Peeking Under The Hood

Low level visibility in Android with eBPF

Alex Calleja
Malware Researcher
alex.calleja(at)zimperium.com

BSides Malaga
14/3/2025

Outline

- Introduction to eBPF
- Motivation: eBPF for App analysis
- eBPF in Android
- Demo Time!
- eBPF Limitations
- Conclusions

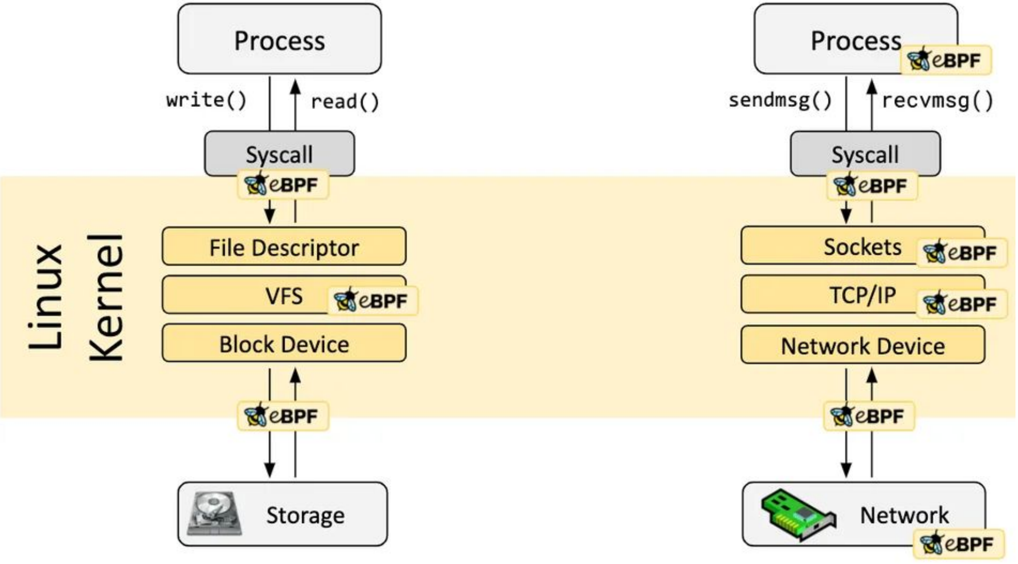
whoami

- Researcher @ Zimperium since 2018
- Android malware
- AOSP internals
- Leading the development of dynamic analysis platform
- Scaled the analysis pipeline to thousands of apps per day

alex.calleja(at)zimperium.com

Introduction to eBPF

- Instrument the kernel with event-driven programs
- Great observability due to privileged execution level
- Sandboxed environment & BPF validator ensure safety
- Applied across various fields, including networking, security, performance analysis, and more



Introduction to eBPF II

- BPF supports different program types with different functionalities

- Tracing programs allow attaching complex event handlers to system tracing probes

`/sys/fs/trace`

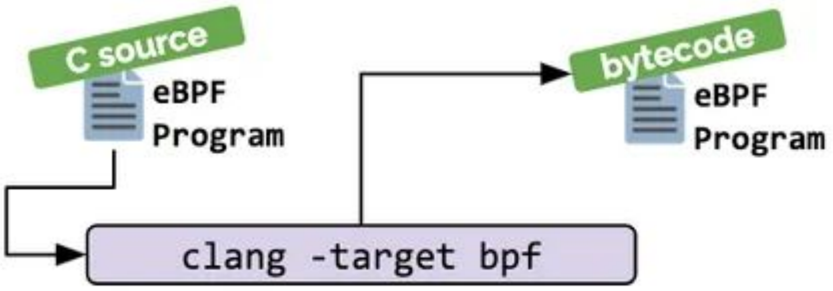
tracepoints
- Allows to instrument specific functions in the kernel, marked as traceable

