

Return to sender

Detecting kernel exploits with eBPF

Guillaume Fournier
August 2022



About me



Guillaume Fournier

Senior Security Engineer @Datadog
gui774ume.fournier@gmail.com

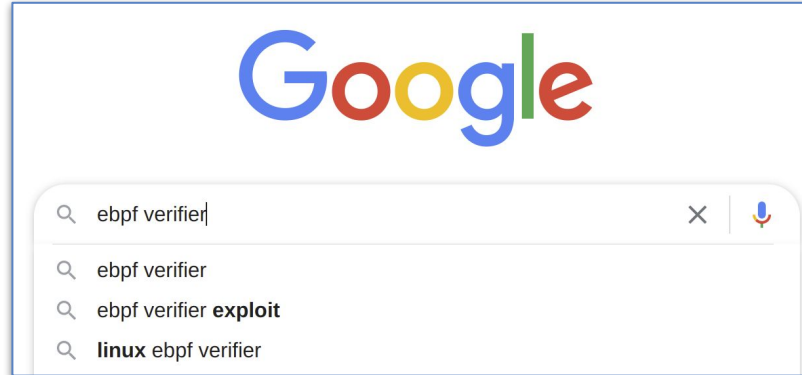
- Cloud Workload Security (CWS)
- Leverage eBPF to detect threats
- Embedded in the Datadog Agent

Agenda

- Context and threat model
- Why eBPF ?
- KRle
 - SMEP & SMAP on a budget
 - Kernel security configuration
 - Kernel runtime alterations
 - Control flow integrity
 - Enforcement
- Performance

Context and threat model

- Critical CVEs are regularly discovered in the Linux Kernel
- Security administrators worry about:
 - Keeping up with security updates
 - Deploying security patches
 - Monitoring & protecting vulnerable hosts



Context and threat model

- Hundreds of ways to exploit the Linux kernel
- This talk targets 3 types of vulnerabilities:
 - Execution flow redirections
 - Logic bugs
 - Post compromise kernel runtime alterations

The goal is to detect (and prevent ?) these attacks with eBPF

Context and threat model

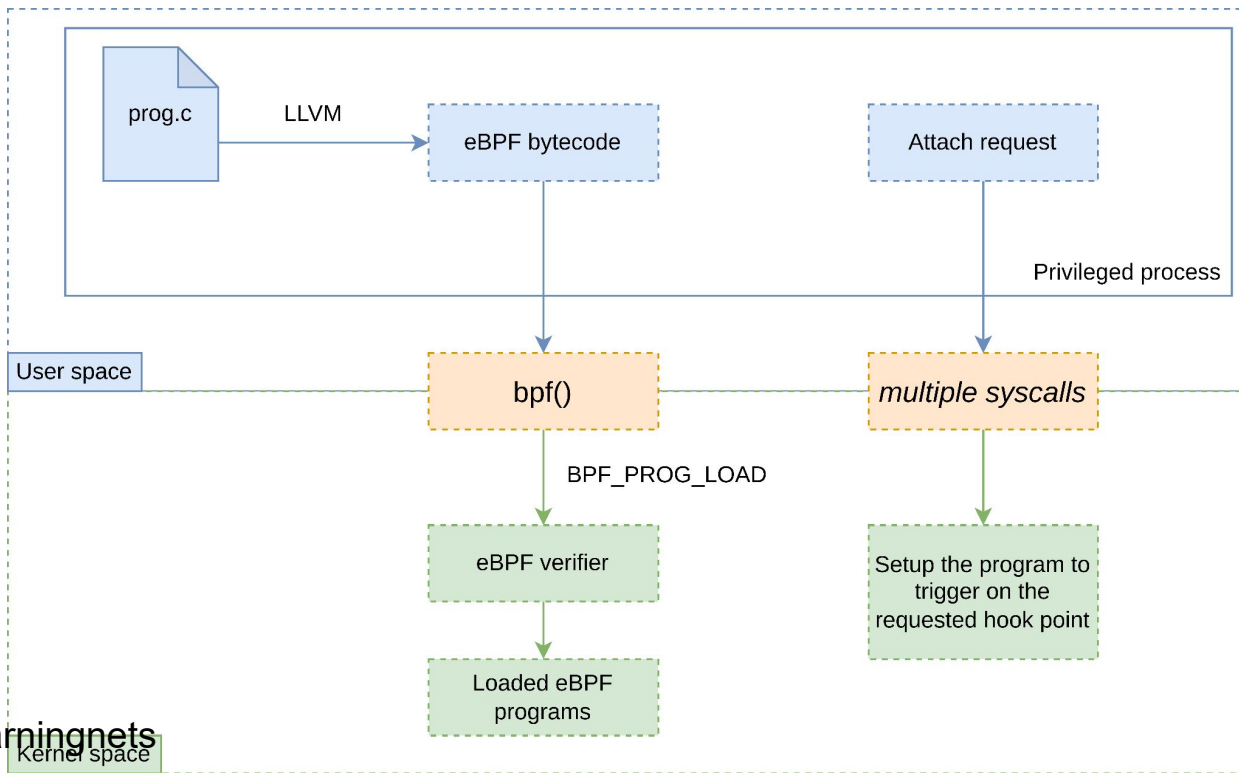
- Hundreds of ways to exploit the Linux kernel
- This talk targets 3 types of vulnerabilities:
 - Execution flow redirection
 - Logic bugs
 - Post compromise kernel runtime alteration

