ARMing Linux Kernel Exploits

Maxwell Bland, Motorola Mobility
mbland@motorola.com bland@sdf.org

Background: The Current Kernel Stack and Ecosystem

The kernel ecosystem now consists of **hardware, privileged security monitors, kernel protections, and userspace isolation mechanisms (e.g. Android JNI isolation for apps).**

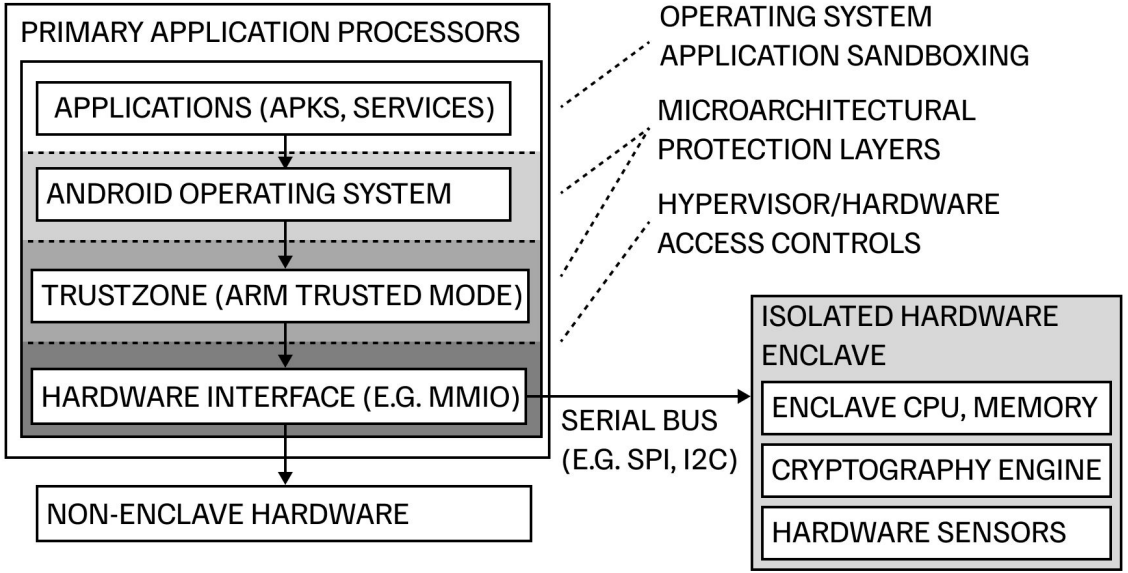
The Modern Kernel Production Stack (from the perspective of Android)

Threats are considered as targeting a specific *application domain*.

For example:
Web: an attack by a malicious webview element which then attempts to break out of the Dalvik VM.

Hardware/Firmware: an attack which targets the modem and attempts to manipulate OS-level modem buffer parsing code.

