eBPF - The coolest-newest kid in town

PatH

whoami, whatisthis

- Pat, Security Researcher @ CrowdStrike
- Digging into eBPF for about a year...ish
- eBPF is revolutionising Linux, want to share why I think so

Two people with Problems

Software Developers & Sysadmins

- Modern code is distributed, complex
  - Microservices, Containers
- Difficulty in debugging bottlenecks
  - Packet Routing?
  - File I/O?
  - Race condition?
- Can't deploy debug code to production
  - Require high throughput while debugging

Malware Analysts & Bug Hunters

- Have some program, what does it do?
  - Files written
  - IP/DNS connections
  - Programs launched
- Binary could be encrypted/packed
- Could download+run 2nd stage
  - Or shell out
- Could have anti-debugging techniques

Solution - Trace the Kernel

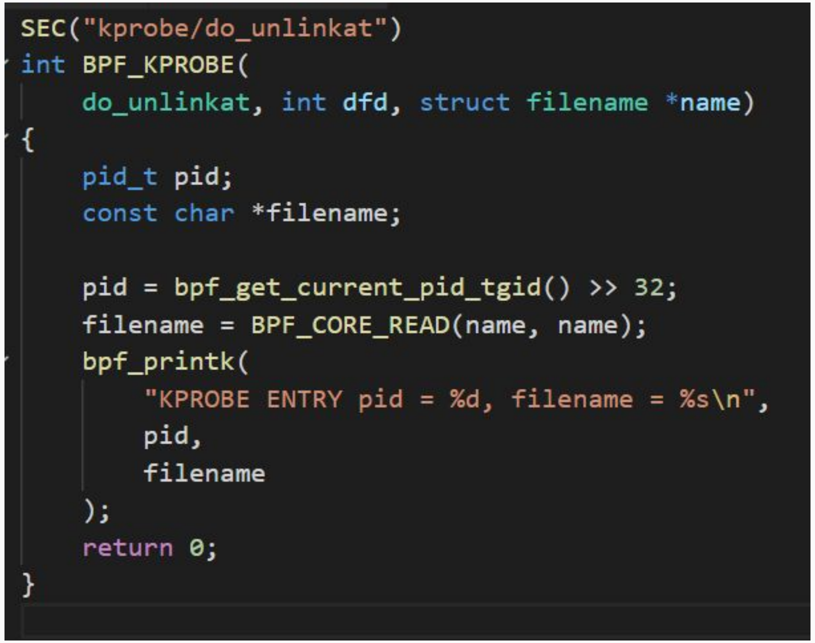
- Kernel sees all network, file, and inter-process activity
  - Trace syscall functions e.g. sys_open
- Kernel has text-based tracing you can setup and read from user space
- Difficult to filter logs. E.g. what if you only wanted to log when:
  - Process *A*
  - Opening File *B*
  - File opened after network connection *C*
  - Data read from file contains *D*
- Doesn’t dereference pointers
  - Can’t log members of an object, buffer, array etc.

Solution - Trace the Kernel

- Can write KProbes to trace functions
- But writing kernel code is:
    - **Hard**: Have to be very comfortable with kernel-specific code (pointers, containerof)
    - **Risky**: Easy to crash entire system
    - **Fragile**: Minor kernel configurations can break code
- Managed K8s often don’t allow you to run arbitrary kernel code anyway

The actual solution - eBPF

- eXtended Berkeley Packet Filtering
    - Much, much, *much* more than 'classic' BPF
- Write small programs to load into the kernel:
    - Filter and log functions and syscalls
    - Filter, log, and alter network packets
- Under the hood uses KProbes, etc.
- Programs guaranteed* to be:
    - **Performant:** Won't slow the system down
    - **Portable:** Can run on any* kernel
    - **Safe:** Won't crash the system



```c
SEC("kprobe/do_unlinkat")
int BPF_KPROBE(
    do_unlinkat, int dfd, struct filename *name)
{
    pid_t pid;
    const char *filename;

    pid = bpf_get_current_pid_tgid() >> 32;
    filename = BPF_CORE_READ(name, name);
    bpf_printk(
        "KPROBE ENTRY pid = %d, filename = %s\n",
        pid,
        filename
    );
    return 0;
}
```

