eBPF - Android Reverse Engineering Superpowers

Terry Chia

$ whoami

- Security Consultant @ Centurion Information Security
- Blog: https://www.ayrx.me
- Github: https://github.com/Ayrx

Agenda

- What is eBPF?
- Why eBPF?
- eBPF on Android
- Writing eBPF programs

What is eBPF?

eBPF

- extended Berkeley Packet Filter
- Framework for tracing a Linux system
- Linux version of Solaris' DTrace
- Requires a relatively new Linux kernel (> 4.1). Newer kernels might have more features.

eBPF

- Write custom code that triggers whenever *something* happens in the system.
- Write eBPF program & compile to bytecode.
- eBPF bytecode is loaded through the *bpf(2)* syscall.
- eBPF code is executed with an in-kernel virtual machine.

eBPF

- Event Sources:
  - kprobes / kretprobes
  - uprobes / uretprobes
  - Tracepoints
  - User Statically-Defined Tracing Probes (USDT)

k(ret)probes, u(ret)probes

- kprobes / kretprobes are used to attach to kernel functions.

- uprobes / uretprobes are used to attach to userspace functions.

Tracepoints

- Tracepoints are used to attach to *events* within the kernel.

- A large number of events are exposed by the kernel. See *[/sys/kernel/debug/tracing/events/]* for a full list.

Tracepoints

```
root@d1visi0n:~# ls /sys/kernel/debug/tracing/events/
alarmtimer  filemap         irq_matrix     module          regmap      thermal_power_allocator
asoc        fs              irq_vectors    mpx             regulator   timer
block       fs_dax          iwlwifi        msr             rpm         tlb
bridge      ftrace          iwlwifi_data   napi            rseq        ucsi
btrfs       gpio            iwlwifi_io     net             rtc         udp
cfg80211    gvt             iwlwifi_msg    nfsd            sched       v4l2
cgroup      hda             iwlwifi_ucode  nmi             scsi        vb2
clk         hda_controller  jbd2           nvme            signal      vmscan
cma         hda_intel       kmem           oom             skb         vsyscall
compaction  header_event    kvm            page_isolation  smbus       wbt
cpuhp       header_page     kvmmmu         pagemap         sock        workqueue
dma_fence   huge_memory     libata         percpu          spi         writeback
drm         hyperv          mac80211       power           sunrpc      x86_fpu
enable      i2c             mac80211_msg   printk          swiotlb     xdp
exceptions  i915            mce            qdisc           sync_trace  xen
ext4        initcall        mdio           random          syscalls    xhci-hcd
fib         intel-sst       mei            ras             task
fib6        iommu           migrate        raw_syscalls    tcp
filelock    irq             mmc            rcu             thermal
root@d1visi0n:~#
```

User Statically-Defined Tracing Probes (USDT)

- USDTs are Tracepoints for userspace.

```c
#include <sys/sdt.h>
#include <sys/time.h>
#include <unistd.h>

int main(int argc, char **argv)
{
    struct timeval tv;

    while(1) {
        gettimeofday(&tv, NULL);
        DTRACE_PROBE1(test-app, test-probe, tv.tv_sec);
        sleep(1);
    }
    return 0;
}
```

eBPF

- eBPF also provides a way for userspace to communicate with a eBPF program.
- BPF_PERF_OUTPUT
  - Fast ring buffer
  - Can create multiple ring buffers per eBPF program

eBPF

- Don’t write eBPF bytecode by hand.
- Use the *bcc* compiler!
  - https://github.com/iovisor/bcc

eBPF

- Running arbitrary code in the kernel is risky.
- eBPF has a validator that tries to ensure that eBPF programs are “safe”
  - eBPF program must terminate
  - Validates stack / register state
  - Validates no out-of-bounds reads

Current eBPF Usage

- eBPF is mainly used for instrumenting *production* Linux systems.
- Especially popular in container / kubernetes environment.
- Firewalls - https://cilium.io/blog/2018/11/20/fb-bpf-firewall/

Why eBPF?

eBPF - Android Reverse Engineering Superpowers

Reverse Engineering

- Reverse Engineering is about understanding an application.
- Three main categories of techniques:
  - Static Analysis - IDA Pro / Ghidra
  - Debugging - GDB / WinDBG / Intel PIN / Frida
  - Behavioural Analysis - strace / ltrace / Procmon

Anti-Reversing

- Anti-Reversing tricks for each technique:
  - Static Analysis - Obfuscation
  - Debugging - Anti-debugging, Root / Jailbreak Detection
  - Behavioural Analysis - Anti-debugging, Root / Jailbreak Detection