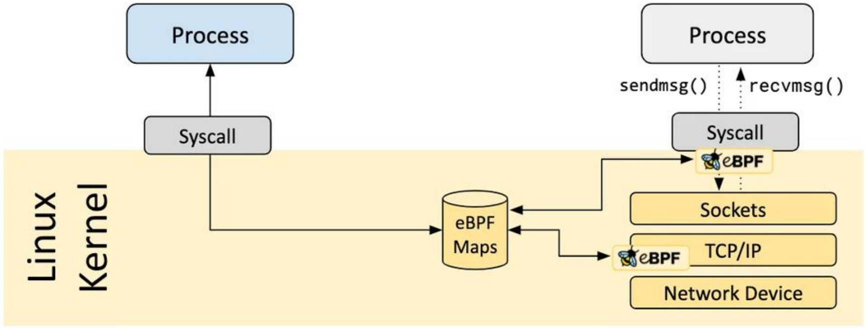
k(ret)probes
- Allows to intercept call/return events of kernel functions

u(ret)probes
- Allows to intercept call/return events of user-space code
- Requires specifying executable program and offset

Introduction to eBPF III

- Common setup consists in a **loader** + a **BPF implant**
- **CO-RE**: ensure compatibility across different kernel versions





- Different **data structures** allow communication between user-space and bpf side

Motivation: eBPF for App Analysis

Malware analysis

- Dynamic analysis often fails due to the detection of the emulated environment
- Frida usually saves the day, but falls short under specific circumstances
- Performance penalty when too much instrumentation is present

Motivation: eBPF for App Analysis

Packers/Runtime app self-protection reversing

- RASPs usually rely on inlined syscalls and other tricks (e.g hooked libc)
- Often include loops checking for root, emulator or dynamic binary instrumentation tooling
- This checks run usually on `.init` functions, which hinders further research

Motivation: Alternatives to eBPF

Dynamic binary Instrumentation (e.g Frida)

💪 Quick & easy solution
💪 Flexibility
💪 Decent compatibility across Android version

👎 Huge footprint
👎 Performance drops as agent size increases

Kernel Instrumentation (e.g Kernel modules)

💪 Full access to kernel APIs/types
💪 Full access to system & process resources
💪 Minimal overhead

👎 Requires painful kernel recompilation loops
👎 System crashes

BPF Programs

✌️ Convenient trade-off between flexibility & observability

eBPF in Android

eBPF is well supported in Android kernels

Since Android13 stock kernels (5.10+) expose BTF debugging info

Android

AOSP contains its own version of libbpf (`libbpf_android.so`)

AOSP includes BPF implants for measuring power & traffic

eBPF in Android

Approach ⚙️	Pros ✅	Cons ❌
AOSP eBPF loader & examples	• Native integration with Android APIs	• Requires setting up AOSP build environment
NDK + `Cilium`	• Good support for complex use-cases • High flexibility for advanced eBPF	• Higher complexity • Requires minimum understanding of NDK and Go's build system
`adeb` + `BCC`	• Ideal for experimental setup • Active community support for BCC	• Performance drops due to BCC overhead • Dependency management
Build `libbpf` + loader from scratch	• Full control over libbpf and program behavior • Highly flexible for specialized use-cases	• High Integration effort • Limited community and support for Android-specific solutions

eBPF in Android

Approach ⚙️	Pros ✅	Cons ❌
AOSP eBPF loader & examples	• Native integration with Android APIs	• Requires setting up AOSP build environment
NDK + `Cilium`	• Good support for complex use-cases • High flexibility for advanced eBPF	• Higher complexity • Requires minimum understanding of NDK and Go's build system
`adeb` + `BCC`	• Ideal for experimental setup • Active community support for BCC	• Performance drops due to BCC overhead • Dependency management
Build `libbpf` + loader from scratch	• Full control over libbpf and program behavior • Highly flexible for specialized use-cases	• High Integration effort • Limited community and support for Android-specific solutions

