

OLF *Live*

MENTORSHIP SERIES



Kernel Livepatching: An Introduction

Joe Lawrence, Principal Software Engineer, Red Hat
Marcos Paulo de Souza, Software Engineer, SUSE

Agenda

- Intro and ksplice beginnings
- kpatch & kGraft open source (re)implementations
- Upstream collaboration
- Examples
- Limitations

Scope and Expectations

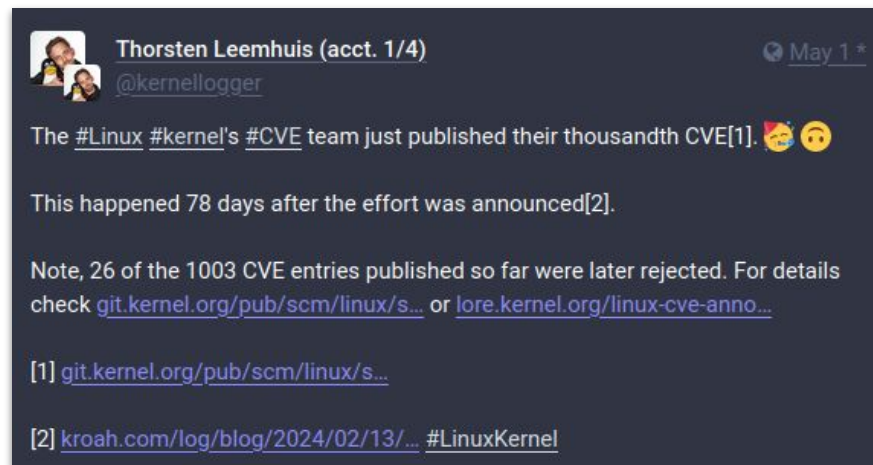
- An overview of kernel livepatching technology
- Introduction of the risks of livepatches
- Resources to look for when creating livepatches




Intro and ksplice beginnings

Why livepatches

- [498 published CVEs for upstream Linux Kernel in 2024 until 08 May 2024](#)
 - Since Kernel became a CNA (CVE Numbering Authority) on 13 Feb 2014.
- Rebooting modern servers can push expensive and limited hardware offline
 - Starting services like databases from boot may require a long time priming caches
 - Some hypervisor guests are not easily migratable



 **Thorsten Leemhuis** (acct. 1/4) May 1 *
[@kernellogger](#)

The [#Linux](#) [#kernel](#)'s [#CVE](#) team just published their thousandth CVE[1]. 🥳🙄

This happened 78 days after the effort was announced[2].

Note, 26 of the 1003 CVE entries published so far were later rejected. For details check [git.kernel.org/pub/scm/linux/s...](#) or [lore.kernel.org/linux-cve-anno...](#)

[1] [git.kernel.org/pub/scm/linux/s...](#)

[2] [kroah.com/log/blog/2024/02/13/... #LinuxKernel](#)



OLFLive MENTORSHIP SERIES

Who creates livepatches?

ORACLE® | Ksplice®

 **TuxCare**
We Take Care of Linux

 Canonical

 aws

 SUSE 

 **Red Hat**

History: ksplice

- [Ksplice: An Automatic System for Rebootless Kernel Security Updates](#), Jeff Arnold (2008)
 - “Since rebooting can cause disruption, system administrators often delay performing security updates, despite the risk of compromises.”
- Announced on the [LKML](#) in 2008
- MIT students create Kpatch, Inc. in 2009
- Acquired by Oracle on 2011, now closed source :(

ksplice key concepts

- Function granularity - most C functions have singular, well defined entry points
- Finding a safe time to update - functions should not be mid-execution by any thread
 - Safety check implemented via kernel's **stop_machine_run()** call
 - Cannot upgrade non-quiescent kernel functions like **schedule()**
- Patches loaded via kernel modules
- Ability to apply subsequent hot patches

kpatch & kGraft - initial LMKL RFCs

- 30 Apr 2014 - [\[RFC 00/16\] kGraft](#)
 - Linux kernel online patching developed at SUSE
 - Per-task consistency: each thread either sees the old version or the new version of functions
 - Transition only when the task exits kernel mode
- 1 May 2014 - [\[RFC PATCH 0/2\] kpatch: dynamic kernel patching](#)
 - Developed by Red Hat, modeled after initial ksplice project
 - **stop_machine()** - all tasks see old or new version of functions
 - Requires reliable stack unwinding (see [Reliable Stacktrace](#) kernel doc)

