1 Slide 1

- Introduce myself
- Here to talk about eBPF
  - all the crazy things you can do with it
  - how it can be a crazy powerful tool for OS security
  - demonstrate this using ebpH, a host-based anomaly detection system I built with eBPF

2 Slide 2

- At a high level, eBPF is a way of injecting user-specified code into the kernel
- With the idea being that this code runs in kernelspace and can be used to instrument essentially all system behavior
  - that means we can see all userspace and all kernelspace events, all at the same time
- The key advantage of eBPF over traditional methods such as LKMs is this idea of safety
  - BPF programs are guaranteed not to crash the kernel or damage a running system
  - they have a verifier for this, which I will explain shortly

3 Slide 3

- Before I talk about security-specific use cases, I want to touch on how pervasive this technology already is in industry
- Lots of big data companies are already using eBPF for performance monitoring, debugging, optimization
  - this is happening in production and at scale
  - companies like Netflix, Facebook, Google, many more
  - creator of eBPF actually works at Facebook
- There is also an established set of tools that cover most basic use cases
  - bcc (sponsored by the Linux foundation) has over 100 of them
  - monitor everything from cache hits/misses to TCP connection requests to setuid system calls
- There's also a really important example of eBPF being used for security in production
  - Cloudflare's DDoS mitigation stack is now largely based on eBPF

4 Slide 4

- Now that you know a bit about the kinds of things you can do with eBPF, we need to talk about how this relates to security
- I think this really starts with the notion that a lot of security is about what you can see
  - if an attacker can break into a system and we don't know about it until it's too late, they've gotten away with it
  - remember, eBPF lets us see everything about our system
  - that means every system call, every login, every scheduler operation, every packet, all at the same time
  - and it can do this with crazy low overhead
- Before eBPF, this kind of system introspection came at a cost
  - generally you have a trade-off between speed, scope, or production safety
  - library call interposition, kernel modification, ptrace is just terrible (REALLY slow, up to 44,200%)
- eBPF can do everything without having to deal with this speed / scope / safety trade-off

5 Slide 5

- Up until now I've just been handwaving and making all these magical claims about eBPF, so here is how it actually works
- We start in userspace where we first need to generate some BPF bytecode to send off to the kernel
  - originally this was done by hand, but now we have fancy LLVM backends that can convert C into BPF instructions
- Once we have our bytecode, we send it off to the kernel using the BPF system call, and this system call immediately traps to the eBPF verifier
- The verifier checks our code for safety and will either immediately reject our code or accept our code, at which point it moves onto a JIT compiler which converts it in real time to machine instructions
- Once the BPF program is running, there are a lot of different things that we can use it to instrument
  - userspace functions
  - kernel functions / data structures
  - hardware performance counters
  - networking stuff
- Everything is event based, so this BPF program is being run every time a specific event occurs

- All BPF programs can store data in maps
  - BPF programs can use maps to interact with each other
  - And maps can be used to interact with userspace (via the BPF system call)
  - The important thing to understand about these maps is that we can control the amount of context switches we need to do, which is a lot of where BPF's performance comes from, over something like Dtrace

6 Slide 6

- The verifier's job is to make sure that BPF programs do not crash the kernel
- It can be considered part of the kernel's reference monitor
- 10,000 lines of C code, pretty complex
- BPF system call traps to verifier on every PROG_LOAD
- How can you guarantee safety?
  - limitations
  - simulation
  - static analysis
- 512 byte stack space
- No unbounded loops
- Hard limit of 4 million BPF instructions per program
- No buffer access with unbounded induction variables

7 Slide 7

- Even with all these limitations, BPF programs can still be quite complex
- The figure on the left is an instruction flow graph of one of ebpH's BPF programs
- Generated with bpftool and graphviz osage
- Just over 1500 BPF instructions, JIT compiled to just over 1900 machine instructions
- A lot of additional complexity comes from maps + tail calls

8 Slide 8

- Before I talk about my anomaly detection system, ebpH, I'll talk a bit about what came before it
- pH or Process Homeostasis was an early anomaly detection by Anil
- The idea was to instrument system calls and build behavioral profiles
  - Delay anomalous system calls proportionally to recent anomalies
- Implementation had a few key problems
  - The stuff that pH was doing is basically impossible to do outside of patching the kernel directly

