Kernel Tracing With eBPF

Unlocking God Mode on Linux

Jeff Dileo
@chaosdatumz

Andy Olsen
@0lsen_

35C3

nccgroup

Who are we?

Jeff Dileo (@chaosdatumz)

- Unix aficionado
- Agent of chaos
- Consultant / Research Director @ NCC Group
- I like to do terrible things to/with/in:
  - programs
  - languages
  - runtimes
  - memory
  - kernels
  - packets
  - bytes
  - ...

Andy Olsen (@0lsen_)

- Ultimate frisbee enthusiast
- Amateur chiptune artist
- Security Consultant @ NCC Group
- Il ne parle pas Français

Outline

- eBPF
- Tracing with eBPF
- Defensive eBPF
- eBPF Secure Coding Gotchas
- Offensive eBPF
- Q&A

eBPF — Background

- ”extended” BPF

eBPF — Background

- ”extended” BPF
- But what is BPF?

eBPF — BPF

- Berkeley Packet Filter
- Limited instruction set for a bytecode virtual machine
- Originally created to implement *FAST* programmatic network filtering in kernel
- has a few (2) 32-bit registers (and a hidden frame pointer)
- load/store, conditional jump (forward), add/sub/mul/div/mod, neg/and/or/xor, bitshift
- `tcpdump -i any -n 'tcp[tcpflags] & (tcp-syn|tcp-ack) != 0'`

```
(000) ldh      [14]
(001) jeq      #0x800           jt 2   jf 10
(002) ldb      [25]
(003) jeq      #0x6             jt 4   jf 10
(004) ldh      [22]
(005) jset     #0x1fff          jt 10  jf 6
(006) ldxb     4*([16]&0xf)
(007) ldb      [x + 29]
(008) jset     #0x12            jt 9   jf 10
(009) ret      #262144
(010) ret      #0
```

eBPF — eBPF

- ”extended” Berkeley Packet Filter
- ”designed to be JITed with one to one mapping”
- ”originally designed with the possible goal in mind to write programs in ’restricted C’”
- socket filters, packet processing, **tracing**, internal backend for ”classic” BPF, and more...
- File descriptor-based API through `bpf(2)` syscall
  - Provide:
    - An array of bytecode instructions
    - Type of eBPF program (e.g. `BPF_PROG_TYPE_SOCKET_FILTER`, `BPF_PROG_TYPE_KPROBE`, etc.)
    - Other type-specific metadata
  - Receive:
    - (on success) A file descriptor referring to the in-kernel compiled eBPF program
- The power of eBPF is really in the kernel APIs that will accept an eBPF descriptor and plug it into things

eBPF — eBPF

```
static int add_lookup_instructions(BPFProgram *p, int map_fd, int protocol, bool is_ingress, int verdict) {
...
    struct bpf_insn insn[] = {
      BPF_JMP_IMM(BPF_JNE, BPF_REG_7, htobe16(protocol), 0),
...
      BPF_MOV64_REG(BPF_REG_1, BPF_REG_6),
      BPF_MOV32_IMM(BPF_REG_2, addr_offset),
      BPF_MOV64_REG(BPF_REG_3, BPF_REG_10),
      BPF_ALU64_IMM(BPF_ADD, BPF_REG_3, -addr_size),
      BPF_MOV32_IMM(BPF_REG_4, addr_size),
      BPF_RAW_INSN(BPF_JMP | BPF_CALL, 0, 0, 0, BPF_FUNC_skb_load_bytes),
...
      BPF_LD_MAP_FD(BPF_REG_1, map_fd),
      BPF_MOV64_REG(BPF_REG_2, BPF_REG_10),
      BPF_ALU64_IMM(BPF_ADD, BPF_REG_2, -addr_size - sizeof(uint32_t)),
      BPF_ST_MEM(BPF_W, BPF_REG_2, 0, addr_size * 8),
      BPF_RAW_INSN(BPF_JMP | BPF_CALL, 0, 0, 0, BPF_FUNC_map_lookup_elem),
      BPF_JMP_IMM(BPF_JEQ, BPF_REG_0, 0, 1),
      BPF_ALU32_IMM(BPF_OR, BPF_REG_8, verdict),
    };
...
```

Listing 1: `systemd/src/core/bpf-firewall.c`

eBPF — Important BPF to eBPF Changes

- now 10 64-bit registers, directly mapped to HW CPU registers
  - `R0`: return value from in-kernel function, and exit value for eBPF program
  - `R1-R5`: arguments from eBPF program to in-kernel function
  - `R6-R9`: callee saved registers that in-kernel function will preserve
  - `R10`: read-only frame pointer to access stack
- new `bpf_call` instruction
  - HW-based register passing convention for zero overhead calls from/to other kernel functions
  - Used to call other eBPF programs *and* ”helper” functions
- Bytecode validator (”verifier”)
- Helper functions
  - Set of native kernel functions exposed to eBPF code
  - Context-dependent (e.g. packet processing eBPF cannot call kernel memory read helper)
  - Argument registers validated against call spec for each helper function

Why eBPF?

- HIGH PERFORMANCE in-plane packet processing

Why eBPF?

- (safe) HIGH PERFORMANCE in-plane packet processing

Why eBPF?

- (safe) HIGH PERFORMANCE in-plane packet processing
- network tunneling

Why eBPF?

- (safe) HIGH PERFORMANCE in-plane packet processing
- network tunneling
- custom iptables rules

Why eBPF?

- (safe) HIGH PERFORMANCE in-~~plane~~kernel ~~packet processing~~ programmatic operations
- network tunneling
- custom iptables rules
- syscall filtering (mix of classic [for now] BPF for seccomp, eBPF for cgroups shenanigans)

Why eBPF?

- (safe) HIGH PERFORMANCE in-~~plane~~kernel ~~packet processing~~ programmatic operations
- network tunneling
- custom iptables rules
- syscall filtering (mix of classic [for now] BPF for seccomp, eBPF for cgroups shenanigans)
- Reduce need for buggy kernel modules

Why eBPF?

- (safe) HIGH PERFORMANCE in-~~plane~~kernel ~~packet processing~~ programmatic operations
- network tunneling
- custom iptables rules
- syscall filtering (mix of classic [for now] BPF for seccomp, eBPF for cgroups shenanigans)
- Reduce need for ~~buggy~~ kernel modules

Why eBPF?

- (safe) HIGH PERFORMANCE in-~~plane~~kernel ~~packet processing~~ programmatic operations
- network tunneling
- custom iptables rules
- syscall filtering (mix of classic [for now] BPF for seccomp, eBPF for cgroups shenanigans)
- Reduce need for ~~buggy~~ kernel modules
- firewall subsystem with rules implemented entirely in eBPF

Why eBPF?

- (safe) HIGH PERFORMANCE in-~~plane~~kernel ~~packet processing~~ programmatic operations
- network tunneling
- custom iptables rules
- syscall filtering (mix of classic [for now] BPF for seccomp, eBPF for cgroups shenanigans)
- Reduce need for ~~buggy~~ kernel modules
- firewall subsystem with rules implemented entirely in eBPF

As more eBPF features have been added in newer kernel versions, the ”why” of eBPF has changed retroactively

Why eBPF? — OK, but really, why?

- eBPF is different things to different people
- Personally, we like being able to selectively instrument an entire OS without making it crawl
- The title of this talk *is* ”Kernel Tracing With eBPF” :)

Why eBPF? — OK, but really, why?

- eBPF is different things to different people
- Personally, we like being able to selectively instrument an entire OS without making it crawl
- The title of this talk *is* ”Kernel Tracing With eBPF” :)
- eBPF has the potential to give DTrace a run for its money

Why eBPF? — OK, but really, why?

- eBPF is different things to different people
- Personally, we like being able to selectively instrument an entire OS without making it crawl
- The title of this talk *is* ”Kernel Tracing With eBPF” :)
- eBPF has the potential to give DTrace a run for its money
- The power of DTrace is in its providers (event/data sources)
    - Linux will not likely gain such unified facilities

Why eBPF? — OK, but really, why?

- eBPF is different things to different people
- Personally, we like being able to selectively instrument an entire OS without making it crawl
- The title of this talk *is* ”Kernel Tracing With eBPF” :)
- eBPF has the potential to give DTrace a run for its money
- The power of DTrace is in its providers (event/data sources)
  - Linux will not likely gain such unified facilities
- eBPF is more programmatic, but lower level
  - It provides a base to build more complicated analysis tooling on

Why eBPF? — OK, but really, why?

- eBPF is different things to different people
- Personally, we like being able to selectively instrument an entire OS without making it crawl
- The title of this talk *is* ”Kernel Tracing With eBPF” :)
- eBPF has the potential to give DTrace a run for its money
- The power of DTrace is in its providers (event/data sources)
  - Linux will not likely gain such unified facilities
- eBPF is more programmatic, but lower level
  - It provides a base to build more complicated analysis tooling on
- DTrace is amazing at one-off human-driven system analysis
- But eBPF enables very efficient dynamic always-on whole system analysis

Let’s talk about tracing

Tracing — An Introduction

- ”Tracing” is a concept
- Wikipedia describes it as

  ”a specialized use of logging to record information about a program’s execution”
- Generally considered *developer-centric* logging
  - Often involves very low-level logging of very low-level information

Tracing — An Introduction

- ”Tracing” is a concept
- Wikipedia describes it as

  ”a specialized use of logging to record information about a program’s execution”
- Generally considered *developer-centric* logging
  - Often involves very low-level logging of very low-level information
- This distinction is unhelpful and misses the point

Tracing — Why is Tracing Useful?

Tracing — Why is Tracing Useful?

- It isn't (for us)

Tracing — Why is Tracing Useful?

