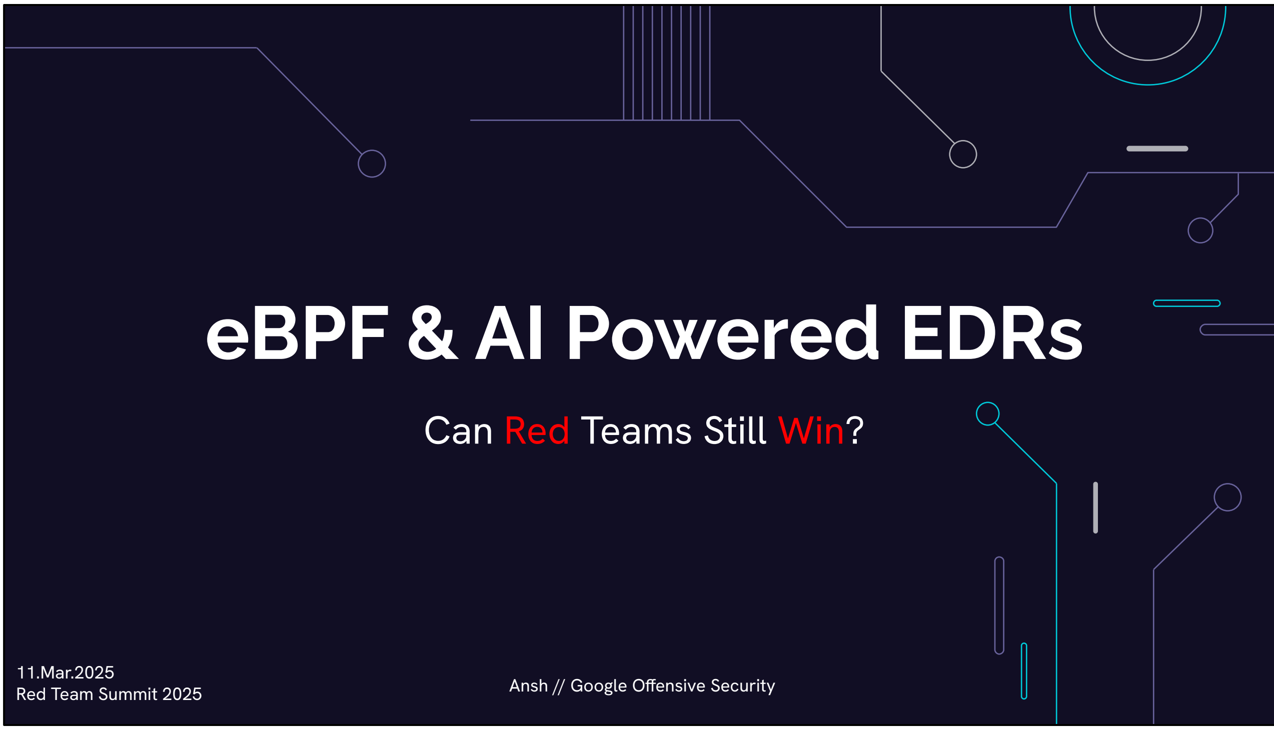
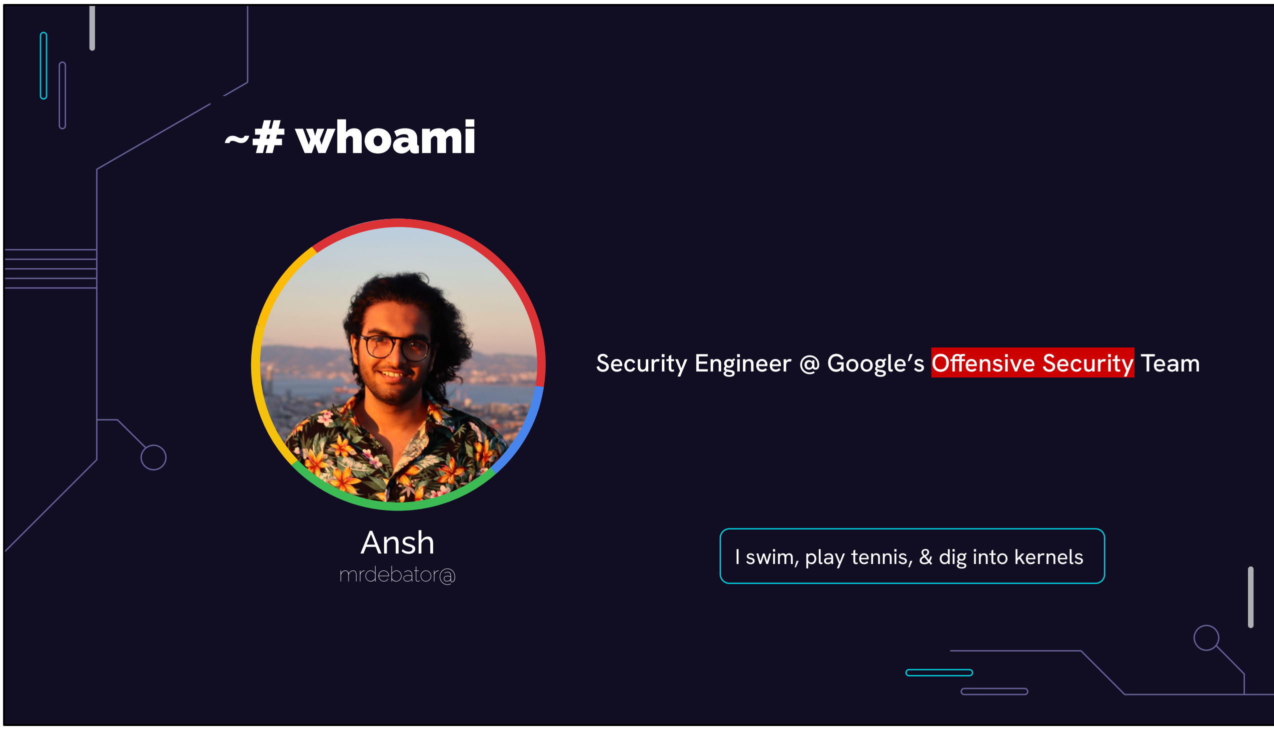


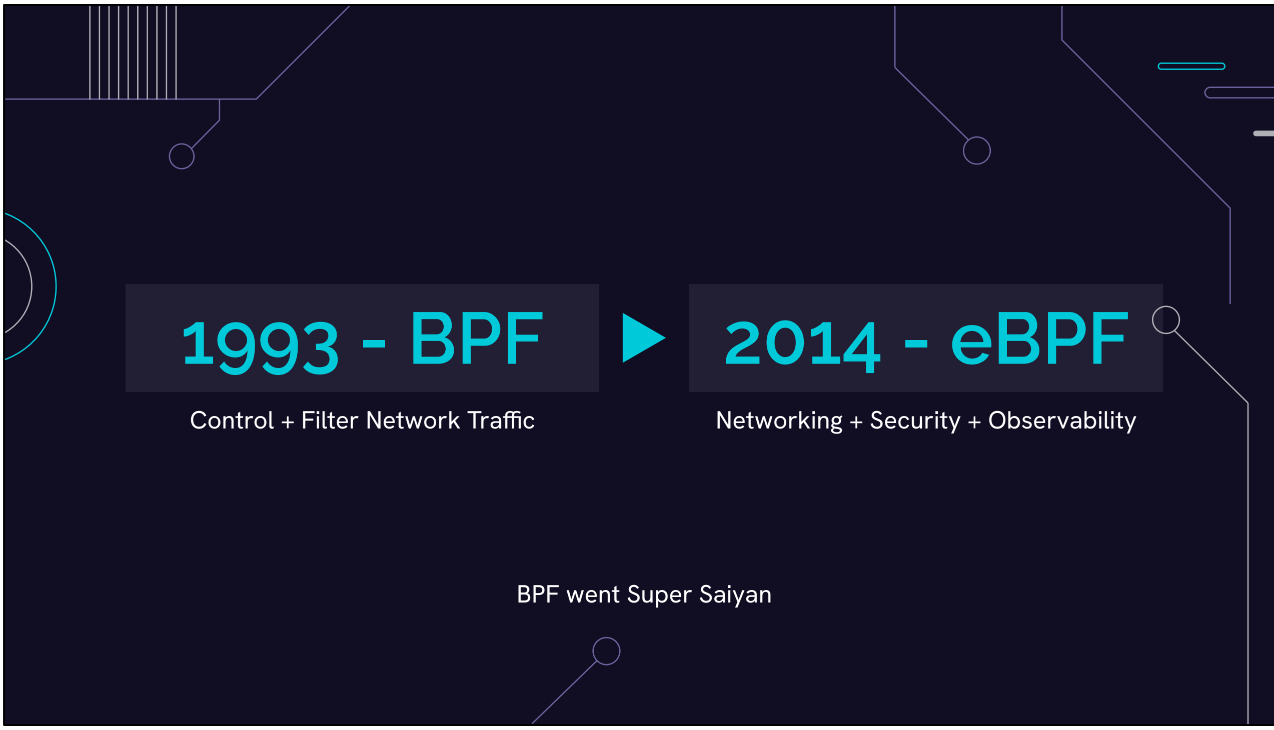
Before we get started, let's have a show of hands, how many of you have worked with or are familiar with the term eBPF / BPF?

