Fast and Easy ~~p~~Tracing with eBPF

(and not ptrace)

Andy Olsen

NCC Group Open Forum September 2019

Who am I

- Ultimate frisbee enthusiast
- Amateur chiptune artist
- Security Constulant @ NCC Group
- Level 10 eBPF druid (the Linux source tree speaks to me)
    - With eBPF Insight feat, gives advantage on eBPF code audit-related checks

What is eBPF?

- “extended” Berkeley Packet Filter (BPF)
  - BPF is the bytecode language used by tcpdump to filter packets in the kernel
- “designed to be JITed with one to one mapping” to mainstream architectures (e.g. x86)
- “originally designed with the possible goal in mind to write programs in ‘restricted C’”
  - The kernel places restrictions on eBPF programs to prevent them from breaking the kernel
- socket filters, packet processing, **tracing**, backend for “classic” BPF, and more...

Tracing

- Dynamic and programmatic logging of code
- “Tracing” as applied to the Linux kernel and processes running on it
  - We want to be able to observe… basically everything
- eBPF makes it easy to use the powerful tracing capabilities of the Linux kernel
  - kprobes (function hooks for kernel code)
  - uprobes (function hooks for userspace code)
  - tracepoints (existing kernel logging functionality that can be enabled at runtime)
  - `perf` events (a bunch of different kernel profiling mechanisms, e.g. interrupt at frequency)
- Except for tracepoints and static profiling with perf events
  - These capabilities are restricted to kernel code and kernel modules

eBPF Tracing

- Pros
  - Lightweight & performant
  - Can observe all processes across a system simultaneously
    - Or filter down to individual processes
  - Can hook kernel functionality in the kernel and read arbitrary kernel memory
  - kprobes are generally unobservable by userspace
    - but can read/write their memory
  - All other benefits of tracepoints/perf events, but with in-kernel processing and filtering
- Cons
  - Complicated to develop for
    - eBPF coding restrictions
    - Build toolchain complexity
    - Lack of standardization in tooling which is also not fully mature
  - Bleeding edge

eBPF Tracing vs.

- vs syscall tracing (e.g. `ptrace(2)` / `strace`)
  - `ptrace(2)` limitations
    - one process at a time
    - Slow
    - detectable
    - Blockable
  - Static logging of only inputs / outputs
    - lags behind on new syscalls
    - and often does not display relevant data (pointer address vs text content)
- vs debuggers (e.g. gdb, lldb)
  - Also built on `ptrace(2)`
    - Extra detectable due to larger footprint
  - Allow dynamically inspecting userspace memory
  - Allows manipulating userspace registers

eBPF Tracing vs.

- vs static program function hooking (e.g. LD_PRELOAD)
  - Injects code into target process via loading an extra shared object
  - Only directly intercepts external functions
    - From dynamically linked shared objects (when called through the PLT)

- vs dynamic program instrumentation (e.g. frida)
  - Injects code into target process (via multiple methods)
  - Advanced management of code execution within process
  - Can hook on functions and even instructions

eBPF Tracing ~~vs.~~ and

- The primary benefit of eBPF tracing is its performance, invisibility, and omnipresence
- If you need more than it gives you, just use something better suited for the job
- Between eBPF tracing, scriptable debuggers, and frida
  - We live in pretty good times for dynamic program analysis of native code

Prior Art – DTrace

- Dynamic tracing framework covering kernel and userspace
- Created by Sun for Solaris, ported to FreeBSD and OS X (neutered in the latter)
- Based on a custom bytecode virtual machine executed in the kernel
  - Does that sound familiar?

```
# dtrace -n 'syscall::open*:entry \
            { printf("%s %s", execname, copyinstr(arg0)); }'
```

bpftrace

- Tracing framework and CLI utility
- Custom high-level tracing language (“bpftrace”) for Linux eBPF based on Dtrace’s D
- Uses LLVM APIs to emit eBPF bytecode
- Supports one-liners and script files using the bpftrace language

```
# bpftrace -e 't:syscalls:sys_enter_openat \
              { printf("%s %s\n", comm, str(args->filename)); }'
```

bpftrace – example

```
# bpftrace -e 'BEGIN { printf("hi open forum\n"); }'
Attaching 1 probe…
hi open forum
^C
```

