

# When FLIRTing gets a CAPE

MERGING STATIC AND DYNAMIC  
DETECTION



# \$whoami

- Malware Detection Researcher @ SentinelOne
- Formerly Security Research @ Uptycs
- Linux nerd
- Work mostly on malware analysis and Security automation
- FOSS contributor and selfhosting nerd

# Motivation

- Imagine this scenario:
- You're working on some binary, doing the typical reverse engineering / malware analysis workflow.

# Motivation

Check for  
packers/encryptors

Static Analysis

Dynamic analysis

Detection

# Motivation

- But there are just way too many functions, possibly library code
- Don't want to actually spend a lot of time in dead code, want to find the useful functions quickly.
- This TechLab will help you do just that.

# Table of Contents

- Prerequisites
- Introduction
- History of eBPF
- eBPF + real-world examples
- Hooking in Linux
- IDA
- FLIRT
- CAPE
- Demo

# Prerequisites

- A linux computer (real or virtualized) - Ubuntu 22.04+ recommended
- IDA Free (Pro would be nice) - for FLIRT signature generation
- Capability to virtualize code - don't run it on host
- Basic understanding of assembly and system calls
- Familiarity with command line tools

# Introduction

- We will introduce you to three four different things:
  - eBPF
  - bpftrace
  - FLIRT
  - Tracing on Linux



# History of eBPF

- DTrace
  - Sun Microsystems created DTrace in 2003 for Solaris
  - Revolutionary dynamic tracing framework
  - Could observe kernel and userspace with minimal overhead
  - "*Don't just reboot your pc, goddamn it, debug it! Come on, you're an educated person, right? Or at least you want to act like one around other educated people!*" - Bryan Cantrill





# Birth of BPF (1992)

- Berkeley Packet Filter created for efficient packet filtering
- Used in tcpdump, Wireshark, and network monitoring tools
- Simple virtual machine in kernel space
- Example: `tcpdump -i wlan0 'tcp port 80'` uses BPF under the hood

# tcpdump

```
tlh@mirai ~> sudo tcpdump -d -i wlan0 'tcp port 80'
(000) ldh      [12]
(001) jeq      #0x86dd        jt 2    jf 8
(002) ldb      [20]
(003) jeq      #0x6          jt 4    jf 19
(004) ldh      [54]
(005) jeq      #0x50         jt 18   jf 6
(006) ldh      [56]
(007) jeq      #0x50         jt 18   jf 19
(008) jeq      #0x800        jt 9    jf 19
(009) ldb      [23]
(010) jeq      #0x6          jt 11   jf 19
(011) ldh      [20]
(012) jset     #0x1fff        jt 19   jf 13
(013) ldxb    4*([14]&0xf)
(014) ldh      [x + 14]
(015) jeq      #0x50         jt 18   jf 16
(016) ldh      [x + 16]
(017) jeq      #0x50         jt 18   jf 19
(018) ret      #262144
(019) ret      #0
tlh@mirai ~> |
```

# eBPF (2013-Current)

- Extended BPF emerged in Linux kernel 3.15
- No longer just for networking - can hook anywhere in kernel
- JIT compilation for better performance
- Verifier ensures safety of loaded programs
- Alexei Starovoitov and Daniel Borkmann led the development



# eBPF

- Wikipedia calls it "a technology that can run programs in a *privileged context* such as the operating system kernel."
- Brendan Gregg (Famous because of Dtrace) calls it "superpowers for Linux".
- But what can you actually get it to do?
- Think of it as a safe way to run custom code in kernel space
- Programs are event-driven and respond to kernel events
- Can observe, modify, and redirect kernel behavior
- Use cases: networking, security, observability, performance analysis

# Real-world Examples

- Netflix uses eBPF for network load balancing
- Facebook uses it for DDoS mitigation
- Cilium uses eBPF for container networking and security
- Falco uses it for runtime security monitoring
- (And these are the ones that talk about it out loud)