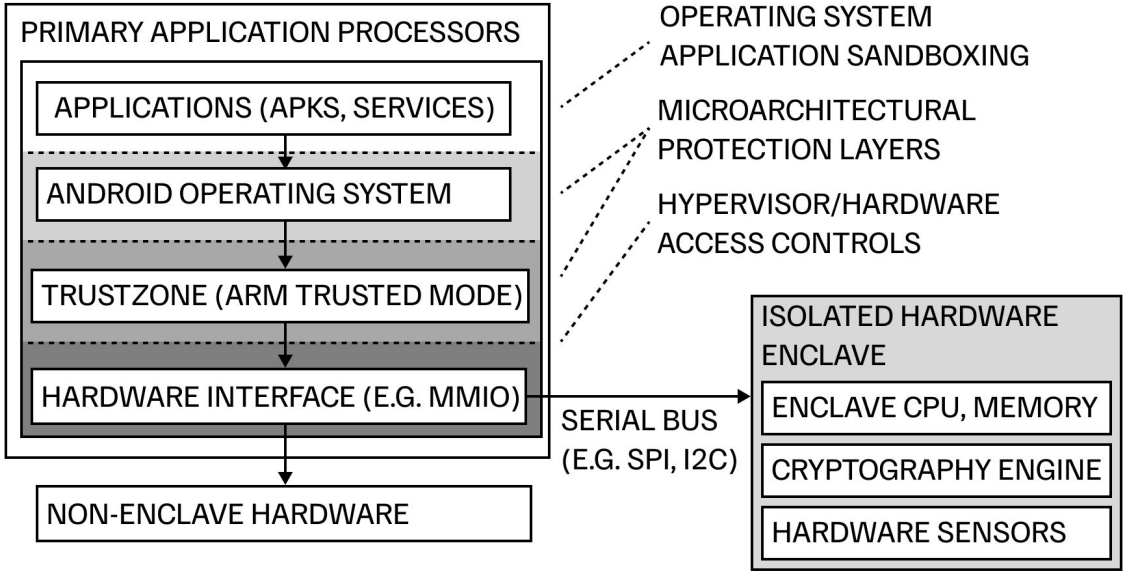
Application: an attack which stages itself from a malicious APK labeled, e.g. “com.evil.minecraft”.



Consider the Microarchitecture (Problems with TrustZone)

In the past, TrustZone, an ARM-specific bit set in peripherals and the processor, was considered to be of benefit to security, but now it increasingly appears to be dangerous, as it increases the threat surface (TZ apps can access all of memory).

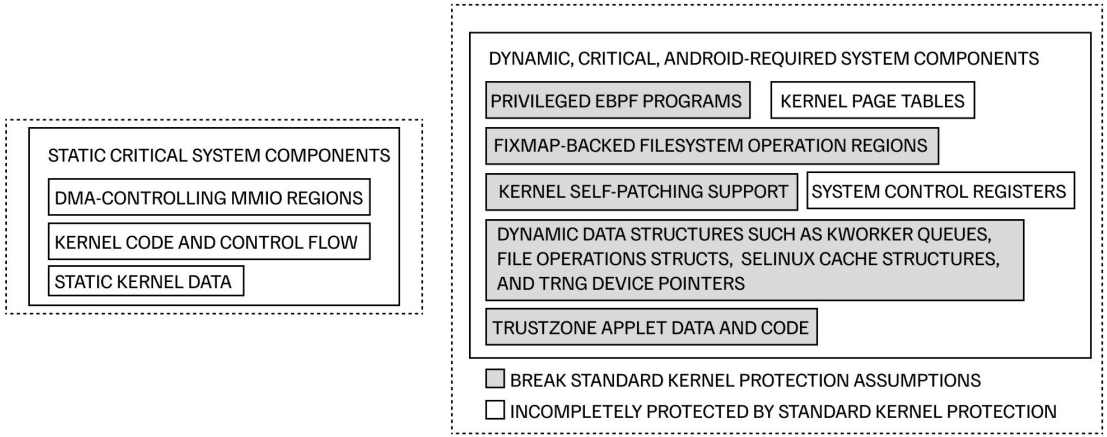
Architecture-specific additional protections can be a good thing, but not when they increase capabilities rather than reduce them!



Overview of New Kernel Threats

We will not cover all of these topics, but this is just a rough sketch of the sheer number of broken things.

Not all broken things are included here, but these are the big areas for the next few years.



Classic Attack Vector: Hardware Manipulation

Many modern exploits start by targeting a **kernel driver** or unprotected **hardware device mapping**.

The New (Old) Crown Jewels: Memory Write Gadgets

- A simple story, you've heard it a million times

1. Find a buffer overflow, use after free, etc.
2. *Write to something you shouldn't*
3. Get control

- And step (2) still breaks everything.