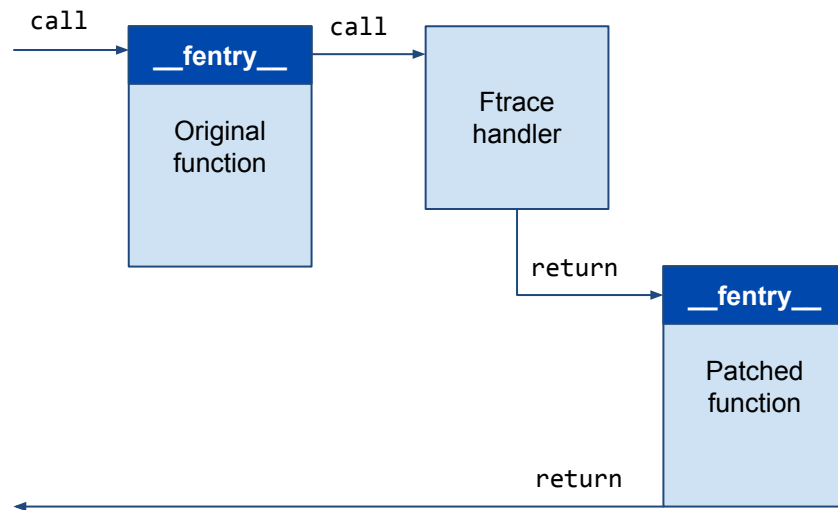


kpatch & kGraft open source (re)implementations

kpatch & kGraft - common ftrace implementation

Both RFCs leveraged the kernel's existing [ftrace function hooking](#) mechanism to detour control flow:

1. Original function entry
2. Ftrace caller redirection
3. Patched function entry
4. Return skips original function, returns directly to its caller



Userspace function padding with mcount

We need to space at the beginning of functions to insert redirection calls. This can be seen in userspace code when building with gcc's -pg profiling option.

Example: <https://godbolt.org/z/6787dxfvM>

When compiling with -pg, this 5-byte sequence is added to the beginning of the **square()** function:

```
e8 00 00 00 00
```

```
call    d <square(int)+0xd>  
        R_X86_64_PLT32 mcount-0x4
```

Kernel function padding with fentry

Similar function entry padding is provided in kernel builds, too.

Dynamic ftrace feature

- Turns on the `-pg` switch in the compiling of the kernel
- Starting with gcc version 4.6, the `-mfentry` switch has been added for x86, which calls “`__fentry__`” instead of “`mcount`”
- On boot up, dynamic ftrace code updates all fentry locations into nops.

See [ftrace.html](#) kernel docs for more on `__fentry__` details

kernel fentry (Background slide)

Step 1 - Determine the latest upstream kernel version:

```
$ git ls-remote --tags \  
  git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git | \  
  grep -o 'v[0-9].[0-9]*$' | sort --version-sort --key=2 | tail --lines=1  
v6.8
```

Step 2 - Shallow clone the latest upstream kernel tree:

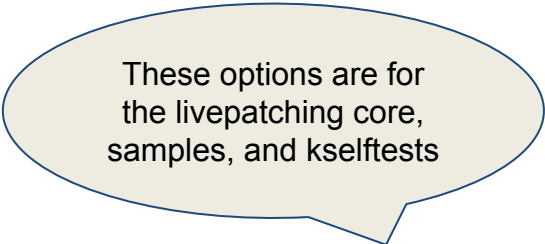
```
$ git clone --depth=1 --branch v6.8 \  
  git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
```

Steps 1 & 2 are optional
if you already have a
local kernel tree.

kernel fentry (Background slide)

Step 3 - Configure the kernel with arch defaults, plus livepatching settings:

```
$ cd linux
$ make defconfig
$ ./scripts/config --set-val CONFIG_FTRACE y \
  --set-val CONFIG_KALLSYMS_ALL y \
  --set-val CONFIG_FUNCTION_TRACER y \
  --set-val CONFIG_DYNAMIC_FTRACE y \
  --set-val CONFIG_DYNAMIC_DEBUG y \
  --set-val CONFIG_LIVEPATCH y \
  --set-val CONFIG_RUNTIME_TESTING_MENU y \
  --set-val CONFIG_SAMPLES y \
  --set-val CONFIG_SAMPLE_LIVEPATCH m
$ make olddefconfig
```



These options are for
the livepatching core,
samples, and kselftests

kernel fentry padding for cmdline_proc_show()

Step 4 - Compile fs/proc/cmdline.c -> cmdline.o:

```
$ make --silent --jobs=$(nproc) fs/proc/cmdline.o
```