Alright, let's do a brief recap on eBPF, explore how it's used in EDRs, lessons I learned creating my own system monitor, and how we as attackers may still be able to defeat the combination of eBPF and EDRs.

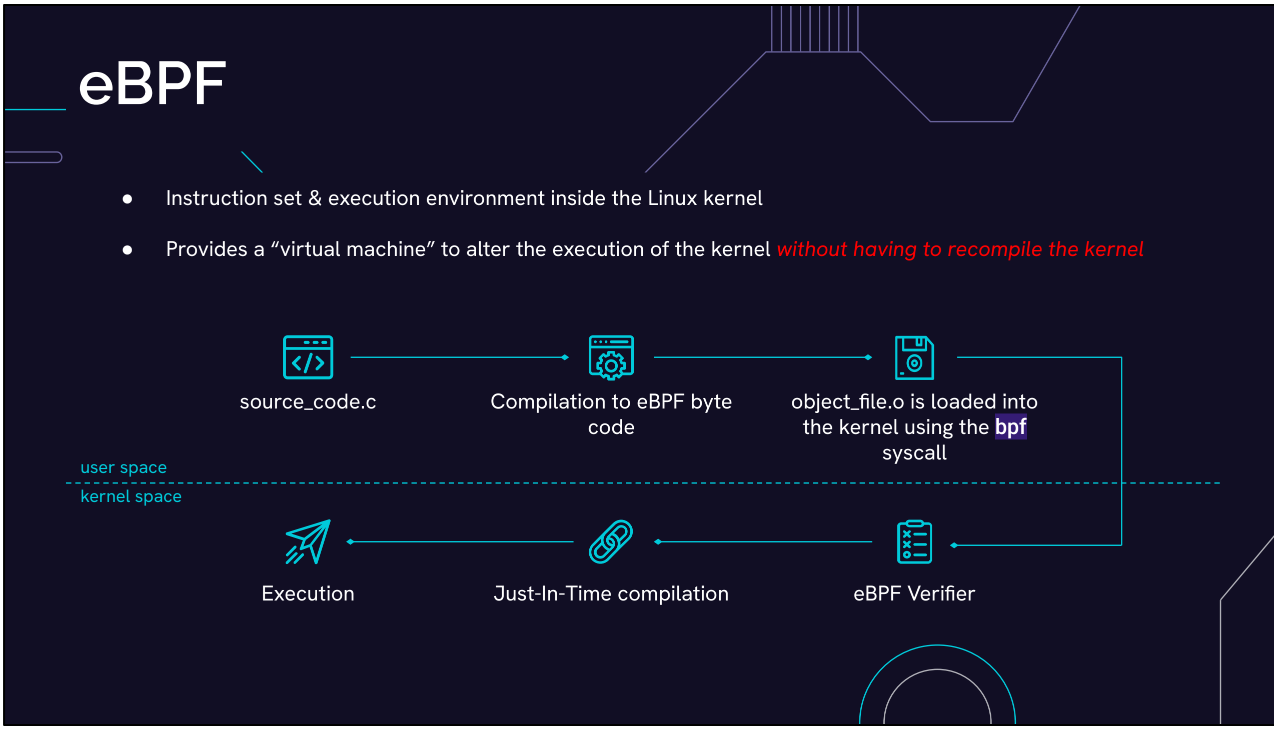


Hey everyone, my name is Ansh. I'm a Security Engineer in Google's Offensive Security team. When I'm not exploring low level system intricacies, I like to cook and play a few sports.

Let's talk about our "frenemy", Endpoint Detection & Response, we need it to keep us secure, but it's also the very thing that makes our jobs harder. And now, eBPF is giving EDR solutions more visibility into our actions than ever before.



In 1993, the Linux kernel added the Berkeley Packet Filter to help filter and control network traffic. About 30 years since, BPF effectively went Super Saiyan and added so many features that we now call it the “extended” Berkeley Packet Filter. These newer features can be used for networking, security, monitoring, and a lot more. Let’s take a look…



1. eBPF allows you to run sandboxed programs within the Linux kernel without modifying kernel source code or loading kernel modules. This capability opens the door the extremely powerful and efficient system observation.
2. It provides various hooks that allow you to trace events. Think about it like a programmable network of sensors and filters inside the kernel – now whatever program you write, takes advantage of these preexisting hooks, and you can set telemetry and actions based on chosen events.
3. An eBPF program starts out as a "restricted" C program. Compared to a regular C program, things like stack size, instruction count, loops, and available functions are limited to protect the operations of the kernel.
4. Once compiled into bytecode and loaded into the kernel, the program is verified by running a Depth-First Search to parse program instructions into a Directed Acyclic Graph. This primarily ensures that program termination must be guaranteed by the program, so no backwards jumps, unbounded loops, or unreachable functions.
5. Then, we wait, the kernel's JIT compiler will execute the program when the relevant event type is triggered.

The benefit of not having to load a kernel module and a dedicated verifying vetting your program, is that in the worst case scenario, the eBPF probe just doesn't load, it doesn't take down kernel operations.

```c
#include "vmlinux.h"
#include <bpf/bpf_helpers.h>
#include <bpf/bpf_tracing.h>
#include <bpf/bpf_core_read.h>

char LICENSE[] SEC("license") = "Dual BSD/GPL";

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;
}

SEC("kretprobe/do_unlinkat")
int BPF_KRETPROBE(do_unlinkat_exit, long ret)
{
    pid_t pid;

    pid = bpf_get_current_pid_tgid() >> 32;
    bpf_printk("KPROBE EXIT: pid = %d, ret = %ld\n", pid, ret);
    return 0;
}
```

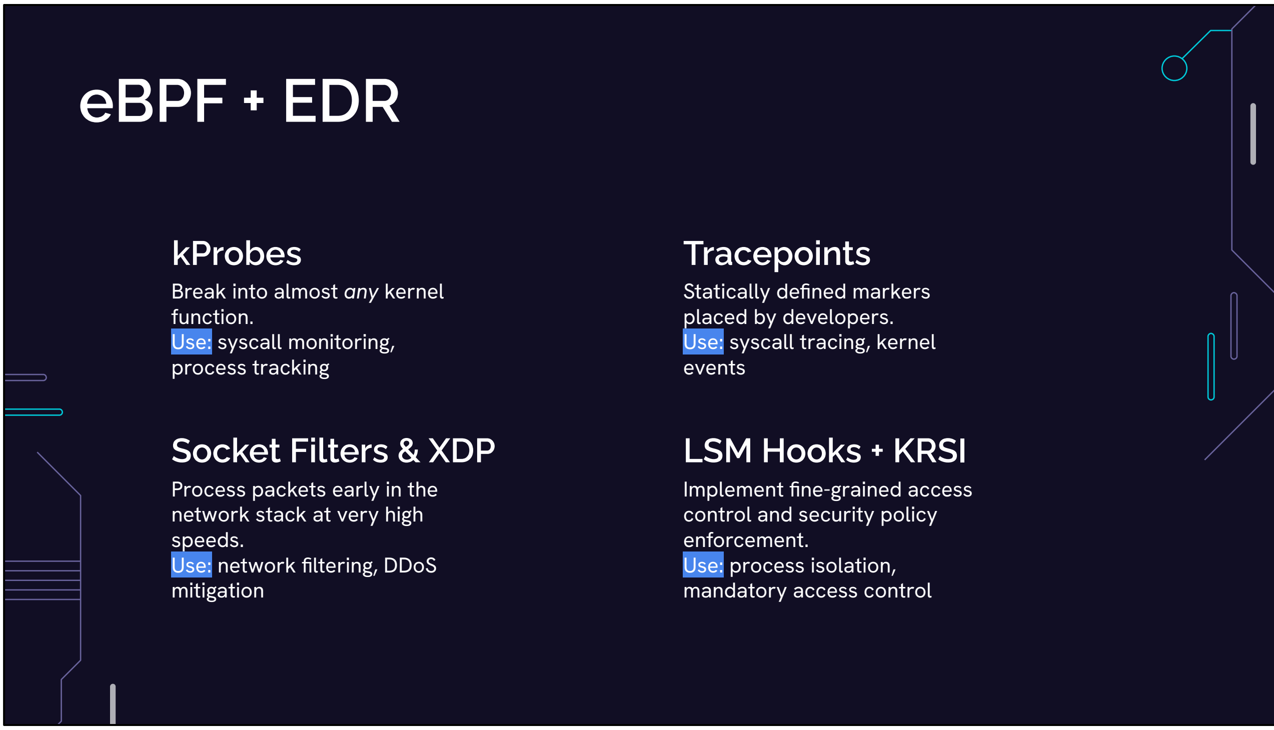
github.com/eunomia-bpf/bpf-developer-tutorial

Here's an example of a simple eBPF program used to monitor and capture the "unlink" system call executed in the Linux kernel. The unlink system call is used to delete a file. The program traces the system call by placing hooks at the entry and exit points of the "do_unlinkat" function.