Technical details credit for this slide in part from https://github.blog/2022-06-16-the-android-kernel-mitigations-obstacle-race/

The New (Old) Crown Jewels: Memory Write Gadgets

- 2021: Maddie Stone finds that out of seven Android 0-days that were detected as exploited in the wild in 2021, five of them targeted GPU drivers
    - Most Android devices use one of a few GPUs, making it possible to obtain universal coverage with relatively few bugs
    - The GPU drivers are also reachable from the untrusted app sandbox in all Android devices
    - Most GPU drivers also handle memory sharing logic between the GPU device and the CPU, meaning unrestricted addressing
- Present work is focused on restricting hardware abstraction layers (HALs), but fails to acknowledge that many write gadget routes exist beyond this layer

Technical details credit for this slide in part from https://github.blog/2022-06-16-the-android-kernel-mitigations-obstacle-race/

An Example GPU CVE

- 5.4 branch of the Qualcomm msm 5.4 kernel when the new kgsl timeline feature, together with some new ioctl associated with it, was introduced
  - Note: “msm” is a branch of the linux kernel maintained for Android
  - Upstream patches to mainline 6.10 are backported by a few maintainers

- ioctl operation for GPU was messed up:

  “IOCTL_KGSL_TIMELINE_DESTROY makes it possible to acquire a reference to a dma_fence in fences after its refcount has reached zero but before it gets removed from fences in timeline_fence_release”

- This dma_fence means “direct memory access” to pretty much arbitrary kernel memory, after some additional, pretty simple manipulation

Technical details credit for this slide in part from https://github.blog/2022-06-16-the-android-kernel-mitigations-obstacle-race/

Another Example CVE (ALSA + GPU)

- Use after free.
- 32-bit compatibility SNDRV_CTL_IOCTL_ELEM_{READ|WRITE}32 ioctls had a race condition, resulting in snd_ctl_elem_write executing with an already freed struct snd_kcontrol input in the ALSA audio driver
    - Some additional GPU JIT compiler functions (REQ_SOFT_JIT_FREE jobs) used to spray the heap and write attacker-controlled data to the free'd location

- Simultaneously, and somewhat prior, we target the Mali GPU's performance tracing facility "timeline stream": we generate tlstream events, placing 16 bytes of controlled data at a known (but safe) kernel address, beating KASLR

- Punchline: the improperly freed struct snd_kcontrol is then overwritten to point to the controlled data provided by the tlstream facility
    - Aside: needed some additional "stabilization" via kernel's VFS subsystem

- Ultimately snd_ctl_elem_write's bad snd_kcontrol == yet another write gadget

Technical details credit for this slide in part from https://googleprojectzero.blogspot.com/2023/09/analyzing-modern-in-wild-android-exploit.html

Contemporary Kernel Exploit Anatomy

Modern exploits target **dynamically allocated resources**.

A Million and One Writable Options

- A decade ago, Samsung hand-waved an incomplete solution: RKP
  - Realtime Kernel Protection: have a security monitor intercept all writes to kernel **code** and **selinux policy** structs

- However, there are innumerable dynamic critical data structures in the kernel

- To name just a few:
  - File Operations Structs
  - TRNG Device Pointers
  - Kernel worker queues
  - …

- Example of this type of exploit:
  - https://github.com/chompie1337/s8_2019_2215_poc
  - Overwrites kernel file operations pointer to arbitrary function pointer

To Make Matters Worse

- Even where prior work was able to build useful primitives from ensuring the immutability of code:

    1. Kernel page tables are no longer static, and might have never been immutable (just switch the data page flag from writable to executable, bypass W^X)

    2. Kprobes were always enabled to support BPF, and required Linux kernel “self patching” code

    3. Since 2018, the EROFS filesystem and other subsystems have used the kernel’s “fixmap” datastructure to remap data pages as executable, and vice-versa

- Meaning that even the base assumptions of RKP’s approach to prevent write gadgets are entirely broken in the baseline Linux kernel

Next: Fight the Good Fight!

