THE LINUX FOUNDATION

OPEN SOURCE SUMMIT KOREA

서울

Kernel Live Patching: Mitigating CVEs With Zero Downtime

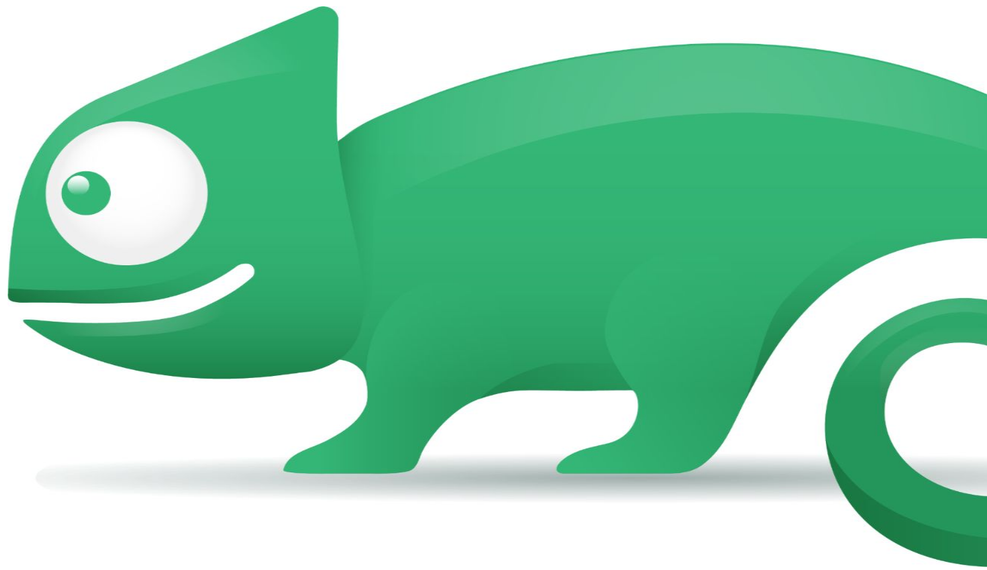
Shung-Hsi Yu · SUSE

12 August 2026

Shung-Hsi Yu

SUSE · based in Taitung, Taiwan

Linux kernel — BPF, and ~~live patching~~



A kernel CVE lands with a CVSS of 8.8.

Your fleet is 400 machines.

Your next maintenance window is in three weeks.

- **Reboot all 400 machines**

  Drain, reboot, verify, repeat. Tonight, because the CVE is 8.8.

- **Stay vulnerable until the maintenance window**

  Sign the exception. Three weeks. Maybe six.

There is a third button.

- **Reboot all 400 machines**

  Drain, reboot, verify, repeat. Tonight, because the CVE is 8.8.

- **Stay vulnerable until the maintenance window**

  Sign the exception. Three weeks. Maybe six.

- **Live-patch it**

  The fix lands now. Nothing stops, nothing restarts. The reboot still happens — later, on your schedule.

Live patching does not abolish reboots.

It decouples the security deadline from the maintenance window.

You still reboot at the maintenance window. It just isn't the CVE that decides when.