There are two operations this program performs with two distinct probes

The kProbe is triggered when the function is entered. It takes two parameters: dfd (file descriptor) and name (filename structure pointer). We retrieve the PID of the current process and then read the filename. Finally, we use the bpf_printk function to print the PID and the filename in the kernel log.

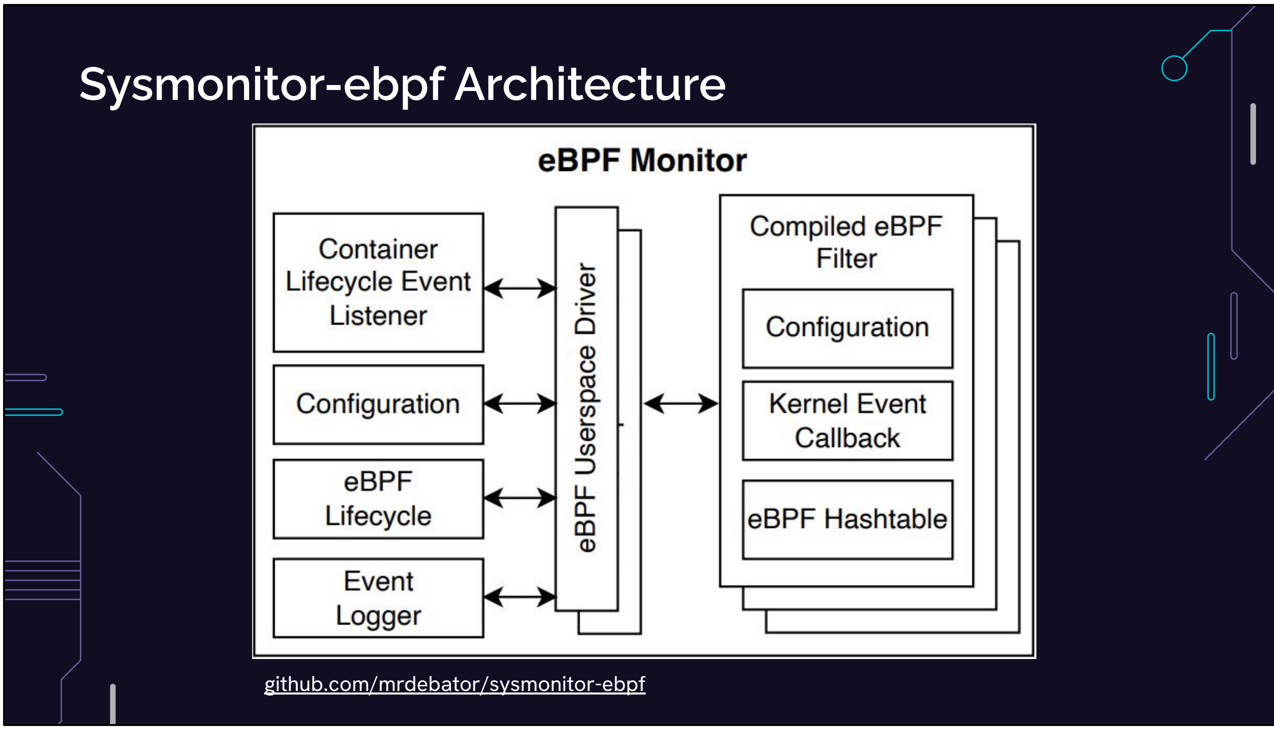
The kRetProbe is triggered when exiting the function. Its purpose is to capture the return value of the function. Once again, we capture the PID and print the PID and return value into the kernel log.



In the previous slide, we looked at a program that could hook onto kernel functions. This was done using a probe type called kProbe.
eBPF provides us with a couple of different kinds of probes. The most commonly used probes for EDRs are the following:

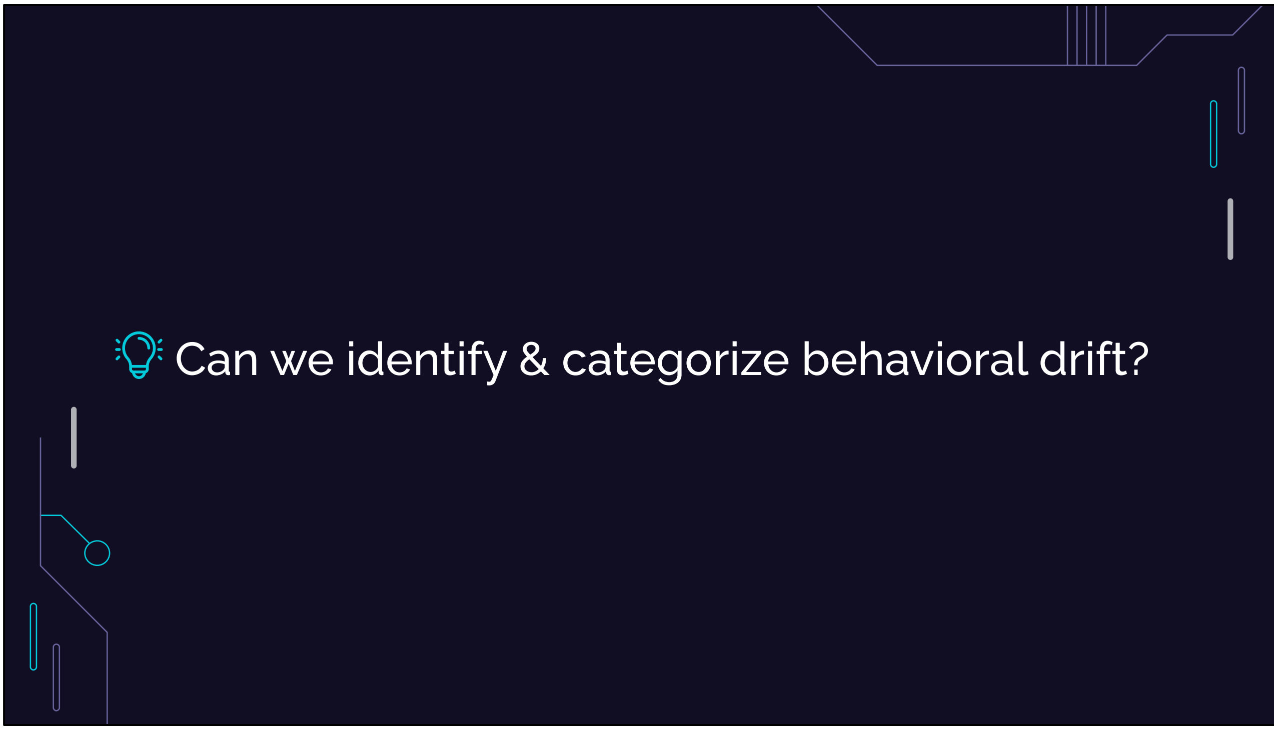
- kProbes help tap into almost any kernel function, on entry, or on return. They're useful for syscall monitoring and process tracking.
- Tracepoints are part of the kernel's stable API and are defined markers placed there by the developers. They're used to trace syscalls, kernel events, and scheduler events.
- Socket Filters and XDP are both network probes that allow network filtering and DDoS mitigation. Use these with caution as you'll be dealing with packets very early in the network stack.
- LSM + KRSI: Traditional LSMs allow admins to enforce mandatory access control policies by executing a security module when a LSM hook is triggered. The Kernel Runtime Security Instrumentation project, created by Google, essentially uses hooks provided by LSMs with the flexibility and safeguards of eBPF. Google's KRSI provides a more granular and adaptable approach compared to traditional LSMs. **The key is that it's built on top of the LSM framework, giving it the power to enforce policies, not just observe.**

These probes allow defenders to receive an extraordinary amount of telemetry with minimal latency and overhead.