~~The goal is to detect (and prevent ?) these attacks with eBPF~~

Make attackers' lives a living hell

What is eBPF ?

- Run sandboxed programs in the Linux kernel



Why eBPF?

- Relatively wide kernel support (4.1 +) depending on eBPF features
- System safety and stability insurances
- Rich feature set with easy to use introspection capabilities
- Some write access and enforcement capabilities

Why ~~eBPF~~ ?

Why is this a terrible idea ?

- Detecting post compromise is fighting a lost battle
- There are dozens of ways to disable an eBPF program
- eBPF can have a significant in kernel performance impact

So what's the point ?

- Script kiddies and OOTB rootkits
- Make it harder to exploit a flaw
- Detecting & blocking pre-compromise is **sometimes** possible

Kernel Runtime Integrity with eBPF (KRle)

- Open source project
- Compile Once Run Everywhere
- Compatible with at least kernels 4.15+ to now
- First version released today !

<https://github.com/Gui774ume/krie>

KRIe: SMEP & SMAP on a budget

Scenario 1: the attacker controls the address of the next instruction executed by the kernel

- Textbook use case for Return Object Programming (ROP) attacks
- Supervisor Mode Access Prevention (SMAP)
- Supervisor Memory Execute Protection (SMEP)

KRIe: SMEP & SMAP on a budget

Scenario 1: the attacker controls the address of the next instruction executed by the kernel

Kernel Executable code		User space memory	
Addresses	Bytecode	Addresses	Bytecode

KRIe: SMEP & SMAP on a budget

Scenario 1: the attacker controls the address of the next instruction executed by the kernel