- Crazy modifications like patching the scheduler, writing new assembly code, etc.
- The key points here: not production safe, certainly not portable

9 Slide 9

- ebpH is a portmanteau of Extended BPF and Process Homeostasis
- The idea is taking this 20 year old technology and re-writing it using modern technology
- Table below compares the original pH and the current version ebpH in some important categories
- (NEXT SLIDE)

10 Slide 10

- Firstly, I want to point out that ebpH solves the problems of portability and production safety
- Two innate properties of BPF
  - perfect forward compatibility (we generally only add features, don't take them away)
  - verifier's safety guarantees
- (NEXT SLIDE)

11 Slide 11

- Note that pH actually has two properties that ebpH currently does not satisfy
  - Low memory overhead and a response component (the system call delays)
- Future work covers this, hopefully I'll be able to convince you that future versions of ebpH will be able to satisfy all of my design goals

12 Slide 12

- Basic idea of ebpH is the same as pH
  - Trace system calls
  - Build profile of lookahead pairs
  - Gather enough data to normalize a profile
  - New lookahead pair entries in a normal profile are considered anomalies
- eBPF makes this safe
  - We are doing this instrumentation in a new way
  - The key difference here is data collection

- Figure on the left shows an example lookahead pair from the ls binary
  - (read, close)

13 Slide 13

- ebpH uses three BPF program types to collect data
- most important is tracepoints
  - instrument system calls
  - instrument the scheduler for process creation and profile association on execve
- kprobes for dynamic instrumentation of signal delivery
  - this allows ebpH to be signal-aware, which helps it deal with non-deterministic behavior from signal handlers
- uprobes
  - instrument userspace shared library called libebph
  - allows userspace to issue more complex commands to the BPF programs
  - on top of direct map access like we have seen before

14 Slide 14

- Daemon manages the lifecycle of all of ebpH's BPF programs
  - loads the BPF programs when it is started
  - BPF programs are invoked every time specific events occur (system calls, process creation, binary loading, signals, etc.)
  - continually polls buffered events from the BPF programs (e.g. anomalies, profile creation, etc.)
  - it also has direct access to all of the BPF hashmaps
- Profiles are stored persistently on disk, log files are kept which describe all ebpH events in detail
- Daemon exposes an API to other userspace programs through a UNIX stream socket
  - this socket is owned by root and is only readable and writeable by root

15 Slide 15

- I have made a lot of claims so far about eBPF's performance, so let's see how ebpH compares with the original pH implementation
- Three distinct benchmarks
- lmbench suite to test
  - basic system call overhead

  - process creation overhead
  - IPC overhead (signals, UDS, pipes) (won't be covering this category)
- kernel compilation benchmarks
  - how does ebpH affect the overhead of real, complex tasks
- bpfbench
  - wrote my own ad-hoc benchmarking tool in eBPF
  - uses BPF programs to instrument latency on system call execution
  - advantage: monitor entire system (see most frequent system calls in practice and measure their overhead)

16 Slide 16

- bpfbench
  - three distinct systems, three distinct workloads
- micro benchmarks
  - bronte only (besides micro benchmarks, bronte was idle)

17 Slide 17

- getppid
  - null system call (almost no kernelspace runtime)
  - essentially involves a namespace lookup + a quick data structure access
  - this shows the worst case for performance overhead
  - about 614%, which seems like a lot, but drops off quickly
- stat(2)
  - more kernelspace runtime
  - this has much better overhead, 65%
- select(2) blocking system call, approaches best case
  - blocks until property is true on one or more open file descriptors
  - overhead as high as 99%, as low as only 2%

18 Slide 18

- process creation
  - three test cases, from least to most complex

- fork + exit creates a new process and exits immediately (this almost never happens in the wild)
- fork + execve creates a new process and executes a simple hello world program (this is probably most common)
- fork + /bin/sh -c same as fork + execve except uses the shell to execute the hello world program
- highest overhead is only 10%, which is acceptable in practice