- BEGIN is a “special probe”
  - A uprobe that hooks a function (BEGIN_trigger) in bpftrace’s own process
  - bpftrace registers these first, calls BEGIN_trigger, then registers all other probes afterwards

bpftrace – real example

```
# bpftrace -e 't:syscalls:sys_enter_execve { printf("%s\n", comm);
                                              join(args->argv); }'
Attaching 1 probe…
zsh
ls --color=auto -v
zsh
git rev-parse --is-inside-work-tree
```

- join() is a bpftrace builtin function
  - Joins a string array with a space char and prints it with a newline
- args is another builtin
  - Struct for tracepoints that contains the tracepoint arguments

bpftrace – realer example

- How does `fork()` work?
  - Man has struggled with this question since the Unix epoch
- Let's find out
  - bpftrace's own examples use the following tracepoint filter:

```
# bpftrace -e 't:sched:sched_process_fork
{ printf("%s\n",comm); cat("/proc/%d/cmdline",pid); printf("\n"); }'
Attaching 1 probe...
zsh
-zsh
^C
```

bpftrace – realer example

- But we want more
- Let’s hook on the real fork syscall tracepoint in the kernel:

```
# bpftrace -e 't:syscalls:sys_enter_fork {printf("%s\n",comm);}'
Attaching 1 probe…
```

bpftrace – realer example

- And we got nothing
- Let’s hook the real kernel implementation of fork:

```
# bpftrace -e 'kprobe:_do_fork {printf("%s\n",comm);}'
Attaching 1 probe...
zsh
zsh
zsh
zsh
^C
```

bpftrace – child PIDs from fork()

- Let’s only get the children processes from `fork()`
- We know from the man page that fork returns 0 in the child process
- Using a kretprobe, we can get the return value of fork

```
#  bpftrace -e 'kretprobe:_do_fork {printf("%d\n",retval)}'
Attaching 1 probe...
35370
35371
35372
```

bpftrace – child PIDs from fork()

- Something is not right
- The return value is never 0
- Why is that?

bpftrace – child PIDs from fork()

- Something is not right
- The return value is never 0
- Why is that?
- Let’s ask the kernel…

bpftrace – child PIDs from fork()

- `_do_fork()` is defined in kernel/fork.c

```
struct task_struct *p;
…
p = copy_process(clone_flags, stack_start, stack_size, parent_tidptr,
                 child_tidptr, NULL, trace, tls, NUMA_NO_NODE);
…
pid = get_task_pid(p, PIDTYPE_PID);
nr = pid_vnr(pid);
…
wake_up_new_task(p);
…
return nr;
```

bpftrace – child PIDs from fork()

- What does copy_process do?

```
struct task_struct *p;
…
p = dup_task_struct(current, node);
…
retval = copy_thread_tls(clone_flags, stack_start, stack_size, p, tls);
…
return p;
```

bpftrace – child PIDs from fork()

- What does copy_thread_tls do?

```
struct pt_regs *childregs;
…
childregs = task_pt_regs(p);
…
childregs->ax = 0;
…
return err;
```

bpftrace – child PIDs from fork()

- _do_fork() is defined in kernel/fork.c

```
struct task_struct *p;
…
p = copy_process(clone_flags, stack_start, stack_size, parent_tidptr,
                 child_tidptr, NULL, trace, tls, NUMA_NO_NODE);
…
pid = get_task_pid(p, PIDTYPE_PID);
nr = pid_vnr(pid);
…
wake_up_new_task(p);
…
return nr;
```

bpftrace – child PIDs from fork()

- While 0 isn't actually "returned," we can get what we need from `wake_up_new_task()`

```
#!/usr/local/bin/bpftrace
#include <linux/sched.h>
kprobe:wake_up_new_task {
  $chld_pid = ((struct task_struct *)arg0)->pid;
  printf("child pid: %d\n", $chld_pid);
}
```

bpftrace – tracking fork+exec

- We want to know when processes are performing fork then exec calls
  - i.e. `system()`
- We can reuse the previous function and add a map to track child PIDs
  - And we'll stash the near time the fork actually happens at

```
kprobe:wake_up_new_task {
  $chld_pid = ((struct task_struct *)arg0)->pid;
  @pids[$chld_pid] = nsecs;
}
```

bpftrace – tracking fork+exec

- Next we hook `execve()` and check if PID is in map
- And we'll do a time comparison for near instant fork+exec pairs

```
tracepoint:syscalls:sys_enter_execve {
  if (@pids[pid]){
    $time_diff = ((nsecs - @pids[pid]) / 1000000);
      if( $time_diff <= 10 ){
        printf("%s => ",comm);
        join(args->argv);
      }
  }
  delete(@pids[pid]);
}
```