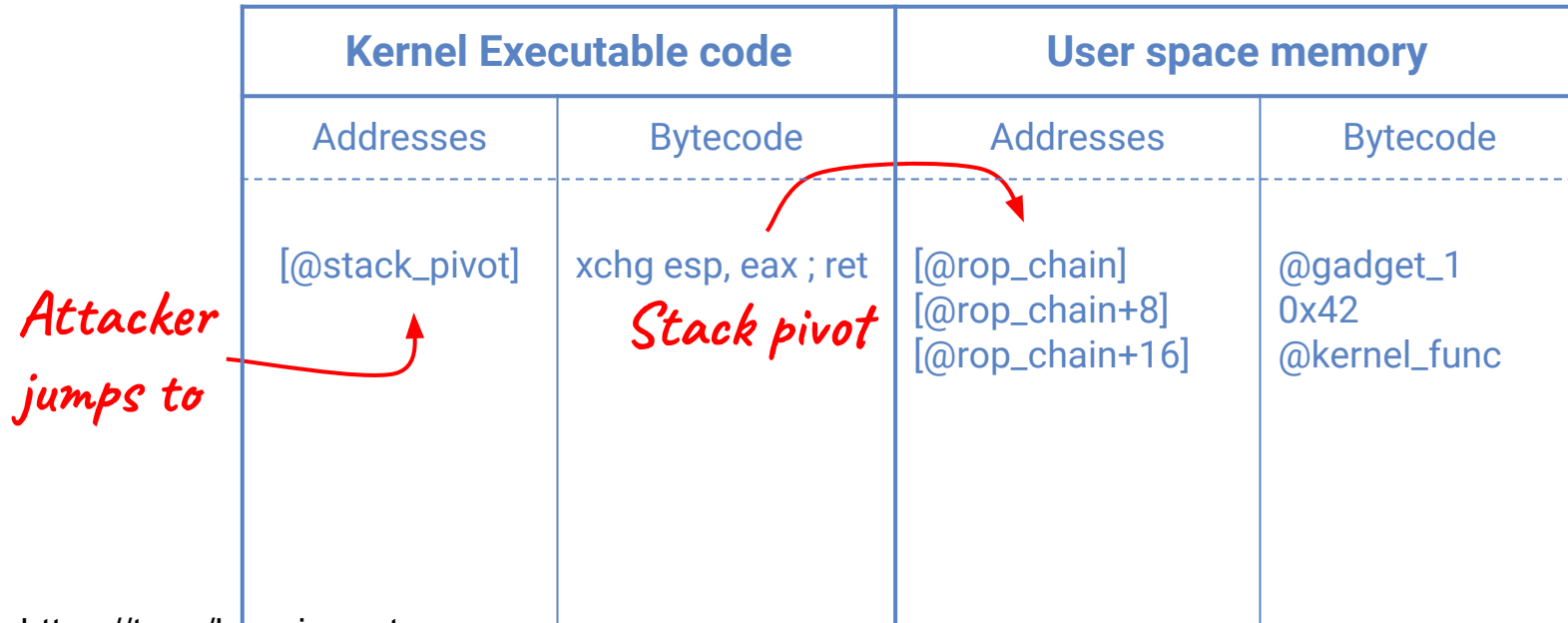
Kernel Executable code		User space memory	
Addresses	Bytecode	Addresses	Bytecode
<code>[@stack_pivot]</code>	<code>xchg esp, eax ; ret</code>		