Step 5 - Disassemble cmdline.o with binutils's objdump utility:

```
$ objdump --disassemble-all --reloc \  
    --section=.text fs/proc/cmdline.o  
  
...  
0000000000000010 <cmdline_proc_show>:  
10:   f3 0f 1e fa                endbr64  
14:   e8 00 00 00 00            call   19 <cmdline_proc_show+0x9>  
15:   R_X86_64_PLT32    __fentry__-0x4
```


kpatch & kGraft - differences

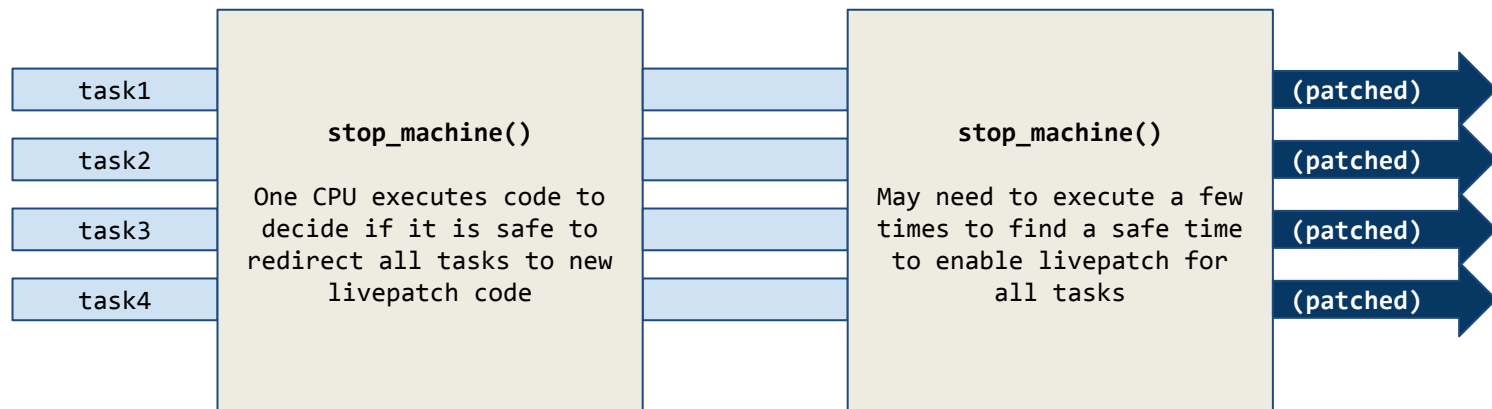
- Kpatch used `stop_machine()` to quiesce the system and apply patches only when safe, retrying when needed.
- kGraft implemented a “lazy” per-task approach that switched tasks individually when ready, leaving the system otherwise running during the transition to an increasingly patched system .

Kpatch RFC: stop_machine() update implementation

Tasks are redirected en masse, but only when it's safe for ALL of them

PROs = conceptually simple

CONs = may require many retries, long latencies

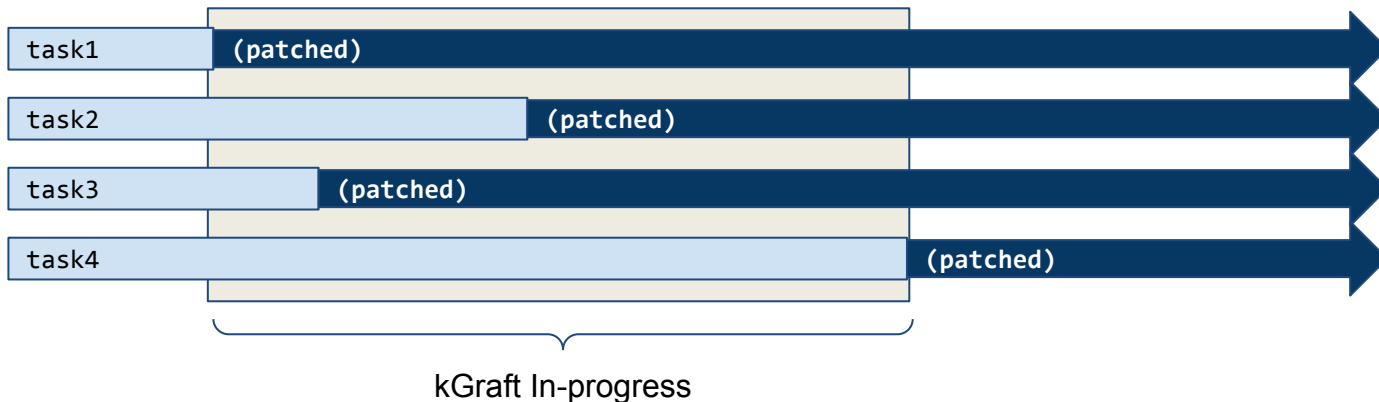


kGraft RFC: “Lazy” per-task switching implementation

Tasks are individually redirected as it is safe for them to do so

PROs = livepatching takes as long as it needs, system remains executing tasks

CONs = conceptually complex





Upstream collaboration

Which RFC to merge?