- It isn’t (for us)
- What is useful is ”dynamic tracing”

Dynamic Tracing — An Introduction

- Two main kinds of dynamic tracing
  - Dynamically enabling/disabling existing logging functionality
  - Dynamically adding logging functionality that wasn’t there before

Dynamic Tracing — An Introduction

- Two main kinds of dynamic tracing
  - Dynamically enabling/disabling existing logging functionality
  - Dynamically adding logging functionality that wasn’t there before
- We mostly care about the latter

Dynamic Tracing — An Introduction

- Two main kinds of dynamic tracing
  - Dynamically enabling/disabling existing logging functionality
  - Dynamically adding logging functionality that wasn’t there before
- We mostly care about the latter
  - But the ”logging” isn’t really that important

Dynamic Tracing — An Introduction

- Two main kinds of dynamic tracing
  - Dynamically enabling/disabling existing logging functionality
  - Dynamically adding logging functionality that wasn’t there before
- We mostly care about the latter
  - But the ”logging” isn’t really that important
  - What’s important is the implementation and its capabilities

Dynamic Tracing — An Introduction

- Two main kinds of dynamic tracing
  - Dynamically enabling/disabling existing logging functionality
  - Dynamically adding logging functionality that wasn’t there before
- We mostly care about the latter
  - But the ”logging” isn’t really that important
  - What’s important is the implementation and its capabilities
- We don’t care about dynamic tracing as much as the *dynamic instrumentation* implementing it

Dynamic Tracing — An Introduction

- Two main kinds of dynamic tracing
  - Dynamically enabling/disabling existing logging functionality
  - Dynamically adding logging functionality that wasn't there before
- We mostly care about the latter
  - But the "logging" isn't really that important
  - What's important is the implementation and its capabilities
- We don't care about dynamic tracing as much as the *dynamic instrumentation* implementing it
- Two main kinds of dynamic instrumentation
  - Function hooking
  - Instruction instrumentation (assembly, bytecode, etc.)
- Depending on the instrumentation target, a function hooking API may be implemented through some amount of instruction modification/instrumentation

Instrumenting Linux With eBPF For Fun and Profit

Jeff Dileo
@chaosdatumz

Andy Olsen
@0lsen_

Linux Tracing — A Purposefully Over-Summarized History

- 2004: kprobes/kretprobes
- 2008: ftrace
- 2009: perf_events
- 2009: tracepoints
- 2012: uprobes
- 2015-present: eBPF tracing integration (Linux 4.1+)

Linux Tracing — A Purposefully Over-Summarized History

- 2004: kprobes/kretprobes
  - Injects jumps into function entry/exit points that go to hook code
  - If jumps can't safely be inserted, falls back to breakpoints and single-stepping from entry to exit
  - API originally exposed to kernel code/kernel modules
- 2008: ftrace
- 2009: perf_events
- 2009: tracepoints
- 2012: uprobes
- 2015-present: eBPF tracing integration (Linux 4.1+)

Linux Tracing — A Purposefully Over-Summarized History

- 2004: kprobes/kretprobes
- 2008: ftrace
  - Provides a filesystem-based userland API to perform various tracing/profiling
- 2009: perf_events
- 2009: tracepoints
- 2012: uprobes
- 2015-present: eBPF tracing integration (Linux 4.1+)

Linux Tracing — A Purposefully Over-Summarized History

- 2004: kprobes/kretprobes
- 2008: ftrace
- 2009: perf_events
  - Does a whole bunch of awesome profiling stuff outside the scope of this talk
- 2009: tracepoints
- 2012: uprobes
- 2015-present: eBPF tracing integration (Linux 4.1+)

Linux Tracing — A Purposefully Over-Summarized History

- 2004: kprobes/kretprobes
- 2008: ftrace
- 2009: perf_events
- 2009: tracepoints
  - Enable-able logging functions that pack log content into documented structs
- 2012: uprobes
- 2015-present: eBPF tracing integration (Linux 4.1+)

Linux Tracing — A Purposefully Over-Summarized History

- 2004: kprobes/kretprobes
- 2008: ftrace
- 2009: perf_events
- 2009: tracepoints
- 2012: uprobes
  - Essentially kprobes applied to userspace memory
- 2015-present: eBPF tracing integration (Linux 4.1+)

Linux Tracing — A Purposefully Over-Summarized History

- 2004: kprobes/kretprobes
- 2008: ftrace
- 2009: perf_events
- 2009: tracepoints
- 2012: uprobes
- 2015-present: eBPF tracing integration (Linux 4.1+)
  - Combined mecha super robot

eBPF Voltron

- eBPF is being integrated with many different kernel technologies, especially the tracing ones
- Core concepts:
  - Attach eBPF program to a data source using perf_events API or `bpf(2)`
  - Use perf_events ring buffer or memory-mapped eBPF maps as output
    - eBPF maps can also be updated from userspace to provide input

eBPF Voltron

- eBPF is being integrated with many different kernel technologies, especially the tracing ones
- Core concepts:
  - Attach eBPF program to a data source using perf_events API or `bpf(2)`
  - Use perf_events ring buffer or memory-mapped eBPF maps as output
    - eBPF maps can also be updated from userspace to provide input
- Sources:
  - k(ret)probes
  - u(ret)probes
  - tracepoints
  - raw tracepoints

eBPF Voltron — Source Attachment

- k(ret)probes (old):
  1. bpf(2) to create a kprobe eBPF program (BPF_PROG_LOAD)
  2. Use ftrace/tracefs API to register a k(ret)probe
  3. Read /id file from it to get kprobe ID
  4. perf_event_open(&attr, <pid>, -1, -1, PERF_FLAG_FD_CLOEXEC)
     - struct perf_event_attr attr;
     - attr.type = PERF_TYPE_TRACEPOINT;
     - attr.config = <kprobe_id>;
  5. ioctl(<perf_fd>, PERF_EVENT_IOC_SET_BPF, <bpf_fd>)
  6. ioctl(<perf_fd>, PERF_EVENT_IOC_ENABLE, 0)
- k(ret)probes (new):
  1. bpf(2) to create a kprobe eBPF program (BPF_PROG_LOAD)
  2. perf_event_open(&attr, <pid>, -1, -1, PERF_FLAG_FD_CLOEXEC)
     - attr.type = 6; // magic number
     - attr.kprobe_func = <addr of str>;
     - attr.probe_offset = <off>; // if attr.kprobe_func != NULL
     - attr.kprobe_addr = <addr>; // if attr.kprobe_func == NULL
  3. Follow steps 4-6 from above

eBPF Voltron — Source Attachment

- u(ret)probes (old/new):
  - Basically identical to the previous slide with minor modifications
- tracepoints
  - Basically identical to the old k(ret)probe attachment
- raw tracepoints
  1. `bpf(2)` to create a raw tracepoint eBPF program (`BPF_PROG_LOAD`)
  2. `bpf(2)` to attach BPF fd to tracepoint by name (`BPF_RAW_TRACEPOINT_OPEN`)

Using eBPF — How (Not) to eBPF

- Don’t write eBPF bytecode assembly by hand
    - It is hard
    - It is basically impossible to do anything more than simple arithmetic and a few comparisons
    - It is not well supported by glibc (not that anything modern is)

Using eBPF — How (Not) to eBPF

- Don’t write eBPF bytecode assembly by hand
  - It is hard
  - It is basically impossible to do anything more than simple arithmetic and a few comparisons
  - It is not well supported by glibc (not that anything modern is)
  - It is highly error prone

Using eBPF — How to eBPF

- Use bcc (BPF Compiler Collection)
  - `https://github.com/iovisor/bcc`
  - Framework for compiling C into eBPF (using LLVM APIs) and hooking it up to sources

Using eBPF — How to eBPF

- Use bcc (BPF Compiler Collection)
  - `https://github.com/iovisor/bcc`
  - Framework for compiling C into eBPF (using LLVM APIs) and hooking it up to sources
- This talk is not ”about” bcc, but it’s the only thing mature enough to suit our purposes
  - As with most modern and useful Linux things:
    - No official userland API other than syscalls
    - Syscall documentation is lacking/wrong
    - Multi-syscall operations are essentially undocumented
    - No support from glibc (everything is generally done with the `syscall()` wrapper)
    - One real consumer of the API, often with varying levels of documentation
    - Kernel APIs often written to support the one consumer, often by the same developers
    - ...
  - bcc is the only real option
  - Everything else either uses at least some of it as a library or cribs from their code

Building Tracing Tools With BCC

- Primarily a Python API, with underlying C/C++ layers to call lower level APIs
- Usually a whole tool is a single Python file
- eBPF C code is generally a Python string
- General structure of bcc-based tracers is the following:
    1. Python imports
    2. Large Python string containing eBPF C code, possibly using custom templating
    3. Argument parsing to codegen templated parts of the eBPF C code
    4. Python `ctypes` struct definitions for eBPF C defined types
    5. Userspace Python callback handlers for events generated by eBPF C
    6. BCC API calls to compile the C code, attach it to sources, and register event handlers

Building Tracing Tools With BCC

- Primarily a Python API, with underlying C/C++ layers to call lower level APIs
- Usually a whole tool is **a single Python file**
    - bcc doesn't handle C `#include ""`s super well
        - Can be done with special function kwargs
        - But need to specify the full path because the default base dir is weird
- eBPF C code is generally a Python string
- General structure of bcc-based tracers is the following:
    1. Python imports
    2. Large Python string containing eBPF C code, possibly using custom templating
    3. Argument parsing to codegen templated parts of the eBPF C code
    4. Python `ctypes` struct definitions for eBPF C defined types
    5. Userspace Python callback handlers for events generated by eBPF C
    6. BCC API calls to compile the C code, attach it to sources, and register event handlers

Let’s write some code!

```python
from bcc import BPF

program = """
#include <asm/ptrace.h>  // for struct pt_regs
#include <linux/types.h> // for mode_t

int kprobe__sys_open(struct pt_regs *ctx,
                     char __user* pathname, int flags, mode_t mode) {
  bpf_trace_printk("sys_open called.\\n");
  return 0;
}
"""

b = BPF(text=program)
b.trace_print()
```

```
$ sudo python code/3-hello-open-world-1.py



...
```

There’s no output! What went wrong?

glibc

```python
from bcc import BPF

program = """
#include <asm/ptrace.h>  // for struct pt_regs
#include <linux/types.h> // for mode_t

int kprobe__sys_open(struct pt_regs *ctx,
                     char __user* pathname, int flags, mode_t mode) {
  bpf_trace_printk("sys_open called.\\n");
  return 0;
}

int kprobe__sys_openat(struct pt_regs *ctx,
                       int dirfd, char __user* pathname, int flags, mode_t mode) {
  bpf_trace_printk("sys_openat called.\\n");
  return 0;
}
"""

b = BPF(text=program)
b.trace_print()
```

```
$ sudo python code/3-hello-open-world-2.py
     gnome-shell-13250 [001] .... 318129.936224: 0x00000001: sys_openat called.
     gnome-shell-13250 [001] .... 318130.022664: 0x00000001: sys_openat called.
         systemd-1     [000] .... 318130.193712: 0x00000001: sys_openat called.
 systemd-journal-339   [000] .... 318130.194966: 0x00000001: sys_openat called.
 systemd-journal-339   [000] .... 318130.194999: 0x00000001: sys_openat called.
 systemd-journal-339   [000] .... 318130.195317: 0x00000001: sys_openat called.
         systemd-1     [000] .... 318130.210087: 0x00000001: sys_openat called.
         systemd-1     [000] .... 318130.210151: 0x00000001: sys_openat called.
      irqbalance-676   [000] .... 319219.767122: 0x00000001: sys_openat called.
      irqbalance-676   [000] .... 319219.767449: 0x00000001: sys_openat called.
     gnome-shell-13250 [000] .... 319224.120910: 0x00000001: sys_openat called.
     gnome-shell-13250 [000] .... 319224.121005: 0x00000001: sys_openat called.
 gnome-control-c-19963 [001] .... 319227.287377: 0x00000001: sys_openat called.
      irqbalance-676   [000] .... 319229.760427: 0x00000001: sys_openat called.
      irqbalance-676   [000] .... 319229.760747: 0x00000001: sys_openat called.
             zsh-14892 [001] .... 319235.284734: 0x00000001: sys_openat called.
             zsh-14892 [001] .... 319235.284914: 0x00000001: sys_openat called.
             zsh-14892 [001] .... 319235.285157: 0x00000001: sys_openat called.
             zsh-14892 [001] .... 319235.285166: 0x00000001: sys_openat called.
...
```

