klp-convert and livepatch relocations

Linux Plumbers Conference 2019

Joe Lawrence
Senior Software Engineer
klp-convert patchset history

(Josh Poimboeuf) RFC:
https://lore.kernel.org/lkml/cover.1477578530.git.jpoimboe@redhat.com/

(João Moreira) v2:
https://lore.kernel.org/lkml/f52d29f7-7d1b-ad3d-050b-a9fa8878faf2@redhat.com/

(Joe Lawrence) v3:
https://lore.kernel.org/lkml/20190410155058.9437-1-joe.lawrence@redhat.com/
v4:
https://lore.kernel.org/lkml/20190509143859.9050-1-joe.lawrence@redhat.com/
v5:
(kbuild cleanup from Masahiro Yamada, review comments from Miroslav Beneš, misc bugfixes)
Creating livepatches: kpatch-build

1. Kernel build (reference)
   - vmlinux
     - functionA(): 1010 1101
     - functionB(): 1100 0101
     - functionC(): 0010 1111
     - functionD(): 0010 1111

2. Kernel build (patched)
   - vmlinux
     - functionA(): 1010 1101
     - functionB(): 0010 0111
     - functionC(): 0010 1111
     - functionD(): 0010 1111

3. livepatch build
   - livepatch.ko
     - boilerplate
       - 0010 0111
     - 0000 1010

VS...
Creating livepatches: source-based

1

module build

livepatch.ko

Makefile
my-livepatch.h
my-livepatch.c
...

Just an “ordinary” kernel module build, no external tooling, real sources.
Problem: unexported symbols

```
EXPORT_SYMBOL(num_socks)
static int hats
static void pretzel_logic()

EXPORT_SYMBOL(countdown)
static int count_it()
```

How to access unexported symbols from livepatches?
Workaround 1: unexported symbols

Use kallsyms to manually lookup symbol names, access via pointer indirection, or ...
Workaround 2: klp-convert, part a

Kernel (and module) build generates a database of objects and their symbols...

vmlinux

EXPORT_SYMBOL(num_socks)
static int hats
static void pretzel_logic()

foo.ko

EXPORT_SYMBOL(countdown)
int count_it()

Symbols.list

klp-convert-symbol-data-.0.1
*vmlinux
num_socks
hats
pretzel_logic
*foo
countdown
count_it
Workaround 2: klp-convert, part b

With the symbol database, klp-convert can resolve unique symbols.

Symbols.list

klp-convert-symbol-data-.0.1
*vmlinux
num_socks
hats
pretzel_logic
*foo
countdown
count_it

livepatch.ko

(patch to vmlinux)
if (num_socks && hats > 0)
    pretzel_logic();

(patch to foo)
if (--countdown)
    count_it();

Symbols.list + livepatch.tmp.ko = livepatch.ko
Unresolved symbols are encoded as “livepatch” relocations, placed in specially named sections as specially named symbols.
Livepatch relocations: kernel support

- Documentation/livepatch/module-elf-format.rst

- kernel/livepatch/core.c
  - klp_resolve_symbols()
  - klp_write_object_relocations()

- arch/x86/kernel/livepatch.c
  - arch_klp_init_object_loaded()
  - No klp-convert support

Arch-specific section name format: .klp.arch.objname.section_name

Kernel support for architecture-specific livepatch relocations have been added for x86 (only) .altinstructions and .parainstructions
Special section example: .smp_locks

```c
__used static notrace void foo(void)
{
    asm volatile(LOCK_PREFIX "nop");
}
```

Disassembly of section .text:

```assembly
0000000000000000 <foo>:
  0:   f0 90  lock nop  <-------------
  2:   c3  retq   |
  ...
```

Disassembly of section .smp_locks:

```assembly
0000000000000000 <.smp_locks>:
  0:   00 00  add %al,(%rax)  |
  0:   R_X86_64_PC32  .text  ----
```