- Kernel community requested a single API and livepatching approach
- Some favored **stop_machine()** as a simpler way to make sure livepatching works
- There were no apparent bugs with “lazy” per-task approach, but still couldn’t livepatch kthreads either

2014 Dec 16

[b700e7f03df5](#) (“livepatch: kernel: add support for live patching”)

- Introduces code for the live patching core
- Represents the greatest common functionality set between kpatch and kgraft and can accept patches built using either method
- Does not implement any consistency mechanism to ensure old and new code do not run together

A little while later ...

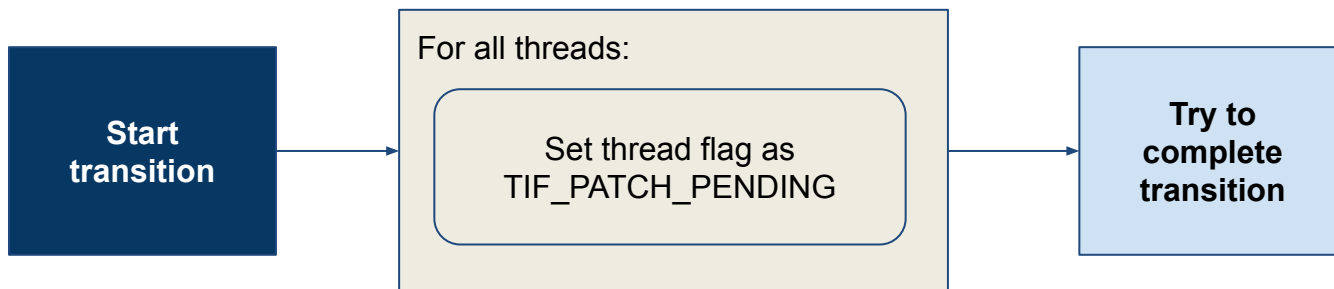
Many proposals, reviews, ideas, and comments from the community, including: Miroslav Benes, Masami Hiramatsu, Seth Jennings, Jiri Kosina, Ingo Molnar, Petr Mladek, Vojtech Pavlik, Josh Poimboeuf, Steven Rostedt, Jiri Slaby and others

2017 Feb 13

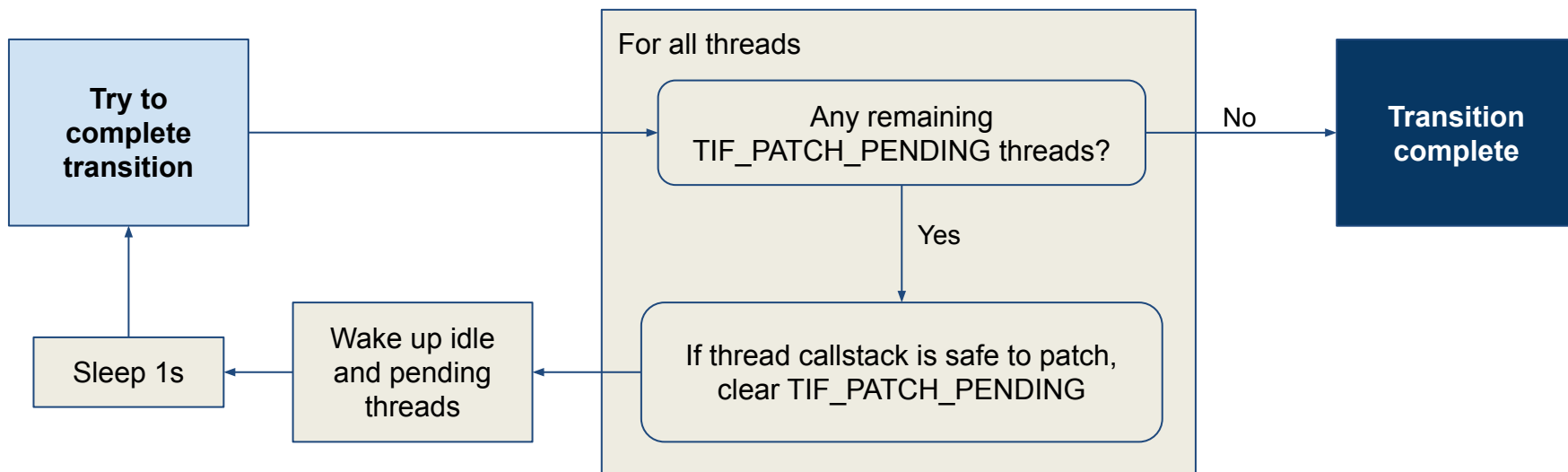
[d83a7cb375ee](#) (“livepatch: change to a per-task consistency model”)