Let’s generalize this code a bit...

```python
from bcc import BPF

program = """
#include <asm/ptrace.h>  // for struct pt_regs
#include <linux/types.h> // for mode_t

int kprobe__do_sys_open(struct pt_regs *ctx,
                        int dirfd, char __user* pathname, int flags, mode_t mode) {
  bpf_trace_printk("do_sys_open called: %s\\n", pathname);
  return 0;
}
"""

b = BPF(text=program)
b.trace_print()
```

```
$ sudo python code/3-hello-open-world-3.py
      irqbalance-676   [000] .... 319659.751235: 0x00000001: do_sys_open called: /proc/interrupts
      irqbalance-676   [000] .... 319659.751685: 0x00000001: do_sys_open called: /proc/stat
     gnome-shell-13250 [000] .... 319661.369193: 0x00000001: do_sys_open called: /proc/self/stat
         systemd-1     [000] .... 319668.190947: 0x00000001: do_sys_open called: /proc/33172/cgroup
         systemd-1     [000] .... 319668.193370: 0x00000001: do_sys_open called: /proc/664/cgroup
 systemd-journal-339   [001] .... 319668.194160: 0x00000001: do_sys_open called: /proc/679/comm
 systemd-journal-339   [001] .... 319668.194253: 0x00000001: do_sys_open called: /proc/679/cmdline
 systemd-journal-339   [001] .... 319668.194276: 0x00000001: do_sys_open called: /proc/679/status
 systemd-journal-339   [001] .... 319668.194319: 0x00000001: do_sys_open called: /proc/679/attr/current
 systemd-journal-339   [001] .... 319668.194335: 0x00000001: do_sys_open called: /proc/679/sessionid
 systemd-journal-339   [001] .... 319668.194349: 0x00000001: do_sys_open called: /proc/679/loginuid
 systemd-journal-339   [001] .... 319668.194363: 0x00000001: do_sys_open called: /proc/679/cgroup
 systemd-journal-339   [001] .... 319668.194406: 0x00000001: do_sys_open called: /run/systemd/units/log
      -extra-fields:dbus.service
 systemd-journal-339   [001] .... 319668.194449: 0x00000001: do_sys_open called: /var/log/journal/
      cd4d5eaa191c4be38b778d3203fb6bbb
 systemd-journal-339   [001] .... 319668.194801: 0x00000001: do_sys_open called: /run/log/journal/
      cd4d5eaa191c4be38b778d3203fb6bbb/system.journa
         systemd-1     [000] .... 319668.213534: 0x00000001: do_sys_open called: /proc/33172/comm
         systemd-1     [000] .... 319668.213615: 0x00000001: do_sys_open called: /proc/33172/comm
         systemd-1     [000] .... 319668.213634: 0x00000001: do_sys_open called: /proc/33172/cgroup
         systemd-1     [000] .... 319668.213687: 0x00000001: do_sys_open called: /sys/fs/cgroup/unified
              /system.slice/systemd-timedated.service/c
...
```

bpf_trace_printk() Considered Harmful

- `bpf_trace_printk()` is like ftrace
- One log buffer shared across the whole system

bpf_trace_printk() Considered Harmful

- `bpf_trace_printk()` is like ftrace
- One log buffer shared across the whole system
- Messages from different tracers will be received by each other

bpf_trace_printk() Considered Harmful

- `bpf_trace_printk()` is like ftrace
- One log buffer shared across the whole system
- Messages from different tracers will be received by each other
- eBPF programs get unloaded on owner process termination
- There is a race condition between termination, kprobe hits, and kprobe detach/eBPF unload

bpf_trace_printk() Considered Harmful

- `bpf_trace_printk()` is like ftrace
- One log buffer shared across the whole system
- Messages from different tracers will be received by each other
- eBPF programs get unloaded on owner process termination
- There is a race condition between termination, kprobe hits, and kprobe detach/eBPF unload
- Messages stick around until read
- The next process to open the log will get existing undelivered messages

```c
#include <asm/ptrace.h>   // for struct pt_regs
#include <bcc/proto.h>    // pulls in types.h
#include <linux/limits.h> // for PATH_MAX

BPF_PERF_OUTPUT(output);

typedef struct notify {
  uint64_t pid;
  uint8_t data[PATH_MAX];
} notify_t;
BPF_PERCPU_ARRAY(notify_array, notify_t, 1);

int kprobe__do_sys_open(struct pt_regs *ctx,
                        int dirfd, char __user* pathname, int flags, mode_t mode) {
  int i = 0;
  notify_t* n = notify_array.lookup(&i);
  if (!n) return 0;

  n->pid = (u32)(bpf_get_current_pid_tgid() >> 32);
  bpf_probe_read_str(&n->data[0], PATH_MAX, pathname);
  output.perf_submit(ctx, n, sizeof(notify_t));

  return 0;
}
```

```c
#include <asm/ptrace.h>
#include <bcc/proto.h>
#include <linux/limits.h>

BPF_PERF_OUTPUT(output); // creates a table for pushing custom events to userspace via ring buffer

typedef struct notify {
  uint64_t pid;
  uint8_t data[PATH_MAX];
} notify_t;
BPF_PERCPU_ARRAY(notify_array, notify_t, 1);

int kprobe__do_sys_open(struct pt_regs *ctx,
                        int dirfd, char __user* pathname, int flags, mode_t mode) {
  int i = 0;
  notify_t* n = notify_array.lookup(&i);
  if (!n) return 0;

  n->pid = (u32)(bpf_get_current_pid_tgid() >> 32);
  bpf_probe_read_str(&n->data[0], PATH_MAX, pathname);
  output.perf_submit(ctx, n, sizeof(notify_t));

  return 0;
}
```

```c
#include <asm/ptrace.h>
#include <bcc/proto.h>
#include <linux/limits.h>

BPF_PERF_OUTPUT(output);

typedef struct notify {
  uint64_t pid;
  uint8_t data[PATH_MAX]; // uint8_t to prevent ctypes from "optimizing" out copy of char[] in userspace
} notify_t;
BPF_PERCPU_ARRAY(notify_array, notify_t, 1); // creates a per-cpu TLS bpf table for off-stack scratch space
                                             // we need this b/c PATH_MAX is 4096 and the bpf stack 512 bytes
int kprobe__do_sys_open(struct pt_regs *ctx,
                        int dirfd, char __user* pathname, int flags, mode_t mode) {
  int i = 0;
  notify_t* n = notify_array.lookup(&i);
  if (!n) return 0;

  n->pid = (u32)(bpf_get_current_pid_tgid() >> 32);
  bpf_probe_read_str(&n->data[0], PATH_MAX, pathname);
  output.perf_submit(ctx, n, sizeof(notify_t));

  return 0;
}
```

```c
#include <asm/ptrace.h>
#include <bcc/proto.h>
#include <linux/limits.h>

BPF_PERF_OUTPUT(output);

typedef struct notify {
  uint64_t pid;
  uint8_t data[PATH_MAX];
} notify_t;
BPF_PERCPU_ARRAY(notify_array, notify_t, 1);

int kprobe__do_sys_open(struct pt_regs *ctx,
                        int dirfd, char __user* pathname, int flags, mode_t mode) {
  int i = 0; // key (array index) into our 1-element scratch-space table
  notify_t* n = notify_array.lookup(&i); // try to get slot for key
  if (!n) return 0; // if no slot found, bail

  n->pid = (u32)(bpf_get_current_pid_tgid() >> 32);
  bpf_probe_read_str(&n->data[0], PATH_MAX, pathname);
  output.perf_submit(ctx, n, sizeof(notify_t));

  return 0;
}
```

```c
#include <asm/ptrace.h>
#include <bcc/proto.h>
#include <linux/limits.h>

BPF_PERF_OUTPUT(output);

typedef struct notify {
  uint64_t pid;
  uint8_t data[PATH_MAX];
} notify_t;
BPF_PERCPU_ARRAY(notify_array, notify_t, 1);

int kprobe__do_sys_open(struct pt_regs *ctx,
                        int dirfd, char __user* pathname, int flags, mode_t mode) {
  int i = 0;
  notify_t* n = notify_array.lookup(&i);
  if (!n) return 0;

  n->pid = (u32)(bpf_get_current_pid_tgid() >> 32); // get pid of calling process from bpf helper
  bpf_probe_read_str(&n->data[0], PATH_MAX, pathname); // copy pathname into scratch space
  output.perf_submit(ctx, n, sizeof(notify_t));

  return 0;
}
```

```c
#include <asm/ptrace.h>
#include <bcc/proto.h>
#include <linux/limits.h>

BPF_PERF_OUTPUT(output);

typedef struct notify {
  uint64_t pid;
  uint8_t data[PATH_MAX];
} notify_t;
BPF_PERCPU_ARRAY(notify_array, notify_t, 1);

int kprobe__do_sys_open(struct pt_regs *ctx,
                        int dirfd, char __user* pathname, int flags, mode_t mode) {
  int i = 0;
  notify_t* n = notify_array.lookup(&i);
  if (!n) return 0;

  n->pid = (u32)(bpf_get_current_pid_tgid() >> 32);
  bpf_probe_read_str(&n->data[0], PATH_MAX, pathname);
  output.perf_submit(ctx, n, sizeof(notify_t)); // copy scratch space down to userspace code

  return 0;
}
```

```python
from __future__ import absolute_import, division, print_function, unicode_literals
import sys, ctypes
from bcc import BPF
text = """..."""

class notify_t(ctypes.Structure): # match layout of eBPF C's notify_t struct
  _fields_ = [("pid", ctypes.c_uint64),
              ("data", ctypes.c_uint8*4096),]

def handle_event(cpu, data, size):
  try:
    notify = ctypes.cast(data, ctypes.POINTER(notify_t)).contents
    data_s = ctypes.cast(notify.data, ctypes.c_char_p).value
    print("{}: {}".format(notify.pid, data_s))
  except KeyboardInterrupt:
    sys.exit(0)

b = BPF(text=text)
b["output"].open_perf_buffer(handle_event)

while True:
  try:
    b.kprobe_poll()
  except KeyboardInterrupt:
    sys.exit(0)
```

```python
from __future__ import absolute_import, division, print_function, unicode_literals
import sys, ctypes
from bcc import BPF
text = """..."""

class notify_t(ctypes.Structure):
  _fields_ = [("pid", ctypes.c_uint64),
              ("data", ctypes.c_uint8*4096),]

def handle_event(cpu, data, size): # handler called on receiving data from eBPF C `output.perf_submit()`
  try:
    notify = ctypes.cast(data, ctypes.POINTER(notify_t)).contents
    data_s = ctypes.cast(notify.data, ctypes.c_char_p).value
    print("{}: {}".format(notify.pid, data_s))
  except KeyboardInterrupt:
    sys.exit(0)

b = BPF(text=text)
b["output"].open_perf_buffer(handle_event) # register handler to eBPF C `BPF_PERF_OUTPUT(output);` table

while True:
  try:
    b.kprobe_poll()
  except KeyboardInterrupt:
    sys.exit(0)
```

```python
from __future__ import absolute_import, division, print_function, unicode_literals
import sys, ctypes
from bcc import BPF
text = """..."""

class notify_t(ctypes.Structure):
  _fields_ = [("pid", ctypes.c_uint64),
              ("data", ctypes.c_uint8*4096),]

def handle_event(cpu, data, size):
  try:
    notify = ctypes.cast(data, ctypes.POINTER(notify_t)).contents # cast raw byte pointer to notify_t
    data_s = ctypes.cast(notify.data, ctypes.c_char_p).value # cast buffer to NUL-terminated C string
    print("{}: {}".format(notify.pid, data_s))
  except KeyboardInterrupt:
    sys.exit(0)

b = BPF(text=text)
b["output"].open_perf_buffer(handle_event)

while True:
  try:
    b.kprobe_poll()
  except KeyboardInterrupt:
    sys.exit(0)
```

```python
from __future__ import absolute_import, division, print_function, unicode_literals
import sys, ctypes
from bcc import BPF
text = """..."""

class notify_t(ctypes.Structure):
  _fields_ = [("pid", ctypes.c_uint64),
              ("data", ctypes.c_uint8*4096),]

def handle_event(cpu, data, size):
  try:
    notify = ctypes.cast(data, ctypes.POINTER(notify_t)).contents
    data_s = ctypes.cast(notify.data, ctypes.c_char_p).value
    print("{}: {}".format(notify.pid, data_s))
  except KeyboardInterrupt:
    sys.exit(0)

b = BPF(text=text)
b["output"].open_perf_buffer(handle_event)

while True:
  try:
    b.kprobe_poll() # poll for perf events from kprobes, call event handlers for events
  except KeyboardInterrupt:
    sys.exit(0)
```