# eBPF gives you superpowers

- eBPF allows you to observe most parts of the kernel, the system-call interface, right up to the application level of random programs
- You can trace the arguments given to a function, the return values, and technically anything in between. (if you can hook into it, you can observe it.)
- Examples of what you can trace:
  - - `opensnoop` - trace file opens
  - - `execsnoop` - trace process execution
  - - `funccount` - count function calls
  - - `profile` - CPU profiling with stack traces

# opensnoop

```
tlh@mirai ~> sudo /usr/share/bcc/tools/opensnoop
PID      COMM              FD  ERR PATH
1901    aw-server          7   0  /home/tlh/.local/share/activitywatch/aw-server/peewee-sqlite.v2.db
1901    aw-server          8   0  /home/tlh/.local/share/activitywatch/aw-server/peewee-sqlite.v2.db-journal
1901    aw-server          9   0  /home/tlh/.local/share/activitywatch/aw-server
1901    aw-server          7   0  /home/tlh/.local/share/activitywatch/aw-server/peewee-sqlite.v2.db
1901    aw-server          8   0  /home/tlh/.local/share/activitywatch/aw-server/peewee-sqlite.v2.db-journal
1901    aw-server          9   0  /home/tlh/.local/share/activitywatch/aw-server
```

# execsnoop

```
tlh@mirai ~> sudo /usr/share/bcc/tools/execsnoop
COMM           PID   PPID   RET ARGS
9              8435   1154    0 /proc/self/fd/9 --deserialize 85 --log-level info --log-target auto
spectacle     8435   1154    0 /usr/bin/spectacle --dbus
```

# funccount

# But how?

tracepoints

kprobes

uprobes

uretprobes

# Tracepoints (+demo)

- "hooks" built into the kernel by developers
- Usually used for stuff like syscalls, scheduler events, memory management
- Hardcoded into the kernel, so cannot be changed
- Stable ABI - won't break between kernel versions
- Example: `syscalls:sys\_enter\_openat` traces all open() syscalls
- List with: `sudo bpftrace -l 'tracepoint:\*'`

# tracepoints

```
tlh@mirai ~> sudo bpftrace -l 'tracepoint:**' | head -n 50
[sudo] password for tlh:
tracepoint:alarmtimer:alarmtimer_cancel
tracepoint:alarmtimer:alarmtimer_fired
tracepoint:alarmtimer:alarmtimer_start
tracepoint:alarmtimer:alarmtimer_suspend
tracepoint:amd_cpu:amd_pstate_epp_perf
tracepoint:amd_cpu:amd_pstate_perf
tracepoint:asoc:snd_soc_bias_level_done
tracepoint:asoc:snd_soc_bias_level_start
tracepoint:asoc:snd_soc_dapm_connected
tracepoint:asoc:snd_soc_dapm_done
tracepoint:asoc:snd_soc_dapm_path
tracepoint:asoc:snd_soc_dapm_start
tracepoint:asoc:snd_soc_dapm_walk_done
tracepoint:asoc:snd_soc_dapm_widget_event_done
tracepoint:asoc:snd_soc_dapm_widget_event_start
tracepoint:asoc:snd_soc_dapm_widget_power
tracepoint:asoc: snd_soc_jack_irq
tracepoint:asoc: snd_soc_jack_notify
tracepoint:asoc: snd_soc_jack_report
tracepoint:avc:selinux_audited
tracepoint:binder:binder_alloc_lru_end
tracepoint:binder:binder_alloc_lru_start
tracepoint:binder:binder_alloc_page_end
tracepoint:binder:binder_alloc_page_start
tracepoint:binder:binder_command
tracepoint:binder:binder_free_lru_end
tracepoint:binder:binder_free_lru_start
tracepoint:binder:binder_ioctl
tracepoint:binder:binder_ioctl_done
tracepoint:binder:binder_lock
tracepoint:binder:binder_locked
tracepoint:binder:binder_read_done
tracepoint:binder:binder_return
tracepoint:binder:binder_transaction
tracepoint:binder:binder_transaction_alloc_buf
tracepoint:binder:binder_transaction_buffer_release
tracepoint:binder:binder_transaction_failed_buffer_release
tracepoint:binder:binder_transaction_fd_recv
tracepoint:binder:binder_transaction_fd_send
tracepoint:binder:binder_transaction_node_to_ref
tracepoint:binder:binder_transaction_received
tracepoint:binder:binder_transaction_ref_to_node
tracepoint:binder:binder_transaction_ref_to_ref
tracepoint:binder:binder_transaction_update_buffer_release
tracepoint:binder:binder_txn_latency_free
tracepoint:binder:binder_unlock
tracepoint:binder:binder_unmap_kernel_end
```

