LF *live*

MENTORSHIP SERIES

LF Live MENTORSHIP SERIES

Kernel Livepatching: Hands On

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

LF live MENTORSHIP SERIES

Agenda

- Common concerns for both approaches
- Approaches of creating livepatches
- The tools
- Hands on

Scope and Expectations

- An overview of the the available ways of creating livepatches
- How to create a livepatch for your system
- How to test your changes

LF Live MENTORSHIP SERIES

Last mentorship session - Kernel Livepatching: An Introduction

Recorded May 22, 2024

Kernel livepatching provides a means of updating a running kernel without suffering the downtime of rebooting. In this session, learn about various livepatching use cases and how the kernel implements this feature. We'll go over a brief subsystem history and how it has evolved to meet the needs of several Linux vendors.

Youtube recording

Download slides

Livepatch creation process

- Creating livepatches is a laborious process and error prone if done manually
- There are a many details that needs to be considered:
  - Symbol visibility (inlined, private, duplicate)
  - Macro expansions
  - Private struct and types
  - Non livepatchable functions or files

Livepatch creation process

- Not every upstream patch is ready to become a livepatch!
- Now think about doing all the previous steps being executed by multiple vendor supported kernels
  - Think about tens of kernel versions, each based on unique upstream kernel versions and with different patches applied, built by their own toolchain version combinations.

LF live MENTORSHIP SERIES

Livepatch creation tools

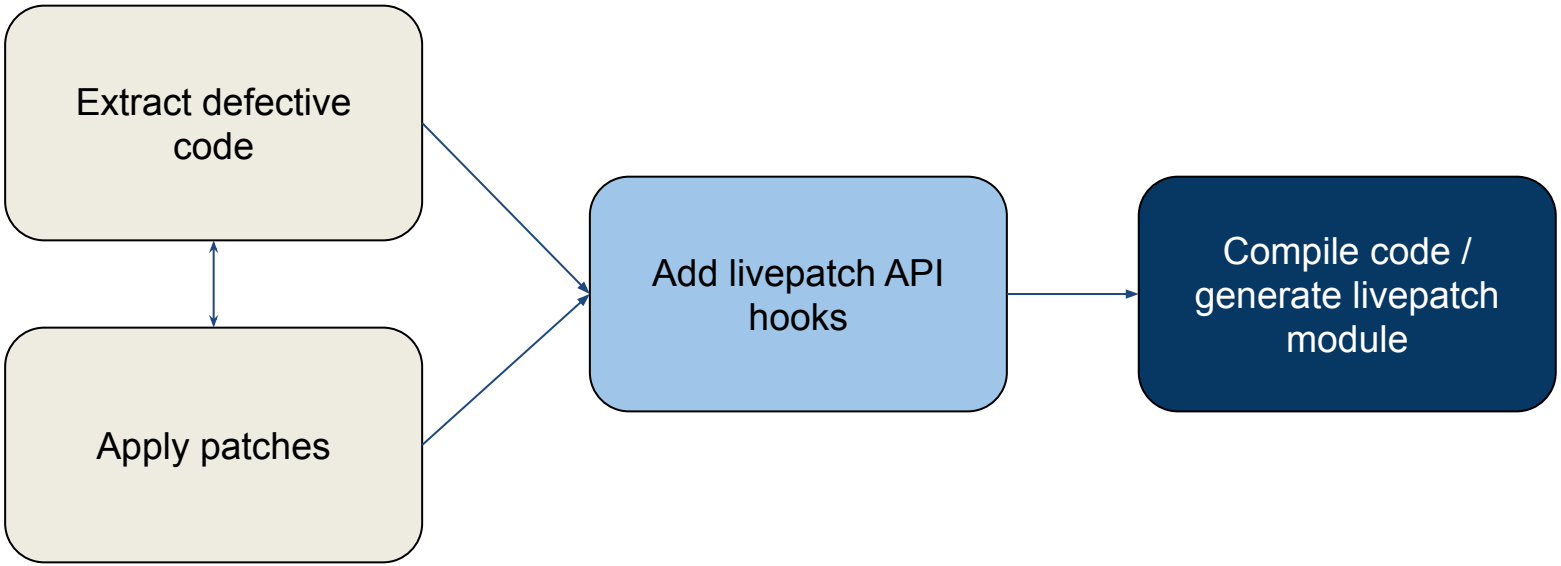
“**A man is a man, but a man with a tool makes two.**”