Attacker jumps to



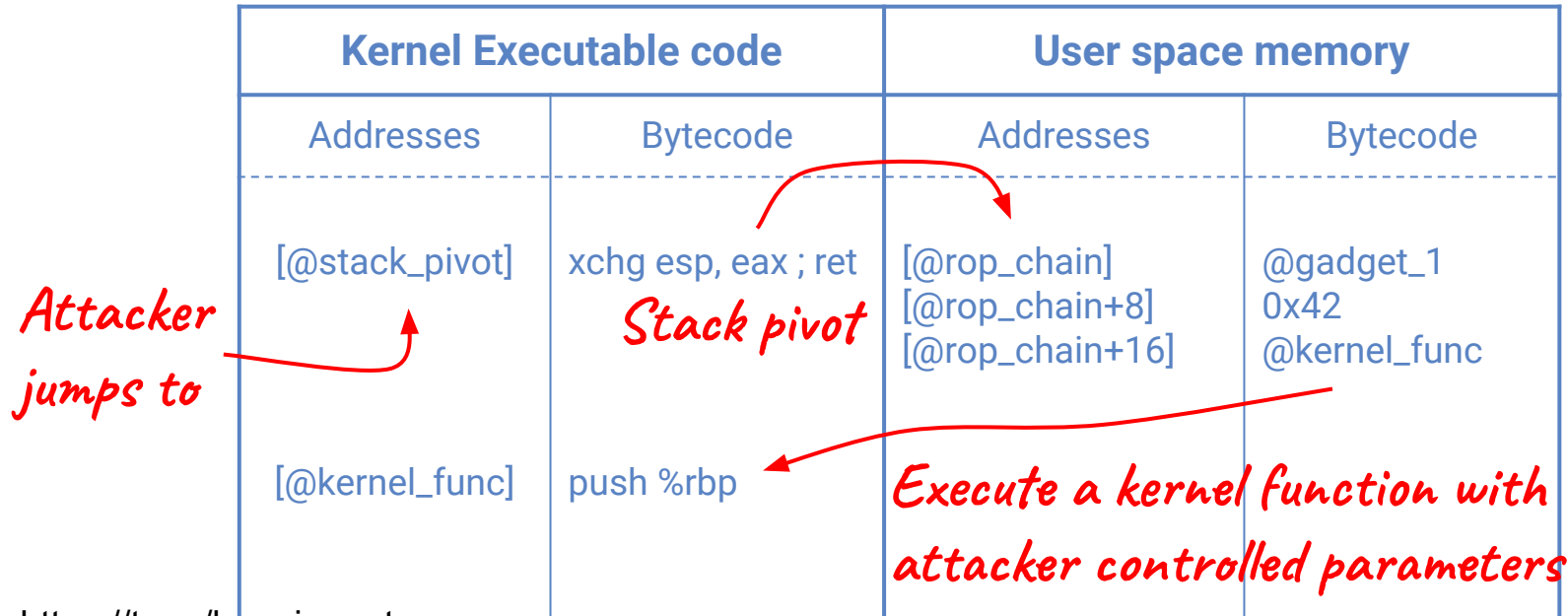
KRIe: SMEP & SMAP on a budget

Scenario 1: the attacker controls the address of the next instruction executed by the kernel



KRIe: SMEP & SMAP on a budget

Scenario 1: the attacker controls the address of the next instruction executed by the kernel



KRIe: SMEP & SMAP on a budget

Scenario 1: the attacker controls the address of the next instruction executed by the kernel

Kernel Executable code		User space memory	
Addresses	Bytecode	Addresses	Bytecode
<i>Attacker jumps to</i> [@stack_pivot]	xchg esp, eax ; ret <i>Stack pivot</i>	[@rop_chain] [@rop_chain+8] [@rop_chain+16]	@gadget 0x42 @kernel_func
@kernel_func	push %rbp	<i>Not possible with SMAP</i>	

KRIe: SMEP & SMAP on a budget

- SMEP would have prevented the CPU from executing code in user space executable memory
- Our example ROP chain will eventually call:

```
commit_creds(prepare_kernel_cred(0))
```

What can we do for machines without SMEP / SMAP ?

KRIe: SMEP & SMAP on a budget

- Place a kprobe on “prepare_kernel_cred” and check if the Stack pointer / Frame pointer / Instruction pointer registers point to user space memory

Demo

(Ubuntu Bionic 18.04 - Kernel 4.15.0-189-generic - SMAP disabled)

KRIe: SMEP & SMAP on a budget

- On a budget because:
 - Need to hook “all the functions called by exploits”
 - Blocking mode only works on 5.3+ kernels
 - An attacker will try to prevent our kprobe from firing ...

KRIe: SMEP & SMAP on a budget

- So ... how can one disable a kprobe ?
 - `echo 0 > /sys/kernel/debug/kprobes/enabled`
 - `sysctl kernel.ftrace_enabled=0`
 - Killing the user space process that loaded the kprobe

KRIe: SMEP & SMAP on a budget

- So ... how can one disable a kprobe ?

- `echo 0 > /sys/kernel/debug/kprobes/enabled`

- `sysctl kernel.ftrace_enabled=0`

- By killing the user space process that loaded the kprobe

→ Let's booby trap everything 🎉

KRle: Kernel security configuration

```
1) echo 0 > /sys/kernel/debug/kprobes/enabled
```

- Global switch that disarms all kprobes on a machine
- The ROP chain can be updated to call

```
write_enabled_file_bool(NULL, "0", 1, NULL)
```

KRle: Kernel security configuration

```
1) echo 0 > /sys/kernel/debug/kprobes/enabled
```

- Global switch that disarms all kprobes on a machine
- The ROP chain can be updated to call

```
write_enabled_file_bool(NULL, "0", 1, NULL)
```

→ Let's put a kprobe on it 🎉

KRIe: Kernel security configuration

```
1) echo 0 > /sys/kernel/debug/kprobes/enabled
```

- Even when enabled, a kprobe can *still* be bypassed:

@write_enabled_file_bool - No kprobe	@write_enabled_file_bool - With a kprobe
0x0: nop dword ptr [...]	0x0: callq 0xffffffff81a01cf0
0x5: push %rbp	0x5: push %rbp
0x6: mov %rsp,%rbp	0x6: mov %rsp,%rbp
0x9: push %r14	0x9: push %r14
0xb: push %r13	0xb: push %r13
0xd: push %r12	0xd: push %r12
...	...