bpftrace – tracing fork+exec

```
# bpftrace fork_exec.bt
Attaching 2 probes...
zsh => ssh localhost
sshd => /usr/sbin/sshd -D -R
…
sshd => -zsh
…
zsh => vim
vim => /bin/zsh -c ls
```

eBPF C

- Manually written “restricted” C programs
  - No loops
  - No non `static inline` calls
- Compiled into eBPF architecture ELF shared library file
- Parsed and then loaded into the kernel with libbpf/bpf_load.c helper functions
- Why?
  - Fine-grained control over what tracing code does
  - Custom userspace code to interact with kernelspace code
  - C memory/struct model (can load kernel headers and directly cast pointers)

eBPF C – setup for tracing

```
clang -S -O2 -emit-llvm -D __BPF_TRACING__ \
  -fno-unwind-tables -fno-asynchronous-unwind-tables \
  -nostdinc -isystem /usr/include/clang/8/include \
  -D__KERNEL__ -D__ASM_SYSREG_H \
  $(CFLAGS) \
  -I/lib/modules/`uname -r`/build/arch/x86/include \
  …
  -I/lib/modules/`uname -r`/build/include/generated/uapi \
  -include /lib/modules/`uname -r`/build/include/linux/kconfig.h \
  -fno-stack-protector \
  -g -c -o - kern.c | llc -march=bpf -filetype=obj -o kern.o
```

eBPF C – setup for tracing

- Basically the same as for Linux kernel module development
- You need the Linux kernel headers for the version of the kernel you're targeting
- Clang (a recent version)
  - Or bleeding edge GCC which apparently just got eBPF support as of yesterday
- Userspace eBPF loader
  - libbpf
  - bpf_load.c
- Very carefully constructed makefile to build eBPF code similarly to Linux kernel code
  - See previous slide
  - ***Note:*** Code compiles targeting actual architecture first, since directly targeting eBPF ISA would break when lower level kernel headers are resolved (i.e. atomics, inline assembly)

eBPF C – real example

- How do namespaces work?
  - In this talk we will not cover how namespaces actually work
- There are a lot of syscalls involved, but we can focus on three
  - `clone` (basically fork/vfork with namespace flags)
  - `unshare` (places process in new namespaces without forking)
  - `setns` (adds a namespace to a process)
- Pulling out and parsing flags is a job for C
  - And so is parsing internal Linux kernel structs for file descriptors and namespaces

eBPF C – real example

```
static inline void parse_namespace_flags(int type){
  char string[8];
  string[0] = (CLONE_NEWNS & type) ? 'M' : '-';
  string[1] = (CLONE_NEWCGROUP & type) ? 'C' : '-';
  string[2] = (CLONE_NEWUTS & type) ? 'T' : '-';
  string[3] = (CLONE_NEWIPC & type) ? 'I' : '-';
  string[4] = (CLONE_NEWUSER & type) ? 'U' : '-';
  string[5] = (CLONE_NEWPID & type) ? 'P' : '-';
  string[6] = (CLONE_NEWNET & type) ? 'N' : '-';
  string[7] = '\0';
  bpf_printk("namespace flags: %s\n", string);
}
```

eBPF C – real example

```
SEC("kprobe/SyS_setns")
int trace_setns(struct pt_regs *ctx){
  // get hooks thread id
  __u64 pid_tgid = bpf_get_current_pid_tgid();
  struct pt_regs backup_regs;
  // write ctx into backup_regs
  bpf_probe_read(&backup_regs, sizeof(struct pt_regs), ctx);
  // update map with tid and regs
  bpf_map_update_elem(&setns_map, &pid_tgid, &backup_regs, BPF_ANY);
  return 0;
}
```

eBPF C – real example

```
SEC("kretprobe/proc_ns_fget")
int trace_ns_fget(struct pt_regs *ctx){
  __u64 pid_tgid = bpf_get_current_pid_tgid();
  struct pt_regs *setns_regs = bpf_map_lookup_elem(&setns_map, &pid_tgid);
  if (setns_regs == NULL) return 0;
  struct file *f = (struct file*)ctx->ax;
  if(IS_ERR_VALUE(f)) return 0;
  struct inode *i; bpf_probe_read(&i, sizeof(i), &f->f_inode);
  struct ns_common *ns; bpf_probe_read(&ns, sizeof(ns), &i->i_private);
  struct proc_ns_operations *ops; bpf_probe_read(&ops,sizeof(ops),&ns->ops);
  int type; bpf_probe_read(&type, sizeof(type), &ops->type);
  parse_namespace_flags(type);
```