The setup

IN MEMORY, ON A MACHINE YOU CAN'T STOP

`old_func`

the function with the CVE

THE FIX, ONCE YOU'VE COMPILED IT

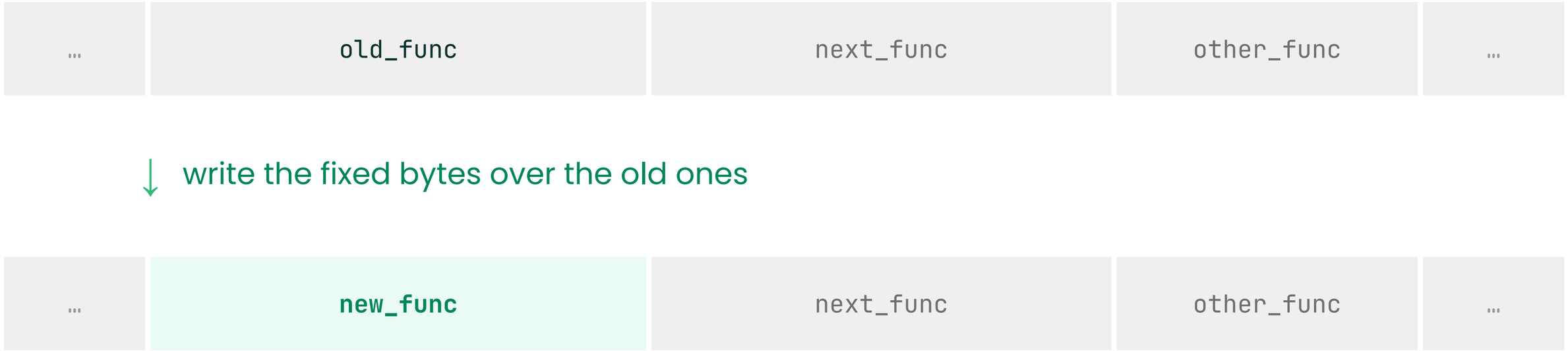
`new_func`

the same function, fixed

Every call that runs **`old_func`** has to run **`new_func`** instead.

The naive idea: just overwrite it

KERNEL TEXT, IN MEMORY

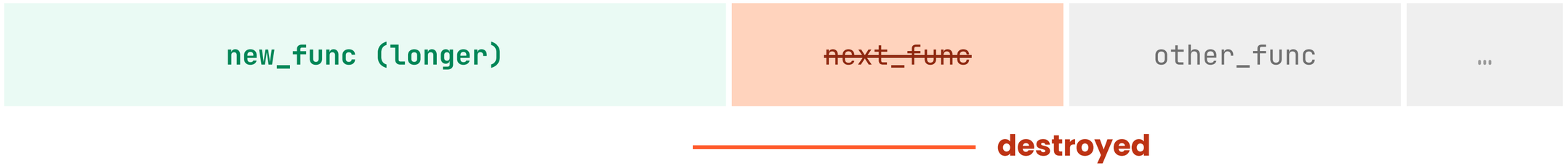


Kernel text is memory like any other. Nothing in the machine stops you from writing to it.

It doesn't fit

BEFORE

AFTER, IF YOU OVERWRITE IN PLACE



No room after a function — the next one starts where it ended.

Push it down, and it lands on `other_func`.

And that's only one of the reasons.

So don't touch it. Put the fixed copy somewhere else.

KERNEL TEXT

`old_func`

untouched, still there

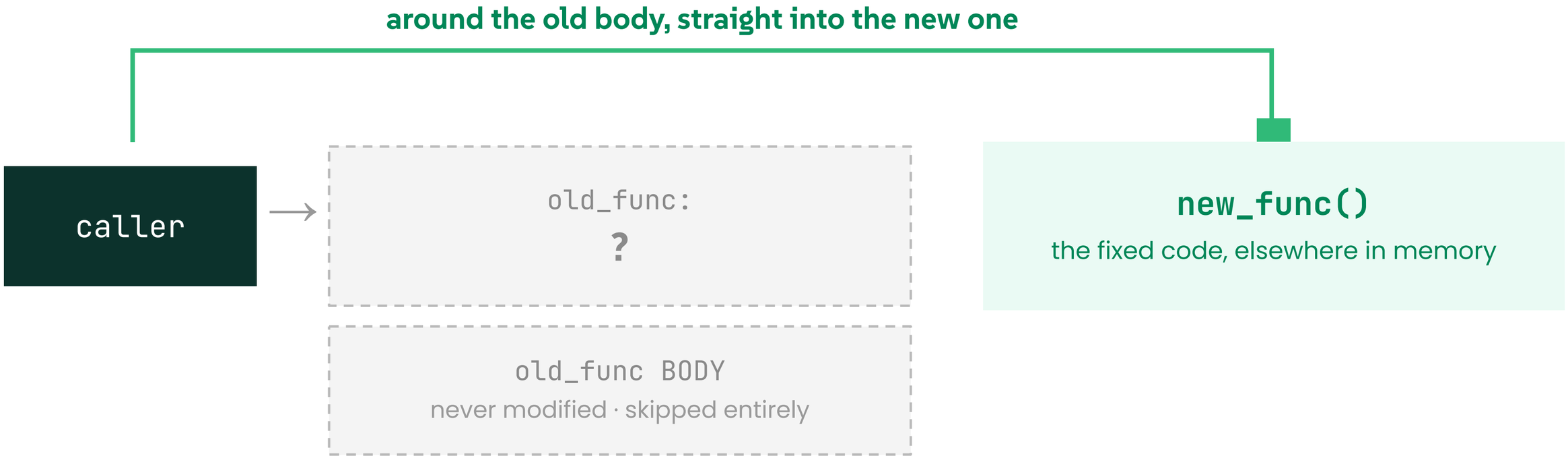
FREE KERNEL MEMORY

`new_func`

the fixed copy, compiled on its own

Nothing has to move. Nothing has to fit. Both versions are in memory at once.