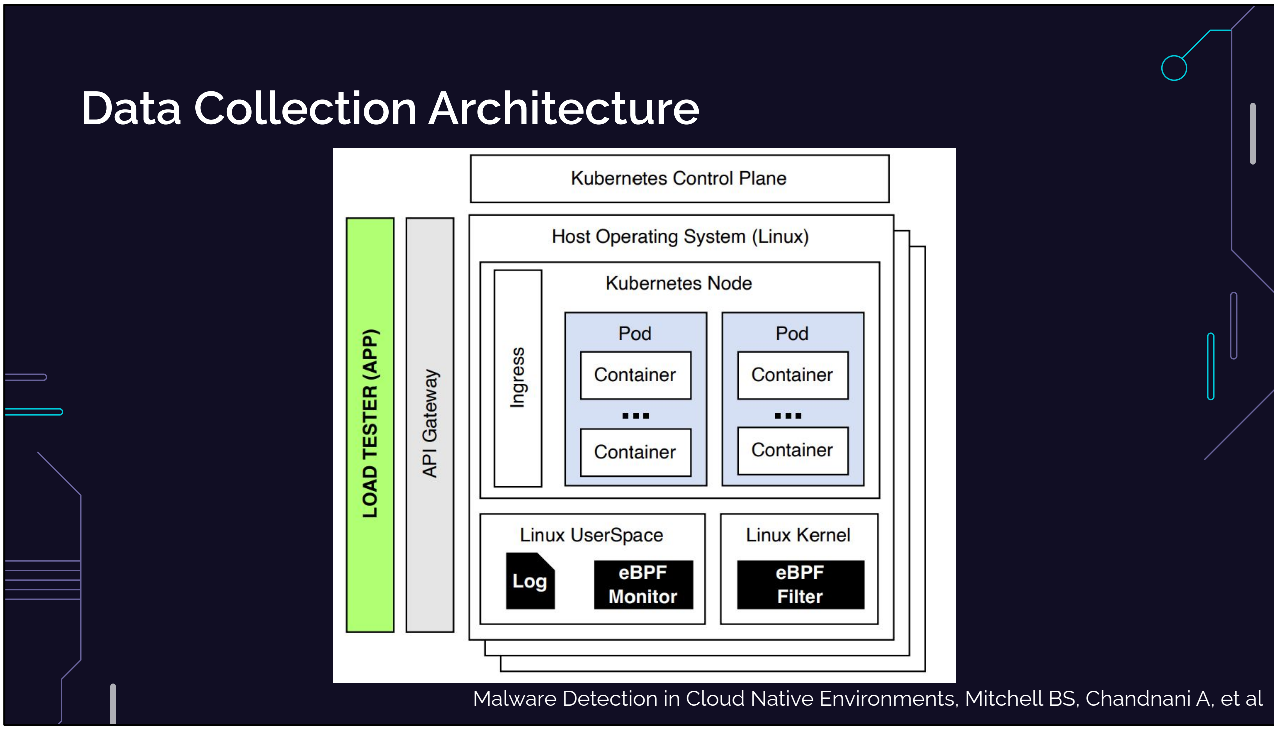


So about 6 months before I completed my undergraduate degree, I set out to better understand what eBPF can really do and how EDRs use it, by writing a systems monitor to aggregate telemetry in cloud native environments from within the kernel. The tool, sysmonitor-ebpf is publicly available if you want to check it out.
My professor and I used tracepoints to isolate the individual containers in the environments by their namespaces and trace the system calls that occur in each container.

Generally, this is an effective alternative to container monitoring solutions that use a conventional sidecar architecture and is used by commercial providers like Falco, Cilium, and Solo.io



Now that we have unparalleled telemetry, a question we posed to ourselves was, can we use this to identify and categorize behavioral drift?

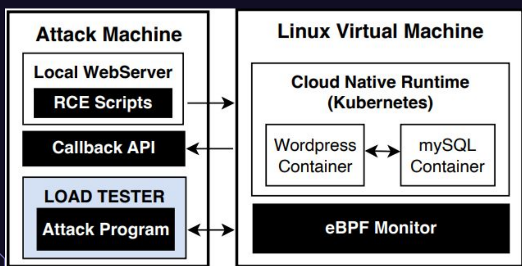


Let's get some data to try and answer this question.

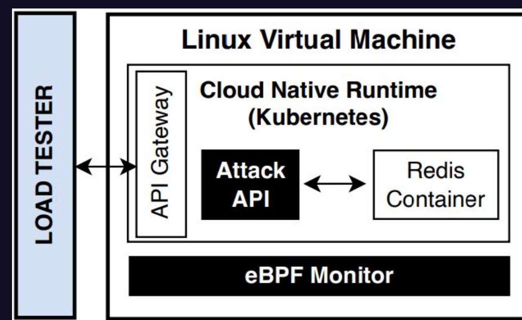
The objective was to gather clean baseline logs for benign and malicious behavior, i.e. regular usage vs active exploitation of the vulnerabilities. This was the architecture we used.

The host underlying our Kubernetes cluster was loaded with our system monitor and user behavior was emulated using automation scripts and a load tester called Locust.

Data Collection Architecture



Attack Strategy, Wordpress CVE-2016-10033



Attack Strategy, Redis CVE-2022-0543

Using the aforementioned architecture, I created two controlled testing environments weakened with known Remote Code Execution (RCE) vulnerabilities in Redis and Wordpress.

eBPF + EDR + AI

Case Study Configuration	Total Events		Total Syscalls		Classifier Results		
	Vulnerable	Patched	Vulnerable	Patched	AUC	F1 Score	Accuracy
Redis Exploit	24,801	8,979	12,277,890	2,050,209	1.0	1.0	1.0
Redis Hybrid	24,744	9,516	7,575,207	1,710,114	1.0	1.0	1.0
WP Exploit	11,134	8,316	4,858,726	3,414,846	1.0	0.98	0.98
WP Hybrid	11,022	8,296	5,116,830	3,772,503	1.0	1.0	1.0

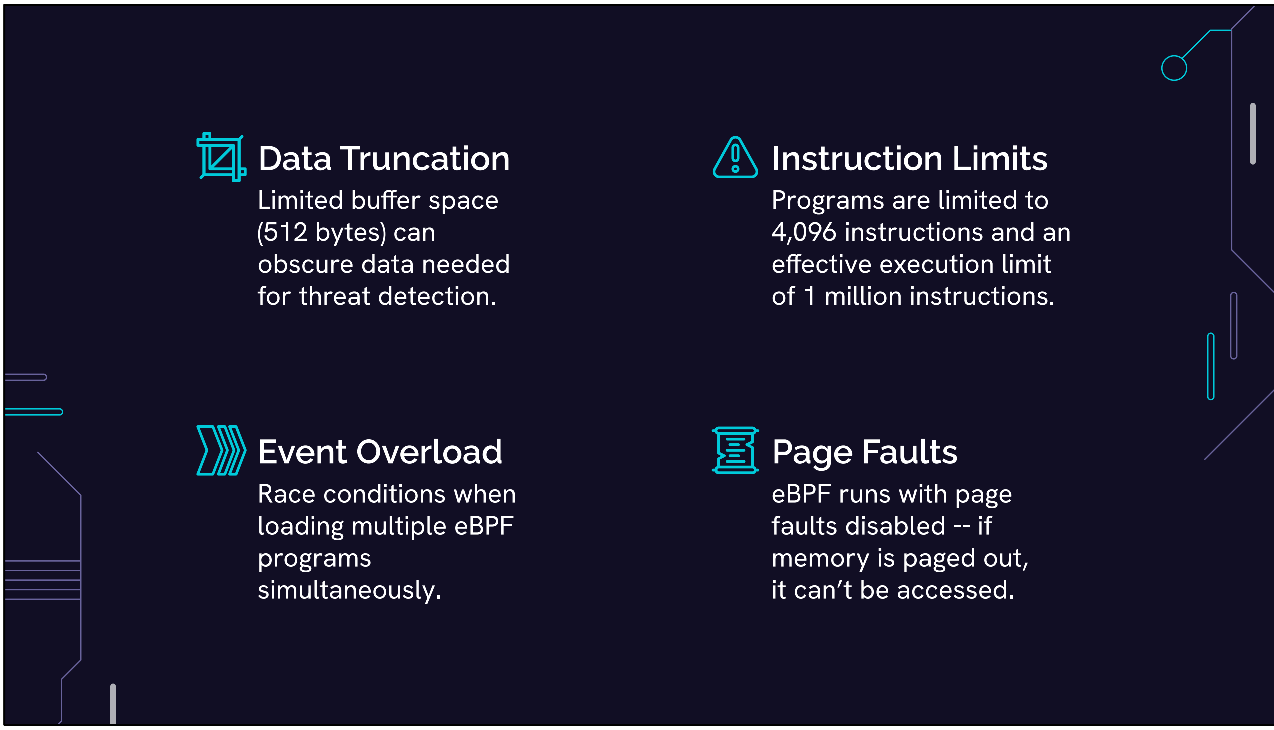
The “exploit” tests you see on each system involved all emulated users attempting to execute attacks on both the vulnerable and patched versions of the instances.

The “hybrid” test involved the emulated users conducting a blend of randomized benign and malicious behavior against both systems.

We trained a voting ensemble classifier that yielded extremely positive results for our Proof-of-Concept

Eight datasets were collected in total. Two datasets were collected for each row – one captured when the vulnerable version was deployed, and one for the patched software. The classifier results were averaged over three separate runs.