- Change livepatch to use a basic per-task consistency model
- Hybrid of kGraft and kpatch: it uses kGraft's per-task consistency and syscall barrier switching combined with kpatch's stack trace switching

Livepatch consistency model: Starting Transition



Livepatch consistency model: Completing Transition



Livepatch API - Basic Data Structures

```
klp_foo() { ... } // replacement function code

struct klp_patch // top-level data structure
    .replace = false // replace all actively used patches

    struct klp_object [] // patch targets:
        .name = "moduleX" // module name or NULL for vmlinux

        struct klp_func [] // original functions and replacements
            .old_name = "old_foo" // old:new function associations
            .new_func = klp_foo()

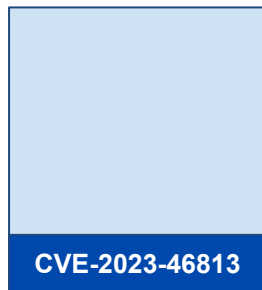
klp_enable_patch(&klp_patch); // do it!
```

Atomic Replace livepatches

- Disable all previously applied livepatches
- Useful when creating new livepatches containing all previous fixes
- Easier to manage when dealing with tens of livepatches

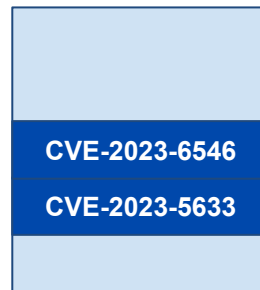
```
// .replace struct member as true
static struct klp_patch patch = {
    .mod = THIS_MODULE,
    .objs = objs,
    .replace = true,
};
```

livepatch-1.ko



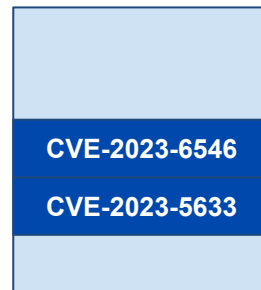
+

livepatch-2.ko



=

System State

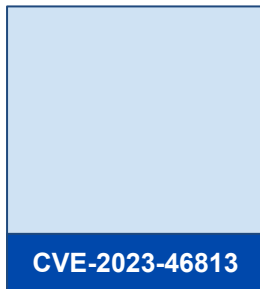


Stacked livepatches

- Applied without interfering with previous livepatches
- Not *stacked*, but each livepatch is managed individually

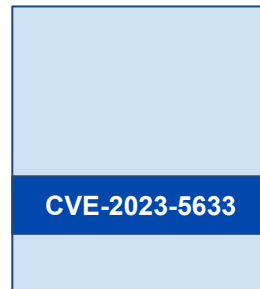
```
// .replace struct member as false
static struct klp_patch patch = {
    .mod = THIS_MODULE,
    .objs = objs,
    .replace = false,
};
```

livepatch-1.ko



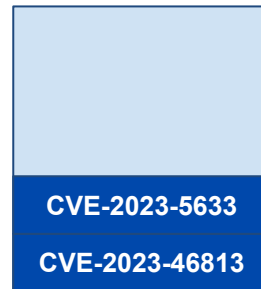
+

livepatch-2.ko



=

System State



Upstream - Currently

- Supported on x86_64, ppc64le, ppc32 and s390x
 - Aarch64 support ongoing
- Multiple companies relying on livepatching for their fleets
 - Meta
 - (...)
- Upstream tests being run by multiple companies to ensure it's working properly
 - Some out-of-tree [tests](#) are being upstreamed as well



Examples

Example - livepatch /proc/cmdline interface

```
// Original function from fs/proc/cmdline.c  
  
static int cmdline_proc_show(struct seq_file *m, void *v)  
{  
    seq_puts(m, saved_command_line);  
    seq_putc(m, '\n');  
    return 0;  
}
```


Example - livepatch /proc/cmdline interface

```
// samples/livepatch/livepatch-sample.c

#include <linux/seq_file.h>

// Replacement function, note the same argument and return interface
static int livepatch_cmdline_proc_show(struct seq_file *m, void *v)
{
    seq_printf(m, "%s\n", "this has been live patched");
    return 0;
}

// ... continued ...
```

Example - livepatch /proc/cmdline interface

```
// klp_func[] array describes all livepatch original and replacement functions
static struct klp_func funcs[] = {
    {
        .old_name = "cmdline_proc_show",
        .new_func = livepatch_cmdline_proc_show,
    }, { }
};

// ... continued ...
```