# Kprobes (+demo)

- "hooks" into the kernel that are dynamic and *can* be changed
- can be put anywhere (as long you follow the instruction boundary)
- Unstable - may break between kernel versions as internal functions change
- Example: `kprobe:vfs\_read` traces the virtual filesystem read function
- Can access function arguments and local variables

# Uprobes (+demo)

- kprobes but in userland
- Can be placed at any offset whatsoever in a binary, and will trigger once that place is reached by the program
- triggers int3 at that point, then the interrupt handler calls uprobe handler
- Works on any ELF binary - no recompilation needed
- Example: `uprobe:/bin/bash:readline` traces when bash reads input
- Perfect for reversing - can trace any function in any binary

# quick keylogger

```
[tlh@mirai ~]$ uname -a
Linux mirai 6.16.8-zen3-1-zen #1 ZEN SMP PREEMPT_DYNAMIC Mon, 22 Sep 2025 22:08:18 +0000 x86_64 GNU/Linux
[tlh@mirai ~]$ █
[tlh@mirai ~]# sudo bpftrace -e 'uretprobe:/usr/lib/libreadline.so.8:readline { printf("probe: %s\n", probe, str(retval));}'
Attaching 1 probe...
probe: uretprobe:/usr/lib/libreadline.so.8:readline, retval: ls -la .
probe: uretprobe:/usr/lib/libreadline.so.8:readline, retval: uname -a
█
```

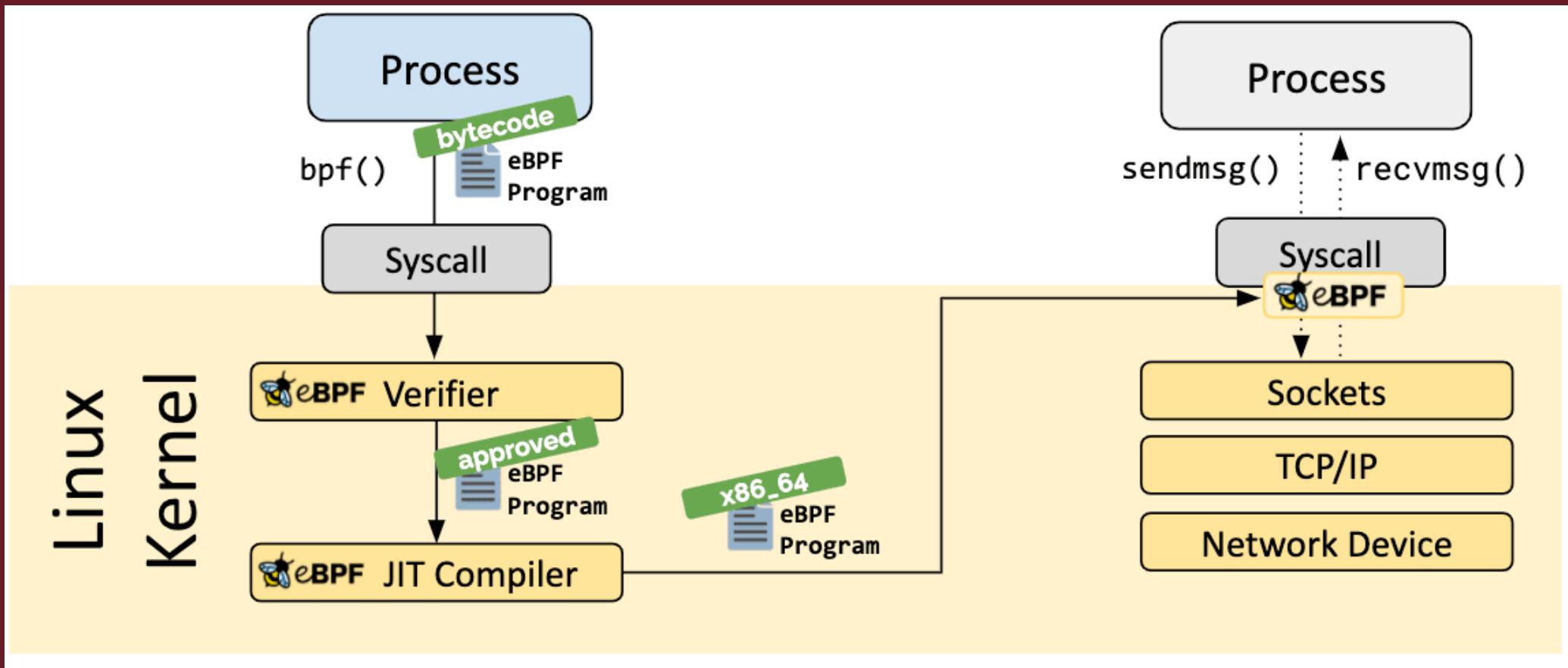
# Uretprobes (+demo)

