and microdroid bootloader (2nd stage) are based on u-boot
  - rely on virtio_blk to get boot configuration and virtio_console for debug output

- **Fuzzing and ASAN for sandbox** patch series enable fuzzing for virtio front-end drives:
  - works for u-boot in sandbox mode
  - provide coverage-guided libFuzzer-based fuzzing
  - enables AddressSanitizer for the fuzz target

- **Findings:**
  - [virtio: Harden and test vring](#) patch series
static void detach_buf(struct virtqueue *vq, unsigned int head)
{
  ...
  while (vq->vring.desc[i].flags & nextflag) {
    virtqueue_detach_desc(vq, i); // <= i is OOB
    i = virtio16_to_cpu(vq->vdev, vq->vring.desc[i].next);
    vq->num_free++;
  }
  ...
}

int bounce_buffer_stop(struct bounce_buffer *state)
{
  ...
  // state is OOB and point to a fully attacker-controlled memory
  if (state->flags & GEN_BB_WRITE)
    memcpy(state->user_buffer, state->bounce_buffer, state->len);

  free(state->bounce_buffer);
  return 0;
}
LKL-based virtio fuzzers continuously run in Google’s internal ClusterFuzz engine.

Virtio fuzzing effort led to identification and proactive mitigation of multiple security and stability issues in GKI and u-boot.

Need your support in improving fuzzing for virtualized interfaces.
Future work

- Write more fuzzers targeting virtio front-end and PCI drivers

- Upstreaming LKL to Linux mainline:
  - first attempt in 2015
  - restarted in 2020\(^1\) -- still ongoing to integrate LKL as a submodule of UML

- Currently focusing on upstreaming LKL to Android Common Kernel mainline:
  - effort to upstream LKL as a separate architecture is WIP
  - share LKL fuzzing work with the open-source community

\(^1\) [https://lwn.net/Articles/811575/](https://lwn.net/Articles/811575/)
Thank you!
Appendix
TrustZone is currently used whenever host isolation is needed.

Historically, a guest VM is completely controllable by the host.

Increasing privilege
Prioritizing host-to-guest attacks in pKVM

Guest-to-host VM escapes is a traditional threat model for modern VMMs and hypervisors.

Android Virtualization Framework in Android 13 doesn’t allow running arbitrary guest VMs.
LKL KASan details

KASan provides actionable reports for invalid memory access:

- OOB, user-after-free, double-free
- covers stack, heap and globals

User-space ASan in LKL:

- ASan shadow memory poisoning routines are invoked in global constructors
- Which might be problematic due to specifics of globals initialization in Linux kernel

LKL implements generic KASan:

- -fsanitize=kernel-address
- arch-specific KASan implementation

![Memory Diagram]

- KASan shadow memory
- LKL globals
- LKL kernel heap
- Stack
- reserved virtual memory
- committed virtual memory
LKL fuzzing coverage

LKL relies on libFuzzer-based fuzzing code coverage instrumentation.

KCOV is an alternative solution:
- needs additional implementation to feed the coverage feedback to libFuzzer engine

Diagram:
- Compile Linux kernel source
- Compile LKL host environment layer
- Compile fuzzer harness
- Incremental link of lkl.o (vmlinux)
- Archive lkl.o and the host environment layer
- Link liblkl.a and the fuzzer harness
- Fuzzer binary
How to develop an LKL fuzzer

- Identify an interface to fuzz
  - use ‘realistic’ attack surface (i.e. reachable from user-space or from the hardware)

- Enable the kernel feature under test in the kernel config
  - which doesn’t depend on aarch64 features or SMP

- Mock-out low-level interfaces if needed
  - LKL already comes with virtio back-end and arch-specific PCI implementations

- Provide fuzzer harness which sends the fuzzer’s entropy to the target kernel interface
Output of virtio_blk fuzzer

./virtio_blk-fuzzer -close_fd_mask=3

...