# Livepatch module dependencies

Explore pros and cons of hard patch module dependencies

Jessica Yu
Associate Software Engineer, Red Hat
Linux Plumbers Conference
November 2, 2016

# Background

Hard dependencies?

- Requiring to-be-patched/target module be loaded before the corresponding patch module
- Livepatch currently allows enabling patches for modules before they are loaded

# Background

This idea has been mentioned on the mailing list a number of times, but we haven’t delved into it yet

Some issues (mainly code duplication) that come with patching modules before they are loaded have been discussed before [1][2][3]


[1] “Bug with paravirt ops and livepatches” Apr 2016
https://lkml.kernel.org/r/20160404161428.3qap2i4vpgda66iw@treble.redhat.com

[2] “Fix issue with alternatives/paravirt patches” Jul 2016
https://lkml.kernel.org/r/20160713011744.GB30925@packer-debian-8-amd64.digitalocean.com

[3] “RFC: removing reloc and module notify code from livepatch” Jul 2015
http://www.spinics.net/lists/live-patching/msg00946.html

# Questions to answer

- Are there any benefits of enforcing hard module dependencies for patch modules?
- Is it better to keep our current scheme (one patch module for many, possibly unloaded, objects) or should we look at alternative solutions?

# Background

Currently, livepatch can patch multiple objects (vmlinux, modules) at once, including unloaded modules.

# Background

Currently, livepatch can patch multiple objects (vmlinux, modules) at once, including unloaded modules.

Delaying operations normally performed by the module loader* enables us to do this.

*: symbol resolution, applying relocations, applying arch-specific sections such as .parainstructions and .altinstructions

# Background

Currently, livepatch can patch multiple objects (vmlinux, modules) at once, including unloaded modules.

Delaying operations normally performed by the module loader* enables us to do this.

This circumvents normal module dependencies, in that we allow loading of a patch module that depends on information/syms from unloaded module(s)

*: symbol resolution, applying relocations, applying arch-specific sections such as .parainstructions and .altinstructions

# Problem Statement

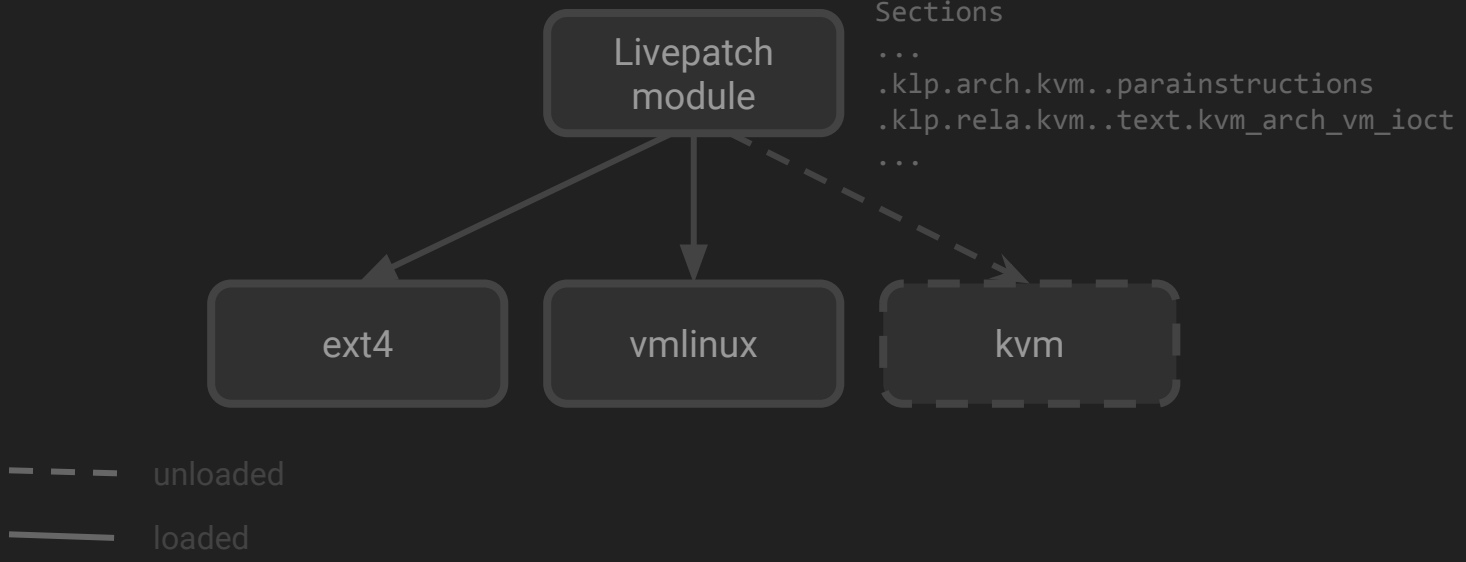
What are the costs of allowing patch modules to load before its target module(s)?