So how does all of this actually work?

```
bpf(BPF_MAP_CREATE, {map_type=BPF_MAP_TYPE_PERF_EVENT_ARRAY, key_size=4, value_size=4, max_entries=128,
                     map_flags=0, inner_map_fd=0, ...}, 72) = 3
bpf(BPF_MAP_CREATE, {map_type=BPF_MAP_TYPE_PERCPU_ARRAY, key_size=4, value_size=4104, max_entries=1,
                     map_flags=0, inner_map_fd=0, ...}, 72) = 4
...
bpf(BPF_PROG_LOAD, {prog_type=BPF_PROG_TYPE_KPROBE, insn_cnt=29, insns=0x7f04a0c697d0, license="GPL",
                    log_level=0, log_size=0, log_buf=0, kern_version=266002, prog_flags=0, ...}, 72) = 5
...
openat(AT_FDCWD, "/sys/kernel/debug/tracing/kprobe_events", O_WRONLY|O_APPEND) = 6
getpid()                                = 43676
write(6, "p:kprobes/p_do_sys_open_bcc_4367"..., 45) = 45
close(6)                                = 0
openat(AT_FDCWD, "/sys/kernel/debug/tracing/events/kprobes/p_do_sys_open_bcc_43676/id", O_RDONLY) = 6
read(6, "1982\n", 4096)                 = 5
close(6)                                = 0
perf_event_open({type=PERF_TYPE_TRACEPOINT, size=0 /* PERF_ATTR_SIZE_??? */, config=1982, ...},
                -1, 0, -1, PERF_FLAG_FD_CLOEXEC) = 6
ioctl(6, PERF_EVENT_IOC_SET_BPF, 0x5)   = 0
ioctl(6, PERF_EVENT_IOC_ENABLE, 0)      = 0
...
perf_event_open({type=PERF_TYPE_SOFTWARE, size=0, config=PERF_COUNT_SW_BPF_OUTPUT, ...},
                -1, 0, -1, PERF_FLAG_FD_CLOEXEC) = 8
ioctl(8, PERF_EVENT_IOC_ENABLE, 0)      = 0
bpf(BPF_MAP_UPDATE_ELEM, {map_fd=3, key=0x7f049aafa0a0, value=0x7f049aafae20, flags=BPF_ANY}, 72) = 0
perf_event_open({type=PERF_TYPE_SOFTWARE, size=0, config=PERF_COUNT_SW_BPF_OUTPUT, ...},
                -1, 1, -1, PERF_FLAG_FD_CLOEXEC) = 9
ioctl(9, PERF_EVENT_IOC_ENABLE, 0)      = 0
bpf(BPF_MAP_UPDATE_ELEM, {map_fd=3, key=0x7f049aafae20, value=0x7f049aafa0a0, flags=BPF_ANY}, 72) = 0
poll([{fd=9, events=POLLIN}, {fd=8, events=POLLIN}], 2, -1) = 1 ([{fd=9, revents=POLLIN}])
...
write(1, "13250: /proc/self/stat\n", 2313250: /proc/self/stat
) = 23
```

```c
#include <asm/ptrace.h>
#include <bcc/proto.h>
#include <linux/limits.h>

BPF_PERF_OUTPUT(output); // creates a table for pushing custom events to userspace via ring buffer

typedef struct notify {
  uint64_t pid;
  uint8_t data[PATH_MAX];
} notify_t;
BPF_PERCPU_ARRAY(notify_array, notify_t, 1);

int kprobe__do_sys_open(struct pt_regs *ctx,
                        int dirfd, char __user* pathname, int flags, mode_t mode) {
  int i = 0;
  notify_t* n = notify_array.lookup(&i);
  if (!n) return 0;

  n->pid = (u32)(bpf_get_current_pid_tgid() >> 32);
  bpf_probe_read_str(&n->data[0], PATH_MAX, pathname);
  output.perf_submit(ctx, n, sizeof(notify_t));

  return 0;
}
```

```c
// Table for pushing custom events to userspace via ring buffer
#define BPF_PERF_OUTPUT(_name) \
struct _name##_table_t { \
  int key; \
  u32 leaf; \
  /* map.perf_submit(ctx, data, data_size) */ \
  int (*perf_submit) (void *, void *, u32); \
  int (*perf_submit_skb) (void *, u32, void *, u32); \
  u32 max_entries; \
}; \
__attribute__((section("maps/perf_output"))) \
struct _name##_table_t _name = { .max_entries = 0 }
```

Listing 2: `bcc/src/cc/export/helpers.h`

BCC — Behind the Curtain

- The previous struct/instance is fake
- It is nothing more than fancy typing to please the first compiler pass
- All operations on it get replaced through LLVM-based codegen
- This is a common idiom in codegen-based APIs

```c
#include <asm/ptrace.h>
#include <bcc/proto.h>
#include <linux/limits.h>

BPF_PERF_OUTPUT(output);

typedef struct notify {
  uint64_t pid;
  uint8_t data[PATH_MAX];
} notify_t;
BPF_PERCPU_ARRAY(notify_array, notify_t, 1);

int kprobe__do_sys_open(struct pt_regs *ctx,
                        int dirfd, char __user* pathname, int flags, mode_t mode) {
  int i = 0;
  notify_t* n = notify_array.lookup(&i);
  if (!n) return 0;

  n->pid = (u32)(bpf_get_current_pid_tgid() >> 32);
  bpf_probe_read_str(&n->data[0], PATH_MAX, pathname);
  output.perf_submit(ctx, n, sizeof(notify_t)); // copy scratch space down to userspace code

  return 0;
}
```

```
} else if (memb_name == "perf_submit") {
  string name = Ref->getDecl()->getName();
  string arg0 = rewriter_.getRewrittenText(expansionRange(Call->getArg(0)->getSourceRange()));
  string args_other = rewriter_.getRewrittenText(expansionRange(SourceRange(GET_BEGINLOC(Call->getArg(1)),
                                                         GET_ENDLOC(Call->getArg(2)))));

  txt = "bpf_perf_event_output(" + arg0 + ", bpf_pseudo_fd(1, " + fd + ")";
  txt += ", CUR_CPU_IDENTIFIER, " + args_other + ")";
}
```

Listing 3: `bcc/src/cc/frontends/clang/b_frontend_action.cc`

BCC — Behind the Curtain

- The `bpf_perf_event_output()` eBPF helper when passed `CUR_CPU_IDENTIFIER` (really `BPF_F_CURRENT_CPU`) will pull a kernel-internal `struct perf_event*` out of the eBPF table (itself a `BPF_MAP_TYPE_PERF_EVENT_ARRAY`) using the current CPU as the index
- This works because the `BPF_MAP_UPDATE_ELEM` `bpf(2)` syscalls set index `0` and `1` with `perf_event` file descriptors

```
bpf(BPF_MAP_CREATE, {map_type=BPF_MAP_TYPE_PERF_EVENT_ARRAY, key_size=4, value_size=4, max_entries=128,
                     map_flags=0, inner_map_fd=0, ...}, 72) = 3
bpf(BPF_MAP_CREATE, {map_type=BPF_MAP_TYPE_PERCPU_ARRAY, key_size=4, value_size=4104, max_entries=1,
                     map_flags=0, inner_map_fd=0, ...}, 72) = 4
...
bpf(BPF_PROG_LOAD, {prog_type=BPF_PROG_TYPE_KPROBE, insn_cnt=29, insns=0x7f04a0c697d0, license="GPL",
                    log_level=0, log_size=0, log_buf=0, kern_version=266002, prog_flags=0, ...}, 72) = 5
...
openat(AT_FDCWD, "/sys/kernel/debug/tracing/kprobe_events", O_WRONLY|O_APPEND) = 6
getpid()                                = 43676
write(6, "p:kprobes/p_do_sys_open_bcc_4367"..., 45) = 45
close(6)                                = 0
openat(AT_FDCWD, "/sys/kernel/debug/tracing/events/kprobes/p_do_sys_open_bcc_43676/id", O_RDONLY) = 6
read(6, "1982\n", 4096)                 = 5
close(6)                                = 0
perf_event_open({type=PERF_TYPE_TRACEPOINT, size=0 /* PERF_ATTR_SIZE_??? */, config=1982, ...},
                -1, 0, -1, PERF_FLAG_FD_CLOEXEC) = 6
ioctl(6, PERF_EVENT_IOC_SET_BPF, 0x5)   = 0
ioctl(6, PERF_EVENT_IOC_ENABLE, 0)      = 0
...
perf_event_open({type=PERF_TYPE_SOFTWARE, size=0, config=PERF_COUNT_SW_BPF_OUTPUT, ...},
                -1, 0, -1, PERF_FLAG_FD_CLOEXEC) = 8
ioctl(8, PERF_EVENT_IOC_ENABLE, 0)      = 0
bpf(BPF_MAP_UPDATE_ELEM, {map_fd=3, key=0x7f049aafa0a0, value=0x7f049aafae20, flags=BPF_ANY}, 72) = 0
perf_event_open({type=PERF_TYPE_SOFTWARE, size=0, config=PERF_COUNT_SW_BPF_OUTPUT, ...},
                -1, 1, -1, PERF_FLAG_FD_CLOEXEC) = 9
ioctl(9, PERF_EVENT_IOC_ENABLE, 0)      = 0
bpf(BPF_MAP_UPDATE_ELEM, {map_fd=3, key=0x7f049aafae20, value=0x7f049aafa0a0, flags=BPF_ANY}, 72) = 0
poll([{fd=9, events=POLLIN}, {fd=8, events=POLLIN}], 2, -1) = 1 ([{fd=9, revents=POLLIN}])
...
write(1, "13250: /proc/self/stat\n", 2313250: /proc/self/stat
) = 23
```