eBPF C – real example

```
SEC("kretprobe/proc_ns_fget")
int trace_ns_fget(struct pt_regs *ctx){
  __u64 pid_tgid = bpf_get_current_pid_tgid();
  struct pt_regs *setns_regs = bpf_map_lookup_elem(&setns_map, &pid_tgid);
  if (setns_regs == NULL) return 0;
  struct file *f = (struct file*)ctx->ax;
  if(IS_ERR_VALUE(f)) return 0;
  struct inode *i; bpf_probe_read(&i, sizeof(i), &f->f_inode);
  struct ns_common *ns; bpf_probe_read(&ns, sizeof(ns), &i->i_private);
  struct proc_ns_operations *ops; bpf_probe_read(&ops,sizeof(ops),&ns->ops);
  int type; bpf_probe_read(&type, sizeof(type), &ops->type);
  parse_namespace_flags(type);
```

eBPF C – real example

```
SEC("kretprobe/proc_ns_fget")
int trace_ns_fget(struct pt_regs *ctx){
  __u64 pid_tgid = bpf_get_current_pid_tgid();
  struct pt_regs *setns_regs = bpf_map_lookup_elem(&setns_map, &pid_tgid);
  if (setns_regs == NULL) return 0;
  struct file *f = (struct file*)ctx->ax;
  if(IS_ERR_VALUE(f)) return 0;
  struct inode *i; bpf_probe_read(&i, sizeof(i), &f->f_inode);
  struct ns_common *ns; bpf_probe_read(&ns, sizeof(ns), &i->i_private);
  struct proc_ns_operations *ops; bpf_probe_read(&ops,sizeof(ops),&ns->ops);
  int type; bpf_probe_read(&type, sizeof(type), &ops->type);
  parse_namespace_flags(type);
```

eBPF C – real example

```
SEC("kretprobe/proc_ns_fget")
int trace_ns_fget(struct pt_regs *ctx){
  __u64 pid_tgid = bpf_get_current_pid_tgid();
  struct pt_regs *setns_regs = bpf_map_lookup_elem(&setns_map, &pid_tgid);
  if (setns_regs == NULL) return 0;
  struct file *f = (struct file*)ctx->ax;
  if(IS_ERR_VALUE(f)) return 0;
  struct inode *i; bpf_probe_read(&i, sizeof(i), &f->f_inode);
  struct ns_common *ns; bpf_probe_read(&ns, sizeof(ns), &i->i_private);
  struct proc_ns_operations *ops; bpf_probe_read(&ops,sizeof(ops),&ns->ops);
  int type; bpf_probe_read(&type, sizeof(type), &ops->type);
  parse_namespace_flags(type);
```

eBPF C – real example

```
SEC("kretprobe/proc_ns_fget")
int trace_ns_fget(struct pt_regs *ctx){
  __u64 pid_tgid = bpf_get_current_pid_tgid();
  struct pt_regs *setns_regs = bpf_map_lookup_elem(&setns_map, &pid_tgid);
  if (setns_regs == NULL) return 0;
  struct file *f = (struct file*)ctx->ax;
  if(IS_ERR_VALUE(f)) return 0;
  struct inode *i; bpf_probe_read(&i, sizeof(i), &f->f_inode);
  struct ns_common *ns; bpf_probe_read(&ns, sizeof(ns), &i->i_private);
  struct proc_ns_operations *ops; bpf_probe_read(&ops,sizeof(ops),&ns->ops);
  int type; bpf_probe_read(&type, sizeof(type), &ops->type);
  parse_namespace_flags(type);
```

eBPF C – real example

```
SEC("kretprobe/proc_ns_fget")
int trace_ns_fget(struct pt_regs *ctx){
  __u64 pid_tgid = bpf_get_current_pid_tgid();
  struct pt_regs *setns_regs = bpf_map_lookup_elem(&setns_map, &pid_tgid);
  if (setns_regs == NULL) return 0;
  struct file *f = (struct file*)ctx->ax;
  if(IS_ERR_VALUE(f)) return 0;
  struct inode *i; bpf_probe_read(&i, sizeof(i), &f->f_inode);
  struct ns_common *ns; bpf_probe_read(&ns, sizeof(ns), &i->i_private);
  struct proc_ns_operations *ops; bpf_probe_read(&ops,sizeof(ops),&ns->ops);
  int type; bpf_probe_read(&type, sizeof(type), &ops->type);
  parse_namespace_flags(type);
```

