



eBPF Tutorial

Teng Jiang, adapted from previous TA, Yannis Zarkadas

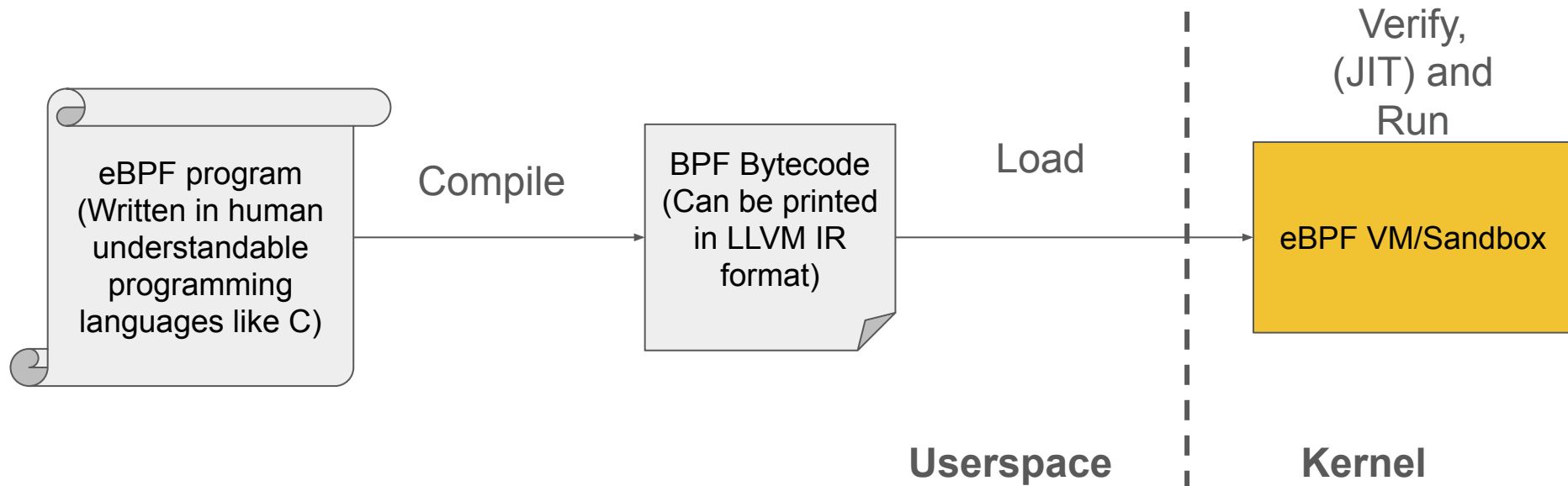
Made for EECS6891 - Extensible OS @ Columbia University, Fall 2024

Lecture Overview and Goals

- eBPF – Extended Berkeley Packet Filter
- Tradition BPF discussed next week
- If you're more interested in the traditional use case (XDP, which we'll briefly discuss), there are good materials online
- Mental framework for eBPF
 - What is it fundamentally?
 - When does it run?
 - How do I get data in/out?
 - Practical examples with code for observability, networking, security, and more novel use cases.

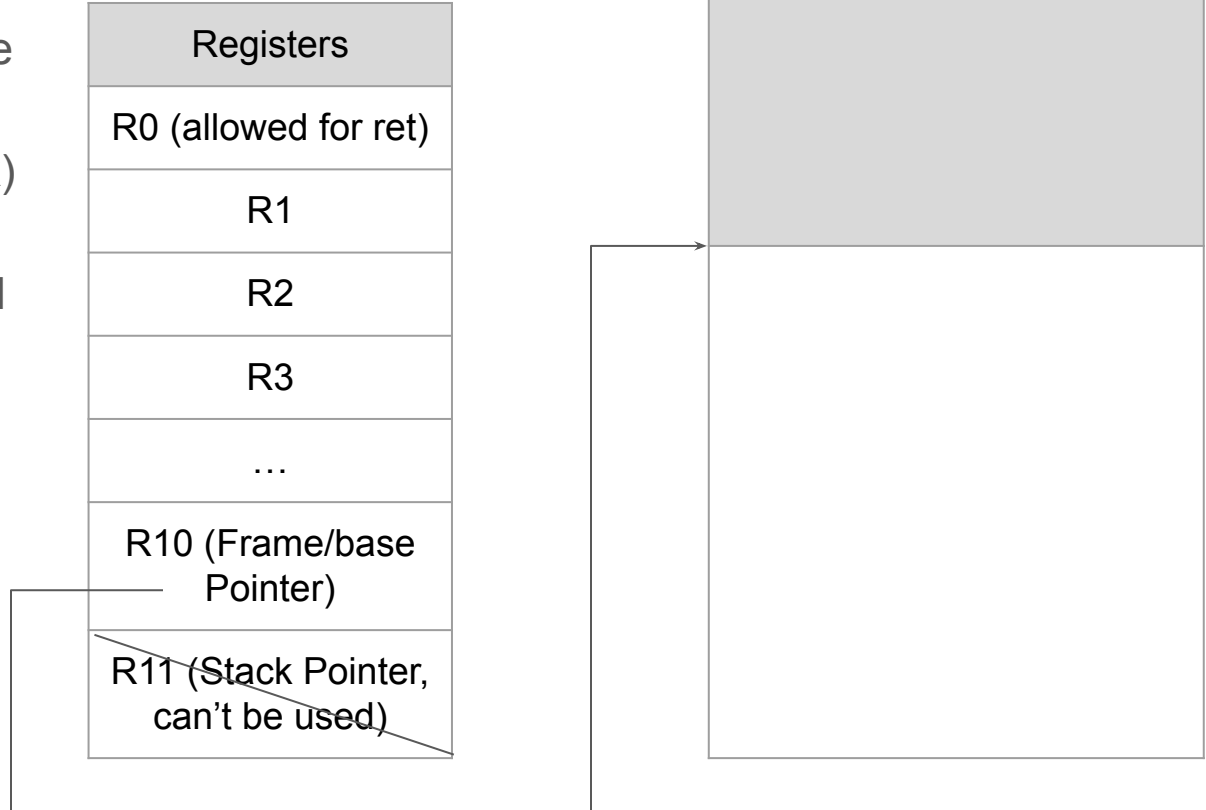
What is eBPF?