What we want



The whole problem is that one box.

That question, and three more, are the rest of the talk.

1. How does the call get **redirected**?
2. How do we know that's safe?
3. How is the patch built?
4. Who does this for me — and when does it not work?

The answer is already in the kernel, and it's called `ftrace`

Nearly every kernel function begins with **five bytes** that do nothing.

AS BOOTED — NOTHING ATTACHED

```
old_func:   +0x00   0f 1f 44 00 00   nop   +0x05
55                  push   rbp
  +0x06   48 89 e5          mov    rbp,rsp
  ...
```

WHEN SOMETHING ATTACHES

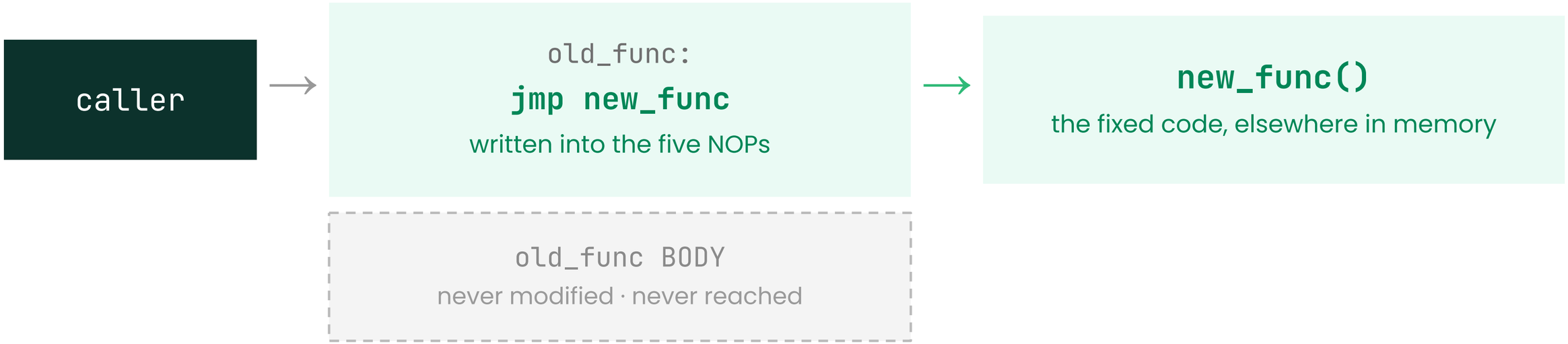
```
old_func:   +0x00   e8 3c a1 12 00   call
ftrace_caller   +0x05   55                  push
rbp
  +0x06   48 89 e5          mov    rbp,rsp
  ...
```

Same address, same five bytes. One instruction swapped for another, on a running kernel.

Enabled on almost every distro kernel. Nothing attached costs one NOP.

Point the hook at the new function

SIMPLIFIED



That's the gist of live patching.

This isn't a product. It's already in the kernel you're running.

```
$ zgrep CONFIG_LIVEPATCH /proc/config.gz CONFIG_LIVEPATCH=y $ ls
/sys/kernel/livepatch/ (empty)
```

Merged in v4.0, 2015. On by default in every major distro kernel.

That directory is empty because nothing is patched — not because you can't.

So what do you load into it?

An ordinary kernel module. One line makes it special.

```
MODULE_INFO(livepatch, "Y");
```

One line, in one function

`cat /proc/cmdline` shows the kernel's boot arguments. It is served by one function: print `saved_command_line`, print a newline, return.

```
--- a/fs/proc/cmdline.c
+++ b/fs/proc/cmdline.c
@@ -8,6 +8,7 @@
 static int cmdline_proc_show(struct seq_file *m, void *v)
 {
   seq_puts(m, saved_command_line);
+  seq_puts(m, " klp_demo=1");    seq_putc(m, '\n');
   return 0;
 }
```

```
=== 1. /proc/cmdline BEFORE patch ===
  console=ttyS0 loglevel=7 rdinit=/init oss_korea_demo

=== 2. loading livepatch-klp-demo-cmdline.ko ===
[    1.102350] livepatch: enabling patch 'livepatch_klp_demo_cmdline'
[    1.106770] livepatch: 'livepatch_klp_demo_cmdline': starting patching transition
[    2.007678] livepatch: 'livepatch_klp_demo_cmdline': patching complete

=== 3. /proc/cmdline AFTER patch ===
  console=ttyS0 loglevel=7 rdinit=/init oss_korea_demo klp_demo=1

=== 4. disabling patch via sysfs ===
[    2.024995] livepatch: 'livepatch_klp_demo_cmdline': starting unpatching transition
[    2.047608] livepatch: 'livepatch_klp_demo_cmdline': unpatching complete

=== 5. /proc/cmdline AFTER disable ===
  console=ttyS0 loglevel=7 rdinit=/init oss_korea_demo
```