eBPF Bytecode

- Code looks like C, but not compiled into native PE
- LLVM used to compile into architecture-agnostic ‘BPF bytecode’
  - Limited set of instructions
- Usermode programs use the ‘bpf’ syscall to send bytes to kernel
- Kernel takes bytes and just-in-time compiles it into machine code
  - Speed of native, but this bytecode limits what can be done
  - Machine code is then loaded into and run in the kernel

eBPF Safety

- Programs checked to ensure they are ‘safe’ at compile, load, and runtime:
- Programs are prevented from:
- **Being too big:**
    - max 4096 BPF instructions in kernels < 5.1 after 1M
- **Being too complex**
    - Limited number of branches and loop iterations
- **Taking too long**
    - Programs can’t sleep, etc.
- **Affect things outside the program**
    - No writing to kernel memory
    - Pointers checked to prevent bad reads
    - Can’t read files, make network connections, run programs

What can an eBPF program do?

- Read arguments passed into functions, and their return values
  - Including following pointers to read structs and arrays
  - Can also write back to userspace 🤨
- Read network packets
  - Parse data in packet and redirect/forward/drop
  - Packets can be read+written by eBPF before firewall 🤨
- Get stack trace
  - Very useful for bug hunting and debugging
- Send Signals to programs
  - E.g. SIG_KILL if program did something bad
- Send and receive data from userspace via eBPF Maps

eBPF Maps

- Main way to send data from kernel program to a userspace
- Key-Value stores
- Common uses:
    - Send events from eBPF program to userspace to be written to disk to sent off
    - Store configuration for eBPF Program
        - E.g. list of IPs to block
- Maps are accessible by any administrator
    - If the have CAP_ADMIN, e.g. root

Only admins can load eBPF programs, right?

- Yes...but no?
- Most eBPF actions require CAP_ADMIN
  - E.g. attaching to syscalls, reading Maps, unloading programs
  - Or CAP_BPF
- To attach to your own socket, you might not
  - Kernel can be configured to allow unpriv users to attack eBPF programs to sockets
  - eBPF Program still runs in the kernel
  - Can be disabled, but a lot of distros by default enable it
- Verifier meant to stop bad devs, not bad people
  - CVE-2017-16995
  - CVE-2020-8835
  - CVE-2021-3444
  - …
- FYI: eBPF programs inside of containers can see everything
  - Can be loaded in containers run with --privileged
  - Containers share same kernel and host