And now for something different...

eBPF Validator Hell

- To make eBPF ”safe,” the Linux kernel validates all eBPF code before loading it
  - eBPF code is not allowed to ”loop” or jump backwards (to prevent infinite loops)

eBPF Validator Hell

- To make eBPF ”safe,” the Linux kernel validates all eBPF code before loading it
  - eBPF code is not allowed to ”loop” or jump backwards (to prevent infinite loops)
- But even if your ”code” doesn’t have loops, the validator may reject it

eBPF Validator Hell

- To make eBPF ”safe,” the Linux kernel validates all eBPF code before loading it
    - eBPF code is not allowed to ”loop” or jump backwards (to prevent infinite loops)
- But even if your ”code” doesn’t have loops, the validator may reject it
- Calls need to be static inline as jumping to a function and returning is considered a ”loop”

eBPF Validator Hell

- To make eBPF ”safe,” the Linux kernel validates all eBPF code before loading it
  - eBPF code is not allowed to ”loop” or jump backwards (to prevent infinite loops)
- But even if your ”code” doesn’t have loops, the validator may reject it
- Calls need to be static inline as jumping to a function and returning is considered a ”loop”
- Compiler optimizations are both a blessing and curse for eBPF code
  - Unrolling loops under some circumstances and adding them in others

eBPF Validator Hell

- To make eBPF ”safe,” the Linux kernel validates all eBPF code before loading it
  - eBPF code is not allowed to ”loop” or jump backwards (to prevent infinite loops)
- But even if your ”code” doesn’t have loops, the validator may reject it
- Calls need to be static inline as jumping to a function and returning is considered a ”loop”
- Compiler optimizations are both a blessing and curse for eBPF code
  - Unrolling loops under some circumstances and adding them in others
- The validator also validates helper calls to ensure they are passed ”safe” arguments

eBPF Validator Hell

- To make eBPF ”safe,” the Linux kernel validates all eBPF code before loading it
  - eBPF code is not allowed to ”loop” or jump backwards (to prevent infinite loops)
- But even if your ”code” doesn’t have loops, the validator may reject it
- Calls need to be static inline as jumping to a function and returning is considered a ”loop”
- Compiler optimizations are both a blessing and curse for eBPF code
  - Unrolling loops under some circumstances and adding them in others
- The validator also validates helper calls to ensure they are passed ”safe” arguments
- This ”logic” is often not thorough enough to properly determine value bounds

eBPF Validator Hell

- To make eBPF ”safe,” the Linux kernel validates all eBPF code before loading it
  - eBPF code is not allowed to ”loop” or jump backwards (to prevent infinite loops)
- But even if your ”code” doesn’t have loops, the validator may reject it
- Calls need to be static inline as jumping to a function and returning is considered a ”loop”
- Compiler optimizations are both a blessing and curse for eBPF code
  - Unrolling loops under some circumstances and adding them in others
- The validator also validates helper calls to ensure they are passed ”safe” arguments
- This ”logic” is often not thorough enough to properly determine value bounds
- Trying to make them obvious is hard as the optimizer will often optimize out ”superfluous” checks

eBPF Validator Hell

- To make eBPF ”safe,” the Linux kernel validates all eBPF code before loading it
  - eBPF code is not allowed to ”loop” or jump backwards (to prevent infinite loops)
- But even if your ”code” doesn’t have loops, the validator may reject it
- Calls need to be static inline as jumping to a function and returning is considered a ”loop”
- Compiler optimizations are both a blessing and curse for eBPF code
  - Unrolling loops under some circumstances and adding them in others
- The validator also validates helper calls to ensure they are passed ”safe” arguments
- This ”logic” is often not thorough enough to properly determine value bounds
- Trying to make them obvious is hard as the optimizer will often optimize out ”superfluous” checks
- Additionally, updating BCC (or the Linux kernel) may potentially result in the validator rejecting once working eBPF C

Some validator errors are downright spooky

We have seen code be rejected or accepted
based on whether a function returned a bool or a size_t (0 or 1)

We have seen code be rejected or accepted

based on whether a function returned a bool or a size_t (0 or 1)
that was being stored in a uint8_t

Surviving eBPF Validator Hell — Correcting the Validator

- At one point, we got really mad at the validator rejecting correct code
- So we wrote a kernel module to neuter its checks

Surviving eBPF Validator Hell — Correcting the Validator

- At one point, we got really mad at the validator rejecting correct code
- So we wrote a kernel module to neuter its checks
- It turned out that the validator is poorly written and tightly coupled to the interpreter

Surviving eBPF Validator Hell — Correcting the Validator

- At one point, we got really mad at the validator rejecting correct code
- So we wrote a kernel module to neuter its checks
- It turned out that the validator is poorly written and tightly coupled to the interpreter
- You can’t skip the verifier because they also tweak and configure the eBPF program

Surviving eBPF Validator Hell — Correcting the Validator

- At one point, we got really mad at the validator rejecting correct code
- So we wrote a kernel module to neuter its checks
- It turned out that the validator is poorly written and tightly coupled to the interpreter
- You can’t skip the verifier because they also tweak and configure the eBPF program
- Instead, you need surgical hooks into it that skip certain checks and set fake ”safe” bounds

Surviving eBPF Validator Hell — yolo-ebpf

- PoC kernel module with a custom function hooking implementation that disables a number of eBPF validator checks
- Caveats:
  - x86_64-only
  - It probably doesn't work with current kernel versions
  - Unsafe eBPF will potentially crash your kernel
- We'll be making the code available anyway to prove a point
- Please don't use this code in production

Surviving eBPF Validator Hell — Tips and Tricks

- Initialize your memory
  - If you put a struct on the stack and fill it in, you may not be able to `perf_submit` it to userspace
  - The validator doesn't like when you try to send uninitialized memory to userspace, including that of padding
  - Eliminate uninitialized padding:
    - By carefully organizing your struct fields
    - By increasing/decreasing the size of struct fields
    - By adding padding fields (or unions) and initializing them
    - By clobbering it with `0`s
    - With `__attribute__((__packed__))`

Surviving eBPF Validator Hell — Tips and Tricks

- Initialize your memory
- Loop elimination
  - You will quickly find that you cann’t even ‘memset(3)‘ among other things
  - Unroll all loops

```
#pragma unroll
for (size_t i=0; i < sizeof(arr); i++) {
  arr[i] = 0;
}
```

  - Inline all calls

```
static inline void foo() {
  // do stuff
}
```

Surviving eBPF Validator Hell — Tips and Tricks

- Initialize your memory
- Loop elimination
- Reimplement kernel code in eBPF valid ways
  - bcc tries to codegen dereferences of non-eBPF memory region pointers into `bpf_probe_read()` calls
  - It often has problems with nested scopes and chained field accesses and fails to convert such code
  - A lot of static inline kernel functions run afoul of the second
  - Due to this, they must often be re-implemented with manual `bpf_probe_read()` calls

Surviving eBPF Validator Hell — Tips and Tricks

- Initialize your memory
- Loop elimination
- Reimplement kernel code in eBPF valid ways
- Ratcheting
  - If you need to implement a ring buffer, you will need logic to wrap the index
  - The validator does not like explicit cases that do this wrap, even if also checked in `default` case
  - Do it only in the `default` case

```
u32 pos = UINT32_MAX;
int key = 0;
sync = sync_buf.lookup(&key);
if (!sync) return 0;

pos = 0;
switch (sync->next) {
  case 0: {
    pos = 0;
    sync->next = 1;
    break;
  };
  case 1: {
    pos = 1;
    sync->next = 2;
    break;
  };
  default: {
    pos = 0;
    sync->next = 1;
  }
}
```

Surviving eBPF Validator Hell — Tips and Tricks

- Initialize your memory
- Loop elimination
- Reimplement kernel code in eBPF valid ways
- Ratcheting
- Dynamic structure parsing
  - Lots of kernel data structures are dynamically sized and structured without using C arrays
  - Best bet is to do a lot of loop unrolling of inlined steps to extract and process data
  - Most important is to detect remaining data that could not be processed due to eBPF limitations

Surviving eBPF Validator Hell — Tips and Tricks

- Initialize your memory
- Loop elimination
- Reimplement kernel code in eBPF valid ways
- Ratcheting
- Dynamic structure parsing
- Static data structures and algorithms
  - Not really feasible to perform nested comparison operations in eBPF code (e.g. ”is value in set?”)
  - Sometimes this can be worked around by using eBPF map operations to implement comparisons
  - Best bet is to statically codegen the C for complete structure walk for algorithm

Surviving eBPF Validator Hell — Tips and Tricks

- Initialize your memory
- Loop elimination
- Reimplement kernel code in eBPF valid ways
- Ratcheting
- Dynamic structure parsing
- Static data structures and algorithms
- Dynamic length byte copying
  - eBPF validator often fails to ascertain variable bounds
  - One pain point is attempting to use an externally sourced length value with `bpf_probe_read()`
  - Explicit checks often get optimized out
  - We've found the following code works, seemingly because using static inline functions prevents certain compiler assumptions

```
static inline
void copy_into_entry_buffer(data_t* entry,
                            size_t const len,
                            char* base,
                            u8 volatile* trunc) {
  int l = (int)len;
  if (l < 0) {
    l = 0;
  }
  if (l >= BUFFER_SIZE) {
    *trunc = 1;
  }
  if (l >= BUFFER_SIZE) {
    // the `- 1` is no longer needed with
    // current bcc on recent kernels
    l = BUFFER_SIZE - 1;
  }
  bpf_probe_read(entry->buffer, l, base);
}
```

Surviving eBPF Validator Hell — Tips and Tricks

- Initialize your memory
- Loop elimination
- Reimplement kernel code in eBPF valid ways
- Ratcheting
- Dynamic structure parsing
- Static data structures and algorithms
- Dynamic length byte copying
- Enable debug output and know why your code works when it shouldn’t
  - bcc can dump out eBPF bytecode annotated with source lines
  - Reading through it when errors occur (or not) can be very helpful
  - Often, code is not itself eBPF friendly, but optimized into a compliant form
  - But adding new code may break compiler assertions needed to optimize
  - So a small change can cause cascading changes that anger the validator

Good luck!

