BPF-NX+CFI

Maxwell Bland, Motorola Mobility
For LSF/MM/BPF 2024
mbland@motorola.com bland@sdf.org

Problem Statement / Overview

- *Verification Bugs:* Privilege Escalation Attacks by classic verifier manipulation
    - CVE-2021-3490 -> ALU32 bounds tracking for bitwise ops did not properly update 32-bit bounds, turning into out of bounds reads and writes in the Linux kernel and therefore, arbitrary code execution https://chomp.ie/Blog+Posts/Kernel+Pwning+with+eBPF+-+a+Love+Story
    - Similar ALU exploit used for cve-2020-8835

- *Exploit Chaining:* Use BPF memory to help other attacks
    - EPF (https://cs.brown.edu/~vpk/papers/epf.atc23.pdf) -> use BPF programs to store the payload for a UAF + heap spray and jump to executable page

- *Unprivileged Misuse:* Malicious BPF Programs (e.g. Symbiote)
    - https://github.com/bfengj/eBPFeXPLOIT/blob/main/ebpf/main.c
    - Mostly not possible with the disabling of unprivileged BPF
        - But still affects Android system-privileged apps (depending on GID)*

*See appendix on protections for bpf() syscall

Background: The x86 BPF-CFI Implementation

- Peter Zijlstra implemented support for the removal of the __nocfi directive from bpf_dispatcher_* by adding CFI-enforcing assembly instructions to BPF programs

```
--- a/include/linux/bpf.h
+++ b/include/linux/bpf.h
...
@@ -1211,7 +1212,11 @@ struct bpf_dispatcher {
 #endif
 };
-static __always_inline __nocfi unsigned int bpf_dispatcher_nop_func(
+#ifndef __bpfcall
+#define __bpfcall __nocfi
+#endif
...
--- a/arch/x86/net/bpf_jit_comp.c
+++ b/arch/x86/net/bpf_jit_comp.c
@@ -315,10 +381,10 @@ static void emit_prologue(u8 **pprog, u3
 {
        u8 *prog = *pprog;
+       emit_cfi(&prog);
```

- Note CFI here is the ENDBRANCH instruction. On indirect branches, x86 machines supporting CFI throw a fault if the next instruction is not ENDBRANCH.

Background: The the aarch64 CFI Implementation

- So now bpf_dispatcher no longer needs __nocfi and all bpf functions are emitted with an ENDBRANCH. Backward-edge CFI supported through shadow stack
  - Mike Rapoport's article here https://lwn.net/Articles/900099/
- For aarch64, prologues have had BTI/PAC instructions since https://lore.kernel.org/bpf/20220711150823.2128542-4-xukuohai@huawei.com
  - but bpf_dispatcher_*_func still had the __nocfi attribute
  - See Mark Rutland + Puranjay Mohan patch: https://lore.kernel.org/all/ZgwJsJPUyPVNdpZb@FVFF77S0Q05N/

```
<bpf_dispatcher_*_func>:
paciasp
stp     x29, x30, [sp, #-0x10]!
mov     x29, sp
+ ldur    w16, [x2, #-0x4]
+ movk    w17, #0x1881
+ movk    w17, #0xd942, lsl #16
+ cmp     w16, w17
+ b.eq    <bpf_dispatcher_*_func+0x24>
+ brk     #0x8222
blr     x2
ldp     x29, x30, [sp], #0x10
autiasp
ret
```

Broader Issues of CFI, PXN and Code Integrity

- From Man Yue Mo’s https://github.blog/2022-06-16-the-android-kernel-mitigations-obstacle-race/

```
1. Add entries to kworker queue using write gadget
2. Because of kCFI, I must call functions with the following signature:
void (func*)(struct work_struct *work)
3. Turns out to be fairly simple: the function call_usermodehelper_exec_work, fits the bill
```

- BPF-CFI patches:

```
> +           emit_kcfi(cfi_get_func_hash(func_addr), ctx);
> so the calling code will fetch the type_id from above the destination
> and compare it with the type_id of the above prototype.
> To make this work with BPF trampolines
...
> we use cfi_get_func_hash() to fetch the type_id and put it above the
> landing location in the trampoline.
```

- For an indirect function call CallSiteTypeId is the first 8 bytes of the xxHash of the function signature (https://clang.llvm.org/docs/ControlFlowIntegrityDesign.html)
- github.com/llvm/llvm-project/blob/531a0b67ea1ad65ea4d1a99c67fee280beeb8fbb/clang/lib/CodeGen/CodeGenModule.cpp#L2112

The XN in BPF-XN+CFI

- CVE-2024-1086: find a way to rewrite/confuse the page-table and flip the PTE bits
- Example: https://github.com/Notselwyn/CVE-2024-1086/blob/main/src/main.c#L376
- Write to eBPF in window between writability and executability qualys.com/2021/07/20/cve-2021-33909/sequoia-local-privilege-escalation-linux.txt
- Fun example on a Boeing 747 https://www.youtube.com/watch?v=r4M9AFZcj2w