- eBPF is a secure sandbox (executes “arbitrary” code) in the kernel!



What's inside of BPF Sandbox?

- RISC virtual Machine
- Virtual states:
Registers and (stack)
memory, to be
executed on a virtual
CPU
- Q: Where to
store larger data
structures?



What's inside of BPF Sandbox?

r0:	stores return values, both for function calls and the current program exit code
r1- r5:	used as function call arguments, upon program start r1 contains the "context" argument pointer
r6- r9:	these get preserved between kernel function calls
r10:	read-only pointer to the per-eBPF program 512 byte stack

```
struct bpf_insn {  
    __u8    code;           /* opcode */  
    __u8    dst_reg:4;      /* dest register */  
    __u8    src_reg:4;      /* source register */  
    __s16   off;           /* signed offset */  
    __s32   imm;           /* signed immediate constant */  
};
```



How is eBPF secure?

- Secure == it doesn't crash and bring down the kernel.
- Method: Verification+Isolation. Run all possible code paths and verify no crashes within a restricted environment

How is eBPF secure? Verification

- General principles when writing code:
 - Bounded loops.
 - Check for NULLs.
 - Check buffer bounds.

See more at:

<https://www.kernel.org/doc/html/v6.1/bpf/verifier.html>

Demo time! (Bounded v.s. Unbounded loop)

How is eBPF secure? Isolation

- Sandboxing-> Isolated from linux execution
- Memory safety -> Cannot access arbitrary (user) memory locations (Comes with verifier)
- Controlled access -> Limited access to kernel internals. Cannot directly access or modify kernel memory, and they can only interact with specific, controlled data structures and interfaces.
- Instruction Set Restrictions -> Cut down on exploitations
- Resource Limiting -> maximum execution time and stack size...

Q: Can BPF call arbitrary kernel functions?

A: NO. BPF programs can only call specific functions exposed as BPF helpers or kfuncs. The set of available functions is defined for every program type.

Q: Can BPF overwrite arbitrary kernel memory?

A: NO.

Tracing bpf programs can read arbitrary memory with `bpf_probe_read()` and `bpf_probe_read_str()` helpers. Networking programs cannot read arbitrary memory, since they don't have access to these helpers. Programs can never read or write arbitrary memory directly.

Q: Can BPF overwrite arbitrary user memory?

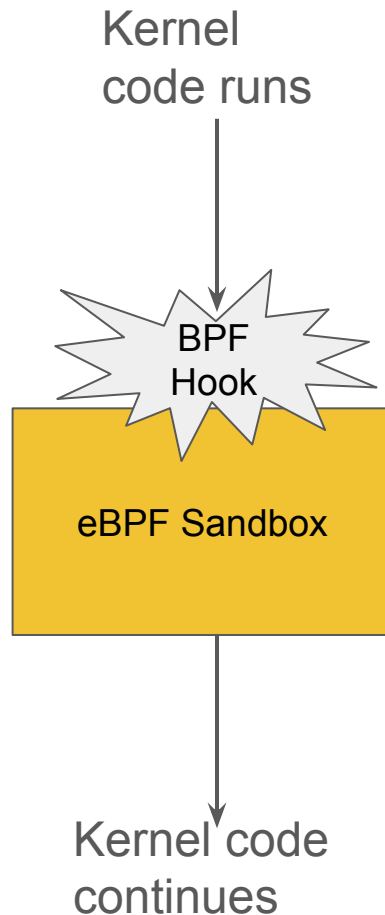
A: Sort-of.

Tracing BPF programs can overwrite the user memory of the current task with `bpf_probe_write_user()`. Every time such program is loaded the kernel will print warning message, so this helper is only useful for experiments and prototypes. Tracing BPF programs are root only.

When does it run?

Points where eBPF runs == BPF Hooks:

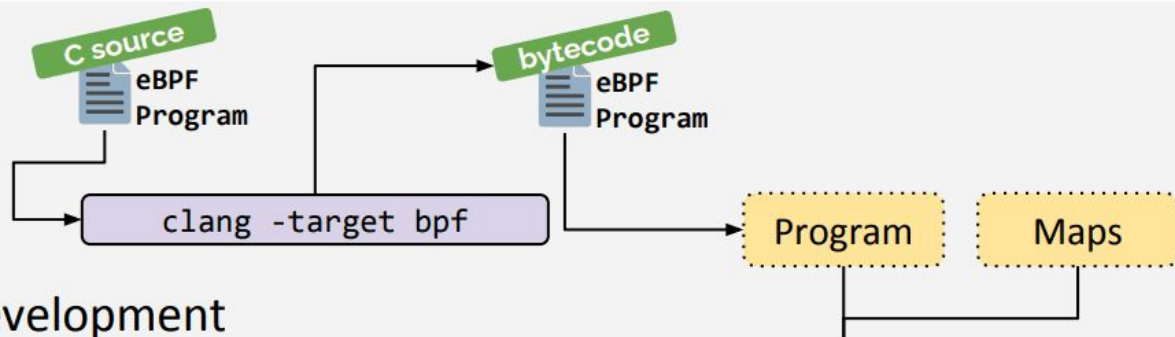
- Tracepoints
- Kprobes
- Networking hooks for inspecting / redirecting packets
- Security hooks for auditing / allowing access to files
- ...