Defensive eBPF?

- Can eBPF be used for defense?

Defensive eBPF?

- Can eBPF be used for defense?
- Why not?
  - eBPF is fast, supposedly 10x faster than auditd
  - We can improve the state of auditing the entire system using just eBPF

Defensive eBPF?

- Can eBPF be used for defense?
- Why not?
  - eBPF is fast, supposedly 10x faster than auditd
  - We can improve the state of auditing the entire system using just eBPF
- What could go wrong? ;)

Defensive eBPF?

- Can eBPF be used for defense?
- Why not?
  - eBPF is fast, supposedly 10x faster than auditd
  - We can improve the state of auditing the entire system using just eBPF
- What could go wrong? ;)
- Let’s give this a try

Defensive eBPF?

- What does security monitoring software do?
  - Watches everything
    - program executions
    - file accesses
    - network traffic
    - administrative operations

Defensive eBPF?

- What does security monitoring software do?
  - Watches everything
    - program executions
    - file accesses
    - network traffic
    - administrative operations
- eBPF kprobes can do all of these things

Defensive eBPF?

- Why would eBPF be good for this?
- Tracing eBPF programs can see all the things
- They can hook into any kernel function
- Observe all user and kernel space memory
- And much more

Defensive eBPF? — Loop-Free Security Monitoring

- Let's implement some trivial security monitoring tasks using eBPF
- To begin, let's watch for file executions from nonstandard directories
  - For simplicity, we'll just hook the `execve(2)` syscall
  - We'll also ignore `mmap(2)` (used for shared libraries)

Defensive eBPF? — Loop-Free Security Monitoring

- Let's implement some trivial security monitoring tasks using eBPF
- To begin, let's watch for file executions from nonstandard directories
  - For simplicity, we'll just hook the `execve(2)` syscall
  - We'll also ignore `mmap(2)` (used for shared libraries)

```
from bcc import BPF
program = """
int kprobe__sys_execve(struct pt_regs *ctx){
  bpf_trace_printk("execve called.\\n");
  return 0;
}
"""
b = BPF(text=program)
b.trace_print()
```

Defensive eBPF? — An attempt at executable whitelisting

- Let's compare the supplied file path against standard directories
- Because of all the issues with eBPF's limitations, we will just process a static number of bytes
- For example, we will start by comparing the first four bytes of the path
    - compare against `/opt`, `/bin`, `/sbi`, `/usr`
    - If it starts with `/usr` we'll continue checking the path
        - It could be `/usr/bin`, `/usr/sbin`, `/usr/local/sbin`, `/usr/local/bin`
    - We could check the path like this to only do processing as we need to
- In the following example, we're only checking against `/bin` to keep it super simple

```python
from bcc import BPF
prog = """
#include <uapi/linux/ptrace.h>
#include <linux/sched.h>
#include <linux/fs.h>
int kprobe__sys_execve(struct pt_regs *ctx, const char __user *filename){
  char bin[] = "/bin";
  #pragma unroll
  for (int i = 0; i < 4; i++)
    if(bin[i] != filename[i]){
      bpf_trace_printk("exec outside /bin\\n");
      return 0;
    }
  return 0;
}
"""
b = BPF(text=prog)
b.trace_print()
```

Defensive eBPF? — An attempt at executable whitelisting

- Can we detect unusual `execve(2)` syscalls from a web application?
- Let's imagine we have a simple web app
    - A wrapper around `ping`
    - It takes in an IP address from user input and runs `ping` on it
        - What could go wrong? ;)
    - We want to know if it's executing anything other than the `ping` binary
    - For simplicity, it does not `fork(2)` before `execve(2)` as the fork-tracking logic is a bit complicated

```c
#include <uapi/linux/ptrace.h>
int kprobe__sys_execve(struct pt_regs *ctx, const char __user *filename){
  size_t pid = (u32)(bpf_get_current_pid_tgid() >> 32);
  #ifdef PID
    if(pid != PID)
      return 0;
  #endif
  char tmp[400];
  int length = bpf_probe_read_str(&tmp[0], 400, filename);
  char ping[] = "/bin/ping";
  if(length != 8){
    bpf_trace_printk("exec of %s\\n", filename);
    return 0;
  }
  #pragma unroll
  for (int i = 0; i < 8; i++)
    if(ping[i] != filename[i]){
      bpf_trace_printk("exec of %s\\n", filename);
      return 0;
    }
  return 0;
}
```

Defensive eBPF? — Loop-Free Security Monitoring

- We are now monitoring file executions
- Next we'll watch for file opens from a specific directory
    - This time we'll hook the `open(2)` syscall

Defensive eBPF? — Loop-Free Security Monitoring

- We are now monitoring file executions
- Next we'll watch for file opens from a specific directory
  - This time we'll hook the `open(2)` syscall

```
from bcc import BPF
program = """
int kprobe__do_sys_open(struct pt_regs *ctx){
  bpf_trace_printk("sys_open called.\\n");
  return 0;
}
"""
b = BPF(text=program)
b.trace_print()
```

Defensive eBPF? — An attempt at file monitoring

- How about we try to detect when a process `open(2)`s a file in `/root` ?
  - Let's compare the file path prefix to `/root`
  - We'll use the filename parameter of `open(2)`
  - Again, we use an unrolled loop to check the first several (5) bytes

```python
from bcc import BPF
prog = """
#include <uapi/linux/ptrace.h>
int kprobe__do_sys_open(struct pt_regs *ctx, int dfd, const char __user *filename){
  char root[] = "/root";
  #pragma unroll
  for(int i = 0; i < 5; i++)
    if(root[i] != filename[i])
      return 0;
  bpf_trace_printk("attempted access: %s\\n", filename);
  return 0;
}
"""
b = BPF(text=prog)
b.trace_print()
```

We have a confession to make

Defensive eBPF — Security-Free Security Monitoring

- All of the previous examples are insecure

Defensive eBPF — Security-Free Security Monitoring

- All of the previous examples are **dangerously** insecure

eBPF Gotchas

- Just because eBPF cannot crash the kernel does not mean that it is safe
- Its limitations in fact make it harder to write secure eBPF code

eBPF Gotchas — Race Conditions

- Time-of-Check-to-Time-of-Use (TOCTTOU)
  - A common vulnerability in kernel code and anything using kprobes
  - Exacerbated by eBPF limitations

eBPF Gotchas — Race Conditions

- Time-of-Check-to-Time-of-Use (TOCTTOU)
  - A common vulnerability in kernel code and anything using kprobes
  - Exacerbated by eBPF limitations
- If you kprobe a syscall
  - User-supplied data you process may change by the time the kernel copies it to *do* the syscall

eBPF Gotchas — Race Conditions

- It’s relatively easy to test for
- Start with a two-thread program
  - First thread repeatedly copies two different filepaths into one char array
  - Second thread repeatedly calls `open(2)` on that char array
- We then kprobe the `open(2)` syscall and the `getname_flags()` internal kernel function
- Then compare the two values obtained from each kprobe

```
a.out-5418  [001] d...  4078.020804: 0x00000001: do_sys_open: /tmp/rupergood
a.out-5418  [001] d...  4078.020805: 0x00000001: getname_flags: /tmp/realrgood
a.out-5418  [001] d...  4084.021083: 0x00000001: NOMATCH
a.out-5418  [001] d...  4084.021088: 0x00000001: do_sys_open: /tmp/supelybad
a.out-5418  [001] d...  4084.021089: 0x00000001: getname_flags: /tmp/reaerybad
a.out-5418  [001] d...  4084.021089: 0x00000001: NOMATCH
a.out-5418  [001] d...  4084.021090: 0x00000001: do_sys_open: /tmp/supelybad
a.out-5418  [001] d...  4084.021091: 0x00000001: getname_flags: /tmp/reaerybad
a.out-5418  [001] d...  4084.021091: 0x00000001: NOMATCH
a.out-5418  [001] d...  4084.021092: 0x00000001: do_sys_open: /tmp/supelybad
a.out-5418  [001] d...  4084.021093: 0x00000001: getname_flags: /tmp/reaerybad
a.out-5418  [001] d...  4084.021093: 0x00000001: NOMATCH
a.out-5418  [001] d...  4084.021094: 0x00000001: do_sys_open: /tmp/supelybad
a.out-5418  [001] d...  4084.021095: 0x00000001: getname_flags: /tmp/reaerybad
a.out-5418  [001] d...  4088.021279: 0x00000001: NOMATCH
a.out-5418  [001] d...  4088.021284: 0x00000001: do_sys_open: /tmp/supergood
a.out-5418  [001] d...  4088.021285: 0x00000001: getname_flags: /tmp/reallgood
```

eBPF Gotchas — Race Conditions

- How do we avoid this problem?

eBPF Gotchas — Race Conditions

- How do we avoid this problem?
- Hook internal kernel functions rather than syscalls
- Preferably a spot where desired value is already copied into kernel memory
- e.g. `sys_execve` vs. `do_execveat_common.isra.34`
- Alternatively, you use an LSM hook function (e.g. `security_bprm_set_creds`)

eBPF Gotchas — File Path Mishandling

- File paths, much like URIs, are slightly complicated
  - If you don't carefully validate them, you might end up in trouble
- Let's rewind to our IDS/endpoint security example
- What didn't we take into account?

eBPF Gotchas — File Path Mishandling

- We didn’t take into account how filenames work on Unix
- For example, what happens if the file isn’t accessed via the absolute path?

eBPF Gotchas — File Path Mishandling

- We didn't take into account how filenames work on Unix
- For example, what happens if the file isn't accessed via the absolute path?
  - An `open(2)` from inside the directory?

eBPF Gotchas — File Path Mishandling

- We didn't take into account how filenames work on Unix
- For example, what happens if the file isn't accessed via the absolute path?
  - An `open(2)` from inside the directory?
  - An `open(2)` on `../../../root/<name>`?