- Code duplication
  - Requiring re-implementation of parts of module loader in livepatch code
    - formerly: duplication of relocation code (now: re-use `apply_relocate_add()`)
    - duplicating portions of `module_finalize()` for application of special sections
- (formerly) usage of module notifiers to finish delayed tasks* upon target module load
- Need to keep elf info around to reuse `apply_relocate_add()`
- Is there a better solution that avoids all this? 🤔

*: symbol resolution, applying relocations, applying arch-specific sections such as .parainstructions

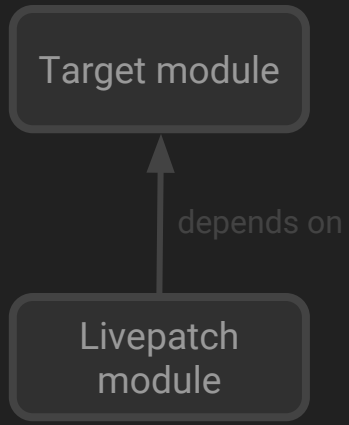
# Current Status

One patch module can patch multiple objects/components at once



# Enforcing hard module deps

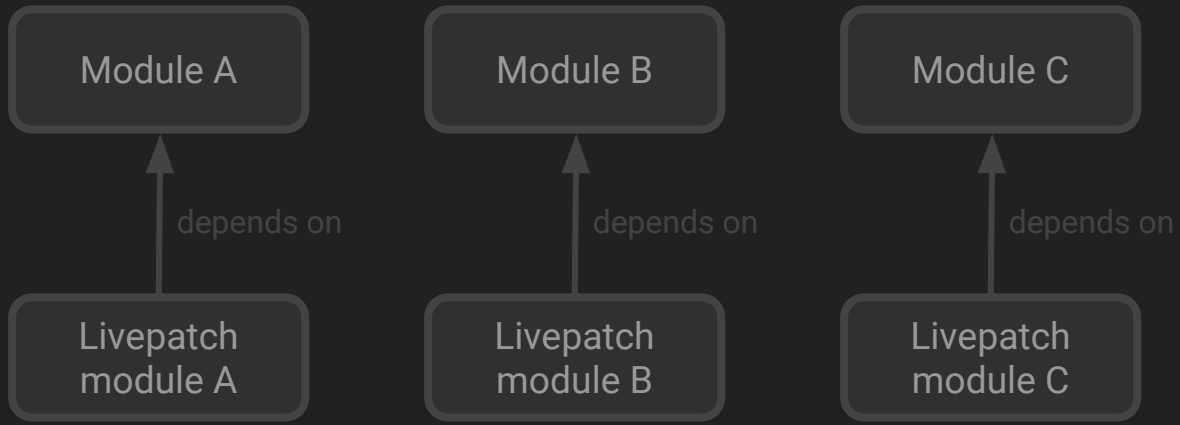
Are there any benefits to enforcing hard module dependencies?



# Enforcing hard module deps

Are there any benefits to enforcing hard module dependencies?

- If we enforce hard module deps, that means all to-be-patched module have to be loaded first
  - Not good, would mean loading a bunch of unneeded modules
- What if we split up patch modules per-object?



# Enforcing hard module deps

Are there any benefits to enforcing hard module dependencies?

- What if we split up patch modules per-object?
  - could reduce a bunch of redundant code by following normal module load order

# Enforcing hard module deps

Are there any benefits to enforcing hard module dependencies?

```
static int load_module(struct load_info *info,...) { // on patch module load...


     ...
     /*
      * can use klp_resolve_symbols() in simplify_symbols() to resolve
      * SHN_LIVEPATCH symbols since target module is already loaded.
      */
     simplify_symbols();
     /* since remaining syms have been resolved, can apply relocs normally */
     apply_relocations();
     ...
     module_finalize(); // for x86 apply_alternatives, apply_paravirt
    /* no longer need for copy_module_elf() for livepatch modules */
}
```

# Enforcing hard module deps

Are there any benefits to enforcing hard module dependencies?

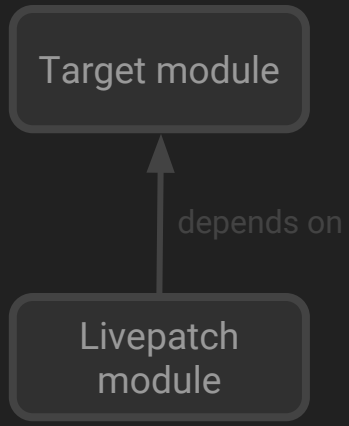
Pros:

- no need to split up rela sections per object (“.klp.rela.objname.” sections) if have 1-1 patch module to target module mapping
- no need to split up arch sections per object (“.klp.arch.objname” sections)
- no need for `arch_klp_init_object_loaded()`
  - (which applies arch-specific sections)
- no need to copy Elf info off of module loader’s load_info struct just to reuse `apply_relocate_add()`
- no need for module notifiers or hard-coded calls to `klp_module_{coming,going}()`

# Enforcing hard module deps

Are there any benefits to enforcing hard module dependencies?