- In truth, ARM Linux kernel maintainers are just starting to integrate CFI
  - Some push towards Rust, but whether Rust or not, uptake will be slow
    - There are still (though less!) CVEs in Rust

- Some potential to fix the dynamic data structure problem by LLVM compiler passes
  - Look at the types, lock/unlock memory at EL2 appropriately
  - Still need to deep dive into a great understanding of *all* kernel semantics

- Goliath of an issue, but a good fight to undertake.
  - Solve problems one at a time: BPF-CFI, File Operations Structure checks, etc.
  - Good news: every patch set can address a specific blog post

- Google approach: throw every app into a VM
- Motorola approach: fix at the root

The Emerging Dangers of eBPF: more than Spectre

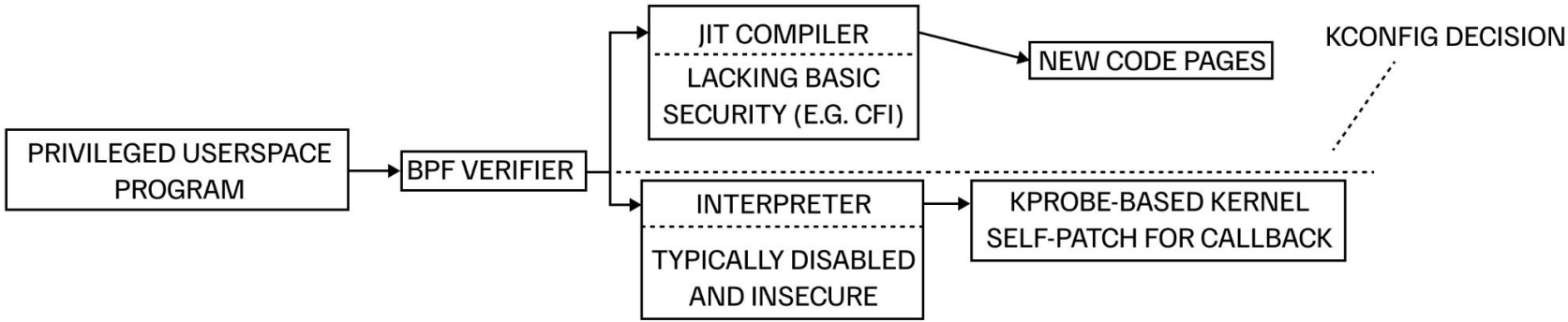
The extended **Berkeley Packet Filter** provides a number of dangerous routes for emerging kernel exploits.

Background on eBPF

- Initially the “Berkeley Packet Filter” was used for networking only: it allowed userspace programs to load efficient, verified programs into the kernel

- Since then, eBPF has expanded to allow userspace programs to load arbitrary verified programs into the kernel

- As you would expect, this is dangerous, and management of eBPF’s strong functionality is imperfect, so most kernels (except android, which restrict BPF through the JNI) disallow unprivileged BPF access altogether