Example - livepatch /proc/cmdline interface

```
// klp_object[] array describes all kernel objects (NULL for vmlinux) and functions to patch
static struct klp_object objs[] = {
    {
        .funcs = funcs,
    }, { }
};

// klp_patch contains all objects to patch
static struct klp_patch patch = {
    .mod = THIS_MODULE,
    .objs = objs,
};

// ... continued ...
```

Example - livepatch /proc/cmdline interface

```
static int livepatch_init(void)
{
    return klp_enable_patch(&patch);
}

static void livepatch_exit(void)
{
}

module_init(livepatch_init); // wire up module init/exit functions to enable the livepatch
module_exit(livepatch_exit);
MODULE_LICENSE("GPL");
MODULE_INFO(livepatch, "Y"); // don't forget to identify the module as a livepatch
```



OLFLive MENTORSHIP SERIES

Demo (<samples/livepatch/livepatch-sample.c>)

Example - livepatch /proc/cmdline interface

```
# Initial conditions
$ cat /proc/cmdline
BOOT_IMAGE=/boot/vmlinuz-6.8.7-1-default root=UUID=9a0ee7fb-04c5-49e1-8333-88f83f8fca75
splash=silent mitigations=auto quiet

# Load the livepatch
$ sudo dmesg -C
$ sudo insmod samples/livepatch/livepatch-sample.ko
```

Example - livepatch /proc/cmdline interface

```
# Dump kernel log to review transition
$ dmesg
[ 14.650630] livepatch_sample: tainting kernel with TAINT_LIVEPATCH
[ 14.650883] livepatch: enabling patch 'livepatch_sample'
[ 14.653698] livepatch: 'livepatch_sample': starting patching transition
[ 16.156028] livepatch: 'livepatch_sample': patching complete

# Test it out!
$ cat /proc/cmdline
this has been live patched
```

Example - livepatch /proc/cmdline interface

```
# Disable the livepatch
$ echo 0 > /sys/kernel/livepatch/livepatch_sample/enabled

# Test it to see initial conditions
$ cat /proc/cmdline
BOOT_IMAGE=/boot/vmlinuz-6.8.7-1-default root=UUID=9a0ee7fb-04c5-49e1-8333-88f83f8fca75
splash=silent mitigations=auto quiet

# Final cleanup
$ sudo rmmmod livepatch_sample
```


Example of an easy livepatch

[2e07e8348ea4](#) ("Bluetooth: af_bluetooth: Fix Use-After-Free in bt_sock_recvmsg")

- [CVE-2023-51779](#)
- Fixes a non-static function
- Don't contain changes to structs

Solution: the fix itself is very simple

```
--- a/net/bluetooth/af_bluetooth.c
+++ b/net/bluetooth/af_bluetooth.c
@@ -309,11 +309,14 @@ int bt_sock_recvmsg(struct socket *sock, struct msghdr *msg, size_t len,
     if (flags & MSG_OOB)
         return -EOPNOTSUPP;

+    lock_sock(sk);
+
     skb = skb_recv_datagram(sk, flags, &err);
     if (!skb) {
         if (sk->sk_shutdown & RCV_SHUTDOWN)
-            return 0;
+            err = 0;
+
+    release_sock(sk);
     return err;
 }

@@ -343,6 +346,8 @@ int bt_sock_recvmsg(struct socket *sock, struct msghdr *msg, size_t len,
     skb_free_datagram(sk, skb);

+    release_sock(sk);
+
     if (flags & MSG_TRUNC)
         copied = skblen;
```

Shadow Variable Example

[1d147bfa6429](#) ("mac80211: fix AP powersave TX vs. wakeup race")

- Adds a **spinlock_t** element to struct `sta_info`
- Changes to functions like **`ieee80211_tx_h_unicast_ps_buf()`** and friends expect to use new **`ps_lock`**
- Livepatches must handle both pre and post patched data structure instances!
 - Many **struct `sta_info`** may have already been created, without **`ps_lock`**