Ahti, the Janitor from Alan Wake 2 game

Livepatch creation tools

- There are currently two open source approaches to create livepatches
  - Source based
  - Binary based
- Both approaches have pros and cons

Source based livepatch creation



Concerns about source based livepatches

- Inlined/optimized functions
  - Copy the functions
- Private types and macros
  - Bring them to the closure
- Private symbols?
  - klp-convert (kallsyms on older kernels)
- Multi-arch livepatch generating

Source based livepatch tools

Present

klp-build

klp-ccp

Future

klp-build

clang-extract

Source based livepatch creation

klp-build

- Created to check the differences of multiple SUSE kernels when creating a livepatch
  - Being adapted to extract code of the host's running kernel
- Uses clang-extract to pull out code that requires changes

clang-extract

- Initially created to extract code for userspace livepatches and later adapted to handle kernel source code
- Uses LLVM machinery to parse the code to be extracted
- Consumes the arguments used to compile the code originally

Source based livepatch creation

klp-build + clang-extract

- Check which kernel configs are being used in the livepatch
  - Check if the symbols exists
  - Files exists
  - Configuration entries are enabled
  - Modules were compiled
  - Apply patches before calling **clang-extract**
  - Applies a templates on the code output from **clang-extract** (livepatch API entry points)

- Extracts the function to be fixed
  - Check inline functions and brings them into the final closure
  - Renames symbols if necessary
  - Check private symbols, adding klp-convert/kallsyms ancillary code as necessary
  - Generates a closure containing the necessary to be compiled as a standalone module kernel module.

klp-build - setup

- klp-build installed (or run from a cloned directory)
- clang-extract installed
  - It depends on LLVM and other more common dependencies (meson, ninja, ...)
- A kernel-source tree compiled with klp-convert (it's not merged upstream yet) and ipa-clones patches applied.
  - vmlinux/modules are used to check if the symbols to be patched are present
  - ipa-clones files are used to check which functions are inlined
  - Module.symvers file is used to check which functions are present on vmlinux and which needs to be relocated/externalized

klp-build

$ cat ~/.config/klp-build/config

[Paths]
work_dir = /home/mpdesouza/klp/livepatches
data_dir = /home/mpdesouza/git/linux

klp-build

```
$ klp-build setup --name lp_cmdline \
                  --conf CONFIG_PROC_FS \
                  --file-funcs fs/proc/cmdline.c cmdline_proc_show
```

klp-build

 # At this point clang-extract will be called to extract the function

$ klp-build **extract** --name **lp_cmdline --apply-patches**

Klp-build - sample patch

$ cat ~/git/linux/fixes/cmdline.patch

```
diff --git a/fs/proc/cmdline.c b/fs/proc/cmdline.c
index a6f76121955f..f511d0afed52 100644
--- a/fs/proc/cmdline.c
+++ b/fs/proc/cmdline.c
@@ -7,8 +7,7 @@

 static int cmdline_proc_show(struct seq_file *m, void *v)
 {
-     seq_puts(m, saved_command_line);
-     seq_putc(m, '\n');
+     seq_printf(m, "%s patched=1\n", saved_command_line);
      return 0;
 }
```

klp-build - code extracted using clang-extract

$ cat ~/klp/livepatches/lp_cmdline/ce/linux/lp/livepatch_lp_cmdline.c

```
#include <linux/fs.h>
#include <linux/init.h>
#include <linux/proc_fs.h>
#include <linux/seq_file.h>
#include <linux/proc_fs.h>
#include <linux/proc_ns.h>
#include <linux/refcount.h>
#include <linux/spinlock.h>
#include <linux/atomic.h>
#include <linux/binfmts.h>
#include <linux/sched/coredump.h>
#include <linux/sched/task.h>
#include <linux/mm.h>

/** clang-extract: from fs/proc/internal.h:31:1 */
struct proc_dir_entry;/* Full definition was removed.  */
```