live kernel, no reboot

captured on a 6.18.1 test VM

What that log shows

1. Nothing was restarted. No process killed, no connection dropped, no reboot — the running kernel changed behaviour underneath everything.

2. It went both ways. One write to sysfs put the original function back — because the original was never modified.

3. **The kernel logged a start and a finish — not a single event. Applying it took time you could measure.**

```
livepatch: 'livepatch_klp_demo_cmdline': starting patching transition
livepatch: 'livepatch_klp_demo_cmdline': patching complete
```

1 How does the call get redirected?

2 How do we know that's **safe**?

3 How is the patch built?

4 Who does this for me — and when does it not work?

Go back to those two lines

```
[    1.106770] livepatch: 'livepatch_klp_demo_cmdline': starting patching transition [
2.007678] livepatch: 'livepatch_klp_demo_cmdline': patching complete
```

a start, then a finish

Why isn't it instant?

A patch is rarely one function

This one changes two — and one of them **calls** the other. The fix moves the lock from the caller down into the callee.

A TASK RUNNING THE **OLD** VERSION OF BOTH

`old_func1`

`mutex_lock(&lock)`

↓ calls

`old_func2`

assumes the caller holds it

✓ **taken exactly once**

A TASK RUNNING THE **NEW** VERSION OF BOTH

`new_func1`

no longer takes the lock

↓ calls

`new_func2`

`mutex_lock(&lock)`

✓ **taken exactly once**

Either version is correct. The trouble is one task running one of each.

So why not switch every task at once?

Because a task can already be **inside the caller** when the patch lands.

	OLD VERSION		NEW VERSION
THE TASK IS ALREADY IN HERE — so it took the lock	**`old_func1`** `mutex_lock(&lock)`	→ patched into	**`new_func1`** no longer takes the lock

THE PATCH LANDS — EVERY TASK, AT ONCE

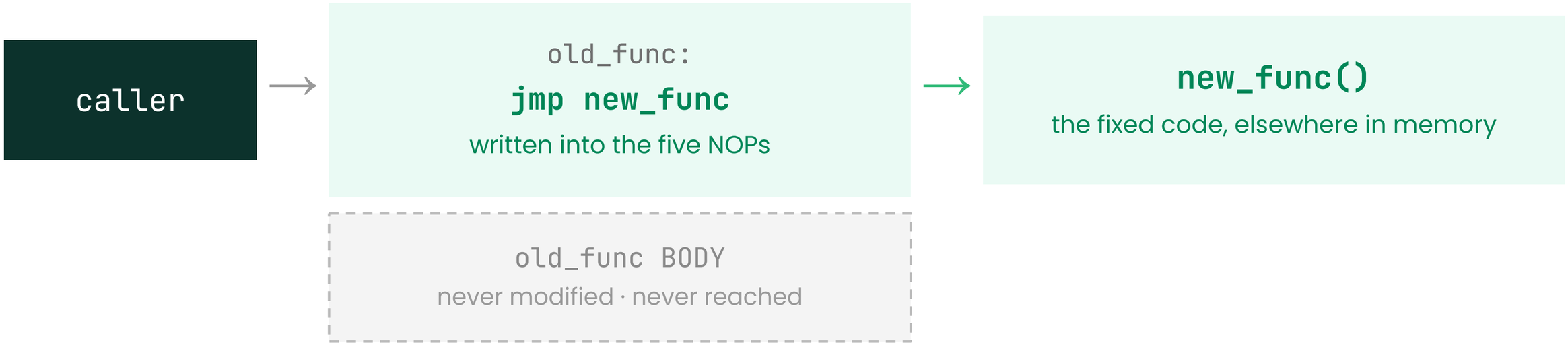
…AND THEN IT CALLS — still holding the lock	**`old_func2`** assumes the caller holds it	→ patched into	**`new_func2`** `mutex_lock(&lock)`

It ran `old_func1` and then `new_func2` — it takes the same lock twice. **Deadlock.**

The task was already inside the old function. There was no safe instant to flip it.