Anti-Reversing

- OWASP MSTG describes some common anti-reversing techniques:
    - https://mobile-security.gitbook.io/mobile-security-testing-guide/android-testing-guide/0x05j-testing-resiliency-against-reverse-engineering
    - https://mobile-security.gitbook.io/mobile-security-testing-guide/ios-testing-guide/0x06j-testing-resiliency-against-reverse-engineering

Android Anti-Reversing

- Android applications commonly utilize a combination of the following tricks:
  - Root Detection
  - Anti-Debugging
  - Obfuscation
  - File Integrity Checks

Android Anti-Reversing

- Tricks that prevent *ptrace* from being used are particularly annoying.
- Many common tools rely on being able to *ptrace* the target process.
  - strace
  - Gdb
  - Frida

Anti-ptrace Example

- Only one tracer can be attached to a process.
- *ptrace*-ing a process that is already being debugged by another process will fail.
- A common technique is for an application to fork a child process that attaches to the parent process.

Anti-ptrace Example

```c
void fork_and_attach()
{
  int pid = fork();

  if (pid == 0)
    {
      int ppid = getppid();

      if (ptrace(PTRACE_ATTACH, ppid, NULL, NULL) == 0)
        {
          waitpid(ppid, NULL, 0);

          /* Continue the parent process */
          ptrace(PTRACE_CONT, NULL, NULL);
        }
    }
}
```

- Fork a child process.

Anti-ptrace Example

```c
void fork_and_attach()
{
  int pid = fork();

  if (pid == 0)
    {
      int ppid = getppid();

      if (ptrace(PTRACE_ATTACH, ppid, NULL, NULL) == 0)
        {
          waitpid(ppid, NULL, 0);

          /* Continue the parent process */
          ptrace(PTRACE_CONT, NULL, NULL);
        }
    }
}
```

- Fork a child process.
- Child process calls *ptrace* on the parent process.
- As long as the child process is alive, no other process can *ptrace* the parent.

Anti-ptrace Example

- In the trivial example, killing the child process will allow the parent process to be debugged.
- In practice, tricks to ensure that the child process remains alive can be used.
  - Forking multiple processes tracing each other
  - Monitoring running processes

Android Anti-Reversing

- Applications with multiple anti-RE tricks implemented can be difficult to analyze.
- Low level (kernel!) capabilities are really helpful to debug such applications.
- You essentially want capabilities at a level of the system that the application cannot subvert.

Android Anti-Reversing

- Previously, this meant writing a kernel module.
  - Error-prone: Bad code means crashing your device
  - Tedious development process: Write -> Compile -> Transfer to Device -> Hope it works
- What other options do we have if we want to run custom code in the kernel? 🤔

eBPF on Android

Requirements & Setup

- eBPF works on Android because Android uses a relatively standard Linux kernel.
- Took me a while to figure out how to get everything working.
- Most of the documentation for the eBPF toolchain assumes standard Linux instead of Android.

bcc toolchain

- *bcc* is the standard eBPF compiler toolchain.
    - https://github.com/iovisor/bcc
- It is a LLVM-based compiler toolchain that compiles C code to eBPF bytecode.
- Requires kernel headers to be present.
- Requires Python.

adeb

- *adeb* makes it easy to setup the bcc toolchain.
  - https://github.com/joelagnel/adeb
- Essentially a Debian-based environment running on the Android device via chroot magic.
- Comes with *bcc* and other useful tools.

Building the Android Kernel

- *adeb* still requires a kernel with the required configs turned on.
- Kernel version 4.9 and above
  - CONFIG_KPROBES=y
  - CONFIG_KPROBE_EVENT=y
  - CONFIG_BPF_SYSCALL=y
  - CONFIG_IKHEADERS=m
  - CONFIG_UPROBES=y
  - CONFIG_UPROBE_EVENT=y

Building the Android Kernel

- No current Android ships with the necessary configs.
  - Have not looked at Android Q yet.
- This means building your own kernel.
- Process differs depending on where you are running Android (Emulator vs device, etc)

adeb

- adeb prepare --full --build --kernelsrc path/to/kernel
    - --arch <amd64/arm64/etc>
    - --buildtar <output dir>
- adeb prepare --archive <output dir>/androdeb-fs.tgz --kernelsrc path/to/kernel

adeb

- adeb shell
- *bcc* by default comes with some really useful utilities.
  - filetop
  - opensnoop (and other *snoop commands)
  - etc

Writing eBPF programs

bcc toolchain

- With *bcc*, you write eBPF programs in Python.
  - import bcc
  - Build a string that contains the eBPF program.
  - Pass string to bcc which invokes LLVM behind the scenes and loads the compiled program into the kernel.