19 Slide 19

- kernel compilation benchmark shows the most impressive result
- the idea was to test a very CPU-intensive task
- involves a lot of userspace time (which ebpH doesn't really affect), but still many system calls
  - tested how many -> 176 million
- ebpH imposes only 10% kernelspace overhead
- the total overhead is under 1%
- this actually shows ebpH outperforming the original pH (kernel implementation)

20 Slide 20

- bpfbench results
- a lot of data, I will summarize for the sake of time
- looked at top 20 system calls by count across three datasets
- overhead from about 5% to 150%, acceptable in practice
- non-idle systems had longer system calls as most frequent
  - which translates to lower overhead where it really matters

21 Slide 21

- in summary:
- ebpH can impose significant overhead on some system calls
  - but this isn't the whole story
  - longer system calls means less overhead
  - systems doing real work tend to invoke longer system calls more frequently
  - system call overhead is not necessarily indicative of tangible performance impact
- ebpH does VERY well on real tasks
  - outperforms the original pH
  - slowdown is imperceptible in practice

22 Slide 22

- we have seen the eBPF can be very effective in data collection
- there are some new additions to eBPF that can make it very effective at responding to attacks
- in Linux 5.3, we got bpf_signal
  - send signals to processes in real time from kernelspace
  - if a signal is coming from the kernel, it happens instantaneously
- Linux 5.5, bpf_signal_thread
  - like bpf_signal, but we now control which thread receives the signal
- Linux 4.16, bpf_override_return
  - targeted error injection on whitelisted kernel functions
  - block system calls from completing successfully

23 Slide 23

- now we have the tools to make ebpH respond to attacks
- send SIGSTOP and SIGCONT for system call delays
- use bpf_override_return to stop anomalous execves
- recall table 1 from the beginning of the talk

24 Slide 24

- I also want to touch on ways that ebpH's memory overhead can be improved
- currently, one big map for processes, one big map for profiles
  - this is way too granular
- original pH used a linked list to keep track of processes and profiles
  - dynamic allocation in the traditional sense is not possible in BPF, needs to be done through maps
- new BPF map types can help make things more fine-grained
  - LRU_HASH enables us to use a smaller map that discards least recently used entries
  - HASH_OF_MAPS nested maps for the sparse array of lookahead pairs, achieve something like dynamic allocation

25 Slide 25

- what other types of security problems can we solve with BPF?
- anomaly detection
  - no reason to stop at system calls
  - ebpH only uses a very tiny subset of eBPF's functionality
  - there are over 1,800 tracepoints alone in Linux 5.5, ebpH uses five of them
- DDoS mitigation and network intrusion detection
  - this is already being done
  - Cloudflare, ntopng, etc.
- Increasing visibility of attacks and misuse
  - ebpH does a bit of this
  - bcc tools are great at this
  - capable
    - monitor every time a capability is used (CAP_SYS_ADMIN, CAP_NET_BIND_SERVIC etc.)
  - eperm
    - monitor every eperm error (useful for constructing policies)
  - setuids
    - monitor all calls to setuid
  - execsnoop
    - monitor every execve system call
  - many others

26 Slide 26

- something I've been thinking about recently is using eBPF to enforce sandboxing rules
  - something like seccomp, but enforce rules externally, totally application transparent
  - bpf_signal could do this easily
- the main takeaway from this talk should be as follows:
  - name anything you want to trace
  - eBPF can do it and it can do it safely and with excellent performance
- ebpH is just the beginning

27 Slide 27

- in conclusion
- ebpH is as fast or faster than the original kernel implementation
- it supports most of the original functionality, and future versions will support all of it
- we can expand on ebpH to leverage more system data with eBPF
  - I envision a whole ecosystem of BPF programs, monitoring various aspects of system state, sharing information with each other
  - take things beyond system call tracing and truly begin to emulate natural homeostatic mechanisms
- the future of eBPF in OS security is bright
  - we are going to see a lot more of this kind of thing
  - we are just at the beginning, and eBPF gets better with every subsequent kernel update

28 Slide 28

- any questions?