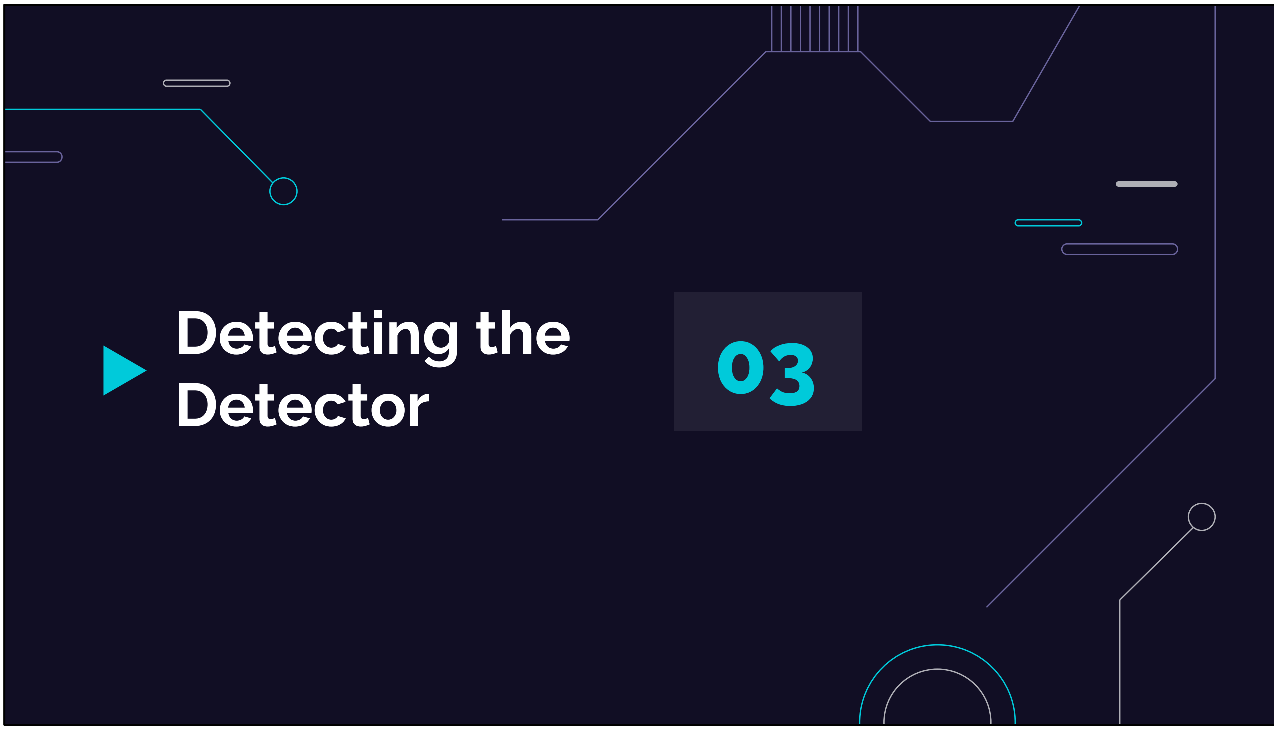
But while eBPF may seem like a silver bullet for monitoring and threat detection, there are some inherent limitations. Remember, eBPF was never built for security.



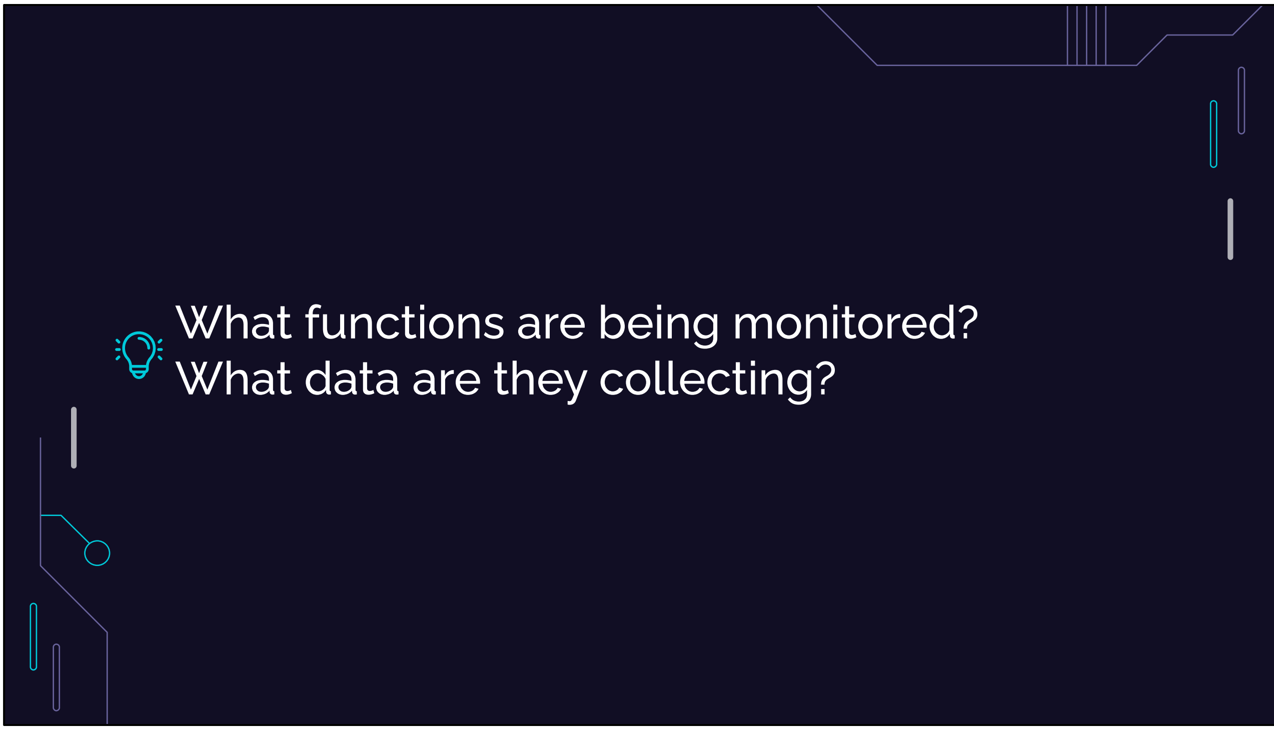
eBPF has some key limitations that impact developers and may benefit attackers alike. I encountered all these limitations while building and testing my tool:

- **Data Truncation:** eBPF's stack space is limited to 512 bytes. When writing code, be mindful of how much scratch data you use and the depth of your call stacks. For instance, 512 bytes is less than the longest permitted file path length of 4,096 bytes.
- **Instruction Limits:** An eBPF program can only have 4,096 instructions, and reusing code (by defining a function) isn't possible. Until recently, loops weren't supported. Now they are, with some guardrails.
  - **This prevents arbitrary long monitoring programs from bring written.**
- **Event Overload:** Because eBPF lacks concurrency primitives and a probe can't block the event producer, an attach point can be easily overwhelmed with events. This can lead to:
  - Missed events (kernel stops calling the probe)
  - Data loss (due to lack of storage)
  - Data loss (due to complete overwriting of older data)
    - A notable encounter with this issue was while processing streams of system calls to log various behaviors on Cloud Native systems, I ended up needed a ring buffer of size 81,920 bytes in user space to handle the sheer number of events coming my way.
  - Data corruption (from partial overwrites or complex data formats)

- **Page Faults:** For various reasons eBPF runs with page faults disabled, this becomes bad news for security monitoring tools if relevant details are paged out.

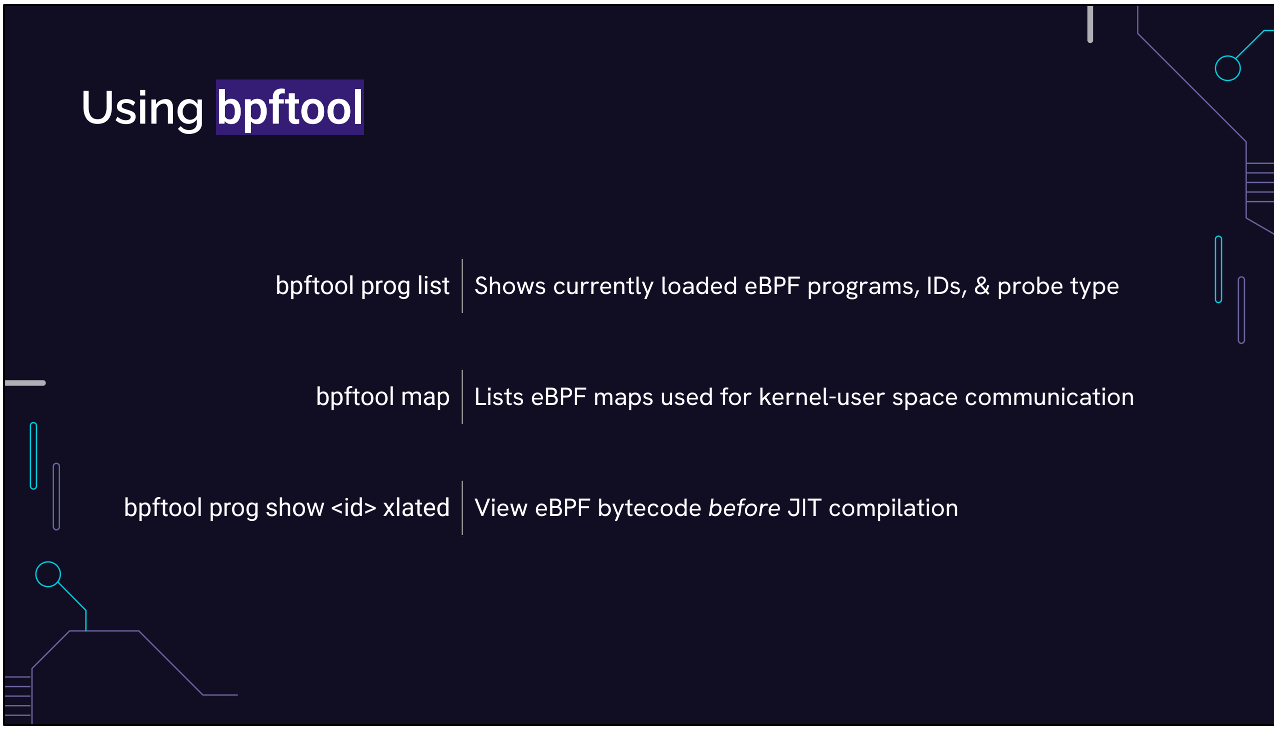


Now that we have a better understanding of eBPF, let's look at footprinting what kinds of eBPF detections may be running on a system.