Hello World

- Write an eBPF program that prints out all the files opened on the system.
  - Attach to an appropriate kernel function
  - Send the pathname being opened to userspace
  - Print the output

Hello World

```
#!/usr/bin/python

from bcc import BPF

program = """
#include <asm/ptrace.h>
#include <uapi/linux/limits.h>

struct open_data_t {
    char fname[NAME_MAX];
};

BPF_PERF_OUTPUT(open_event);

int kprobe__sys_openat(struct pt_regs *ctx,
    int dirfd, char __user* pathname, int flags, mode_t mode) {

    struct open_data_t data = {{}};
    bpf_probe_read(&data.fname, sizeof(data.fname), (void *)pathname);
    open_event.perf_submit(ctx, &data, sizeof(data));

    return 0;
}
"""
```

- *open_data_t* is a struct that stores data we want to send to userspace.
- BPF_PERF_OUTPUT opens up a ring buffer called *open_event* .

Hello World

```
#!/usr/bin/python

from bcc import BPF

program = """
#include <asm/ptrace.h>
#include <uapi/linux/limits.h>

struct open_data_t {
    char fname[NAME_MAX];
};

BPF_PERF_OUTPUT(open_event);

int kprobe__sys_openat(struct pt_regs *ctx,
    int dirfd, char __user* pathname, int flags, mode_t mode) {

    struct open_data_t data = {{}};
    bpf_probe_read(&data.fname, sizeof(data.fname), (void *)pathname);
    open_event.perf_submit(ctx, &data, sizeof(data));

    return 0;
}
"""
```

- *kprobe__* syntax tells *bcc* that the function is a kprobe.
- The function arguments to the syscall can be omitted if your eBPF program does not use them.

Hello World

```
#!/usr/bin/python

from bcc import BPF

program = """
#include <asm/ptrace.h>
#include <uapi/linux/limits.h>

struct open_data_t {
    char fname[NAME_MAX];
};

BPF_PERF_OUTPUT(open_event);

int kprobe__sys_openat(struct pt_regs *ctx,
    int dirfd, char __user* pathname, int flags, mode_t mode) {

    struct open_data_t data = {{}};
    bpf_probe_read(&data.fname, sizeof(data.fname), (void *)pathname);
    open_event.perf_submit(ctx, &data, sizeof(data));

    return 0;
}
"""
```

- Initialize an instance of *open_data_t* to store the file name.
- Use the special *bpf_probe_read* function to copy the data into the *fname* array.
- *open_event.perf_submit* sends the initialized *open_data_t* instance to userspace.

Hello World

```python

def print_open_event(cpu, data, size):
    event = b["open_event"].event(data)
    print event.fname

b = BPF(text=program)
b["open_event"].open_perf_buffer(print_open_event)

while True:
    try:
        b.perf_buffer_poll()
    except KeyboardInterrupt:
        exit()

```

- *print_open_event* is a callback function that can be made to trigger when a perf event is received.

Hello World

```python

def print_open_event(cpu, data, size):
    event = b["open_event"].event(data)
    print event.fname

b = BPF(text=program)
b["open_event"].open_perf_buffer(print_open_event)

while True:
    try:
        b.perf_buffer_poll()
    except KeyboardInterrupt:
        exit()

```

- *print_open_event* is a callback function that can be made to trigger when a perf event is received.
- Initialize an eBPF program and opens up the perf buffer called *open_event*.

Hello World

```python
def print_open_event(cpu, data, size):
    event = b["open_event"].event(data)
    print event.fname

b = BPF(text=program)
b["open_event"].open_perf_buffer(print_open_event)

while True:
    try:
        b.perf_buffer_poll()
    except KeyboardInterrupt:
        exit()
```

- *print_open_event* is a callback function that can be made to trigger when a perf event is received.
- Initialize an eBPF program and opens up the perf buffer called *open_event*.
- Polls all opened buffers in an infinite loop.

Code Generation

- Codegen is a common pattern that you will see in eBPF programs.
- Useful if there is a part of the code you want to change every time you run the program.
  - PID filtering is one example.
- Codegen is also useful to get around eBPF limitations.