eBPF in Android: Building BPF programs from scratch

- An user space `loader` program must be written in order to load the bpf ELF file
- Alternatively, use `bpftool` and load `skel.h` file
- Linking loader against `libbpf` allows us to use `bpf_prog__*` API
- The loader must also contain the logic to consume the data sent from the implant or read from `trace_pipe`

```c
struct bpf_object* obj;
struct bpf_program * prog;
obj = bpf_object__open_file("/data/local/tmp/implant.o.bpf", NULL);
/* load program, this issues a BPF(BPF_PROG_LOAD, ) syscall */
if (bpf_object__load(obj)) {
    fprintf(stderr, "ERROR: loading BPF object file failed. Errno: %d\n");
    return 1;
}
/* find program in loaded ELF */
prog = bpf_object__find_program_by_name (obj, "handle_tp_sys_enter");
if (!prog) {
        fprintf(stderr, "ERROR: finding a prog in obj file failed\n");
        goto cleanup;
    }
struct bpf_link *link = NULL;
/* associate program with tracepoint */
link = bpf_program__attach(prog);
```

```c
void ringbuff_callback(void *ctx, int cpu, void *data, unsigned int data_sz){
    printf("New event received!\n")
}
int fd = bpf_object__find_map_fd_by_name(obj, "ringbuf_map");
if(fd<0){
    fprintf(stderr, "ERROR: could not find fd for map\n");
    return -1;
}
struct ring_buffer * rb  = ring_buffer__new (fd, ringbuff_callback, NULL, NULL );
while (1) {
        err = ring_buffer__poll(rb, 100);
        if (err < 0) {
            printf("Error polling ring buffer: %d\n", err);
            break;
        }
    }
```

eBPF in Android: Building BPF programs from scratch

- SEC macro defines sections within the resulting `ELF` file
- The program type defines the parameters that are accepted

```c
SEC("tp/raw_syscalls/sys_enter")
int handle_tp_sys_enter(struct trace_event_raw_sys_enter* ctx) {
    int pid = bpf_get_current_pid_tgid() >> 32;
    long syscall_nr = ctx->id;
    bpf_printk("PID %d issues syscall:  %ld\n", pid, syscall_nr);
    return 0;
}
```

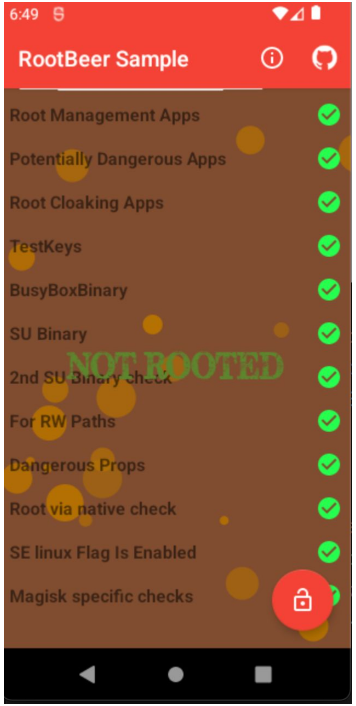
```makefile
implant: bpf_example.c
	$(CLANG) -g -O2 -target bpf    \
	         -include vmlinux.h    \
	         -I../loader/dependencies/libbpf/src/    \
	         I../loader/dependencies/libbpf/include/bpf    \
	         -c $^ -o $(IMPLANT_OBJ)
```

- Invoke `clang` with `-target bpf`

- Resulting bytecode can be checked with `llvm-objdump -S`

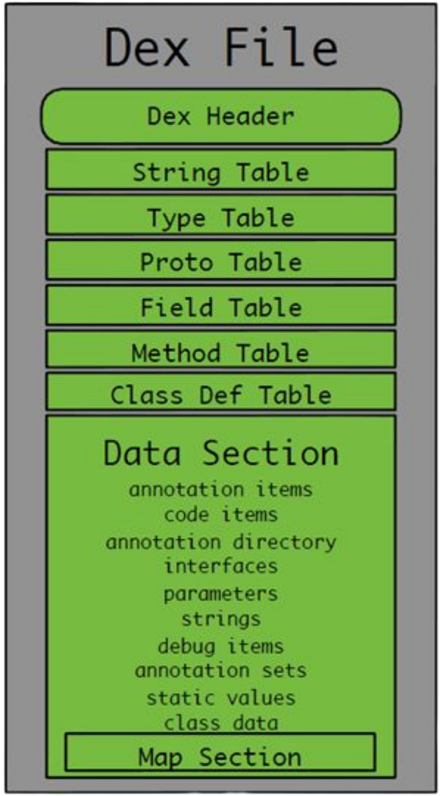
```
Disassembly of section tp/raw_syscalls/sys_enter:

0000000000000000 <handle_tp_sys_enter>:
; int handle_tp_sys_enter(struct trace_event_raw_sys_enter* ctx) {
       0:       bf 16 00 00 00 00 00 00 r6 = r1
;       int pid = bpf_get_current_pid_tgid() >> 32;
       1:       85 00 00 00 0e 00 00 00 call 0xe
;       long syscall_nr = ctx->id;
       2:       79 64 08 00 00 00 00 00 r4 = *(u64 *)(r6 + 0x8)
;       int pid = bpf_get_current_pid_tgid() >> 32;
       3:       77 00 00 00 20 00 00 00 r0 >>= 0x20
;       bpf_printk("PID %d issues syscall:  %ld\n", pid, syscall_nr);
       4:       18 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 r1 = 0x0 ll
       6:       b7 02 00 00 1d 00 00 00 r2 = 0x1d
       7:       bf 03 00 00 00 00 00 00 r3 = r0
       8:       85 00 00 00 06 00 00 00 call 0x6
;       return 0;
       9:       b7 00 00 00 00 00 00 00 r0 = 0x0
      10:       95 00 00 00 00 00 00 00 exit
```