Where we got to

SIMPLIFIED

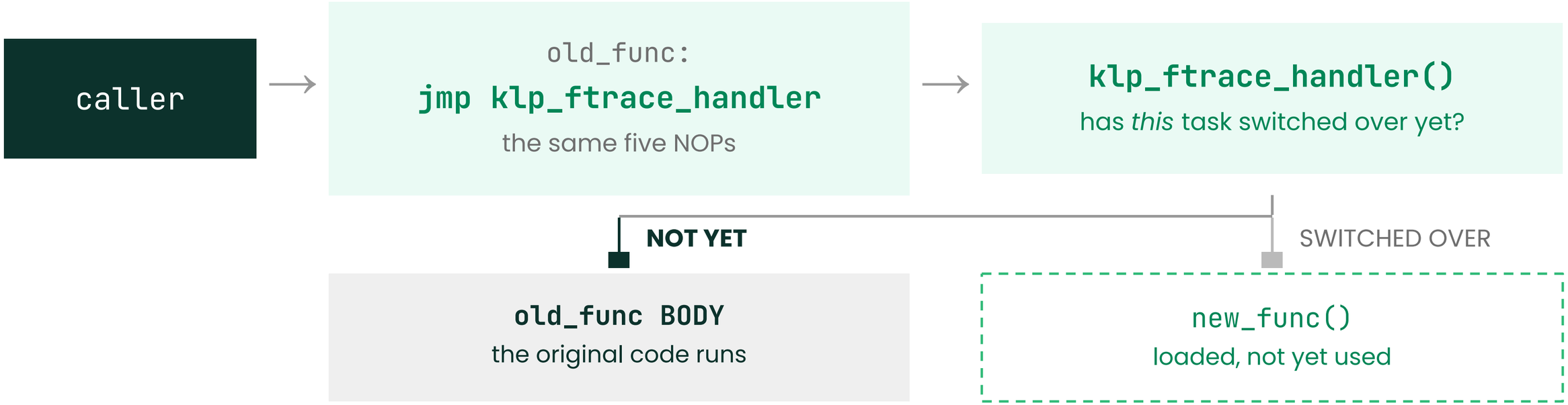


That's the version I said was simplified.

It's actually more complicated than that

STILL SIMPLIFIED

The jump goes to a handler, not to the fix.

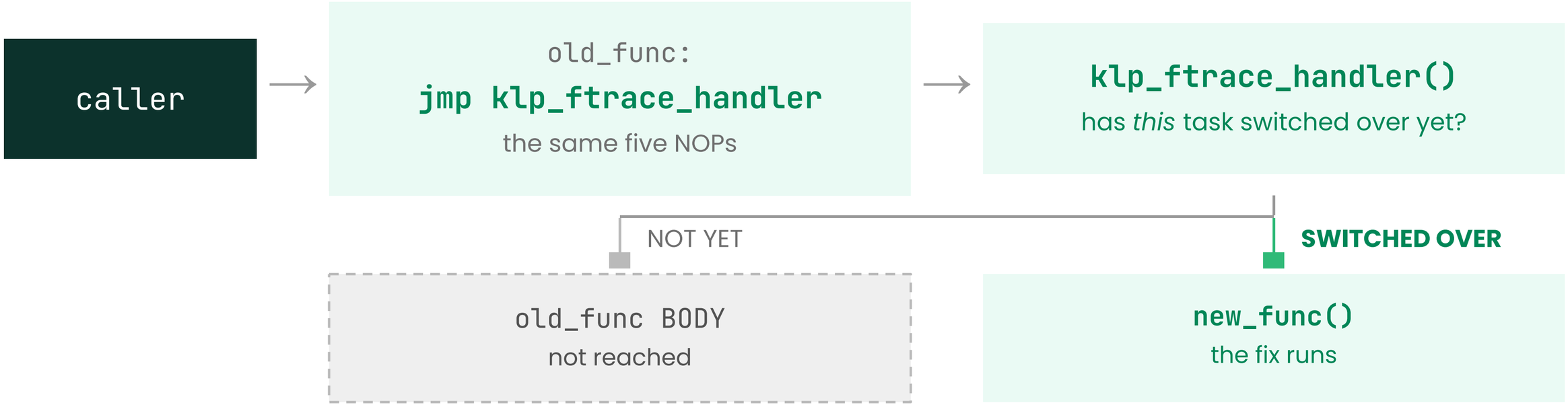


Until this task switches over, the original code is still what runs.