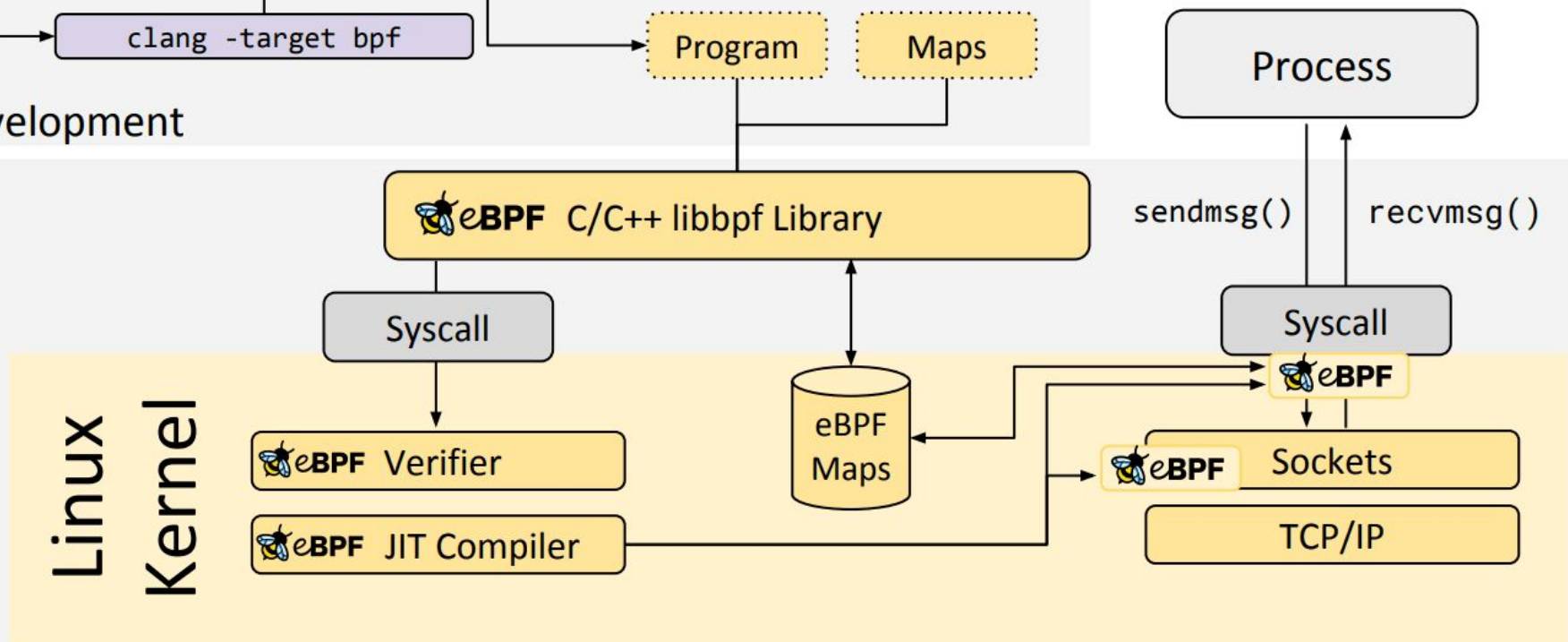
We're going to start with some examples

- There are tons of great tools for using the eBPF infrastructure.
- But, we're going to cover the most difficult/flexible tool: libbpf C/C++ library (go library also available)
- 3 most widely used tools are:
 - BCC is a framework that enables users to write Python programs with eBPF programs embedded inside them, built on top of libbpf
 - bpftrace is a high-level tracing language for Linux eBPF, inspired by awk and C and built on top of bcc
 - Cilium for eBPF Kubernetes
- When to use:
 - Libbpf: When you want to start messing with Kernel code
 - BCC if you want flexibility, and your C is not as good as your Python
 - Bpfttrace for simple probe into the kernel

Development



Runtime



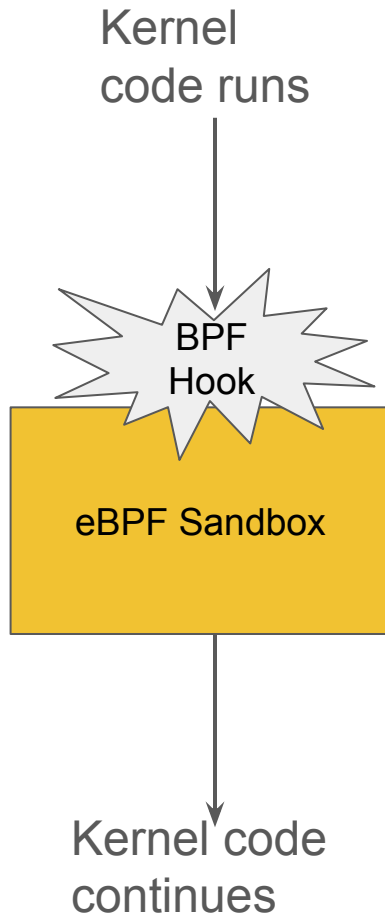
Example 1: Hello world

BPF Hook:

- Tracepoint syscalls/sys_enter

What are tracepoints?

- A tracepoint placed in code provides a hook to call a function (probe) that you can provide at runtime. Linux sprinkles them in interesting spots (e.g., syscalls, io start/end, ...) ~120K as of now
- Very stable interface to build on!
- Full list on your machine:
ls /sys/kernel/debug/tracing/events/



Example 1: Hello world

```
// All linux kernel type definitions are in vmlinux.h
#include "vmlinux.h"
// BPF helpers
#include <bpf/bpf_helpers.h>

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

// SEC name is important! libbpf infers program type from it.
// See: https://docs.kernel.org/bpf/libbpf/program\_types.html#program-types-and-elf
SEC("tracepoint")
int handle_tracepoint(void *ctx) {
    // bpf_get_current_pid_tgid is a helper function!
    int pid = bpf_get_current_pid_tgid() >> 32;
    bpf_printk("BPF triggered from PID %d.\n", pid);

    return 0;
}
```

Boilerplate

BPF helpers

**First way of
output!**

Example 1: Hello world

Let's compile it to BPF bytecode!

```
clang -O2 -target bpf -g -c hello_world.c -o hello_world.o
```

What does it look like?

```
llvm-objdump -d -S ./hello_world.bpf.o
```

Example 1: Hello world

```
./hello_world.bpf.o:    file format elf64-bpf
```

```
Disassembly of section tracepoint:
```

```
0000000000000000 <handle_tracepoint>:
;   int pid = bpf_get_current_pid_tgid() >> 32;
;       0:   85 00 00 00 0e 00 00 00 call 14
;       1:   77 00 00 00 20 00 00 00 r0 >= 32
;   bpf_printk("BPF triggered from PID %d.\n", pid);
;       2:   18 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 r1 = 0
ll  4:   b7 02 00 00 1c 00 00 00 r2 = 28
;       5:   bf 03 00 00 00 00 00 00 r3 = r0
;       6:   85 00 00 00 06 00 00 00 call 6
;   return 0;
;       7:   b7 00 00 00 00 00 00 00 r0 = 0
;       8:   95 00 00 00 00 00 00 00 exit
```

**This is BPF
bytecode!**

**Assembly for a
CPU that does not
exist!**

Example 1: Hello world

```
llvm-objdump -x -d hello_world.bpf.o
```

- You can even view it's LLVM IR, sections, symbol table...