klp-build - code extracted using clang-extract

```
/** clang-extract: from fs/proc/cmdline.c:8:1 */
int klpp_cmdline_proc_show(struct seq_file *m, void *v)
{
        seq_printf(m, "%s patched=1\n", saved_command_line);
        return 0;
}

#define KLP_RELOC_SYMBOL_POS(LP_OBJ_NAME, SYM_OBJ_NAME, SYM_NAME, SYM_POS) \
    asm("\".klp.sym.rela." #LP_OBJ_NAME "." #SYM_OBJ_NAME "." #SYM_NAME "." #SYM_POS "\"")
#define KLP_RELOC_SYMBOL(LP_OBJ_NAME, SYM_OBJ_NAME, SYM_NAME) \
    KLP_RELOC_SYMBOL_POS(LP_OBJ_NAME, SYM_OBJ_NAME, SYM_NAME, 0)

extern char *saved_command_line KLP_RELOC_SYMBOL(vmlinux, vmlinux, saved_command_line);
```

klp-build - code extracted using clang-extract

```
#include <linux/livepatch.h>
static struct klp_func vmlinux_funcs[] = {
        {
                .old_name = "cmdline_proc_show",
                .new_func = klpp_cmdline_proc_show,
        }, {}
};

static struct klp_object objs[] = {   {

     .funcs = vmlinux_funcs

  }, {}
};
```

klp-build - code extracted using clang-extract

```
static struct klp_patch patch = {
	.mod = THIS_MODULE,
	.objs = objs,
};

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

static void livepatch_lp_cmdline_cleanup(void)
{
}
module_init(livepatch_lp_cmdline_init);
module_exit(livepatch_lp_cmdline_cleanup);
MODULE_LICENSE("GPL");
MODULE_INFO(livepatch, "Y");
```

klp-build - final livepatch module

$ cd ~/klp/livepatches/**lp_cmdline**/ce/linux/lp/

$ make

$ ls
livepatch_lp_cmdline.c ... livepatch_lp_cmdline.mod.c **livepatch_lp_cmdline.ko**

klp-build - testing the livepatch

$ cat /proc/cmdline
virtme_hostname=virtme-ng nr_open=1048576 …

$ insmod ./**livepatch_lp_cmdline.ko**
$ cat /proc/cmdline
virtme_hostname=virtme-ng nr_open=1048576 … **patched=1**

$ echo 0 >/sys/kernel/livepatch/**livepatch_lp_cmdline**/enabled

$ cat /proc/cmdline
virtme_hostname=virtme-ng nr_open=1048576 …

Important!

- klp-build is still under heavy development
  - Expect subcommands and other arguments to change soon
  - Always check the latest version on https://github.com/SUSE/klp-build

- Same applies to clang-extract
  - Fixes are applied frequently
  - Always check the latest version on https://github.com/SUSE/clang-extract

LF live MENTORSHIP SERIES

kpatch

kpatch

- Homepage: https://github.com/dynup/kpatch/
- Multi-distro support: RHEL, Amazon Linux, OpenEuler, Anolis OS, Ubuntu, and others
- Multi-arch support: x86_64, ppc64le, s390x
- Converts a .patch file into a livepatch kernel object .ko
  - Builds a reference kernel with new options `--ffunction-sections` and `--fdata-sections`
  - Builds a patched kernel (with same options)
  - Performs a **binary** comparison of the builds, extracts new and modified parts into a new .o object file
  - Adds boilerplate code to “wire” it up and writes a kernel object .ko

kpatch utility command

- `kpatch install / uninstall` - copies livepatch .ko to `/var/lib/kpatch/<kernel-version>` and enables kpatch systemd service to load it on boot
- `kpatch build` - create a livepatch .ko from a .patch
- `kpatch load / unload` - load or unload a livepatch .ko on a running system
- `kpatch list` - list installed and loaded kpatches

kpatch-build: organized elves

GCC build option `--ffunction-sections` separates functions into their own ELF object file section:

source .c file

```
void functionA(void) { ... }
void functionB(void) { ... }
void functionC(void) { ... }
```