eBPF Gotchas — File Path Mishandling

- We didn't take into account how filenames work on Unix
- For example, what happens if the file isn't accessed via the absolute path?
  - An `open(2)` from inside the directory?
  - An `open(2)` on `../../../root/<name>`?
  - An `execve(2)` on `/bin/../tmp/foo`?

eBPF Gotchas — File Path Mishandling

- We didn't take into account how filenames work on Unix
- For example, what happens if the file isn't accessed via the absolute path?
  - An `open(2)` from inside the directory?
  - An `open(2)` on `../../../root/<name>`?
  - An `execve(2)` on `/bin/../tmp/foo`?
  - An `open(2)` on a symlink in `/tmp`?
- How can we fix those issues?

eBPF Gotchas — File Path Mishandling

- Things we could try:
    - Compare value against a known set
    - Attempt to canonicalize the path

eBPF Gotchas — File Path Mishandling

- Things we could try:
  - Compare value against a known set
  - Attempt to canonicalize the path
    - Linux's internal `struct file` and `struct path` are complicated to parse from eBPF
    - This adds to the amount of work eBPF has to do
    - It may not be even be possible to fully follow the object to recreate the path

eBPF Gotchas — File Path Mishandling

- Things we could try:
  - Compare value against a known set
  - Attempt to canonicalize the path
    - Linux's internal `struct file` and `struct path` are complicated to parse from eBPF
    - This adds to the amount of work eBPF has to do
    - It may not be even be possible to fully follow the object to recreate the path
  - Try to find an internal function that has access to an absolute path?
    - For example, the `security_bprm_set_creds` LSM hook

eBPF Gotchas — File Path Mishandling

- Things we could try:
  - Compare value against a known set
  - Attempt to canonicalize the path
    - Linux's internal `struct file` and `struct path` are complicated to parse from eBPF
    - This adds to the amount of work eBPF has to do
    - It may not be even be possible to fully follow the object to recreate the path
  - Try to find an internal function that has access to an absolute path?
    - For example, the `security_bprm_set_creds` LSM hook
    - This won't work
    - The path string it receives is the same one from the user (i.e. not canonical, nor absolute)
    - We would still need to parse the structs

eBPF Gotchas — Parsing Externally-Supplied Binary Data

- bcc has example code to use eBPF to do network monitoring

eBPF Gotchas — Parsing Externally-Supplied Binary Data

- bcc has example code to use eBPF to do network monitoring
- We found that it didn't properly calculate IP header offsets
  - Specifically, it didn't account for the fact that TCP options are variable-length

eBPF Gotchas — Parsing Externally-Supplied Binary Data

- bcc has example code to use eBPF to do network monitoring
- We found that it didn’t properly calculate IP header offsets
  - Specifically, it didn’t account for the fact that TCP options are variable-length
- It was possible to spoof a TCP header in the options and bypass the checks it performed

eBPF Gotchas — Parsing Externally-Supplied Binary Data

- bcc has example code to use eBPF to do network monitoring
- We found that it didn’t properly calculate IP header offsets
    - Specifically, it didn’t account for the fact that TCP options are variable-length
- It was possible to spoof a TCP header in the options and bypass the checks it performed
- So we sent them a PoC

eBPF Gotchas — Parsing Externally-Supplied Binary Data

- bcc has example code to use eBPF to do network monitoring
- We found that it didn't properly calculate IP header offsets
  - Specifically, it didn't account for the fact that TCP options are variable-length
- It was possible to spoof a TCP header in the options and bypass the checks it performed
- So we sent them a PoC
- and a patch :)
  - https://github.com/iovisor/bcc/commit/3d9b687

```diff
diff --git a/examples/networking/http_filter/http-parse-complete.c \
  b/examples/networking/http_filter/http-parse-complete.c PYZbs
index 61bb0f0a3..dff16b940 100644
--- a/examples/networking/http_filter/http-parse-complete.c
+++ b/examples/networking/http_filter/http-parse-complete.c
@@ -56,6 +56,19 @@ int http_filter(struct __sk_buff *skb) {
         struct Key         key;
         struct Leaf zero = {0};

+        //calculate ip header length
+        //value to multiply * 4
+        //e.g. ip->hlen = 5 ; IP Header Length = 5 x 4 byte = 20 byte
+        ip_header_length = ip->hlen << 2;    //SHL 2 -> *4 multiply
+
+        //check ip header length against minimum
+        if (ip_header_length < sizeof(*ip)) {
+                goto DROP;
+        }
+
+        //shift cursor forward for dynamic ip header size
+        void *_ = cursor_advance(cursor, (ip_header_length-sizeof(*ip)));
+
         struct tcp_t *tcp = cursor_advance(cursor, sizeof(*tcp));

         //retrieve ip src/dest and port src/dest of current packet
```

eBPF Gotchas — Assuming Userspace Isn’t Evil

- In general, values obtained from untrusted places (i.e. userspace) require strict validation

eBPF Gotchas — Assuming Userspace Isn't Evil

- In general, values obtained from untrusted places (i.e. userspace) require strict validation
- eBPF does not have a `copy_from_user()` helper function

eBPF Gotchas — Assuming Userspace Isn't Evil

- In general, values obtained from untrusted places (i.e. userspace) require strict validation
- eBPF does not have a `copy_from_user()` helper function
- If you blindly run `bpf_probe_read()` on a user-supplied pointer
  - you may be tricked into reading kernel memory

eBPF Gotchas — Assuming Userspace Isn’t Evil

- In general, values obtained from untrusted places (i.e. userspace) require strict validation
- eBPF does not have a `copy_from_user()` helper function
- If you blindly run `bpf_probe_read()` on a user-supplied pointer
  - you may be tricked into reading kernel memory
- Instead, you have to manually verify pointers
- This can be done by comparing against `((struct task_struct*)bpf_get_current_task())->mm->highest_vm_end`
  - However, this will need to be broken up or the eBPF validator will reject it

Defensive eBPF?

- Can eBPF be used for defense?
- Why not?

Defensive eBPF?

- Can eBPF be used for defense?
- ~~Why~~ **not**~~?~~ **directly**
- eBPF’s limitations make it hard to use securely in general, let alone as a security mechanism

Defensive eBPF?

- Can eBPF be used for defense?
- ~~Why~~ **not**~~?~~ **directly**
- eBPF's limitations make it hard to use securely in general, let alone as a security mechanism
- Instead, eBPF is much more useful for tracking data as it flows through the system

unixdump

- `tcpdump` for Unix domain sockets
- Originally created to reverse engineer `ptrace(2)`ing processes (e.g. Frida)
- Demonstrates our successful fight against eBPF validator
- Features:
  - Captures full streams
  - Captures ancillary data messages (e.g. passed file descriptors)
  - Filter/exclude by PID or socket path
  - Full support for abstract namespace, including binary ”paths”
- Link at end of slides :)

unixdump

- Retrieves `msghdr` buffer contents and metadata from `unix_stream_sendmsg` and `unix_dgram_sendmsg`
- Uses a custom ring buffer to share data with userspace while limiting byte copies
- Uses python to generate C code dynamically
- CLI arguments to tweak C array sizes

unixdump — Code Generation

- Python is used to generate eBPF C code
- This allows us to tweak the eBPF program at ”runtime” using defines and ifdefs
  - Ring buffer size, pids to exclude, `sun_path` to filter on
  - Increases performance by reducing the amount of events receiving heavier processing
- This also helps to get around loop restriction
  - Can’t loop through an array of PIDs so we codegen a static C BST lookup

```c
// generated by $ unixdump -x 1 2 3
static inline bool is_excluded_pid(u32 needle) {
  if (needle == 2) {
    return true;
  }
  if (needle < 2) {
    if (needle == 1) {
      return true;
    }
    return false;
  } else {
    if (needle == 3) {
      return true;
    }
    return false;
  }
}
```

unixdump — Code Generation

- We use another percpu array of size 1 to store the current ring buffer slot
- We can’t loop, so we generate a ratcheting switch statement

```python
def gen_ratchet_switch(sz):
  preamble = '''switch (sync->next) {
  '''
  entry_template = '''
    case {}: {{
      nxt = {};
      sync->next = {};
      break;
    }};
  '''
  end = '''
    default: {
      nxt = 0;
      sync->next = 1;
    }
  }
  '''
  out = ""
  out += preamble
  for i in range(sz):
    out += entry_template.format(i, i, i+1)
  out += end
  return out
```

unixdump — Event Notification

- The ring buffer is an eBPF percpu array mapped to userspace
- It holds large structs we fill with stream content
- The structs also have an in-use status field
- We check the in-use flag is cleared in eBPF, set it, and notify userspace
- Userspace checks that the flag is set, processes the data, and clears the flag
- This prevents race conditions due to async updating of kernel-userspace mapped pages

If eBPF isn't that good at defense, what else can we use it for?

Let's talk about offense

Offensive eBPF

- Let's assume someone bad gets some privileges on a modern Linux system
  - E.g. `CAP_SYS_ADMIN` in a container (it's more common than you might think)
- What could they do with eBPF?

Offensive eBPF

- Let's assume someone bad gets some privileges on a modern Linux system
  - E.g. `CAP_SYS_ADMIN` in a container (it's more common than you might think)
- What could they do with eBPF?
- A lot actually

Offensive eBPF

- Let's assume someone bad gets some privileges on a modern Linux system
    - E.g. `CAP_SYS_ADMIN` in a container (it's more common than you might think)
- What could they do with eBPF?
- A lot actually
- Tracing eBPF programs (kprobes, uprobes, tracepoints, raw tracepoints) can see everything

Offensive eBPF

- Let's assume someone bad gets some privileges on a modern Linux system
  - E.g. `CAP_SYS_ADMIN` in a container (it's more common than you might think)
- What could they do with eBPF?
- A lot actually
- Tracing eBPF programs (kprobes, uprobes, tracepoints, raw tracepoints) can see everything
- They can also write userspace memory

Offensive eBPF

- Let’s assume someone bad gets some privileges on a modern Linux system
  - E.g. `CAP_SYS_ADMIN` in a container (it’s more common than you might think)
- What could they do with eBPF?
- A lot actually
- Tracing eBPF programs (kprobes, uprobes, tracepoints, raw tracepoints) can see everything
- ***THEY CAN ALSO WRITE USERSPACE MEMORY***

Offensive eBPF — The Rootkit Principle

- `bpf_probe_write_user()`
  - Intended for use ”to debug, divert, and manipulate execution of semi-cooperative processes”
  - Enables writing to *writable* userspace memory
    - ~~Text~~
    - Stack
    - Heap
    - Static data
- Is there anything useful in those memory regions?