SYMBOL TABLE:

0000000000000000	l	df *ABS*	0000000000000000	hello_world.bpf.c
0000000000000000	l	d	tracepoint/	0000000000000000 tracepoint/
0000000000000000	l	d	.debug_loclists	0000000000000000 .debug_loclists
0000000000000000	l	d	.debug_abbrev	0000000000000000 .debug_abbrev
0000000000000000	l	d	.debug_rnglists	0000000000000000 .debug_rnglists
0000000000000000	l	d	.debug_str_offsets	0000000000000000 .debug_str_offsets
0000000000000000	l	d	.debug_str	0000000000000000 .debug_str
0000000000000000	l	d	.debug_addr	0000000000000000 .debug_addr
0000000000000000	l	d	.debug_frame	0000000000000000 .debug_frame
0000000000000000	l	d	.debug_line	0000000000000000 .debug_line
0000000000000000	l	d	.debug_line_str	0000000000000000 .debug_line_str
0000000000000000	g	F	tracepoint/	0000000000000000a0 handle_tracepoint
0000000000000000	g	0	license	0000000000000000d LICENSE

Example 1: Hello world

Let's load it in the kernel!

Simplified loader

program:

1. Load
2. Attach

```
#include <bpf/libbpf.h>
#include <bpf/bpf.h>

int main() {
    struct bpf_object *obj;
    struct bpf_program *prog;
    struct bpf_link *link;
    int prog_fd;

    // Load and verify BPF application
    obj = bpf_object__open_file("hello_world.bpf.o", NULL);
    bpf_object__load(obj)

    // Attach BPF program
    prog = bpf_object__find_program_by_name(obj, "handle_tracepoint");
    prog_fd = bpf_program__fd(prog);
    link = bpf_program__attach_tracepoint(prog, "raw_syscalls",
    "sys_enter");
    printf("BPF tracepoint program attached. Press ENTER to exit...\n");
    getchar();
}
```

Example 1: Hello world

Let's see the code the kernel
JIT compiler generates for
that program!

```
bpftool prog dump jited name handle_tracepoint
```

**This is ARM64
assembly!**

```
int handle_tracepoint(void * ctx):
bpf_prog_b36af1abfd77d74e_handle_tracepoint:
; int pid = bpf_get_current_pid_tgid() >> 32;
0:    add x9, x30, #0x0
4:    nop
8:    paciasp
c:    stp x29, x30, [sp, #-16]!
10:   mov x29, sp
14:   stp x19, x20, [sp, #-16]!
18:   stp x21, x22, [sp, #-16]!
1c:   stp x25, x26, [sp, #-16]!
20:   stp x27, x28, [sp, #-16]!
24:   mov x25, sp
28:   mov x26, #0x0                                // #0
2c:   sub x27, x25, #0x0
30:   sub sp, sp, #0x0
34:   mov x10, #0xffffffffffffb9b0                // #-18000
...
```

Example 1: Hello world

Let's view the output of **bpf_printk**!

It goes to the system tracing buffer.

```
sudo cat /sys/kernel/debug/tracing/trace_pipe
```

```
root@dev-vm:~# cat /sys/kernel/debug/tracing/trace_pipe
<...>-13197  [000] d...1 14391.036697: bpf_trace_printk: BPF triggered from PID 13197.
<...>-13197  [000] d...1 14391.036697: bpf_trace_printk: BPF triggered from PID 13197.
<...>-13197  [000] d...1 14391.036698: bpf_trace_printk: BPF triggered from PID 13197.
<...>-13197  [000] d...1 14391.036699: bpf_trace_printk: BPF triggered from PID 13197.
```

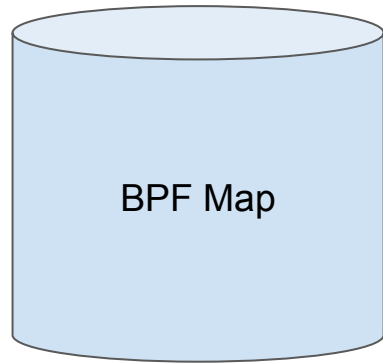
Example 1: Hello world

Conclusion:

- See the essence of eBPF in action: code -> BPF bytecode -> assembly
- Learn to write / compile / attach a simple eBPF program.
- Learn to output and read logs with `bpf_printk`.

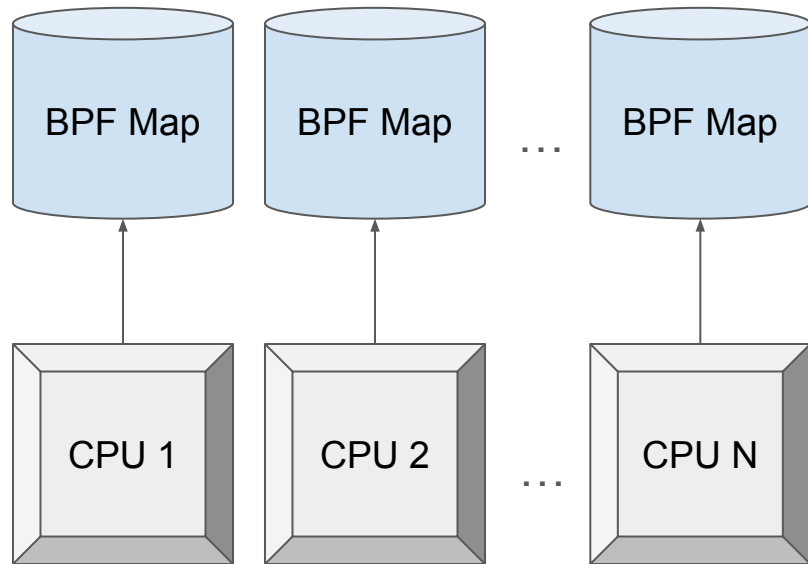
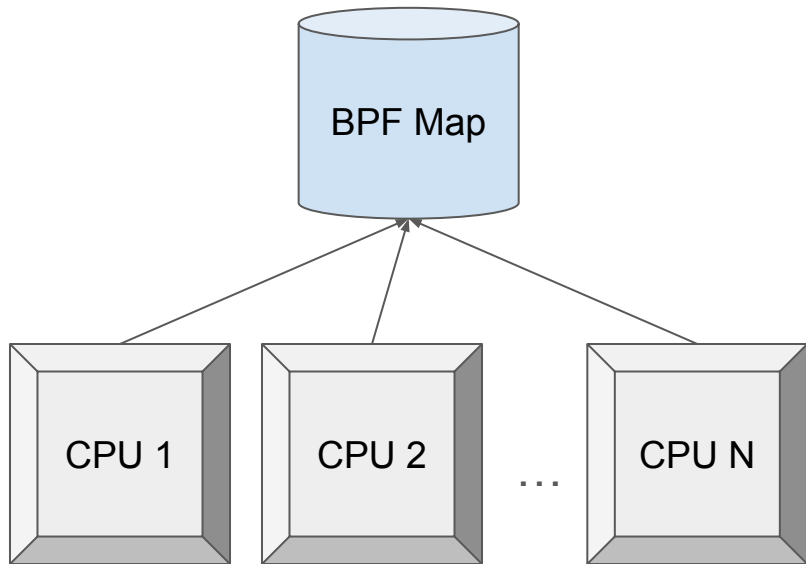
eBPF Maps - Maintaining state