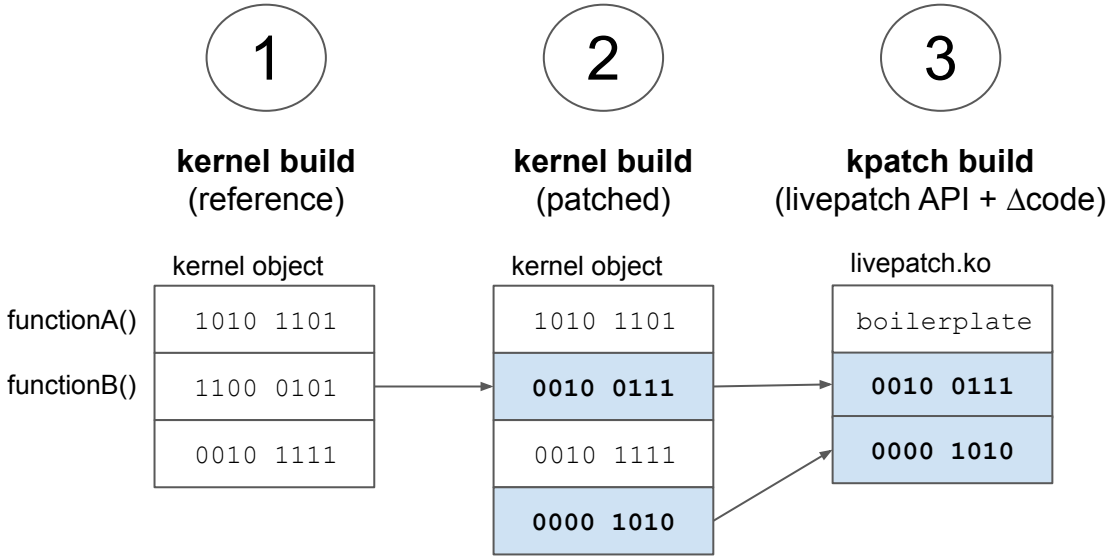
object code .o file

```
.text.functionA                [PROGBITS]
  [ ... functionA code ... ]

.text.functionB                [PROGBITS]
  [ ... functionB code ... ]

.text.functionC                [PROGBITS]
  [ ... functionC code ... ]
```

kpatch-build: “easy” as 1-2-3



kpatch-build: setup

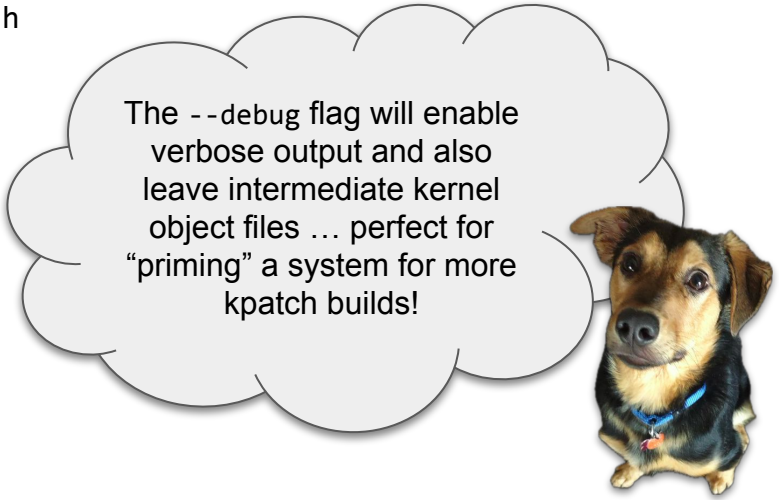
- Create a VM, container, or grab a test box
- Install and boot desired target kernel
  - Requires target kernel source
  - Kernel debuginfo packages or kernel build tree
- Install kpatch-build dependencies and build the utility:
  - ```
    make dependencies && \              # may need customization for new distros
      make -j$(nproc) && \
      make install
    ```
    ```

kpatch-build: example build invocation

- Build an example kpatch:

```
$ kpatch-build/kpatch-build --debug \
      ~/kpatch-git/examples/cmdline-string.patch
```

- Go grab a coffee and walk the dog!



kpatch-build: examples/cmdline.patch

Adds a “kpatch” string to `/proc/cmdline`, replacing `seq_put{s,c}()` calls with `seq_printf()`:

```
diff -Nupr src.orig/fs/proc/cmdline.c src/fs/proc/cmdline.c
--- src.orig/fs/proc/cmdline.c  2022-10-24 15:41:08.858760066 -0400
+++ src/fs/proc/cmdline.c       2022-10-24 15:41:11.698715352 -0400
@@ -6,8 +6,7 @@