When footprinting eBPF detectors, ask yourself:

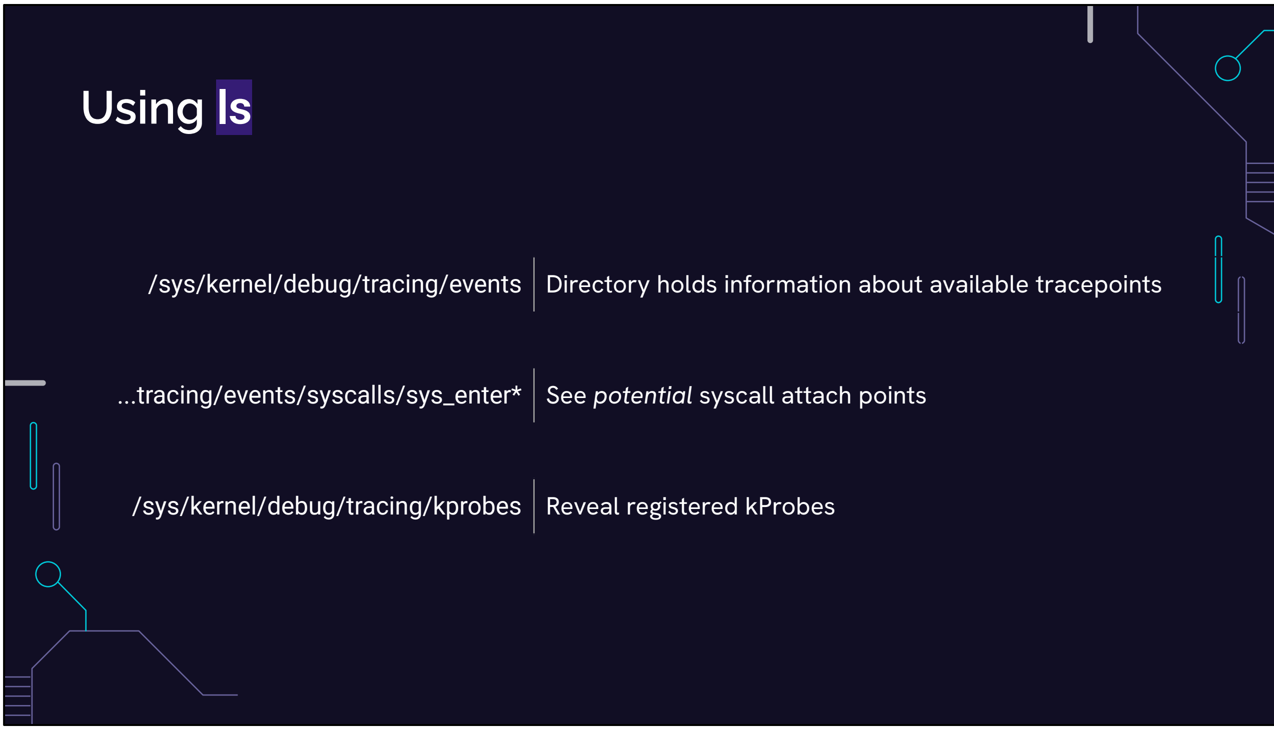
- What functions would be monitored?
  - execve for process creation
  - openat for file access
  - socket for network connections
  - ptrace for process tracing
- What data are they collecting?
  - How is this data being communicated?
  - Can we examine the eBPF maps used to communicate between kernel and user space?



Bpftool is the primary command utility for all things eBPF. Contingent on the nuances of your code execution, here are three commands you can use to better understand what you're up against.

1. First, try listing the loaded eBPF programs, this will also shed some light on what probe types are being used
2. Check out the various eBPF maps currently in use. BPF Maps are used to communicate between kernel and userspace components of monitoring softwares. This should tell us what kind of telemetry is being tracked.
3. Finally, we can also retrieve the bytecode of the program and attempt to reverse it at our convenience.

bpftool prog unload <id>

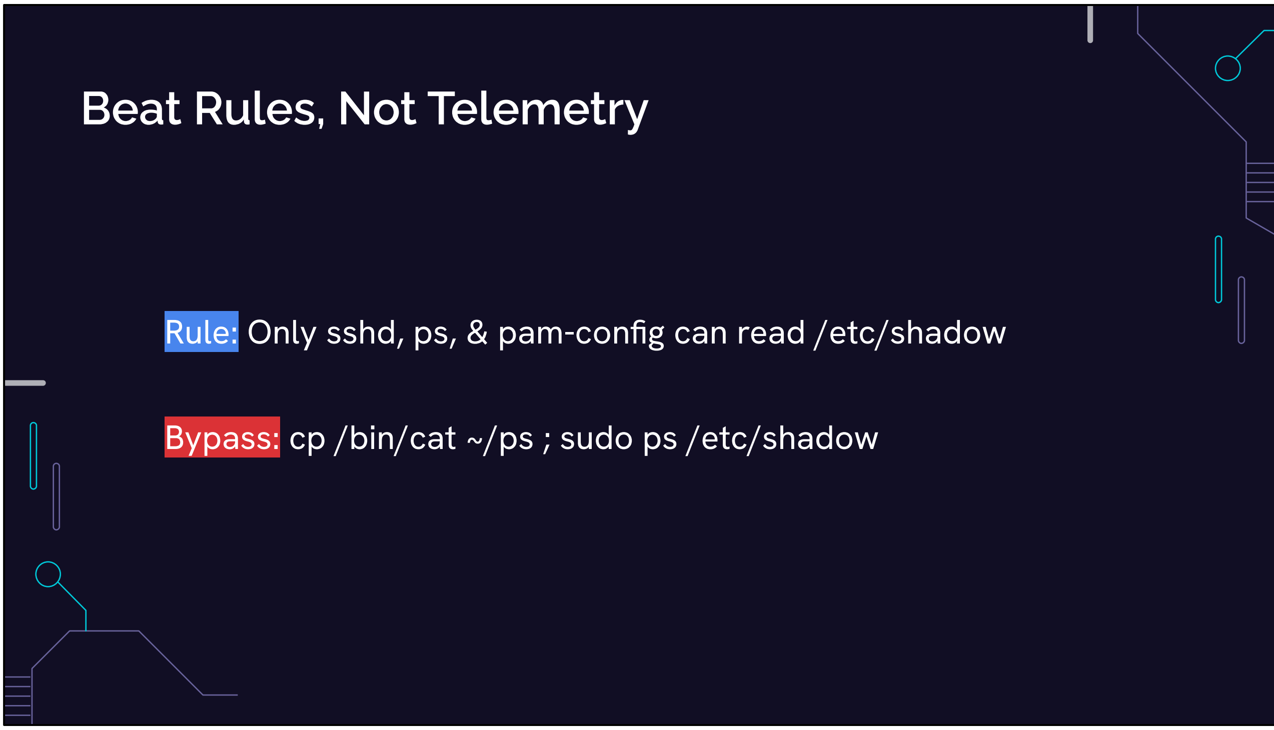


A safer, yet more manual method is to simply use the `ls` command.
The /sys/kernel/debug directory and its subdirectories hold information registered probes

For example, listing the first directory reveals information about available tracepoints
You can take this a step further and try to list potential attach points for system calls as well

Similarly, we can list out kProbes, uProbes, and more.

We've found our foothold into the world of eBPF EDRs, let's take it further.



Ultimately, our battle is with what agents do with gathered telemetry, not from gathering telemetry in the first place.
Your telemetry can be top notch, but if enforced with ambiguous rules, we can craft a bypass nonetheless.