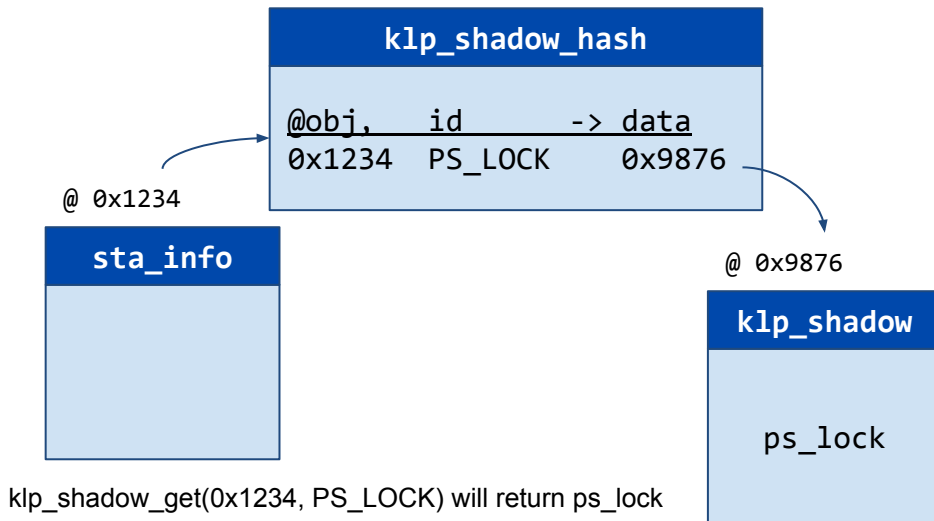
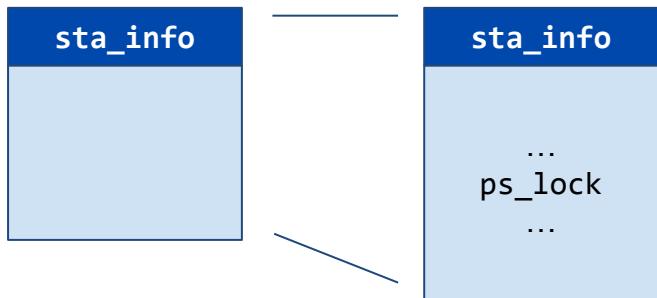
Solution: [shadow variable API](#)

```
diff --git a/net/mac80211/sta_info.h b/net/mac80211/sta_info.h
index d77ff709063038..d3a6d8208f2f85 100644
--- a/net/mac80211/sta_info.h
+++ b/net/mac80211/sta_info.h
@@ -267,6 +267,7 @@ struct ieee80211_tx_latency_stat {
 * @drv_unblock_wk: used for driver PS unblocking
 * @listen_interval: listen interval of this station, when we're acting as AP
 * @_flags: STA flags, see &enum ieee80211_sta_info_flags, do not use directly
+ * @ps_lock: used for powersave (when mac80211 is the AP) related locking
 * @ps_tx_buf: buffers (per AC) of frames to transmit to this station
 *           when it leaves power saving state or polls
 * @tx_filtered: buffers (per AC) of frames we already tried to
@@ -356,10 +357,8 @@ struct sta_info {
 /* use the accessors defined below */
 unsigned long _flags;

- /*
-  * STA powersave frame queues, no more than the internal
-  * locking required.
-  */
+ /* STA powersave lock and frame queues */
+ spinlock_t ps_lock;
 struct sk_buff_head ps_tx_buf[IEEE80211_NUM_ACS];
 struct sk_buff_head tx_filtered[IEEE80211_NUM_ACS];
 unsigned long driver_buffered_tids;
```

Shadow Variable Example

Before vs. After



A conventional patch adds a **spinlock_t** member to the structure, changing its memory layout to squeeze in **sizeof(spinlock_t)** + alignment bytes.

A livepatch conversion allocates a separate shadow variable to hold a **spinlock_t**, maintaining the original structure size and layout.

Shadow Variable Example - Matching parent's lifecycle

- Allocate and release shadow variables at the same time as its parent structure, i.e. **sta_info_alloc()**
- Suitable for data structures that are frequently created and destroyed

```
#define PS_LOCK 1
struct sta_info *sta_info_alloc(...)
{
    struct sta_info *sta;
    spinlock_t *ps_lock;

    /* Parent structure is created */
    sta = kzalloc(sizeof(*sta) + hw->sta_data_size, gfp);

    /* Attach shadow variable, then initialize it */
    ps_lock = klp_shadow_alloc(sta, PS_LOCK,
        sizeof(*ps_lock), gfp, NULL, NULL);
    if (!ps_lock)
        goto shadow_fail;
    spin_lock_init(ps_lock);
    ...
}
```

Shadow Variable Example - In-flight creation

- Allocate shadow variables as they are needed, i.e.
`ieee80211_sta_ps_deliver_wakeup()`
- Suitable for long lived data structures, or those that only need require a subset of patching

```
int ps_lock_shadow_ctor(..., void *shadow_data, ...)
{
    spin_lock_init((spinlock_t *) shadow_data);
    return 0;
}

#define PS_LOCK 1
void ieee80211_sta_ps_deliver_wakeup(struct sta_info *sta)
{
    spinlock_t *ps_lock;
    /* sync with ieee80211_tx_h_unicast_ps_buf */
    ps_lock = klp_shadow_get_or_alloc(sta, PS_LOCK,
        sizeof(*ps_lock), GFP_ATOMIC,
        ps_lock_shadow_ctor, NULL);
    /* code continues */
    if (ps_lock)
        spin_lock(ps_lock);
    ...
}
```

Complexity

How it started

Super easy!
Simple NULL-ptr check
in one function



How it's going

“Rocket-surgery” !@#^%\$!
So many functions to edit!
Lots of shadow variables!
Patch upgrades?