Boring example, relocation is local to the module, so no klp-convert implications.
Special section example: .altinstructions

alternative("call foo1", "call foo2", X86_FEATURE_FPU)

```
.altinstructions     .rela.altinstructions
----    old = (reloc) patch spot      a
| --    new = (reloc) alt instr spot    b
| |      feature
| |      old_len, new_len, pad_len
| |
| | .text                                  .rela.text
|--->  call (reloc) foo1                 c
|  < nop pads >
|
| .altinst_replacement                   .rela.altinst_replacement
--->  call (reloc) foo2                 d
```

.rel.a.ltinstructions is module-local, but .rela.text and .rela.altinst_replacement possibly not.
Special section example: .altinstructions

load_module

apply_relocations

post_relocation

module_finalize

    apply_alternatives  << pick new or old to patch in

...

do_init_module

do_one_initcall(mod->init)

__init patch_init [kpatch-patch]

klp_register_patch

klp_init_patch

klp_for_each_object(patch, obj)

klp_init_object

klp_init_object_loaded

    klp_write_object_relocations  << resolve livepatch relocs
Special section example: .altinstructions

Ordering problem:

1. Load patch module
2. Apply alternatives to livepatch module
3. Apply per-object relocations to livepatch module when target module loads, clobbering (2)

Correct order:

1. Load patch module
2. Apply per-object relocations to livepatch module
3. Apply alternatives and paravirt patches to patch module

Delay alternatives patching until after livepatch relocations are applied.
Special section example: .altinstructions

Kpatch-build already handles this and moves sections:

```
.altinstructions  ->  .klp.arch.<obj>..altinstructions
.rela.altinstructions  ->  .rela.klp.arch.<obj>..altinstructions
```

The `.altinst_replacement` section remains intact, but kpatch-build does move its converted relocations `.klp.rela.<obj>..altinst_replacement` as per usual livepatch symbol/relocation conversion.

klp-convert:

TODO
special section example: __jump_table

```
extern struct static_key_false module_key;
__used static notrace void foo(void)
{
    if (static_branch_likely(&module_key))
        asm("nop 1");
    else
        asm("nop 2");

    asm("nop 3");
}
```
Special section example: __jump_table

Disassembly of section .text:

```
0000000000000000 <foo>:
    0:   e9 11 00 00 00     jmpq 16 <foo+0x16>
    5:   0f 1f 04 25 01 00 00    nopl  0x1
    c:   00
    d:   0f 1f 04 25 03 00 00    nopl  0x3
    14:  00
    15:  c3    retq
---->  16:   0f 1f 04 25 02 00 00    nopl  0x2
    1d:  00
    1e:   eb ed     jmp d <foo+0xd>

| | Offset Type Value Addend   Name
| --  0000000000000000 X86_64_PC32 0000000000000000 +0  .text
----  0x0000000000000000 X86_64_PC32 0000000000000000 +22 .text
  0x0000000000000008 X86_64_PC64 0000000000000000 +0 module_key
```

Static key code and target are module-local relocations

But the key may be external.
Special section example: __jump_table

- kpatch-build, klp-convert: TODO

- Once again, we will need to do some relocation / section book-keeping:
  - For any jumpy label key-value relocation that requires livepatch relocation type
    - Move it into an arch-specific section
  - Update arch_klp_init_object_loaded() to initialize this particular static key

- TBD: is this enough? Does the jump label code make assumptions about __jump_table and whether all structures can be considered “live”
  - e.g. need to resize and dynamically manage struct module’s jump_entries array?
More TODO

● How many other arch-specific sections do we need to worry about?
  ○ We will need good regression tests to aid long-term stability.

● External modules: should we support out-of-tree livepatch builds that require klp-convert?
  ○ Can out-of-tree modules provide their own Symbols.list?

● BFD library bug: bz-24456
  ○ Doesn’t like multiple relocation sections to same (.text) section
  ○ Affects objdump, gdb, crash utility
  ○ Mitigation recently checked into binutils
    ■ a7ba389645d1 ("Stop the BFD library from failing when encountering a second set of relocations for the same section.")
THANK YOU