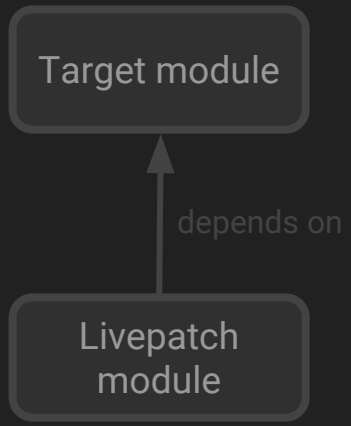
- Can we manage patch mod dependencies through modprobe/depmod?



# Enforcing hard module deps

Are there any benefits to enforcing hard module dependencies?

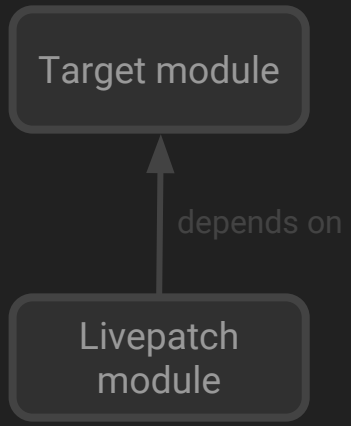
- Can we manage patch mod dependencies through modprobe?
  - Two potential cons/issues with this:
    - [1] Currently no way to manually specify hard dep
      - depmod will recognize a hard dependency if modA uses exported symbols from modB.
        - With patch module -> target module relationships we don’t necessarily meet this requirement
      - There is MODULE_DEPEND() macro in FreeBSD that allows one to manually list dependencies but we don’t have an analogous feature (softdeps come close, but not strict enough)



# Enforcing hard module deps

Are there any benefits to enforcing hard module dependencies?

- Can we manage patch mod dependencies through modprobe?
  - Two potential issues with this:
    - [2] Dependency needs to go both ways
      - It is not enough that the target module gets loaded before the patch module
      - Patch module must also be loaded after target module (post-dependency)
      - One can specify post dependencies with softdeps, but  not strict enough



# Enforcing hard module deps

Are there any benefits to enforcing hard module dependencies?

Cons:

- 1-1 scheme works very well if every patch were isolated to one object (vmlinux or module)
  - But we do not live in a perfect world…
  - If a single patch spans across multiple objects, can become a pain to safely apply patch modules and to manage them together
  - example: CVE-2016-7097
    - involves change to VFS api that needed to be propagated to many file systems
      - Patched in commit 073931017b49d9458aa351605b43a7e34598caef (“Clear SGID bit when setting file permissions”)
    - probably not suitable for 1-1 patch module to target module scheme

# Enforcing hard module deps

Are there any benefits to enforcing hard module dependencies?

Cons:

- May need to ship many modules for a single patch instead of just one patch module
- Patch management/coordination between patch modules can become cumbersome
  - If one patch/logical change is spread out to multiple patch modules, they need to be treated as a unit, and enabled/disabled together as a unit
  - Have each patch module register own `klp_patch` or just initialize new `klp_object` struct and add to an existing `klp_patch`?
    - if implementing the latter, then need to figure out who initializes the `klp_patch` for patch modules that load later

# Enforcing hard module deps

Are there any benefits to enforcing hard module dependencies?

Cons:

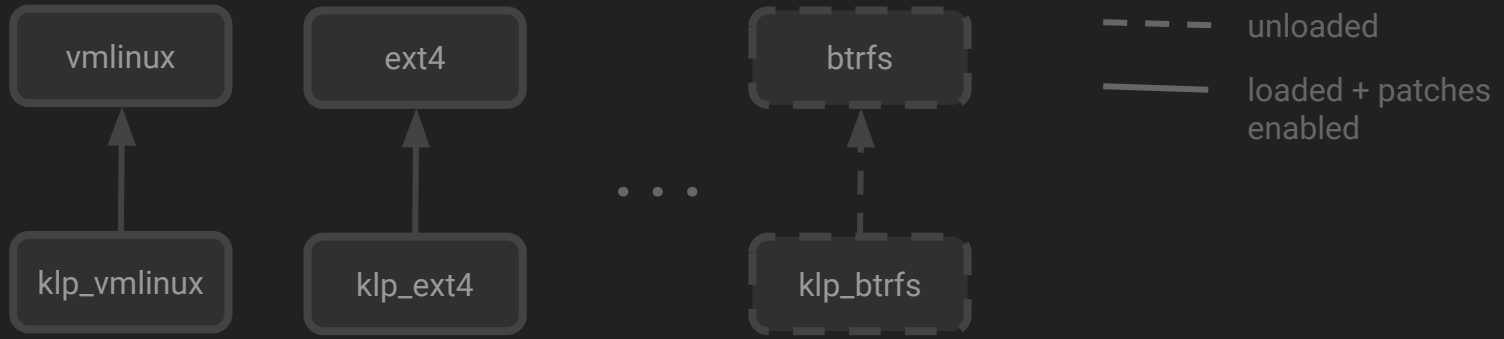
- Without `klp_module_going()`, if a target module wants to unload, patch module still needs to be notified somehow (to disable `klp_object`)
  - or just never unload target module...and layover new patch modules over it
- Without `klp_module_coming()`, might run into  unsafe situation where old and new code run together..