Example uses of eBPF

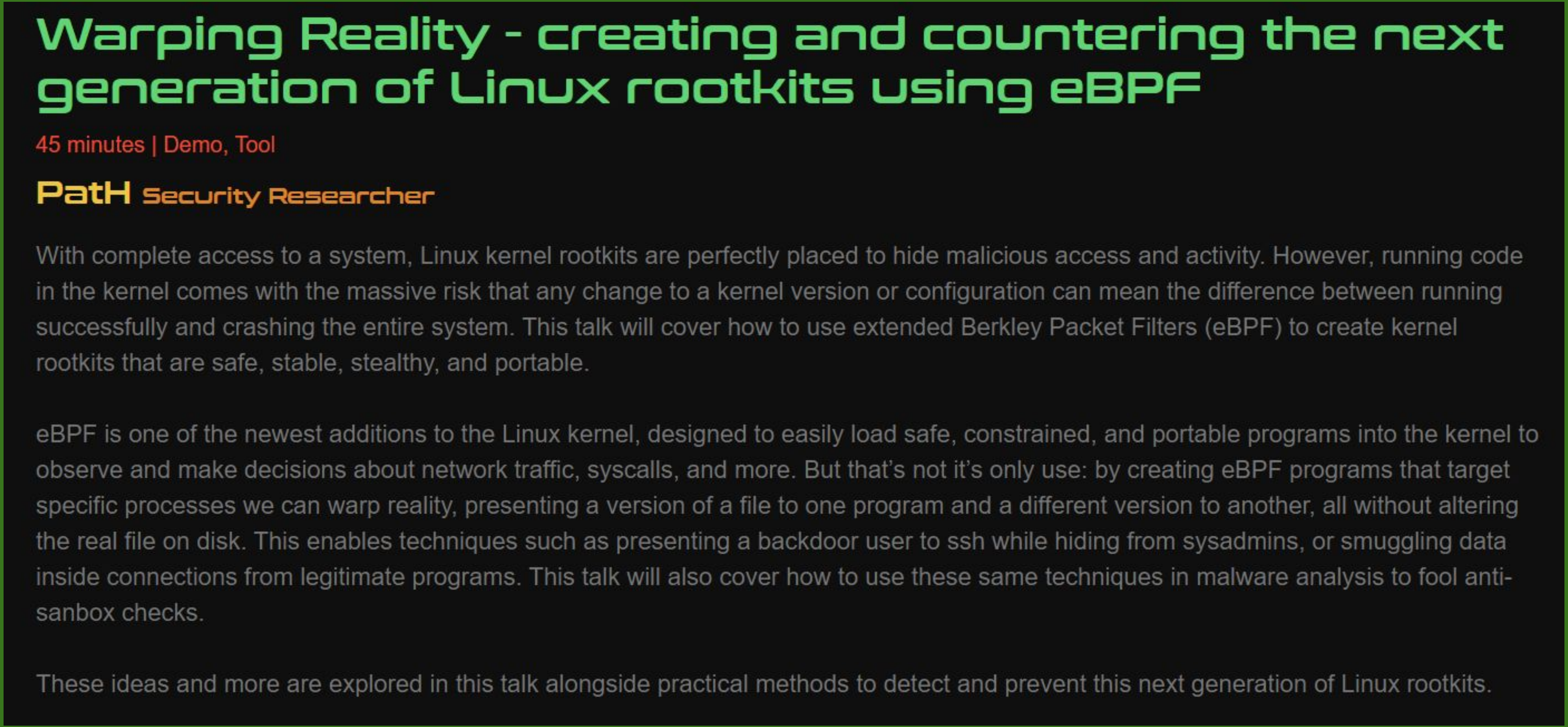
- Cilium
  - https://cilium.io
  - ‘Identity-aware’ network visibility and packet routing
  - Logs, filters, and redirects packets for Kubernetes
  - Is becoming the core of Google’s Managed Kubernetes network plane (GKE)
- Tracee
  - https://github.com/aquasecurity/tracee
  - System wide tracing
  - Like STrace, but for the entire system
  - eBPF means not PTracing the processes
- LibBPF
  - https://github.com/libbpf/libbpf-bootstrap
  - C usermode Library, part of the Linux source tree
  - Makes it easy to load and run eBPF Programs
  - Libbpf-bootstrap has lots of example programs in C and Rust

What if I like making sysadmins cry?

- We've discussed legitimate uses, what about illegitimate?
- Remember the 🤨:
  - Kernel Code that intercepts syscalls?
  - With the ability to read and write userspace memory??
  - With the ability to read and write packets pre-firewall??

- Sounds like a rootkit to me!
  - And it makes rootkits portable and safer to build and run?
  - And it can run in managed K8s environments?
  - And it allows you to behave differently to different programs and users?

🏄‍♀️ Come to DEF CON 29 🏄‍♂️



Warping Reality - creating and countering the next generation of Linux rootkits using eBPF

45 minutes | Demo, Tool

PatH Security Researcher

With complete access to a system, Linux kernel rootkits are perfectly placed to hide malicious access and activity. However, running code in the kernel comes with the massive risk that any change to a kernel version or configuration can mean the difference between running successfully and crashing the entire system. This talk will cover how to use extended Berkley Packet Filters (eBPF) to create kernel rootkits that are safe, stable, stealthy, and portable.

eBPF is one of the newest additions to the Linux kernel, designed to easily load safe, constrained, and portable programs into the kernel to observe and make decisions about network traffic, syscalls, and more. But that's not it's only use: by creating eBPF programs that target specific processes we can warp reality, presenting a version of a file to one program and a different version to another, all without altering the real file on disk. This enables techniques such as presenting a backdoor user to ssh while hiding from sysadmins, or smuggling data inside connections from legitimate programs. This talk will also cover how to use these same techniques in malware analysis to fool anti-sanbox checks.

These ideas and more are explored in this talk alongside practical methods to detect and prevent this next generation of Linux rootkits.

Questions?

Website, Blog:
https://blog.tofile.dev

GitHub, Slack, Twitter:
@pathtofile

Email:
path[at]tofile[dot]dev