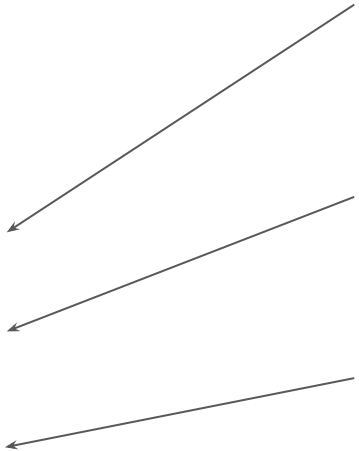
 static int cmdline_proc_show(struct seq_file *m, void *v)
 {
-       seq_puts(m, saved_command_line);
-       seq_putc(m, '\n');
+       seq_printf(m, "%s kpatch=1\n", saved_command_line);
        return 0;
 }
```

kpatch-build: example build output

```
(kpatch.git) $ ./kpatch-build/kpatch-build --skip-cleanup \
                    examples/cmdline-string.patch
Skipping cleanup
RHEL distribution detected
Downloading kernel source for 5.14.0-284.48.1.el9_2.x86_64
Unpacking kernel source
Testing patch file(s)
Reading special section data
Building original source
Building patched source
Extracting new and modified ELF sections
cmdline.o: changed function: cmdline_proc_show

Patched objects: vmlinux
Building patch module: livepatch-cmdline-string.ko

SUCCESS
```



Affected object code function(s)

Affected objects: module(s) and/or vmlinux

Final status: success or fail

kpatch-build: kpatch-build filesystem output (abridged)

```
$ tree ~/.kpatch/
├── build.log                          < kpatch-build log file
├── tmp/
│   ├── orig/                          < original kernel build object files (step 1)
│   │   └── fs/
│   │       └── proc/
│   │           └── cmdline.o
│   ├── patched/                       < patched kernel build object files (step 2)
│   │   └── fs/
│   │       └── proc/
│   │           └── cmdline.o
│   └── output/                        < extracted differences from original vs. patched (step 3)
│       └── fs/
│           └── proc/
│               └── cmdline.o
├── patch/...                          < files for building output livepatch.ko module
└── src/...                          < kernel source tree used for kernel builds
```

kpatch-build: original cmdline_proc_show()

```
0000000000000000 <cmdline_proc_show>:
   0:   e8 00 00 00 00          callq  5 <cmdline_proc_show+0x5>
                        1: R_X86_64_PLT32       __fentry__-0x4
   5:   55                      push   %rbp
   6:   48 8b 35 00 00 00 00    mov    0x0(%rip),%rsi
                        9: R_X86_64_PC32        saved_command_line-0x4
   d:   48 89 fd                mov    %rdi,%rbp
  10:   e8 00 00 00 00          callq  15 <cmdline_proc_show+0x15>
                        11: R_X86_64_PLT32      seq_puts-0x4
  15:   48 89 ef                mov    %rbp,%rdi
  18:   be 0a 00 00 00          mov    $0xa,%esi
  1d:   e8 00 00 00 00          callq  22 <cmdline_proc_show+0x22>
                        1e: R_X86_64_PLT32      seq_putc-0x4
  22:   31 c0                   xor    %eax,%eax
  24:   5d                      pop    %rbp
  25:   e9 00 00 00 00          jmpq   2a <cmdline_proc_show+0x2a>
                        26: R_X86_64_PLT32      __x86_return_thunk-0x4
```

Original object code

Note the calls to `seq_puts()` and `seq_putc()`

kpatch-build: patched cmdline_proc_show()

```
0000000000000000 <cmdline_proc_show>:
   0:   e8 00 00 00 00          callq  5 <cmdline_proc_show+0x5>
                        1: R_X86_64_PLT32       __fentry__-0x4
   5:   48 8b 15 00 00 00 00    mov    0x0(%rip),%rdx
                        8: R_X86_64_PC32        saved_command_line-0x4
   c:   48 c7 c6 00 00 00 00    mov    $0x0,%rsi
                        f: R_X86_64_32S .rodata.cmdline_proc_show.str1.1
  13:   e8 00 00 00 00          callq  18 <cmdline_proc_show+0x18>
                        14: R_X86_64_PLT32      seq_printf-0x4
  18:   31 c0                   xor    %eax,%eax
  1a:   e9 00 00 00 00          jmpq   1f <cmdline_proc_show+0x1f>
                        1b: R_X86_64_PLT32      __x86_return_thunk-0x4
```

Patched object code

Note the new call to `seq_printf()`

LF live MENTORSHIP SERIES

kpatch: testing livepatch-cmdline-string.ko