- The simple hello world example is not very useful.
- Most useful programs need **STATE**.
- BPF Maps are how BPF programs **maintain state** and **get data in/out to userspace!**
- Maps persist and are not tied to eBPF program execution lifetime.
- Many types of maps:
 - Array
 - Hash (key-value store)
 - Global and per-cpu variants



eBPF Maps - Global and Per-CPU variants

Q: difference?



BPF Helpers

A BPF program cannot call arbitrary kernel functions. To accomplish certain tasks with this limitation, “helper” functions that BPF can call have been provided (later there’re also kfuncs which is used in projects such as sched_ext). For example:

- What CPU am I running on?
- In what PID’s context is the eBPF program running right now?
 - We saw **bpf_get_current_pid_tgid** before!
- Get / set map key-values.
- ...

BPF helpers are to eBPF what system calls are to userspace programs.

An interface to more privileged actions.

Some other helpers

- An almost exhaustive list:
<https://ebpf-docs.dylanreimerink.nl/linux/helper-function/>
- <https://manpages.ubuntu.com/manpages/focal/en/man7/bpf-helpers.7.html>
- Most commonly used for beginners:
 - Interact with map: `bpf_map_push/pop/peek_elem()`
 - XDP helpers
 - Print helpers
 - Process info helpers
- You can also define your own helper after substantial engineering efforts (modify some kernel code)

A recap: What is eBPF?

```
// SPDX-License-Identifier: GPL-2.0 OR BSD-3-Clause
/* Copyright (c) 2020 Facebook */
/* Adapted by yannisark in 2024 */

// All linux kernel type definitions are in vmlinux.h
#include "vmlinux.h"
// BPF helpers
#include <bpf/bpf_helpers.h>

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

// SEC name is important! libbpf infers program type from it.
// See: https://docs.kernel.org/bpf/libbpf/program_types.html#program-types-and-elf-sec("tracepoint/")
int handle_tracepoint(void *ctx) {
    // bpf_get_current_pid_tgid is a helper function!
    int pid = bpf_get_current_pid_tgid() >> 32;
    bpf_printk("BPF triggered from PID %d.\n", pid);

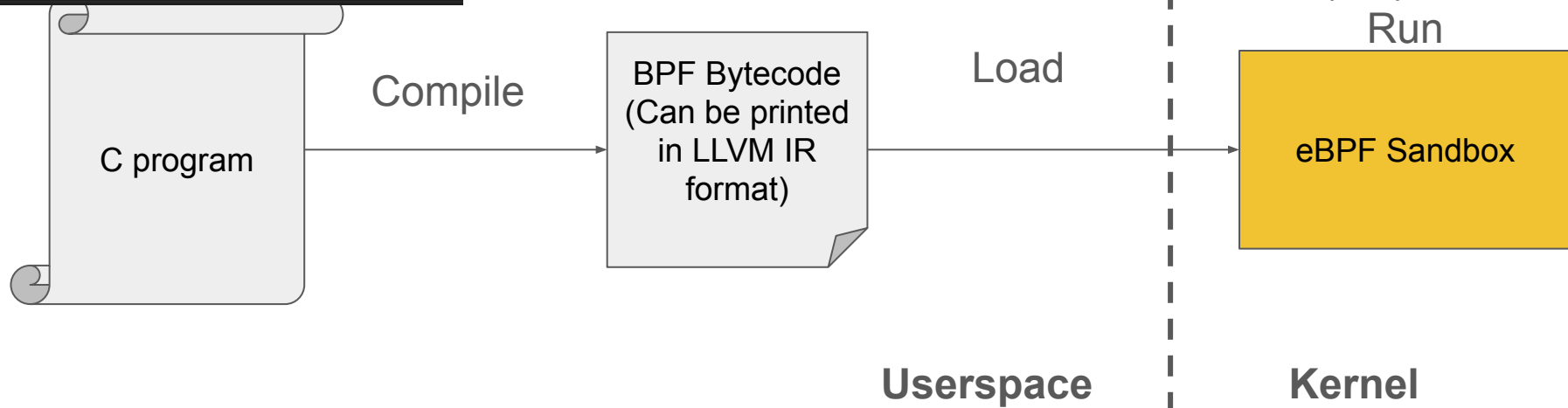
    return 0;
}
```

```
./hello_world.bpf.o:   file format elf64-bpf

Disassembly of section tracepoint:

0000000000000000 <handle_tracepoint>:
;   int pid = bpf_get_current_pid_tgid() >> 32;
;   0: 85 00 00 00 0e 00 00 00 call 14
;   1: 77 00 00 00 20 00 00 00 r0 >>= 32
;   ;   bpf_printk("BPF triggered from PID %d.\n", pid);
;   2: 18 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 r1 = 0
;   4: b7 02 00 00 1c 00 00 00 r2 = 28
;   5: bf 03 00 00 00 00 00 00 r3 = r0
;   6: 85 00 00 00 06 00 00 00 call 6
;   ;   return 0;
;   7: b7 00 00 00 00 00 00 00 r0 = 0
;   8: 95 00 00 00 00 00 00 00 exit
```

```
int handle_tracepoint(void * ctx):
bpf_prog_b36af1abfd77d74e_handle_tracepoint:
; int pid = bpf_get_current_pid_tgid() >> 32;
;   0: add x9, x30, #0x0
;   4: nop
;   8: paciasp
;   c: stp x29, x30, [sp, #-16]!
;  10: mov x29, sp
;  14: stp x19, x20, [sp, #-16]!
;  18: stp x21, x22, [sp, #-16]!
;  1c: stp x25, x26, [sp, #-16]!
;  20: stp x27, x28, [sp, #-16]!
;  24: mov x25, sp
;  28: mov x26, #0x0
;  2c: sub x27, x25, #0x0
;  30: sub sp, sp, #0x0
;  34: mov x10, #0xffffffffffffb9b0
; ...
```