# Enforcing hard module deps

Are there any benefits to enforcing hard module dependencies?

Cons:

- Without `klp_module_going()`, if a target module wants to unload, patch module still needs to be notified somehow (to disable `klp_object`)
  - or just never unload target module...and layover new patch modules over it
- Without `klp_module_coming()`, might run into  unsafe situation where old and new code run together..



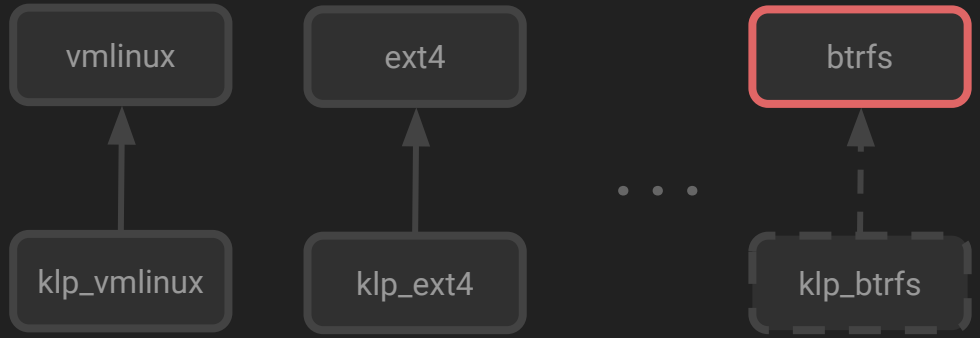

CVE 2016 7097

# Enforcing hard module deps

Are there any benefits to enforcing hard module dependencies?

Cons:

- Without `klp_module_going()`, if a target module wants to unload, patch module still needs to be notified somehow (to disable `klp_object`)
  - or just never unload target module...and layover new patch modules over it
- Without `klp_module_coming()`, might run into  unsafe situation where old and new code run together..




modprobing order (hard deps)

- `insmod btrfs.ko`
- `insmod klp_btrfs.ko`

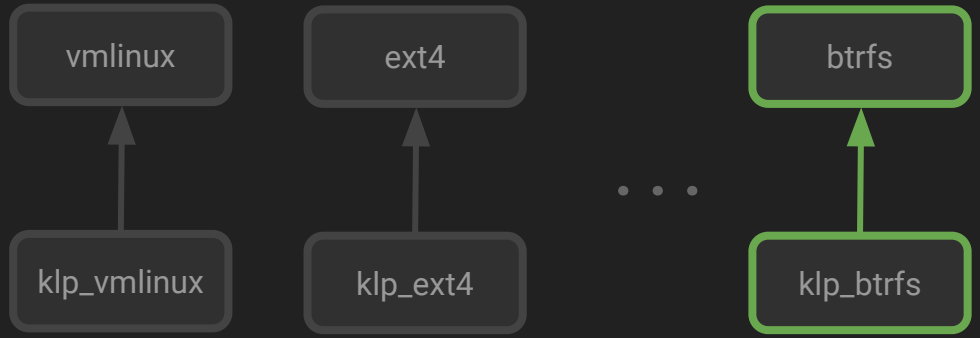
CVE 2016 7097

# Enforcing hard module deps

Are there any benefits to enforcing hard module dependencies?

Cons:

- Without `klp_module_going()`, if a target module wants to unload, patch module still needs to be notified somehow (to disable `klp_object`)
  - or just never unload target module...and layover new patch modules over it
- Without `klp_module_coming()`, might run into  unsafe situation where old and new code run together..




modprobing order (hard deps)

- `insmod btrfs.ko`
- `insmod klp_btrfs.ko`

btrfs must be loaded for klp_btrfs to load, remains unpatched until that happens

CVE 2016 7097

# Thanks!