```
$ kpatch load livepatch-cmdline-string.ko
loading patch module: livepatch-cmdline-string.ko
waiting (up to 15 seconds) for patch transition to complete...
transition complete (2 seconds)
$ cat /proc/cmdline
BOOT_IMAGE=(hd0,gpt2)/vmlinuz-5.14.0-427.34.1.el9_4.x86_64 [ ... snip ...] kpatch=1

$ kpatch unload livepatch-cmdline-string.ko
disabling patch module: livepatch_cmdline_string
waiting (up to 15 seconds) for patch transition to complete...
transition complete (2 seconds)
unloading patch module: livepatch_cmdline_string
$ cat /proc/cmdline
BOOT_IMAGE=(hd0,gpt2)/vmlinuz-5.14.0-427.34.1.el9_4.x86_64 [ ... snip ...]
```

LF Live MENTORSHIP SERIES

kpatch-build: just because the kernel builds …

- Successful kernel builds do NOT imply a successful kpatch build!
  - For example, changing data and related sections like:

```
diff -Nupr src.orig/fs/proc/kmsg.c src/fs/proc/kmsg.c
---- src.orig/fs/proc/kmsg.c     2022-10-24 15:41:08.858760066 -0400
+++ src/fs/proc/kmsg.c  2022-10-24 15:41:11.698715352 -0400
@@ -55,7 +59,7 @@ static const struct proc_ops kmsg_proc_ops = {
       .proc_poll    = kmsg_poll,
       .proc_open    = kmsg_open,
       .proc_release   = kmsg_release,
-      .proc_lseek   = generic_file_llseek,
+      .proc_lseek   = kpatch_llseek,
 };

 static int __init proc_kmsg_init(void)
```

```
ERROR: kmsg.o: 1 unsupported
section change(s)
create-diff-object:
unreconcilable difference
kmsg.o: changed section
.rela.rodata.kmsg_proc_ops
not selected for inclusion
```

kpatch-build: just because the kpatch builds …

- Successful kpatch builds do NOT imply a safe livepatch!
  - For example, consider a kpatch that:
    - Modifies functions that allocate `struct foo` with a new structure definition
    - Modifies all code to use `foo`'s new definition
  - This may build a livepatch.ko, but:
    - Does not consider pre-existing instances of `foo` (before the livepatch loads)
    - Is not safe to unload, as it exposes the original kernel code to new `struct foo` layout (after the livepatch unloads)

kpatch-build: endless binary comparison …

- For better or worse, kpatch-build will try to compare *all* binary changes. Changes to header files will cause *all* its includers to rebuild and force comparison.

```
diff -Nupr src.orig/include/linux/kernel.h src/include/linux/kernel.h
--- src.orig/include/linux/kernel.h     2022-10-24 15:41:08.858760066 -0400
+++ src/include/linux/kernel.h  2022-10-24 15:41:11.698715352 -0400
@@ -25,6 +25,7 @@

 #include <uapi/linux/kernel.h>

+#define KPATCH_VALUE   12345
 #define STACK_MAGIC       0xdeadbeef

 /**
```

LF Live MENTORSHIP SERIES

kpatch Patch Author Guide

https://github.com/dynup/kpatch/blob/master/doc/patch-author-guide.md

- Patch analysis
- kpatch vs livepatch vs kGraft
- Patch upgrades
- Data structure changes
- Data semantic changes
- Init code changes
- Header file changes
- Dealing with unexpected changed functions
- Removing references to static local variables
- Code removal
- Once macros
- inline implies notrace
- Jump labels and static calls
- Sibling calls
- Exported symbol versioning
- System calls

LF live MENTORSHIP SERIES

Livepatch creation best practices

- Minimize livepatch-sets to focus on the problem at hand
- Try using cumulative livepatches (one big patch) instead of stacked (multiple) livepatches
- Not every upstream patch is reasonable to convert to a livepatch
- Carefully read the documentation in the previous slide and docs.kernel.org

LF live MENTORSHIP SERIES

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.