OLFLive MENTORSHIP SERIES





Limitations

Limitations

- The fix patches a file that has tracing disabled: [CVE-2023-46813](#)
- [b9cb9c45583b](#) ("x86/sev: Check IOBM for IOIO exceptions from user-space")
- This is usually true when patching low level architecture code

```
# File arch/x86/kernel/Makefile
```

```
ifdef CONFIG_FUNCTION_TRACER
```

```
# Do not profile debug and lowlevel utilities
```

```
...
```

```
CFLAGS_REMOVE_ftrace.o = -pg
```

```
CFLAGS_REMOVE_early_printk.o = -pg
```

```
CFLAGS_REMOVE_head64.o = -pg
```

```
CFLAGS_REMOVE_head32.o = -pg
```

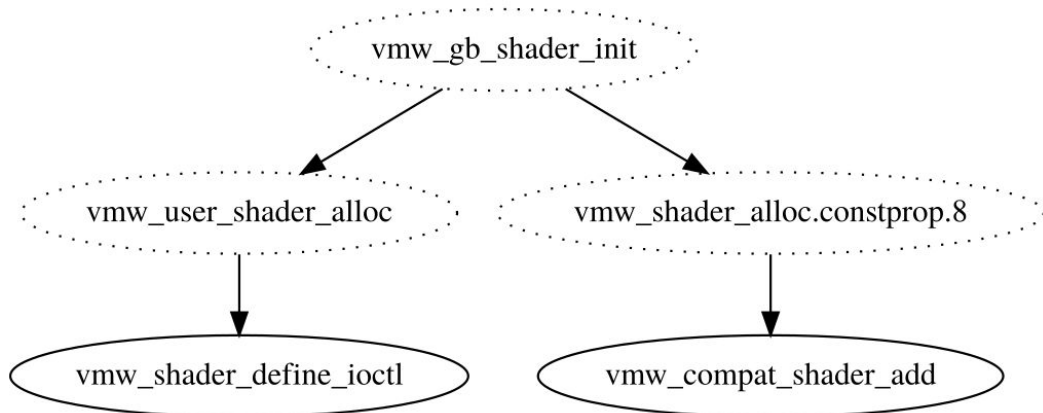
```
CFLAGS_REMOVE_sev.o = -pg
```

```
...
```

} (Omits the 5-byte ftrace hook at the beginning of functions)

Limitations

- The fix patches an inline function: [CVE-2023-5633](#)
- [91398b413d03](#) ("drm/vmwgfx: Keep a gem reference to user bos in surfaces")
- Some functions that were patched are inlined, so we need to livepatch their callers



Limitations

- There are a significant number of situations where a patch cannot be transformed into a livepatch:
 - A fix can patch a function that contains **notrace** macro, which disables tracing
 - The livepatch needs be based on the top most caller which can be patched
 - Example: [9d2231c5d74e](#) (“lib/iov_iter: initialize "flags" in new pipe_buffer”)
 - Modifies untraceable code, inlines into several exported functions
- Be careful when creating a livepatch targeting multiple architectures
 - Endianness can be a problem (same patch for x86_64 and s390x, for example)
 - A fix can patch static functions, and the compiler is free to inline these into different functions
 - Different compilers on different architectures have different inlining behaviors
 - Code can be optimized differently on different architectures
 - Symbols can be available on one architecture and missing in others
- **Hand-crafted livepatches may be dangerous! Consider using a tool to help creating them!**

Additional References

- [SUSE's livepatch tests](#)
- [Kpatch's patch author guide](#)
- [Livepatching mailing list](#)



Thank you for joining us today!

We hope it will be helpful in your journey to learning more about effective and productive participation in open source projects. We will leave you with a few additional resources for your continued learning:

- The [LF Mentoring Program](#) is designed to help new developers with necessary skills and resources to experiment, learn and contribute effectively to open source communities.
- [Outreachy remote internships program](#) supports diversity in open source and free software
- [Linux Foundation Training](#) offers a wide range of [free courses](#), webinars, tutorials and publications to help you explore the open source technology landscape.
- [Linux Foundation Events](#) also provide educational content across a range of skill levels and topics, as well as the chance to meet others in the community, to collaborate, exchange ideas, expand job opportunities and more. You can find all events at events.linuxfoundation.org.