The same call, once the task has switched over

STILL SIMPLIFIED

Nothing else moved.



Same idea, one extra stop — and the stop is where the decision gets made.

So the kernel switches over one task at a time.

Each task flips only at a moment when the kernel can see it isn't inside anything the patch touches. Until the last one flips, old and new run side by side, on purpose.

That's the transition. That's the gap between those two log lines.

Three things that follow

1. Applying a patch has a **duration** , not an instant. Usually fast enough that nothing notices; on a busy machine, longer.

2. Occasionally one task sits somewhere awkward and holds things up. The kernel keeps retrying, and nudges it along.

3. **You can reverse a transition while it's still running. A patch that can't finish never wedges the machine.**

1 How does the call get redirected?

2 How do we know that's safe?

3 How is the patch **built**?

4 Who does this for me — and when does it not work?

Two ways to make one of these

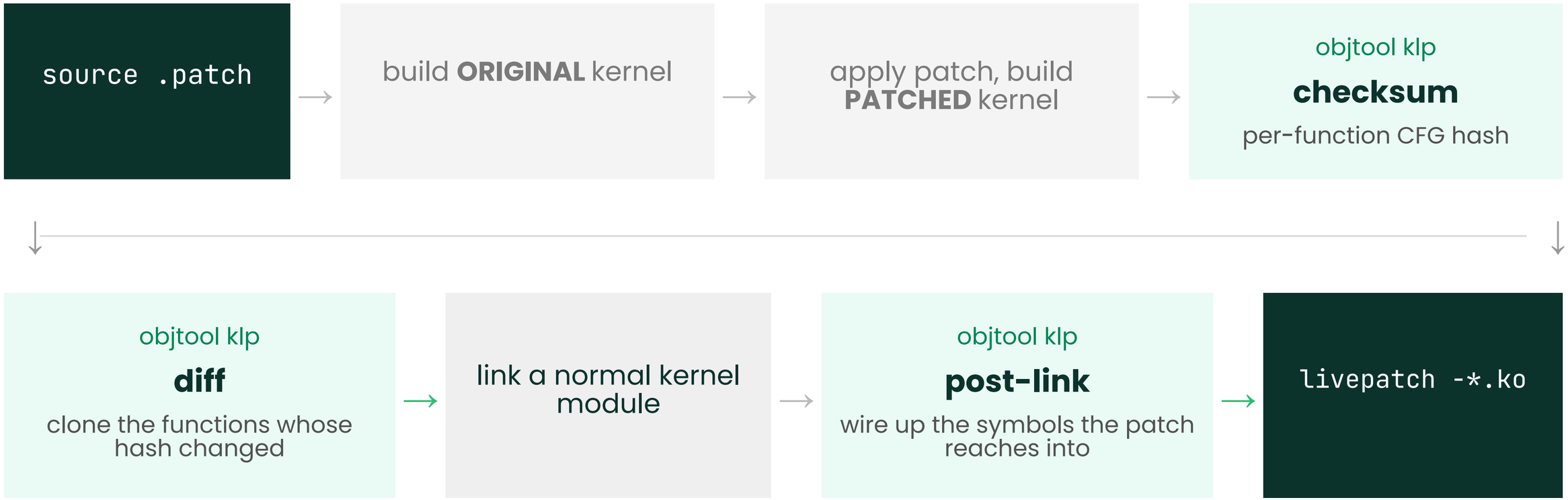
Write it by hand	Diff two kernels
write the replacement function yourself	hand it a source `.patch`
simple, works everywhere	it builds the kernel twice and extracts what changed
you manage every reference yourself	it works out what actually changed, including what the compiler did

The tool that does this was merged **into the kernel tree** in v7.0.

`scripts/livepatch/klp-build`

Twelve years out-of-tree as kpatch-build — now maintained next to the code that breaks it. x86_64 for now.

What it actually does



It builds the whole kernel, twice.

Compilers don't produce identical code for identical source

inlining decisions · line numbers shifting · section layout · jump labels

A naive binary diff flags half the kernel.