KRle: Kernel security configuration

```
1) echo 0 > /sys/kernel/debug/kprobes/enabled
```

- Even when enabled, a kprobe can *still* be bypassed:

@write_enabled_file_bool - No kprobe	@write_enabled_file_bool - With a kprobe
0x0: nop dword ptr [...]	0x0: callq 0xffffffff81a01cf0
0x5: push %rbp	0x5: push %rbp
0x6: mov %rsp, %rbp	0x6: mov %rsp, %rbp
0x9: push %r14	0x9: push %r14
0xb: push %r13	0xb: push %r13
0xd: push %r12	0xd: push %r12
...	...

Jump here with the ROP chain →

KRIe: Kernel security configuration

```
1) echo 0 > /sys/kernel/debug/kprobes/enabled
```

→ Booby trap the function at random offsets 🎉

@write_enabled_file_bool - No kprobe	@write_enabled_file_bool - With kprobe(s)
0x0: nop dword ptr [...]	0x0: callq 0xffffffff81a01cf0
0x5: push %rbp	0x5: push %rbp
0x6: mov %rsp, %rbp	0x6: callq 0xffffffff81a01cf0
0x9: push %r14	0xb: push %r13
0xb: push %r13	0xd: callq 0xffffffff81a01cf0
0xd: push %r12	...
...	

KRle: Kernel security configuration

```
1) echo 0 > /sys/kernel/debug/kprobes/enabled
```

- “write_enabled_file_bool” writes 0 or 1 to a global variable called “kprobes_all_disarmed”
- An attacker could try to write 1 to it directly

KRIe: Kernel security configuration

```
1) echo 0 > /sys/kernel/debug/kprobes/enabled
```

- “write_enabled_file_bool” writes 0 or 1 to a global variable called “kprobes_all_disarmed”
 - An attacker could try to write 1 to it directly
- We can use a `BPF_PROG_TYPE_PERF_EVENT` program to periodically check the values of all sensitive kernel parameters 🎉

KRIe: Kernel security configuration

```
2) sysctl kernel.ftrace_enabled=0
```

- There is an eBPF program type dedicated to monitoring and enforcing `sysctl` commands :

```
BPF_PROG_TYPE_CGROUP_SYSCTL (kernels 5.2+)
```

- (Almost) all `sysctl` parameters are checked by KRIE's periodical check

KRle: Kernel runtime alterations

Scenario 2: the attacker is root on the machine and wants to persist its access by modifying the kernel runtime

- Insert a rogue kernel module
- Hook syscalls to hide their tracks
 - Using kprobes
 - By hooking the syscall table directly
- BPF filters are used to silently capture network traffic
- eBPF programs can also be used to implement rootkits

KRIe: Kernel runtime alterations

Scenario 2: the attacker is root on the machine and wants to persist its access by modifying the kernel runtime

→ KRIE monitors:

- ◆ All bpf() operations and insertion of BPF filters
 - ◆ Kernel module load / deletion events
 - ◆ K(ret)probe registration / deletion / enable / disable / disarm events
 - ◆ Ptrace events
 - ◆ Sysctl commands
 - ◆ Execution of hooked syscalls
- ... and more to come !