- uprobes but for return values
- Automatically captures return values and execution time
- Uses kernel's return probe mechanism
- Example: trace malloc() calls and their return values to track memory allocation

# eBPF Verifier

- The verifier is a program that is used to check if the program is viable to run in the runtime
- Static analysis of eBPF bytecode before loading into kernel
- Key safety checks:
  - - *Bounds checking* - no out-of-bounds memory access
  - - *Termination* - programs must terminate (no infinite loops)
  - - *Register state* - all registers properly initialized
  - - *Pointer arithmetic* - safe pointer operations only

# eBPF Verifier



# Verifier Rejection Examples

- - error: back-edge from insn 2 to 0 - infinite loop detected
- - error: R1 invalid mem access 'inv' - invalid memory access
- - error: program too large - exceeds instruction limit

# bpftrace

- bpftrace is a high-level language that's JIT compiled and loaded into the verifier
- high-level, relatively simple syntax for writing eBPF programs
- Compiles to eBPF bytecode using LLVM
- Built on top of BCC (BPF Compiler Collection)
- Much easier than writing raw eBPF C code

# Basic bpftrace examples (+demo)

- Count syscalls by name
  - `bpftrace -e 'tracepoint:syscalls:sys_enter_* { @[probe] = count(); }'`
- Trace file opens
  - `bpftrace -e 'tracepoint:syscalls:sys_enter_openat { printf("%s opened %s\n", comm, str(args->filename)); }'`
- Function call frequency
  - `bpftrace -e 'uprobe:/bin/bash:* { @[probe] = count(); }'`

# Some built-in bpftrace variables

- pid - process ID
- comm - command name
- args - function arguments
- retval - return value (in uretprobe)
- str() - convert pointer to string
- hist() - histogram
- @ - map/associative array

# Debugging with bpftrace (+demo)

- `bpftrace -l` can be used to list available probes
- `bpftrace -l 'uprobe:/bin/bash:*`' - list all functions in bash
- `bpftrace -l 'kprobe:*tcp*`' - list all kernel functions containing "tcp"
- `bpftrace -l 'tracepoint:syscalls:*`' - list all syscall tracepoints

# Useful one-liners

- Trace all function calls in a binary
  - `bpftrace -e 'uprobe:/path/to/binary:* { printf("%s called\n", probe); }'`
- Trace function arguments
  - `bpftrace -e 'uprobe:libc:malloc { printf("malloc(%d)\n", arg0); }'`
- Trace return values
  - `bpftrace -e 'uretprobe:libc:malloc { printf("malloc returned %p\n", retval); }'`
- Stack traces on function calls
  - `bpftrace -e 'uprobe:/bin/bash:* { print(ustack); }'`