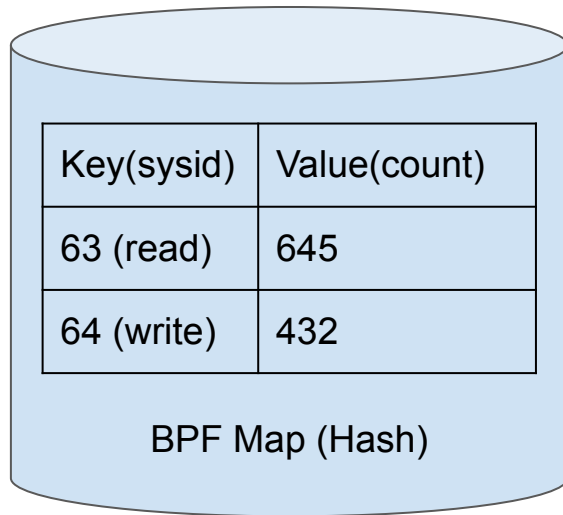
Example 2: syscount

Let's extend hello world to do something more useful:

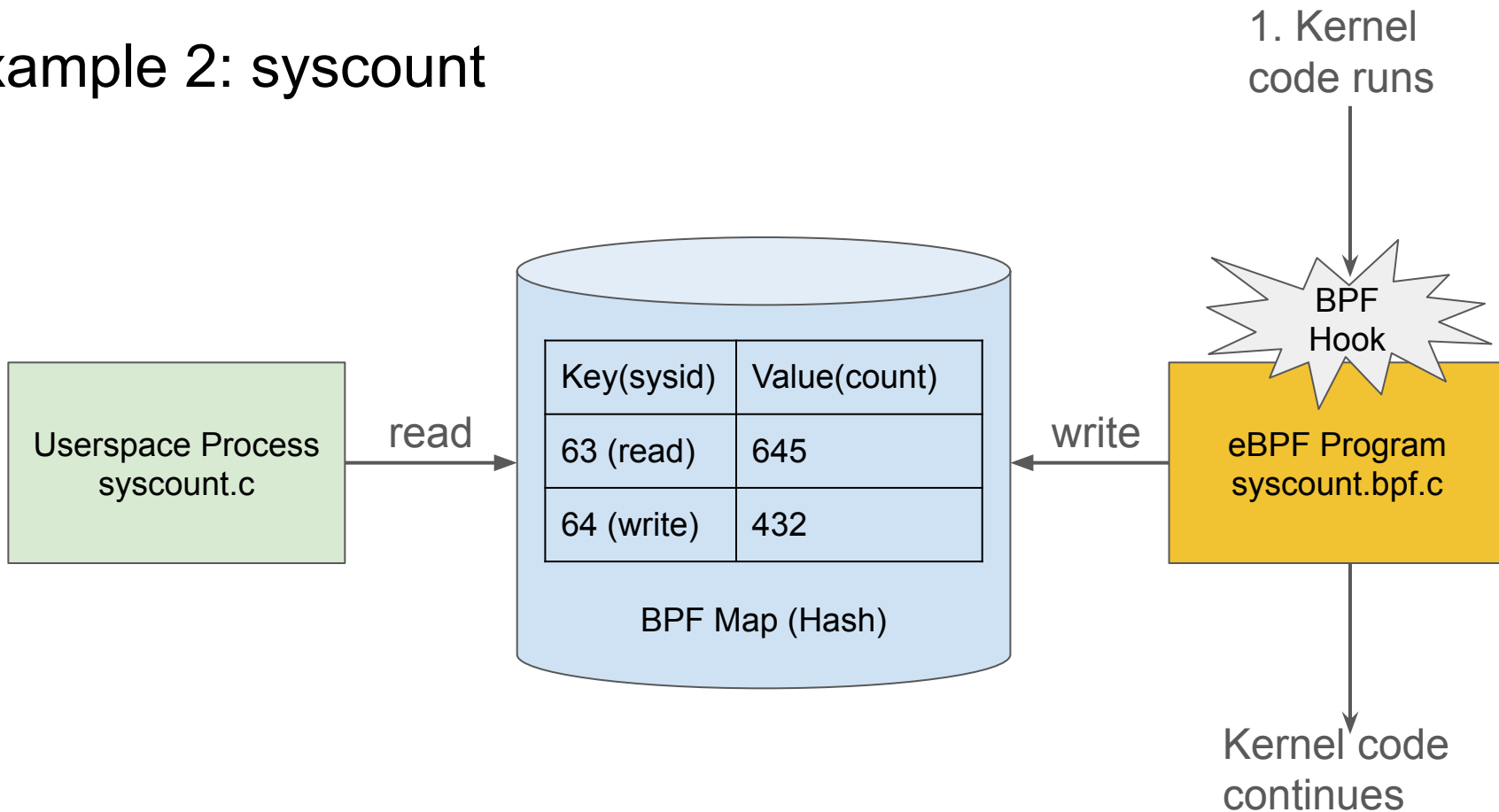
- Count how many times each syscall was used system-wide.
- I.e. **construct a key-value map of syscall-id to count.**

Introducing:

- BPF_MAP_TYPE_HASH
- See all here:
<https://docs.kernel.org/bpf/maps.html>



Example 2: syscount



Example 2: syscount - BPF part


Let's define the map in eBPF! Why do we use a per-cpu map?

```
// Map of type hash (essentially a key-value store)
// Key: syscall number
// Value: number of times the syscall was called
struct {
    __uint(type, BPF_MAP_TYPE_PERCPU_HASH); ———> Map Type
    __type(key, u64); ———> Key Type
    __type(value, u64); ———> Value Type
    __uint(max_entries, 500); // most linux systems have 300-400 syscalls
} syscall_id_to_count SEC(".maps");
```

Example 2: syscount - BPF part

/sys/kernel/debug/tracing/events/

```
SEC("tracepoint/syscount")
int syscount(struct trace_event_raw_sys_enter *ctx) {
    // Interpret ctx
    u64 syscall_id = ctx->id;
    u64 *value;
    // Get the value from the map and increment
    value = bpf_map_lookup_elem(&syscall_id_to_count, &syscall_id);
    if (value) {
        *value += 1;
    } else {
        u64 zero = 0;
        bpf_map_update_elem(&syscall_id_to_count, &syscall_id, &zero, BPF_ANY);
    }
    return 0;
}
```



How do we know this?

Example 2: syscount - Userspace part

For each syscall id (key), add the values from all CPUs.

```
__u64 *curr_key = NULL;
__u64 next_key;
__u64 *values = (__u64 *)malloc(roundup(sizeof(__u64), 8) * num_cpus);
while (bpf_map_get_next_key(map_fd, curr_key, &next_key) == 0) {
    // Get value
    bpf_map_lookup_elem(map_fd, &next_key, values);
    // Calculate total from all CPUs
    __u64 total = 0;
    for (int i = 0; i < num_cpus; i++) {
        total += values[i];
    }
    printf("Syscall %s - Count %llu", syscall_id_to_name[next_key], total);
    // Update key
    curr_key = &next_key;
}
```

Example 2: syscount

Conclusion:

- Learn how to maintain state with BPF maps.
 - Syscall counts persist across invocations!
- Learn how to export state to userspace with BPF maps.

How do I know which tracepoint I should hook to?

+ :: For tracepoint:

```
# SEC("tracepoint/syscalls/sys_enter")  
  
ls /sys/kernel/debug/tracing/events/  
sudo perf list tracepoints
```

For kprobe/kretprobe:

```
# SEC("k(ret)probe/___x64_sys_execve")  
  
cat /proc/kallsyms | grep " [tT] "
```

For uprobe/uretprobe:

```
# SEC("uprobe/your_binary:your_function")  
  
objdump -T /path/to/binary | grep "FUNC"
```

Other:

```
SEC("xdp")  
SEC("perf_event") # Google for a perf example
```

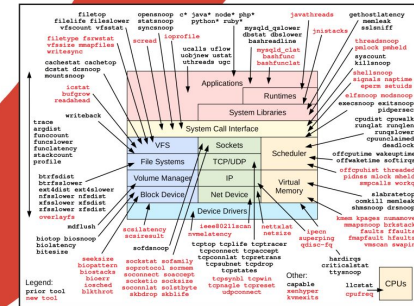
More on Observability

- If you're keen on eBPF for observability topics, please check out Brennan Gregg's Book
- Most papers we're going to read is not going to cover observability topics, so it will complement with the topics of our course nicely

BPF Performance Tools

Linux System and Application Observability

Brendan Gregg



Foreword by Alexei Starovoitov,
creator of the new BPF

Taking actions with eBPF

- So far we've mainly focused on observability use-cases.
- However, many BPF program types can take **actions**.
- For example:
 - Networking: BPF programs can reject / forward a packet
 - Security: BPF programs can allow / block a filesystem operation

XDP: BPF for networking

Program type **BPF_PROG_TYPE_XDP**

```
int xdp_program(struct xdp_md *ctx) {  
    return XDP_DROP;  
}
```

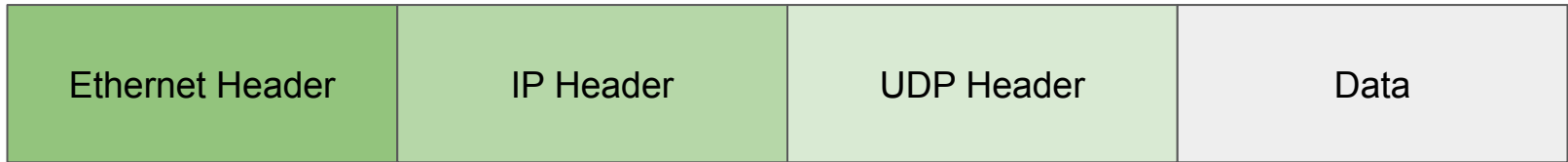
Takes action with return code:

- **XDP_DROP**: Drop the packet.
- **XDP_PASS**: Continue processing as normal.
- **XDP_TX / XDP_REDIRECT**: Redirect the package to the same / another NIC.

Example 3: Simple Firewall

- Let's use XDP to make a simple firewall! It will just block UDP port 11111.
- XDP program see ethernet frames. Need to parse.

Message Buffer:



Example 3: Simple Firewall - BPF part

```
SEC("xdp")
int xdp_firewall(struct xdp_md *ctx) {
    // We see the raw ethernet frame here.
    // To filter on higher-level protocols, we need to parse it.
    void *data = (void *)(long)ctx->data;
    void *data_end = (void *)(long)ctx->data_end;

    // Parse IP
    struct iphdr *ip = data + sizeof(struct ethhdr);
    // Verifier check.
    if ((void *)(ip + 1) > data_end) return XDP_PASS;
    if (ip->protocol != IPPROTO_UDP) {
        return XDP_PASS;
    }
}
```

Example 3: Simple Firewall - BPF part

```
// Parse UDP
struct udphdr *udp =
    (struct udphdr *)(data + sizeof(struct ethhdr) + sizeof(struct iphdr));
// Verifier check.
if ((void *)(udp + 1) > data_end) return XDP_PASS;

// Block 11111
if (udp->dest == bpf_htons(11111)) {
    bpf_printk("Dropping packet to port 11111\n");
    return XDP_DROP;
}
return XDP_PASS;
}
```

Example 3: Simple Firewall - Userspace part

Same as hello_world, just different attach function:

```
// Get ifindex of interface
char* ifname = "lo";
unsigned int ifindex = if_nametoindex(ifname);
link = bpf_program__attach_xdp(prog, ifindex);
if (libbpf_get_error(link)) {
    fprintf(stderr, "ERROR: Attaching BPF program to interface failed\n");
    return 1;
}
```


Example 3: Simple Firewall - Demo

Let's see it running live!