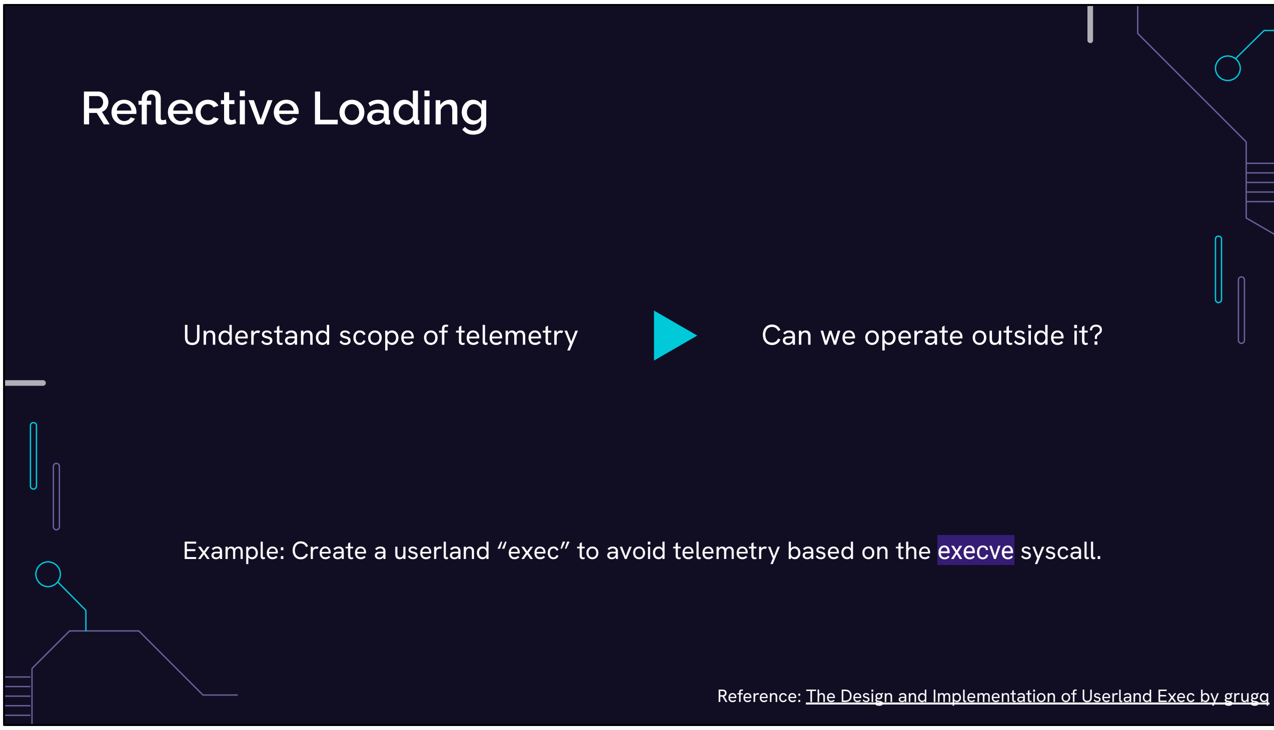
Let’s take a look at a rule from Falco’s example ruleset that allows only some binaries to read /etc/shadow

Beat Rules, Not Telemetry

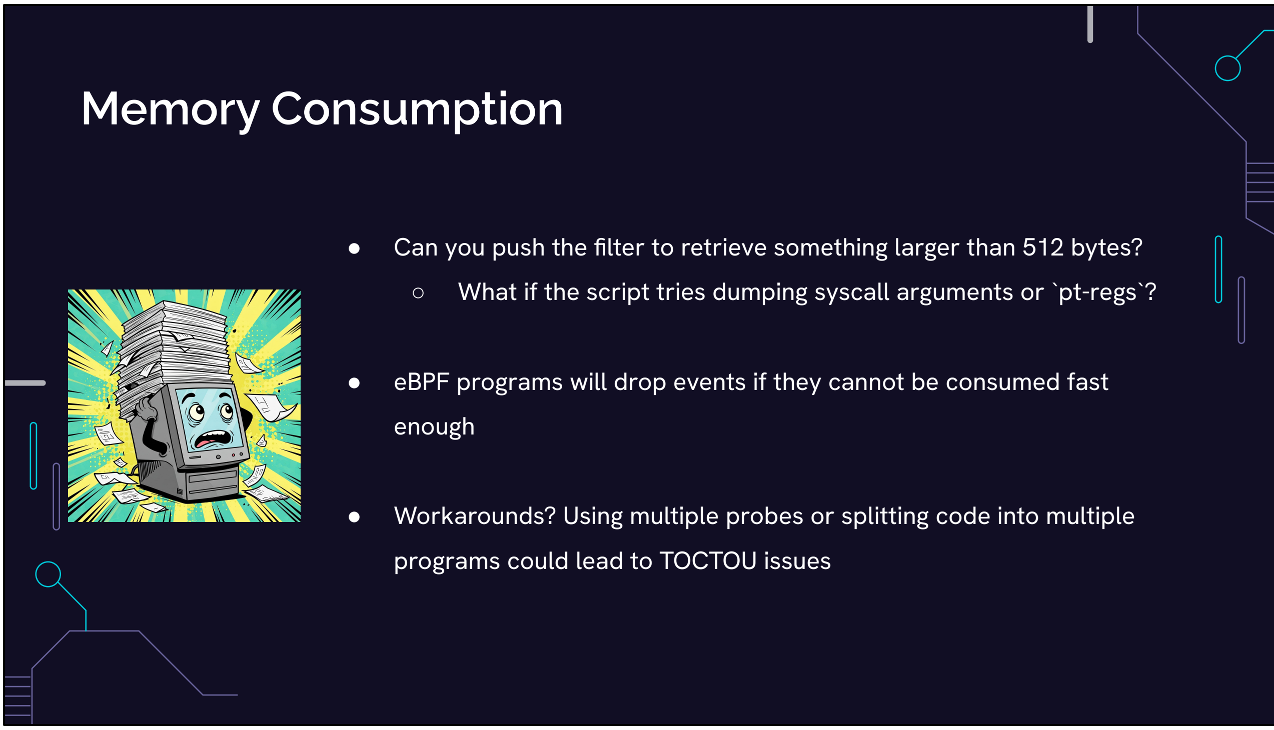
```
  condition: >
    open_read
    and sensitive_files
    and server_procs
    and not proc_is_new
    and proc.name!="sshd"
    and not user_known_read_sensitive_files_activities
```

```
- list: read_sensitive_file_binaries
  items: [
    iptables, ps, lsb_release, check-new-relea, dumpe2fs, accounts-daemon, sshd,
    vsftpd, systemd, mysql_install_d, psql, screen, debconf-show, sa-update,
    pam-auth-update, pam-config, /usr/sbin/spamd, polkit-agent-he, lsattr, file, sosreport,
    scxcimservera, adclient, rtvscand, cockpit-session, userhelper, ossec-syscheckd
    ]
```

However, the way this rule is implemented only mentions the names of the binaries, not the full path or any kind of signature.