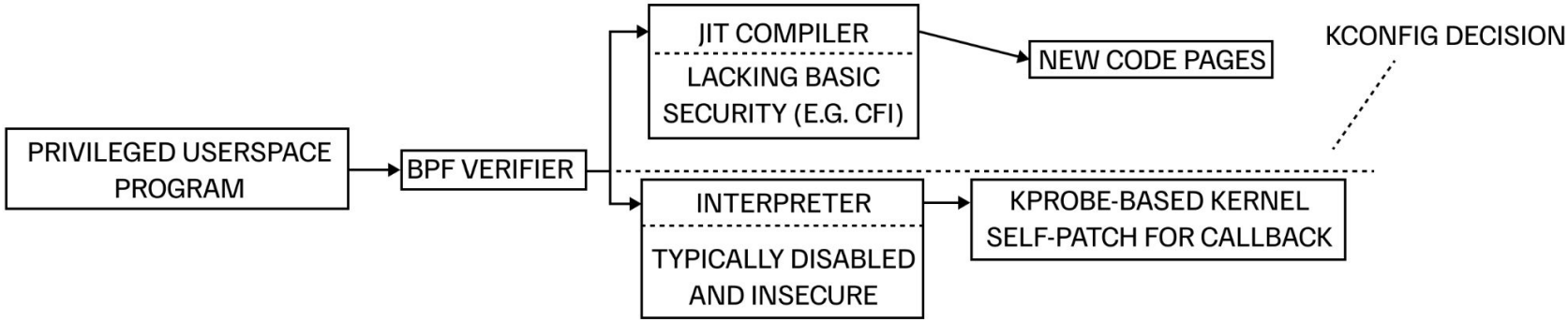


What Exploits Look Like

- Many, many examples of BPF use as a second step once malware is loaded https://www.trendmicro.com/vinfo/us/security/news/threat-landscape/how-bpf-enabled-malware-works-bracing-for-emerging-threats

- An example: Symbiote first gets privileged execution, and then uses a BPF program to filter and control network traffic to/from the infected device

- Open research area: loaded BPF programs could potentially contain exploitable code, and if data can be passed into this chain, it does not have CFI restrictions



Emerging Threats: Building a eBPF Exploit

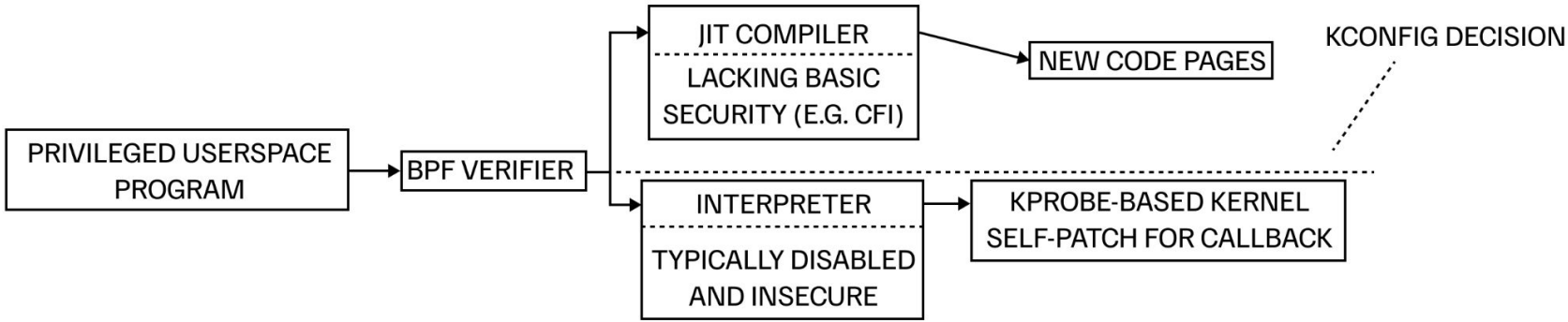
- BPF verifier guarantees code does not contain loops and does not write to memory outside of its dedicated area, but can have issues (CVE-2021-3490), creating:

1) Find a program that allows you to load a BPF program
2) Find a way to load a BPF program yourself (e.g. for a while, any user could load eBPF programs and run them by attaching them to a socket they own)

- BPF programs are approximately verified C, so ideally are totally safe, though

3) Find an existing BPF program conditioned on user input that allows for some form of confused deputy attack