KRIe: Kernel runtime alterations

- All syscall tables are checked periodically with the `BPF_PROG_TYPE_PERF_EVENT` program trick
- KRIE is also able to detect and report when a process executes a hooked syscall

Demo

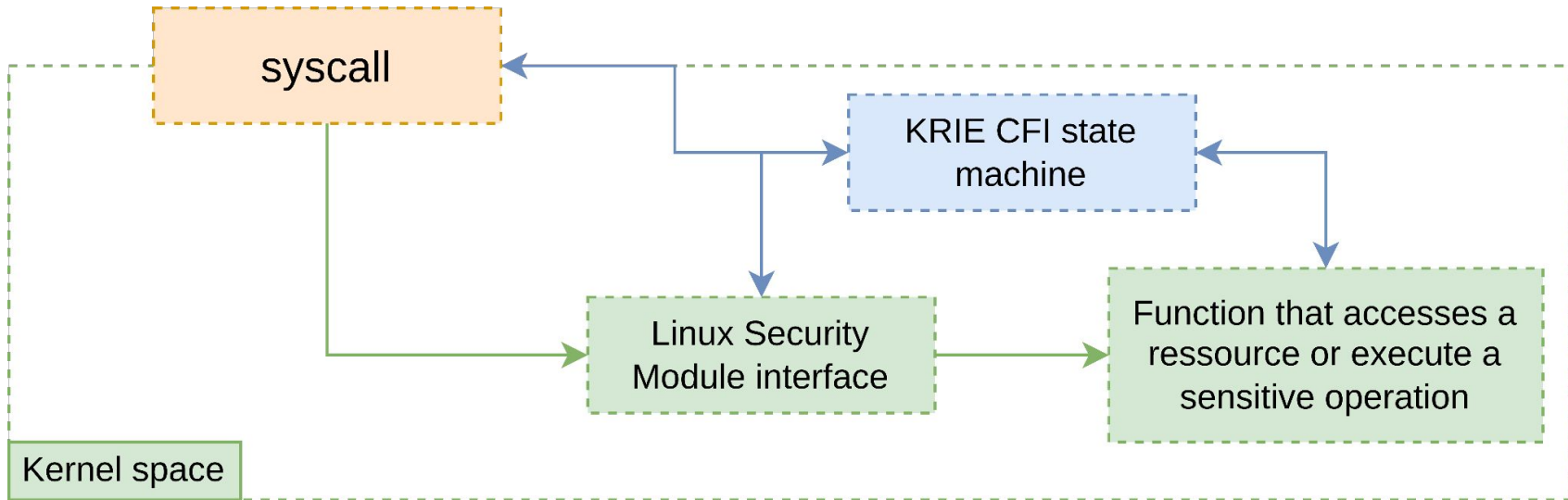
(Ubuntu Jammy 22.04 - Kernel 5.15.0-43-generic)

KR1e: Control flow Integrity (CFI)