So it fingerprints each function's control flow first, and compares the fingerprints.

```
Validating patch(es)
Building original kernel
Copying original object files
Fixing patch(es)
Building patched kernel
Copying patched object files
```

[16 minutes later]

```
Diffing objects vmlinux.o: changed function: skcipher_sock_destruct
vmlinux.o: changed function: skcipher_recvmsg
vmlinux.o: changed function: aead_sock_destruct
vmlinux.o: changed function: aead_recvmsg
vmlinux.o: changed function: af_alg_pull_tsgl
vmlinux.o: changed function: af_alg_count_tsgl
Building patch module: livepatch-copyfail.ko SUCCESS
```

`livepatch-copyfail.ko` — 27,712 bytes on disk · 6 changed functions · a real CVE fix

Live patching is cheap to **apply** and expensive to **produce**.

That asymmetry is exactly why vendors do it for you.

1 How does the call get redirected?

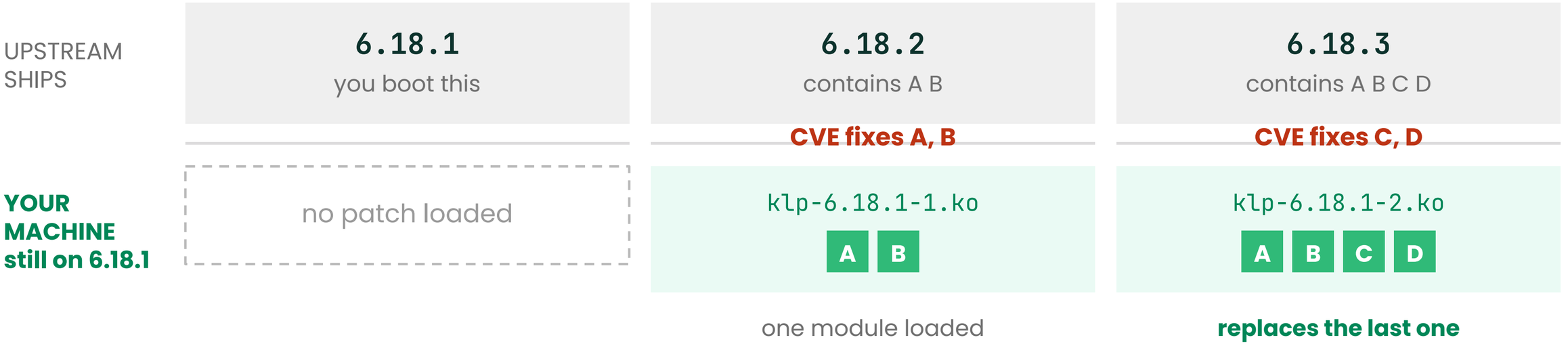
2 How do we know that's safe?

3 How is the patch built?

4 Who **does this for me** — and when does it not work?

In production, patches are cumulative

Each new patch carries every fix before it, and atomically retires the one it replaces.