# Common tracepoints

- syscalls:sys\_enter\_\* - system call entry
- syscalls:sys\_exit\_\* - system call exit
- sched:sched\_process\_exec - process execution
- signal:signal\_deliver - signal delivery
- kmem:\* - memory allocation events

# IDA

- IDA needs no explanation, most widely used disassembler/decompiler
- Interactive DisAssembler - first released in 1990 by Ilfak Guilfanov
- Can be used for static as well as dynamic analysis, has a lot of useful plugins
- Supports 50+ processor architectures
- Python scripting interface for automation
- Hex-Rays decompiler produces pseudo-C code
- I want to talk about FLIRT, however, and the technology behind it



# FLIRT

- Fast Library Identification and Recognition Technology
- FLIRT is a technology used to create "signature" files for functions
- Part of IDA, but does not necessarily need full IDA to function
- Identifies library functions in stripped binaries
- Works by creating signatures of function prologs and other characteristics

# Open source FLIRT tools

- Some great people have already done the really hard work of an open-source parser for .idb and .i64 databases
- `flirt-rs` - Rust implementation of FLIRT
- `python-flirt` - Python tools for FLIRT signatures
- `r2flirt` - Radare2 FLIRT plugin
- Can extract signatures without full IDA license

# Signature Databases

- IDA ships with signatures for common libraries (libc, Windows APIs, etc.)
- Custom signatures can be created for proprietary libraries
- Community maintains signature databases for malware families
- Hex-Rays provides additional signature packs

# Idea

- eBPF can be used to debug literally anywhere in a program
- FLIRT can be used to tell you what the actual functions are in a stripped program
- Merge the two, and we have magic

# The combination

- Use FLIRT to identify known library functions in stripped binary
- Use eBPF to trace function calls during execution
- Combine traces with function names for readable execution flow
- Identify hot paths and critical functions automatically
- Focus reverse engineering efforts on the most important code



# DEMO TIME!

# Demo - Easy hello-world

```
#include <stdio.h>
#include <unistd.h>

int two() {
    printf("Hello, world!\n");
}

int main() {
    while(1) {
        sleep(1);
        two();
    }
}
```

# Demo - slightly harder malware

- The malware we're looking at is xmrig

# CAPE

- CAPE (Config And Payload Extraction) is a fork of Cuckoo Sandbox focused on automated malware analysis and payload extraction
- Built on top of Cuckoo's foundation but adds advanced unpacking and configuration extraction capabilities
- Designed specifically for Windows malware analysis but has been extended to support Linux environments
- Maintains Cuckoo's agent-based architecture while adding more sophisticated analysis modules

# Final Demo (with CAPE!)

Quick Overview Behavioral Analysis Detailed Behaviour (Tracee) FLIRT Analysis Network Analysis Dropped Files (1) Compare this analysis to...

### FLIRT Analysis Summary

**63**  
Total Functions

**63**  
FLIRT Functions

**42**  
Total Calls

**3**  
Functions Called

#### FLIRT Signature Coverage

100.0% of functions have FLIRT signatures

#### Most Active Functions

#	Function	Calls	Address
1	j_.free_0 <span>FLIRT</span>	22	0x5d54c0
2	j_j_.free_0_8 <span>FLIRT</span>	18	0xe9cb0
3	j_j_.free_0_12 <span>FLIRT</span>	2	0xec310

# Future work

- Currently, this works well for C,C++ binaries with <100 unknown functions.
- Will need to debug how to make it work for ~1000 probes
- Figuring out ways to re-add debugging info (DWARF) for ELF binaries for better analysis and more capabilities with eBPF

# Thank you!

[hello@nischay.me](mailto:hello@nischay.me)

<https://www.linkedin.com/in/thatloststudent>

<https://www.twitter.com/thatloststudent>

<https://infosec.exchange/@thatloststudent>

# eBPF gives you superpowers