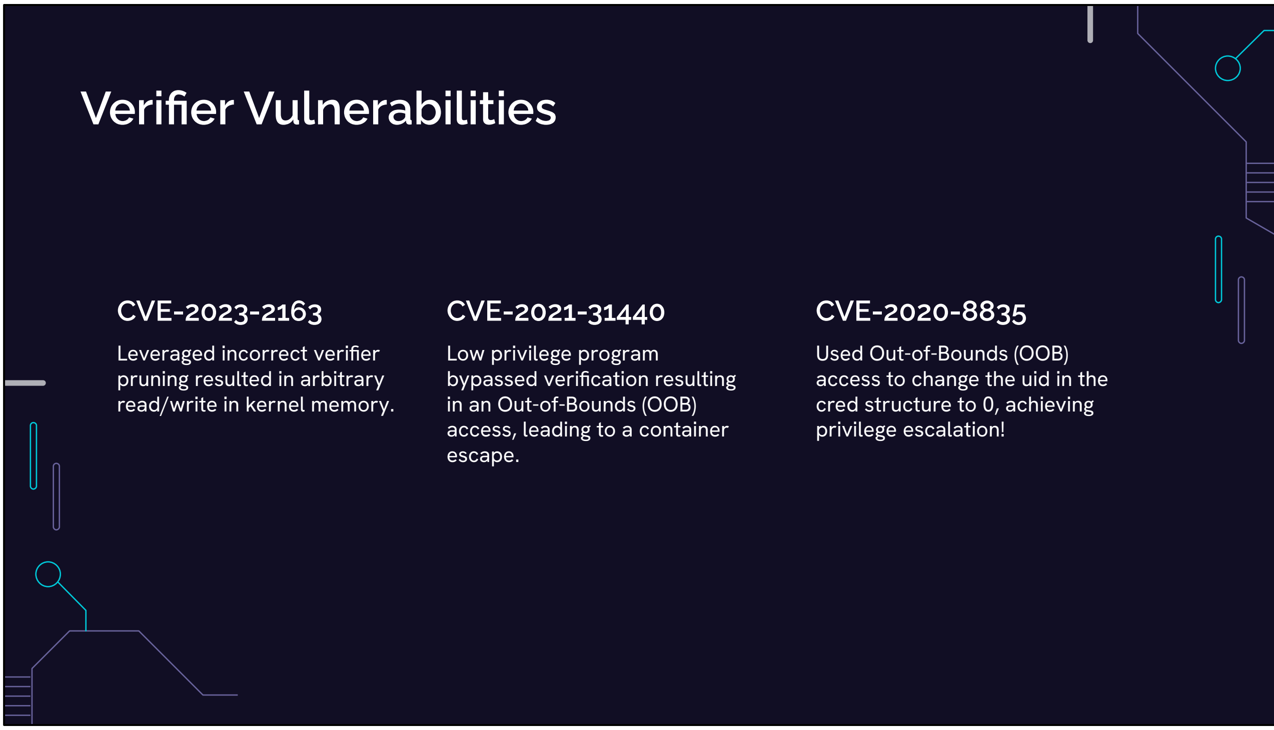


A common Over-the-Shelf approach is reflective loading.
When tackling eBPF instrumentation, it's crucial to understand what telemetry is gathered. Understanding the extent of eBPFs telemetry will, in the most literal sense, help us think outside the box
For instance, if the `execve` syscall is being monitored, it's possible to create a userland version of the system call to create new processes that the eBPF program won't have visibility into.
There are plenty of areas that can't be reasonable instrumented with BPF due to performance reasons. There could be areas where excessive probing could lead to a performance overhead pushing development team to ignore them. Carefully auditing the eBPF program is paramount.



If you can't avoid noise, be extra noisy

- eBPF programs have limited stack space, which can cause data truncation.
  - Remember, eBPF will drop events if they can't be consumed fast enough, instead of dragging down the performance of the entire system with it.
- Even when workarounds are used, such as
  - multiple probes to trace the same events but capture different data
  - Splitting the code into multiple programs that call each other using a program map
- There's still room to abuse its native behavior, not to mention all the potential TOCTOU issues that may arise.



Finally, we have verifier vulnerabilities. Bugs in the very system that's meant to guardrail eBPF programs. Here's three notable examples:

1. Incorrect verifier pruning in BPF led to unsafe code paths being incorrectly marked as safe, resulting in arbitrary read/write in kernel memory, lateral privilege escalation, and container escapes
2. OOB access may lead to container escapes and the ability to modify key structures, such as the cred structure to change your uid and elevate privileges

Thank you!

Acknowledgements

- Brian Mitchell
- Ash Fox
- Niru Ragupathy
- Shea Polansky
- Peter Moody

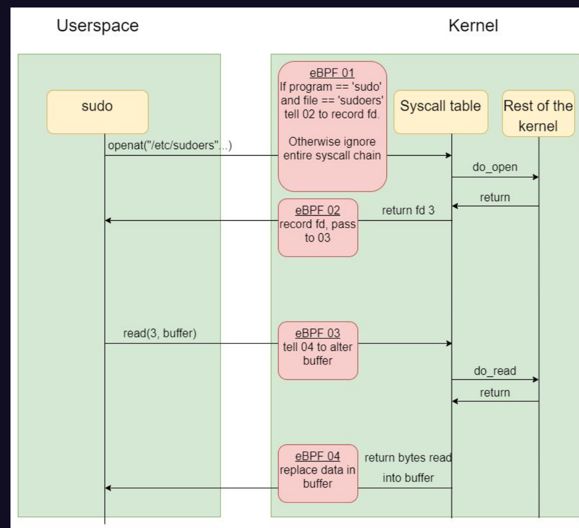
References

1. eBPF: A new frontier for malware - Red Canary, https://redcanary.com/blog/threat-detection/ebpf-malware/
2. Malware Detection in Cloud Native Environments, https://www.cs.drexel.edu/~bmitchell/pubs/AICCC_2024_Malware_Final.pdf
3. Pitfalls of relying on eBPF for security monitoring (and some solutions) | Trail of Bits Blog, https://blog.trailofbits.com/2023/09/25/pitfalls-of-relying-on-ebpf-for-security-monitoring-and-some-solutions/
4. Why Traditional EDRs Fail at Server D&R in the Cloud - Sysdig, https://sysdig.com/blog/traditional-edr-solutions-cloud/
5. On Bypassing eBPF Security Monitoring - Doyensec's Blog, https://blog.doyensec.com/2022/10/11/ebpf-bypass-security-monitoring.html
6. What is eBPF? The Hacker's New Power Tool for Linux, https://cymulate.com/blog/ebpf_hacking/
7. Warping reality using eBPF, https://blog.tofile.dev/2021/08/01/bad-bpf.html

Under these perceived dire circumstances, let’s look at how we can still sneak past

Exploit eBPF Helper Functions

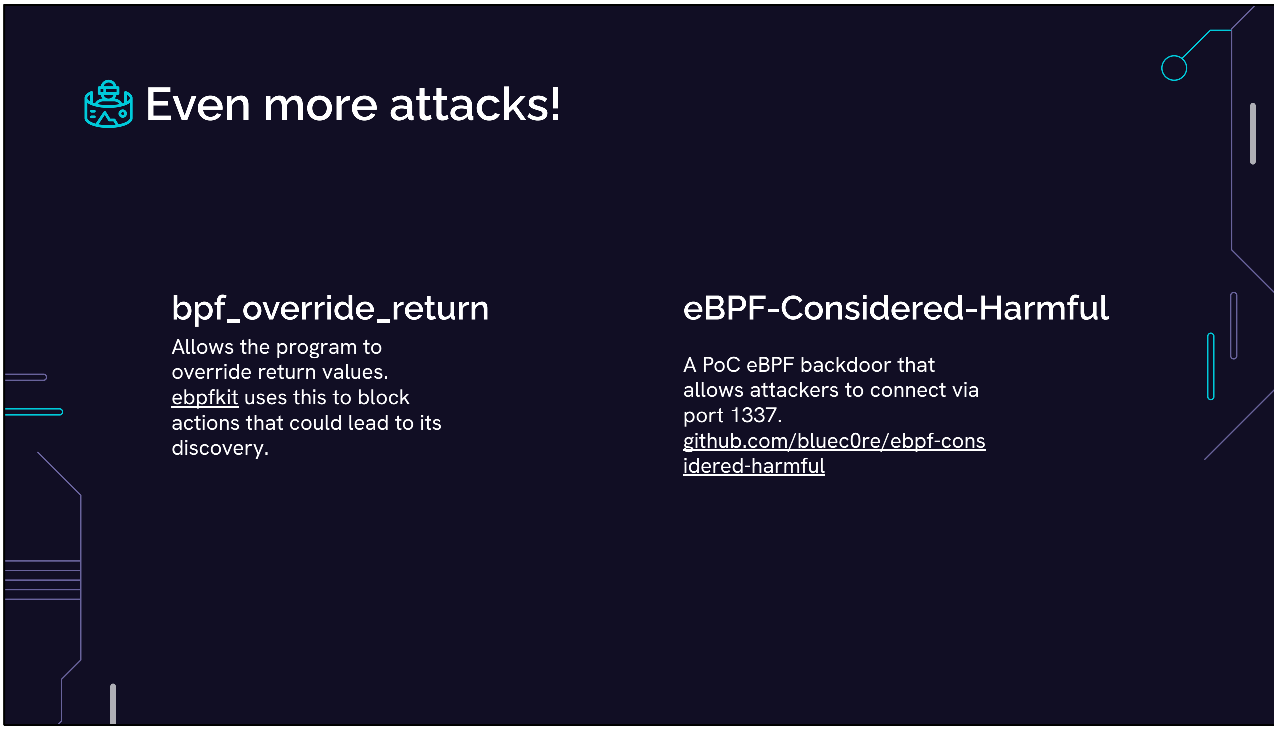
- `bpf_probe_write_user()` can write to the user-space memory of the current process.
- bad-bpf writes to user-space memory when `sudo` tries to read `/etc/sudoers`



- Bad user land configuration, we write to /etc/sudoers (this will give us system)
- Major ref: https://blog.tofile.dev/2021/08/01/bad-bpf.html
- new title: liar liar, sudo lights the system on fire?

- Example situation: compromised web server, with temp root access
- While a reverse webshell can be used as a reentry, the web server runs as the low priv user.
    - This user is not on sudo list!
- This is where eBPF enters:
- Sudo is in the end a list, requires a open/read sys call.
- We can use eBPF to hijack these calls!

- Obsidian notes
- Timo’s GitHub repo
- Small ref: Guillaume Fournier Sylvain Afchain Sylvain Baubeau - eBPF, I thought we were friends.pdf
- Cool factor for hiding processes.
    - Ps, ls, crowdstrike (EDR) raw sys calls for listing processes, all use the same syscall getdents, we override the function and use pointer manipulation to obfuscate our process entry.
    - A → B → C; A → sysdent + pointer → C
- Using an eBPF rootkit to `hide` processes



And there’s more,