Example 3: Simple Firewall

Conclusions:

- Learn how eBPF can route / drop packets using XDP.
- Implement a simple firewall.
- Food for thought: Think about what we could build by adding maps to our simple firewall. We can enable userspace to encode a great amount of complex rules that can change at runtime.
 - Facebook uses something similar for their firewall!
 - http://vger.kernel.org/lpc_net2018_talks/ebpf-firewall-LPC.pdf
- Why XDP?
 - Performance
 - Flexibility

BPF in Security - LSM Hooks

Linux Security Modules (LSM)

- Framework for implementing new security models in Linux.
- TLDR: It's a bunch of hooks in strategic locations (mainly file operations).
 - File open
 - File permission (read / write)
 - File mmap
 - ...
- See: `security/security.c` in Linux kernel

BPF in Security - LSM Hooks

Linux Security Modules (LSM)

- Framework for implementing new security models in Linux.
- TLDR: It's a bunch of hooks in strategic locations (mainly file operations).
 - File open
 - File permission (read / write)
 - File mmap
 - ...
- See: security/security.c in Linux kernel
- Traditionally implemented with custom Linux Kernel Modules
- But now, we can also attach BPF programs!

Example 4: Simple file access control

Let's build a simple access control system with eBPF.

- On each file operation, check if a user is possibly compromised.
- If they are, disallow all interactions with the filesystem.

Example 4: Simple file access control - BPF part

```
// Map of type hash (essentially a key-value store)
// Key: user id
// Value: true if user is compromised, false otherwise
struct {
    __uint(type, BPF_MAP_TYPE_HASH);
    __type(key, u64);
    __type(value, bool);
    __uint(max_entries, 100);
} compromised_users SEC(".maps");

inline bool is_user_compromised() {
    u64 uid_gid = bpf_get_current_uid_gid();
    u64 uid = uid_gid & 0xFFFFFFFF;
    bool *is_compromised = bpf_map_lookup_elem(&compromised_users, &uid);
    if (is_compromised != NULL && *is_compromised) {
        // Yes, user is compromised.
        bpf_printk("User %d is compromised\n", uid);
        return true;
    }
    return false;
}
```

Example 4: Simple file access control - BPF part

```
SEC("lsm/file_open")
int BPF_PROG(lsm_access_control_open, struct file *file, int
ret) {
    // ret is the return value from the previous BPF program
    // or 0 if it's the first hook.
    if (ret != 0) {
        return ret;
    }
    // Is intrusion detected?
    if (is_user_compromised()) {
        return -EPERM;
    }
    return 0;
}
```

Example 4: Simple file access control - BPF part

```
SEC("lsm/file_permission")
int BPF_PROG(lsm_access_control_file_permission, struct file
*file, int mask,
            int ret) {
    // ret is the return value from the previous BPF program
    // or 0 if it's the first hook.
    if (ret != 0) {
        return ret;
    }
    // Is intrusion detected?
    if (is_user_compromised()) {
        return -EPERM;
    }
    return 0;
}
```


Example 4: Demo

Example 4: Conclusions

- One more use-case where eBPF can actually take decisions on behalf of the kernel.
- Again, think how this could be combined with a auditing and detection tool.
- Example eBPF security projects:
 - Falco
 - Tracee

Advanced Topics

Don't really need to know any of them to do useful things, but you may see them in online resources and I want you to have an idea of what they are:

- BPF CO-RE (BTF)
- Libbpf - skeleton
- Iterators

Advanced Topics - BPF CO-RE

CORE == Compile Once Run Anywhere

- Aims to solve the problem of portability
- Imagine you had v1 code that accessed: **kernel_struct->b**
- What would happen if you ran it in v2?

v1

```
struct kernel_struct {  
    int a;  
    int b;  
}
```

v2

```
struct kernel_struct {  
    int a;  
    int a1;  
    int b;  
}
```

Advanced Topics - BPF CO-RE

Solution: BPF Type Format (BTF)

- See: <https://nakryiko.com/posts/bpf-portability-and-co-re/>

Basically:

- Record all types for accessed kernel structs in BPF programs (object files), using the BTF format.
- When loading the BPF program, field accesses are matched based on **name** and **type**.

Advanced Topics - Libbpf Skeleton

- Generated helper code by libbpf: **bpftool gen skeleton**
- Quality of life improvement for working with BPF programs.
- See:

https://docs.kernel.org/bpf/libbpf/libbpf_overview.html#bpf-object-skeleton-file

Features like:

- Easier interaction with global vars and maps.
- Bytecode embedded in skeleton, no need to load anything.

We didn't use it for the examples as it was a bit too "magic". But it is recommended for stuff that will hit production.

Advanced Topics - BPF Iterators

- So far, we've seen that BPF programs are triggered as part of the kernel control flow.
- We can also iterate through certain structures of the kernel (e.g., tasks) and trigger a BPF program for each one.
- These are called BPF iterators.

See: https://docs.kernel.org/bpf/bpf_iterators.html

Cool eBPF Projects to Try

Observability:

- BCC tools:
 - <https://github.com/iovisor/bcc>
- BPFTrace
 - <https://github.com/bpftrace/bpftrace>
- Auto-instrumentation, aka metrics without changing the executables:
 - <https://github.com/grafana/beyla>

Scheduling:

- sched_ext: Write custom schedulers in eBPF + userspace code
 - <https://github.com/sched-ext/scx>

Good source of learning

- <https://github.com/zoidyzoidzoid/awesome-ebpf>
- https://docs.google.com/presentation/d/1abYBW7L8kAupgG9YkFPRGayZSXm9hGv_Dvp7ADBkfyg/edit#slide=id.g13ff8f89a35_2_2225
- <https://www.youtube.com/watch?v=TJgxjVTZtfw>
- <https://www.brendangregg.com/blog/2021-06-15/bpf-internals.html>
- Ask Teng/Hubertus