- Disabling unprivileged BPF became a redhat linux requirement in 09/2023



Emerging Threats: Building a eBPF Exploit

- What can you do in a BPF program?
  - Pretty much C, still has buffer parsing/verifier, but … no CFI (yet!)

```
/* Vulnerable BPF function example */
SEC("kprobe/do_unlinkat")
int BPF_KPROBE(do_unlinkat, int dfd, struct filename *name)
{
    char *filename;
    char foo[20] = {0};

    filename = BPF_CORE_READ(name, name);
    bpf_probe_read(foo, 10, filename);
    bpf_trace_printk(foo, sizeof(foo)); /* format string injection */
    return 0;
}

$ touch test
$ rm test
$ touch %x%x%x
$ rm %x%x%x
$ sudo cat /sys/kernel/debug/tracing/trace_pipe
           <...>-6447    [014] d..31  3197.540982: bpf_trace_printk: test
           <...>-6447    [014] d..31  3197.541013: bpf_trace_printk: KPROBE EXIT: pid = 6447, ret = 0
           <...>-6393    [012] d..31  3110.270100: bpf_trace_printk: 110fefefeff
           <...>-6393    [012] d..31  3110.270154: bpf_trace_printk: KPROBE EXIT: pid = 6393, ret = 0
```


Exploit Mitigation Research
Fixing up the kernel with **hard work** and **empirical measurement**.

Preventing Write Gadgets

- Our baseline support is W^X, but this is applied inconsistently to kernel virtual memory mapping and filesystem support
  - i.e. what if I need to load a kernel module dynamically, or want to compress part of my code memory until it is loaded to free up ram?

- general idea was partitioning code out from data, however, this is not consistently applied across the kernel because of another protection: ASLR
  - https://lore.kernel.org/all/20240220203256.31153-1-mbland@motorola.com/

UXN	PXN	AP[2:1]	SCTLR_EL1.WXN	Access from EL1	Access from EL0
0	0	00	0	R, W, Executable	Executable
			1	R, W, Not executable[a]	Executable
		01	0	R, W, Not executable[b]	R, W, Executable
			1	R, W, Not executable	R, W, Not executable[c]
		10	x	R, Executable	Executable
		11	x	R, Executable	R, Executable
0	1	00	x	R, W, Not executable	Executable
		01	0	R, W, Not executable	R, W, Executable
			1	R, W, Not executable	R, W, Not executable[c]
		10	x	R, Not executable	Executable
		11	x	R, Not executable	R, Executable

Protecting Dynamics: Rust to the Rescue?

- But what about memory that must remain dynamic?
  - Open problem:
    - difficulty involved in tracking the appropriate use of the large number of pointers in the kernel
    - difficulty involved in determining whether a write is malicious or not generically: is this change in the TRNG device pointer “normal?”

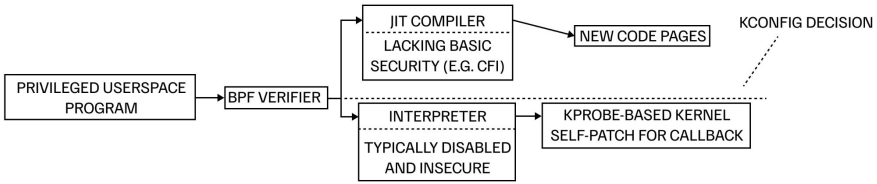
- Potential solutions:
  - Remove all buffer overflows, coding errors, everything
    - won’t solve the problem, new code gets added all the time
  - Switch everything over to rust and ban unsafe rust altogether
    - maybe … Torvalds is in the "wait and see" camp
  - Microarchitecture: ARM rolled out “pac cfi” for stack protection
    - unique key associated with pointer modifications
  - Something you come up with!

What eBPF indicates about Hardening the Future

- eBPF stands as a great case study that our work may never be complete
  - A new feature is added and opens new exploits we may not be fully aware of

- But, let's recall "Disabling unprivileged BPF became a redhat linux requirement in 09/2023"
  - New features need to get "contained" somehow, but Android didn't do it

- So we run into another set of diverging paths:
  - Virtualize, virtualize, virtualize: Android is rolling out "microkernels", which put potentially every program into its own VM
    - Or just put eBPF into its own environment
  - My current approach: solve the problem at its root!
    - Remove the interpreted path from BPF
    - Treat BPF code like any other dynamic kernel module



Thank you! Questions?

Maxwell Bland, Motorola Mobility