Offensive eBPF — The Rootkit Principle

- `bpf_probe_write_user()`
  - Intended for use ”to debug, divert, and manipulate execution of semi-cooperative processes”
  - Enables writing to *writable* userspace memory
    - ~~Text~~
    - Stack
    - Heap
    - Static data
- Is there anything useful in those memory regions?
- Buffers for reading/writing data through syscalls

Offensive eBPF — The Rootkit Principle

- `bpf_probe_write_user()`
  - Intended for use ”to debug, divert, and manipulate execution of semi-cooperative processes”
  - Enables writing to *writable* userspace memory
    - ~~Text~~
    - Stack
    - Heap
    - Static data
- Is there anything useful in those memory regions?
- Buffers for reading/writing data through syscalls
- What if we intercepted `read(2)`s on a sensitive file descriptor

Offensive eBPF — The Rootkit Principle

- `bpf_probe_write_user()`
  - Intended for use ”to debug, divert, and manipulate execution of semi-cooperative processes”
  - Enables writing to *writable* userspace memory
    - ~~Text~~
    - Stack
    - Heap
    - Static data
- Is there anything useful in those memory regions?
- Buffers for reading/writing data through syscalls
- What if we intercepted `read(2)`s on a sensitive file descriptor
  - That is used by a privileged process outside of the container?

Spoofing cron jobs with Conjob

- Cron auto-pwner
- Hooks all `*stat(2)` syscalls
  - If `stat(2)`-ing `/etc/crontab`, triggers kretprobe logic
  - In kretprobe, modifies the kernel-written `struct stat` to update the last modified time
  - This triggers `cron` to reload the file
- Hooks `openat(2)` and `close(2)`
  - If `openat(2)`-ing `/etc/crontab`, triggers kretprobe logic
  - In `openat(2)` kretprobe, saves the file descriptor returned to userspace
  - In `close(2)` kprobe, clears the mapping if the `/etc/crontab` fd is closed
- Hooks `read(2)`
  - If `read(2)`-ing from a known `/etc/crontab` fd, triggers kretprobe logic
  - In kretprobe, modifies the kernel-written buffer to inject root commads at the beginning of the ”file”

Demo

Conjob — Fun Facts

- Uses percpu maps to have kprobes and associated kretprobes communicate with each other
- Uses eBPF hash maps to have different pairs of k(ret)probes share fds with each other
- Uses the `bpf_ktime_get_ns()` helper to keep `/etc/crontab` ”recently updated”

What else can we do with eBPF?

Go for broke

Offensive eBPF — ROP 'til You Drop

- If you'll recall, we can write to the stack

Offensive eBPF — ROP ’til You Drop

- If you’ll recall, we can write to the stack
- The stack has return addresses

Offensive eBPF — ROP ’til You Drop

- If you’ll recall, we can write to the stack
- The stack has return addresses
- We can also read the stack and all of userspace memory

Offensive eBPF — ROP ’til You Drop

- If you’ll recall, we can write to the stack
- The stack has return addresses
- We can also read the stack and all of userspace memory
- We can scan for the text section and shared libraries

glibcpwn — The fastest way to a man’s heart is through his init daemon

- Systemd auto-pwner
- Scans PID 1 memory for `libc.so`
- Backs up stack content at the return address for libc syscall stub
- Injects a ROP payload targeting `libc.so` into the stack
- ROP payload calls glibc-internal `dlopen(3)` wrapper
- Loads malicious shared library into PID 1
- Completely cleans up after itself as if nothing happened

Demo

glibcpwn — Implementation Details Pt. 1

1. Hooks `timerfd_settime(2)`, a syscall `systemd` reliably calls once every minute
2. Scans *forward* from the stack-based `struct itimerspec` passed to the kernel
3. Looks for return address from `timerfd_settime(2)` stub function
   1. Follows each possible return address
   2. Scans back for and parses `jmp` and `call` instructions
   3. Applies relative offsets and scans for syscall stub or PLT stub
      - If the latter, parses the `jmp` to get function start
4. Calculates offset to start of `libc.so`
5. Returns stack return address and address of `__libc_start_main` to userland tracer code

glibcpwn — Implementation Details Pt. 2

1. Hooks `timerfd_settime(2)` and `close(2)`
2. In kretprobe for `timerfd_settime(2)`
   1. Copies stack for safekeeping
   2. Writes a ROP chain into return address
3. Kernel returns to userspace
4. `timerfd_settime(2)` returns into ROP chain
   1. Sets up rdi, rsi, rdx, rcx
   2. Returns into `__libc_dlopen_mode` to load shared library
   3. Sets rax to 3 (`close(2)`)
   4. Sets rdi to a magic negative value
   5. Returns into raw syscall gadget
5. `close(2)` kprobe hit
   1. Checks if fd matches magic value, writes *most of* original stack back
   2. Does not write over remaining gadgets in original chain
   3. Writes a new ROP chain past the end of where the stack originally was
6. Kernel returns to userspace
7. Last gadget shifts rsp to newly written ROP chain

glibcpwn — Implementation Details Pt. 3

1. New ROP chain fires
   1. Writes back original stack values over the last original gadget
   2. `xor rax, rax` to mark success for original `timerfd_settime(2)` syscall
   3. Returns back to next instruction after `syscall` in `timerfd_settime(2)` stub
2. Process execution continues as normal

glibcpwn — Fun Facts

- glibc is fairly stable, even between different versions on different distros
    - All gadgets have identical or nigh-identical equivalents across the board

What *else* can we do with eBPF?

Use it as intended

eBPF Rootkits — Omniscience and Omnipotence

- Once eBPF is running in a kprobe, it can prevent processes from interacting with the kernel

eBPF Rootkits — Omniscience and Omnipotence

- Once eBPF is running in a kprobe, it can prevent processes from interacting with the kernel
- For example, it can prevent processes from:

eBPF Rootkits — Omniscience and Omnipotence

- Once eBPF is running in a kprobe, it can prevent processes from interacting with the kernel
- For example, it can prevent processes from:
    - Listing running eBPF programs and kprobes

eBPF Rootkits — Omniscience and Omnipotence

- Once eBPF is running in a kprobe, it can prevent processes from interacting with the kernel
- For example, it can prevent processes from:
  - Listing running eBPF programs and kprobes
  - Creating eBPF kprobes

eBPF Rootkits — Omniscience and Omnipotence

- Once eBPF is running in a kprobe, it can prevent processes from interacting with the kernel
- For example, it can prevent processes from:
  - Listing running eBPF programs and kprobes
  - Creating eBPF kprobes
  - Loading kernel modules

eBPF Rootkits — Omniscience and Omnipotence

- Once eBPF is running in a kprobe, it can prevent processes from interacting with the kernel
- For example, it can prevent processes from:
  - Listing running eBPF programs and kprobes
  - Creating eBPF kprobes
  - Loading kernel modules
  - Phoning home about a detected compromise

eBPF Rootkits — Omniscience and Omnipotence

- Once eBPF is running in a kprobe, it can prevent processes from interacting with the kernel
- For example, it can prevent processes from:
  - Listing running eBPF programs and kprobes
  - Creating eBPF kprobes
  - Loading kernel modules
  - Phoning home about a detected compromise
    - This is important because using `bpf_probe_write_user()` causes a dmesg notification
    - The only way to escape is to have read dmesg *first* and already had memory-mapped direct packet I/O configured to send an SOS *without* using a syscall

eBPF Rootkits — Omniscience and Omnipotence

- Once eBPF is running in a kprobe, it can prevent processes from interacting with the kernel
- For example, it can prevent processes from:
  - Listing running eBPF programs and kprobes
  - Creating eBPF kprobes
  - Loading kernel modules
  - Phoning home about a detected compromise
    - This is important because using `bpf_probe_write_user()` causes a dmesg notification
    - The only way to escape is to have read dmesg *first* and already had memory-mapped direct packet I/O configured to send an SOS *without* using a syscall
    - Even then, it's probably possible to use non-writing (k|u)probes to burn time until it can kill the process

eBPF Rootkits — Omniscience and Omnipotence

- Once eBPF is running in a kprobe, it can prevent processes from interacting with the kernel
- For example, it can prevent processes from:
  - Listing running eBPF programs and kprobes
  - Creating eBPF kprobes
  - Loading kernel modules
  - Phoning home about a detected compromise
    - This is important because using `bpf_probe_write_user()` causes a dmesg notification
    - The only way to escape is to have read dmesg *first* and already had memory-mapped direct packet I/O configured to send an SOS *without* using a syscall
    - Even then, it's probably possible to use non-writing (k|u)probes to burn time until it can kill the process
    - Also, `bpf_override_return()` is supposed to allow eBPF kprobes to force a syscall to bail, but it didn't work for us when we tried it...

eBPF Rootkits — Nigh-Omnipresence

- The one downside of eBPF is that it needs to be tied to a running process to stay alive

eBPF Rootkits — Nigh-Omnipresence

- The one downside of eBPF is that it needs to be tied to a running process to stay alive
- What if we could make our eBPF kprobes functionally immortal?

eBPF Rootkits — Anchor Pivoting

- Once you take over a process like PID 1, you can run rootkit eBPF kprobes from PID 1 itself

eBPF Rootkits — Anchor Pivoting

- Once you take over a process like PID 1, you can run rootkit eBPF kprobes from PID 1 itself
- This means they will stay alive until the system shuts down

eBPF Rootkits — Anchor Pivoting

- Once you take over a process like PID 1, you can run rootkit eBPF kprobes from PID 1 itself
- This means they will stay alive until the system shuts down
- And vice-versa if PID 1 crashes, so to does the system

eBPF Rootkits — Anchor Pivoting

- Once you take over a process like PID 1, you can run rootkit eBPF kprobes from PID 1 itself
- This means they will stay alive until the system shuts down
- And vice-versa if PID 1 crashes, so to does the system
- Which is great for us, because everyone will think `systemd` is being unstable as usual

Conclusion

- eBPF is useful for *everyone*

Conclusion

- eBPF is useful for *everyone*
- *Except* people trying to build IDS on top of it

Conclusion

- eBPF is useful for *everyone*
- *Except* people trying to build IDS on top of it
- It needs to get much better at supporting that use case, and it simply isn’t there right now

Conclusion — Pleas to eBPF Kernel Devs

- Please add more helper functions:
    - `copy_from_user()`
    - To aid in reading tricky kernel data structures
        - Like files/paths
    - Direct string/memory comparison operations
    - Also, `memset(3)`

Greetz — Thanks for the code and the blogs!

- The BCC developers
- Julia Evans
- Brendan Gregg
- Jessie Frazelle

You can’t hide from the future.

Questions?
Pull Requests?

https://github.com/nccgroup/ebpf

jeff.dileo@nccgroup.com
@chaosdatumz

andy.olsen@nccgroup.com
@0lsen_

Kernel Tracing With eBPF

Unlocking God Mode on Linux

Jeff Dileo
@chaosdatumz

Andy Olsen
@0lsen_

35C3