eBPF C – real example

```
SEC("kretprobe/proc_ns_fget")
int trace_ns_fget(struct pt_regs *ctx){
…
  if (!(type & CLONE_NEWUSER)) return 0;
  struct user_namespace* uns = container_of(ns, struct user_namespace, ns);
  struct user_namespace* p; bpf_probe_read(&p, sizeof(p), &user_ns->parent);
  int level; bpf_probe_read(&level, sizeof(level), &user_ns->level);
  kuid_t owner; bpf_probe_read(&owner, sizeof(owner), &user_ns->owner);
  kuid_t group; bpf_probe_read(&group, sizeof(group), &user_ns->group);
  bpf_printk("parent: 0x%lx, level: %d\n", (__u64)parent, level);
  bpf_printk("owner: %ld, group: %ld\n", owner, group);
```

eBPF C – real example

```
SEC("kretprobe/proc_ns_fget")
int trace_ns_fget(struct pt_regs *ctx){
…
  if (!(type & CLONE_NEWUSER)) return 0;
  struct user_namespace* uns = container_of(ns, struct user_namespace, ns);
  struct user_namespace* p; bpf_probe_read(&p, sizeof(p), &user_ns->parent);
  int level; bpf_probe_read(&level, sizeof(level), &user_ns->level);
  kuid_t owner; bpf_probe_read(&owner, sizeof(owner), &user_ns->owner);
  kuid_t group; bpf_probe_read(&group, sizeof(group), &user_ns->group);
  bpf_printk("parent: 0x%lx, level: %d\n", (__u64)parent, level);
  bpf_printk("owner: %ld, group: %ld\n", owner, group);
```

eBPF C – real example

```
SEC("kretprobe/proc_ns_fget")
int trace_ns_fget(struct pt_regs *ctx){
…
  if (!(type & CLONE_NEWUSER)) return 0;
  struct user_namespace* uns = container_of(ns, struct user_namespace, ns);
  struct user_namespace* p; bpf_probe_read(&p, sizeof(p), &uns->parent);
  int level; bpf_probe_read(&level, sizeof(level), &uns->level);
  kuid_t owner; bpf_probe_read(&owner, sizeof(owner), &uns->owner);
  kuid_t group; bpf_probe_read(&group, sizeof(group), &uns->group);
  bpf_printk("parent: 0x%lx, level: %d\n", (__u64)p, level);
  bpf_printk("owner: %ld, group: %ld\n", owner, group);
```

eBPF C – real example – lxc-start

```
lxc-usernsexec-2820  [001] ....   274.075547: 0x00000001: unshare
lxc-usernsexec-2820  [001] .N..   274.075570: 0x00000001: namespace flags: M---U--

lxc-usernsexec-2820  [001] ....   274.084084: 0x00000001: unshare
lxc-usernsexec-2820  [001] ....   274.084098: 0x00000001: namespace flags: M------

         <...>-2825  [000] ....   274.126341: 0x00000001: unshare
         <...>-2825  [000] ....   274.126358: 0x00000001: namespace flags: ------N

  lxc-user-nic-2835  [001] ....   274.156965: 0x00000001: setns: fd: 4; type: 0x40000000; pid: 2835
  lxc-user-nic-2835  [001] dN..   274.156980: 0x00000001: namespace flags: ------N

  lxc-user-nic-2835  [001] ....   274.162965: 0x00000001: setns: fd: 3; type: 0x40000000; pid: 2835
  lxc-user-nic-2835  [001] dN..   274.162978: 0x00000001: namespace flags: ------N
```

eBPF C – real example – lxc-start

```
      lxc-start-2825  [000] ....   274.163998: 0x00000001: unshare
      lxc-start-2825  [000] ....   274.164021: 0x00000001: namespace flags: -C-----

          <...>-2911  [000] ....   274.356758: 0x00000001: unshare
          <...>-2911  [000] ....   274.356785: 0x00000001: namespace flags: M------

          <...>-2921  [000] ....   274.395521: 0x00000001: unshare
          <...>-2921  [000] ....   274.395549: 0x00000001: namespace flags: M------

 systemd-journal-2906  [001] ....   274.426977: 0x00000001: clone:
 systemd-network-2921  [000] ....   274.437340: 0x00000001: clone flags: VM|FS|FL|SH|--|--|--|--|TH|SV|ST|PS|CC|--|--|--|-
-

     (resolved)-2945  [001] ....   274.462602: 0x00000001: unshare
     (resolved)-2945  [001] .N..   274.462613: 0x00000001: namespace flags: M------
```