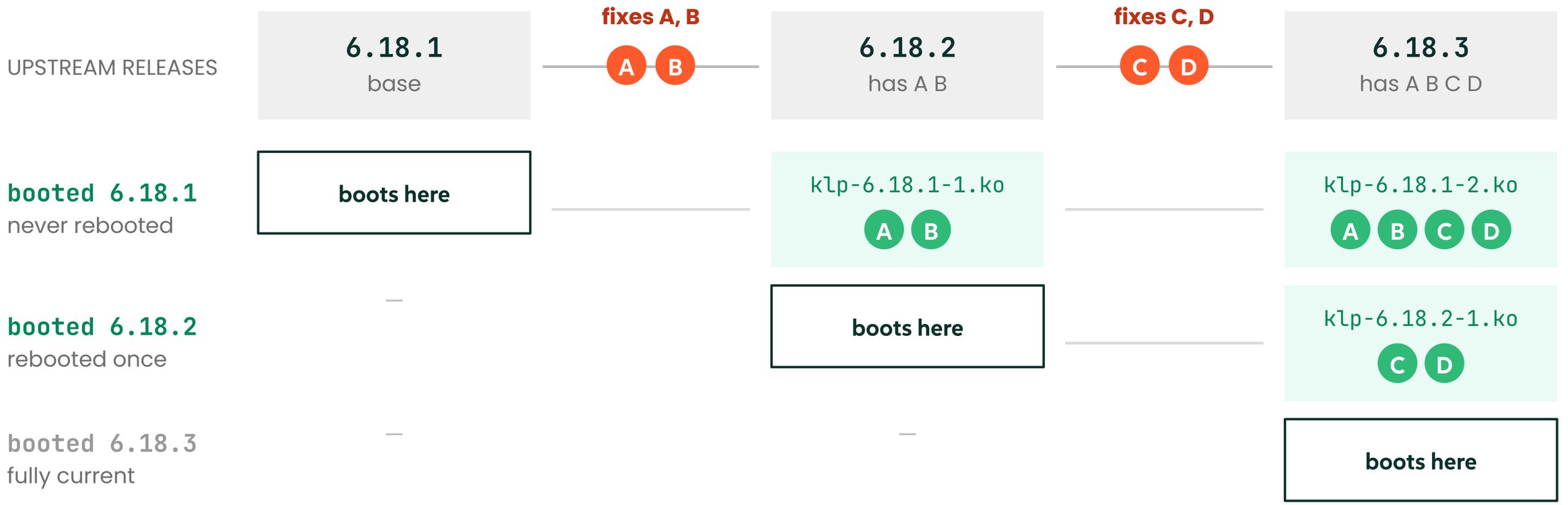


One module is loaded at a time — never a stack of four.

Reboot into 6.18.3 and all four fixes are in the kernel itself. The patch stream starts over, empty.

Now do that for every version still in the field

A patch built for one base version won't load on another. Every version anyone still runs needs its own stream.



Three releases, three live modules — and it's a triangle, not a line.

A support cycle is dozens of releases. Twenty of them is 190 modules, each built, tested and signed separately.

MODULES TO MAINTAIN

`N(N-1)/2`

And the fix rarely applies cleanly

The upstream fix is written against mainline. Your machine is running a kernel that moved on months ago.

`the upstream fix`
written against mainline

→

`your base kernel`
the code around it has moved

→

doesn't apply
not a build problem

1. Someone rewrites the fix against the code that's actually there.
2. Then proves the rewrite still closes the CVE — and changes nothing else.
3. Then again, for every stream still in the field.

That isn't build time. That's a person who knows the subsystem.

Who ships live patching

	PRODUCT	LINEAGE	NOTABLE
SUSE	SLE Live Patching	kGraft	also live-patches userspace libraries (libpulp)
Red Hat	RHEL kernel live patching	kpatch	kpatch-build is the direct out-of-tree ancestor of klp-build
Canonical	Ubuntu Livepatch	own service	bundled with Ubuntu Pro; free tier for personal use on a small number of machines
Oracle	Ksplice	predates upstream livepatch entirely	also covers userspace and the hypervisor
TuxCare	KernelCare	third party	multi-distro — the option when your distro doesn't offer one

...or build your own — new this year.

Checked 2026-08-05 · re-verify every row before presenting

What can and can't be live-patched

Change what a function **does** — a bounds check, a missing lock, an error path	**Yes.** The sweet spot, and it's most security fixes.
Change the **shape of data** already sitting in memory	**Mostly no.** There are workarounds for adding a field, but they're fiddly and the patch author has to design for it.
Change how the system was set up **at boot**	**No.** That code already ran.
New features, big refactors, new hardware support	**No** — and that's not what it's for.

What every one of them has in common

1. **A curated subset** — typically high/critical severity. Not every CVE.
2. **Tied to your exact kernel build.** You can't take one vendor's patch to another distro, or even to a different kernel version on the same distro.
3. **The stream has an end date**, tied to the base kernel's support life.

---

4. **You still reboot eventually.**

Do you actually need this?

Yes, if

rebooting is expensive for you — large fleet, stateful workloads, long drain times, HA choreography

and your compliance clock is shorter than your maintenance window

Probably not, if

you can already reboot on a weekly cadence without anyone noticing

rolling reboots are simpler, and simpler wins

SUSE

SUSE Linux Enterprise Live Patching

The kGraft lineage, on the same in-kernel machinery this talk just walked through.

Shung-Hsi Yu · SUSE Open Source Summit Korea 2026

suse.com/products/live-patching

Live patching does not abolish reboots.

It decouples the security deadline from the maintenance window.

Three things to take away

1. It's the kernel's existing tracing hook, pointed somewhere new.

2. The hard part isn't applying a patch, it's **producing** one — which is why you'll almost certainly buy it rather than build it.

3. It buys you **scheduling freedom**, not immortality.

Questions?

WHERE TO LOOK NEXT

```
Documentation/livepatch/
samples/livepatch/
scripts/livepatch/klp-build
tools/testing/selftests/livepatch/
```

Shung-Hsi Yu · SUSE Open Source Summit Korea 2026