Demo Time!



- RootBeer can detect if the device has been rooted or tampered with
- The detection is mainly based on several file system checks (i.e `su`, `magisk` binaries)
- It also checks for specific property values by calling `getprop` (`ro.debuggable`, `ro.secure`) and grepping its output
- Additionally calls `mount` to check for `+rw` permissions on `system` partition
- **How can we evade this detections with kprobes+eBPF?**
    - Identify relevant syscalls (attach to `sys_enter` tracepoint)
    - Kprobe relevant syscals
    - Use `bpf_probe_write_user` to smash arguments passed to the syscall

Demo Time!

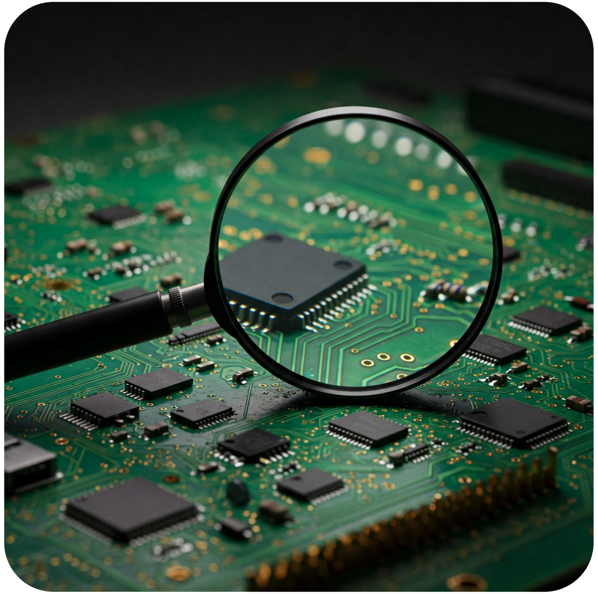


- Often malware loads different DEX files during its execution
- This technique is used for hiding payloads and evade static analysis
- Frida can be used to instrument an unpacking point and dump the loaded DEX file
- What if the app is hardened against Frida?
- **Uprobes in action: dumping dex files without Frida**
  - Locate relevant function in user space (`OpenCommon` from `libdexfile.so`)
  - Place a Uprobe
  - Send back to user space {`pid`, `dex_base`, `dex_size`}
  - Use `process_vm_readv` to read `dex_size` bytes from target process memory

BPF Limitations

- Kernel memory is not writable
- Just a small subset of Kernel APIs/types are available
- The validator imposes several limitations such as:
  - Very restrictive pointer arithmetic
  - Loops must be deterministic (i.e it must be possible to check if a loop ends at load time)
- Program size restricted to 4096 instructions
- Despite efforts, compatibility is still an issue, mostly with older (<11) Android versions

Conclusions



- BPF provides a way to safely instrument the kernel with event-driven programs to gain observability of low level events
- It can be used for malware analysis and app analysis to overcome limitations of other approaches
- eBPF is well supported in Android kernels
- Still, some integration effort is required in Android
- Overall, eBPF is a valuable addition to your app analysis toolkit

Thank you!

Alex Calleja

Researcher

alex.calleja(at)zimperium.com

Code & slides available in github.com/alximw/BSidesMalaga25