eBPF C – real example – lxc-start

```
    rsyslogd-2949  [000] ....   274.621476: 0x00000001: clone:
    rsyslogd-2949  [000] ....   274.621507: 0x00000001: clone flags: VM|FS|FL|SH|--|--|--|--|TH|SV|ST|PS|CC|--|--|--|--

 in:imuxsock-2955  [000] ....   274.629593: 0x00000001: clone:
 in:imuxsock-2955  [000] ....   274.629620: 0x00000001: clone flags: VM|FS|FL|SH|--|--|--|--|TH|SV|ST|PS|CC|--|--|--|--

       <...>-2959  [000] ....   274.637363: 0x00000001: unshare
       <...>-2959  [000] ....   274.637405: 0x00000001: namespace flags: ------N

  (ostnamed)-2959  [000] ....   274.639527: 0x00000001: unshare
  (ostnamed)-2959  [000] ....   274.639558: 0x00000001: namespace flags: M------

networkd-dispat-2947  [001] ....   274.716177: 0x00000001: clone:
networkd-dispat-2947  [001] .N..   274.716189: 0x00000001: clone flags: VM|FS|FL|SH|--|--|--|--|TH|SV|ST|PS|CC|--|--|--|--
```

eBPF C – real example – lxc-attach

```
lxc-usernsexec-2965  [001] ....   282.262893: 0x00000001: unshare
lxc-usernsexec-2965  [001] .N..   282.262912: 0x00000001: namespace flags: M---U--

lxc-usernsexec-2965  [001] ....   282.271559: 0x00000001: unshare
lxc-usernsexec-2965  [001] ....   282.271571: 0x00000001: namespace flags: M------

          <...>-2970  [000] ....   282.305322: 0x00000001: setns: fd: 3; type: 0x10000000; pid: 2970
          <...>-2970  [000] d...   282.305358: 0x00000001: namespace flags: ----U--
          <...>-2970  [000] d...   282.305359: 0x00000001: dumping user_namespace!
          <...>-2970  [000] d...   282.305360: 0x00000001: parent: 0xffffffff988541e0, level: 1
          <...>-2970  [000] d...   282.305361: 0x00000001: owner: 1000, group: 1000
```

eBPF C – real example – lxc-attach

```
<...>-2970  [000] ....   282.305374: 0x00000001: setns: fd: 4; type: 0x20000; pid: 2970
<...>-2970  [000] d...   282.305382: 0x00000001: namespace flags: M------
<...>-2970  [000] ....   282.305390: 0x00000001: setns: fd: 5; type: 0x20000000; pid: 2970
<...>-2970  [000] d...   282.305393: 0x00000001: namespace flags: -----P-
<...>-2970  [000] ....   282.305396: 0x00000001: setns: fd: 6; type: 0x4000000; pid: 2970
<...>-2970  [000] d...   282.305398: 0x00000001: namespace flags: --T----
<...>-2970  [000] ....   282.305401: 0x00000001: setns: fd: 7; type: 0x8000000; pid: 2970
<...>-2970  [000] d...   282.305403: 0x00000001: namespace flags: ---I---
<...>-2970  [000] ....   282.305406: 0x00000001: setns: fd: 8; type: 0x40000000; pid: 2970
<...>-2970  [000] d...   282.305408: 0x00000001: namespace flags: ------N
<...>-2970  [000] ....   282.305411: 0x00000001: setns: fd: 9; type: 0x2000000; pid: 2970
<...>-2970  [000] d...   282.305413: 0x00000001: namespace flags: -C-----
```

Conclusion

- eBPF tracing can be tricky, but it’s a useful addition to the security toolkit
  - As a supplement, not a replacement!
- Go forth and listen to your kernels
  - And make them reveal their secrets!

Future Work

- eBPF tracing + containers
- Rewriting a bunch of tooling using eBPF C for better and more stable performance
- eBPF + all the things

Questions?

Andy Olsen

andy.olsen@nccgroup.com

Fast and Easy ~~p~~Tracing with eBPF

(and not ptrace)

Andy Olsen

NCC Group Open Forum September 2019