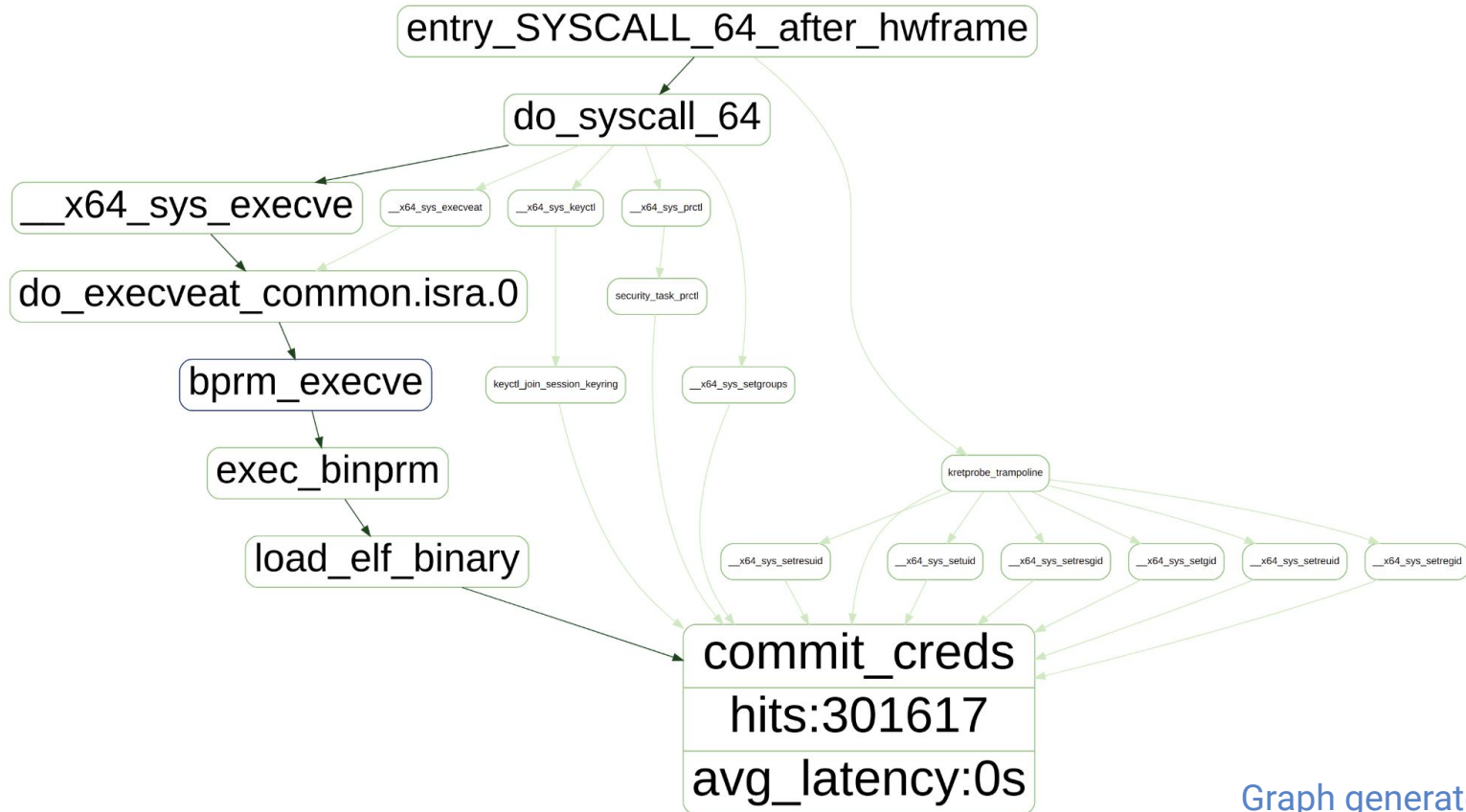
- Locks down the execution flows in the kernel by controlling call sites at runtime
- Usually added at compile time or implemented in hardware
- CFI is a great way to prevent ROP attacks
- These features aren't always available; specifically the hardware ones

KRIe: Control flow Integrity (CFI)

- KRIE locks down jumps between control points
- Both hook points and parameters are checked



KRIe: Control flow Integrity (CFI)



KRIe: Control flow Integrity (CFI)

The goal:

- Catch malicious calls to sensitive functions (via ROP)
- Detect logic bugs

But:

- Tedious process
- Hook points limitations

KRIe: Enforcement

- KRIE enables blocking features when available:
 - `bpf_override_return` helper (4.16+)
 - `BPF_PROG_TYPE_CGROUP_SYSCTL` programs (5.2+)
 - `bpf_send_signal` helper (5.3+)
 - LSM programs (5.7+)
- Every detection is configurable:
 - Log
 - Block
 - Kill
 - Paranoid

Performance

- 2 parts to consider
- Linux kernel compilation time

	User space CPU time		Kernel space CPU time		Total elapsed time
Without KRle	4,320s	88%	568s	12%	5:53.14
With KRle (all features)	4,517s	68%	2,097s	32%	8:15.76
	+4.5%		+270%		+40%
With KRle (syscall hook check disabled on syscall entry)	4,380s	88%	585s	12%	5:58.36
	+1%		+3%		+1%

<https://t.me/learningsnets> (Benchmark run on a 5.15.0 kernel, 11th Gen Intel(R) Core(TM) i9-11950H @ 2.60GHz, 32GB of RAM, average on 10 iterations)

Thanks

- Powerful defensive tools can be implemented with eBPF
- eBPF is not really the ideal technology to detect kernel exploits
- KRle is realistically a last resort, not a bulletproof strategy

<https://github.com/Gui774ume/krie>



<https://t.me/learningnets>