Code Generation

```
#!/usr/bin/python

from bcc import BPF

program = """
#include <asm/ptrace.h>
#include <uapi/linux/limits.h>

struct open_data_t {
    char fname[NAME_MAX];
};

BPF_PERF_OUTPUT(open_event);

int kprobe__sys_openat(struct pt_regs *ctx,
    int dirfd, char __user* pathname, int flags, mode_t mode) {

    PID_FILTER

    struct open_data_t data = {{}};
    bpf_probe_read(&data.fname, sizeof(data.fname), (void *)pathname);
    open_event.perf_submit(ctx, &data, sizeof(data));

    return 0;
}
"""
```

Code Generation

```python
def insert_pid_filter(bpf_text, pid):
    bpf_text = "#define FILTER_PID {}\n".format(pid) + bpf_text
    pid_filter = """
    u64 pid_tgid = bpf_get_current_pid_tgid();
    if (pid_tgid >> 32 != FILTER_PID) {
        return 0;
    }
    """
    bpf_text = bpf_text.replace("PID_FILTER", pid_filter)

    return bpf_text

program = insert_pid_filter(program, sys.argv[1])
```

- *insert_pid_filter* replaces the PID_FILTER placeholder in the eBPF program string with C code.

Code Generation

```python
def insert_pid_filter(bpf_text, pid):
    bpf_text = "#define FILTER_PID {}\n".format(pid) + bpf_text
    pid_filter = """
    u64 pid_tgid = bpf_get_current_pid_tgid();
    if (pid_tgid >> 32 != FILTER_PID) {
        return 0;
    }
    """
    bpf_text = bpf_text.replace("PID_FILTER", pid_filter)

    return bpf_text

program = insert_pid_filter(program, sys.argv[1])
```

- *insert_pid_filter* replaces the PID_FILTER placeholder in the eBPF program string with C code.
- The value of the FILTER_PID macro depends on the value of *sys.argv[1]*.

strace.py

- Syscall tracing utility implemented with eBPF
- Trace mode vs Aggregate mode
- Filter by PID / Process Name
- Filter only syscalls you are interested in
- Disclaimer: Pretty ugly code

Modifying the system with eBPF

- eBPF can *write* to userspace memory with the *bpf_probe_write_user* function.
    - int bpf_probe_write_user(void *dst, const void *src, u32 len)
- This only works for userspace memory that already has write permissions in place.
    - So no writing to the .text segment with eBPF
    - You can however write to the stack, heap, etc

Modifying the system with eBPF

- As an example, we can use this capability to bypass simple root detection techniques.
- OWASP MSTG - UnCrackable-Level3.apk
- Looks for the presence of certain files on the system.

Modifying the system with eBPF

/sbin/su	/system/app/Superuser.apk
/system/sbin/su	/system/xbin/daemonsu
/apex/com.android.runtime/bin/su	/system/etc/init.d/99SuperSUDaemon
/system/bin/su	/system/bin/.ext/.su
/system/xbin/su	/system/etc/.has_su_daemon
/odm/bin/su	/system/etc/.installed_su_daemon
/vendor/bin/su	/dev/com.koushikdutta.superuser.daemon
/vendor/xbin/su	

Modifying the system with eBPF

- *faccessat* syscall is used to test for the presence of the files.
    - int faccessat(int dirfd, const char *pathname, int mode, int flags);
- We can use *bpf_probe_write_user* to modify the value of the *pathname* parameter.
- Redirect all *faccessat* calls on those files to a non-existent file.

Closing Notes

Limitations

- The verifier can get in the way of writing complex or substantial programs.
- Requires familiarity with Linux Kernel APIs.
- No stable kernel API. An eBPF program working on one version might break after an upgrade.
    - https://github.com/torvalds/linux/blob/master/Documentation/process/stable-api-nonsense.rst

Limitations

- An eBPF program cannot write to kernel memory.
- TOCTOU issues are a huge problem when hooking syscalls.
  - Exploiting races in system call wrappers - https://lwn.net/Articles/245630/

Awesome-ness

- eBPF offers a lot of power while being relatively simple to write.
  - Good for ad-hoc tracing or scripting
- Want to see: a set of eBPF programs like the ones in *bcc* but focused on security / RE.

Useful Resources

- *bcc* Reference Guide - https://github.com/iovisor/bcc/blob/master/docs/reference_guide.md
- *man* and other Linux kernel programming references
- BPF Performance Tools
    - http://www.brendangregg.com/bpf-performance-tools-book.html

Code:
https://github.com/CenturionInfoSec/ebpf-examples

Slides:
https://bit.ly/2kUnlrg