- With this mechanism, a BPF program can be counterfeited using vmalloc data
  - Bypass CFI by calculating or writing the desired CallsiteTypeId hash

- Not just a BPF but a mm and storage issue?
  - kprobes/jump labels (see appendix self-patching exploit) and storage, particularly the EROFS filesystem's loading-in of executable code via fixmap

- A sort of option: list of "reserved" vaddr ranges which cannot be allocated unless explicitly requested by "vstart" of find_vmap_lowest_match ← alloc_vmap_area
  - Paired with further program verification and checking after marking executable
  - Complications detailed on next slide

Complications to NX introduced by BPF (and kprobes)

- Overrides between arch and core BPF allocation regions added because MODULES_VADDR/VMALLOC_START are not supported by arch code

`fdadd04931c2 ("bpf: fix bpf_jit_limit knob for PAGE_SIZE >= 64K")`

- Case study: allocation restriction entirely lifted in aarch64
https://lore.kernel.org/bpf/1636131046-5982-2-git-send-email-alan.maguire@oracle.com

```
“The practical reason to relax this restriction on JIT memory is that 128MB of JIT memory can be quickly exhausted,
especially where PAGE_SIZE is 64KB - one page is needed per program. In cases where seccomp filters are applied to
multiple VMs on VM launch - such filters are classic BPF but converted to BPF - this can severely limit the number of
VMs that can be launched.”
```

- Proposal: large BPF range allocation should be an opt-in, to allow for greater security on systems (Android, embedded) that use a limited number of BPF program allocations and want integrity checking on these allocations

- Build on top of Mike Rapoport execmem patch:
https://lore.kernel.org/all/20240505160628.2323363-1-rppt@kernel.org/

What’s Next?

- *Interim:* Puranjay’s patch for aarch64 CFI
- *CFI/Clang:* Better hashing/auth mechanisms — mix in a dynamic nonce value?
- *BPF:* Introduce kconfig to manage the allocation size provided to BPF to provide a baseline for the restriction of code allocations / integrity checks for restricted BPF environments
- *Storage*: Figure out a decent mechanism for security/monitoring fixmap updates in EROFS uncompression operations
- *MM*: Provide a mechanism to support ASLR-respecting code allocations while ensuring a verifiable difference between code and data (for purposes of identifying payload or PTE/PMD modification attacks)

Thank you!

Maxwell Bland, Motorola Mobility

Appendix

How Android Prevents BPF Misuse

- Android allows unprivileged BPF loading
  - Unprivileged BPF can use BPF as a tool to hid RAT in latin america targeting financial sector socket handling https://intezer.com/blog/research/new-linux-threat-symbiote/
- FS-level restrictions for rwx by GID on /sys/fs/bpf/, GID's are tied into the Android permission system, e.g. android.permission.UPDATE_DEVICE_STATS
  - Protections provided by the "bpfloader" selinux context and framework library, which restricts the core init-time loading of BPF programs
    - But bpf() syscall protected by Seccomp filter, not Loader.cpp?

- UID restrictions on bpf_prog_load bpf system call func
  - Unfortunately there are a lot of 3rd party system-signed apps and signing keys leak: https://bugs.chromium.org/p/apvi/issues/detail?id=100

- Potential improvement: integrity checks on loaded BPF programs
  - Fine-grained BPF capability restriction?

Unsafe Parameters?

- The use of functions with unsafe parameters unknowingly:

```
/* 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
```

An Example GPU Write Gadget 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

- 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"

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

Are we doomed to exploitability forever? Writable Options

- Digging up an old skeleton: 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: https://github.com/chompie1337/s8_2019_2215_poc
  - Overwrites kernel file operations pointer to arbitrary function pointer

- UAF tend to depend on heap-spray attacks though: we could avoid understanding data structure semantics by providing more fine-grained control over pages and then more fine-grained preventions against heap-spray attacks

Example Misuse of Self-Patching interface

```
/*
 * Critically, the code for a jump entry is calculated using the 64 bits
 * of the address of the jump entry struct's code member, and then this is
 * added to the value of the code member, so we must take
 * this into account when writing an address by allocating a fake jump
 * entry using an existing data structure in the same upper 32 bit memory
 * region. Spectre_bhb_state was chosen for no particular reason, other
 * than being in the BSS and having enough adjacent memory
 */
#define ATTACK_KERNEL_CODE                                              \
	do {                                                            \
		fake_je = (struct jump_entry *)kallsyms_lookup_name_ind(       \
			"spectre_bhb_state");                                 \
		attack_addr = kallsyms_lookup_name_ind("udp_recvmsg");        \
		if (register_kprobe(&kp2)) {                                  \
			return -1;                                            \
		}                                                             \
		arch_jump_label_transform =                                   \
			(arch_jump_label_transform_t)kp2.addr;                \
		fake_je->code = attack_addr - (unsigned long)&(fake_je->code); \
		fake_je->target = stext - (unsigned long)&(fake_je->target);   \
		arch_jump_label_transform(fake_je, JUMP_LABEL_JMP);           \
		return 0;                                                     \
	} while (0)
```