(Un)patching Callbacks

Livepatch (un)patch-callbacks provide a mechanism for livepatch modules to execute callback functions when a kernel object is (un)patched. They can be considered a **power feature** that **extends livepatching abilities** to include:

- Safe updates to global data
- "Patches" to init and probe functions
- Patching otherwise unpatchable code (i.e. assembly)

In most cases, (un)patch callbacks will need to be used in conjunction with memory barriers and kernel synchronization primitives, like mutexes/spinlocks, or even stop_machine(), to avoid concurrency issues.

1. Motivation

Callbacks differ from existing kernel facilities:

- Module init/exit code doesn't run when disabling and re-enabling a patch.
- A module notifier can't stop a to-be-patched module from loading.

Callbacks are part of the klp_object structure and their implementation is specific to that klp_object. Other livepatch objects may or may not be patched, irrespective of the target klp_object's current state.

2. Callback types

Callbacks can be registered for the following livepatch actions:

- Pre-patch
    - before a klp_object is patched
- Post-patch
    - after a klp_object has been patched and is active across all tasks
- Pre-unpatch
    - before a klp_object is unpatched (ie, patched code is active), used to clean up post-patch callback resources
- Post-unpatch
    - after a klp_object has been patched, all code has been restored and no tasks are running patched code, used to cleanup pre-patch callback resources

3. How it works

Each callback is optional, omitting one does not preclude specifying any other. However, the livepatching core executes the handlers in symmetry: pre-patch callbacks have a post-unpatch counterpart and post-patch callbacks have a pre-unpatch counterpart. An unpatch callback will only be executed if its corresponding patch callback was executed. Typical use cases pair a patch handler that acquires and configures resources with an unpatch handler tears down and releases those same resources.

A callback is only executed if its host klp_object is loaded. For in-kernel vmlinux targets, this means that callbacks will always execute when a livepatch is enabled/disabled. For patch target kernel modules, callbacks will only execute if the target module is loaded. When a module target is (un)loaded, its callbacks will execute only if the livepatch module is enabled.

The pre-patch callback, if specified, is expected to return a status code (0 for success, -ERRNO on error). An error status code indicates to the livepatching core that patching of the current klp_object is not safe and to stop the current patching request. (When no pre-patch callback is provided, the transition is assumed to be safe.) If a pre-patch callback returns failure, the kernel's module loader will:

- Refuse to load a livepatch, if the livepatch is loaded after targeted code.

  or:

- Refuse to load a module, if the livepatch was already successfully loaded.

No post-patch, pre-unpatch, or post-unpatch callbacks will be executed for a given klp_object if the object failed to patch, due to a failed pre_patch callback or for any other reason.

If a patch transition is reversed, no pre-unpatch handlers will be run (this follows the previously mentioned symmetry -- pre-unpatch callbacks will only occur if their corresponding post-patch callback executed).

If the object did successfully patch, but the patch transition never started for some reason (e.g., if another object failed to patch), only the post-unpatch callback will be called.

4. Use cases

Sample livepatch modules demonstrating the callback API can be found in samples/livepatch/ directory. These samples were modified for use in kselftests and can be found in the lib/livepatch directory.

Global data update

A pre-patch callback can be useful to update a global variable. For example, 75ff39ccc1bd ("tcp: make challenge acks less predictable") changes a global sysctl, as well as patches the tcp_send_challenge_ack() function.

In this case, if we're being super paranoid, it might make sense to patch the data *after* patching is complete with a post-patch callback, so that tcp_send_challenge_ack() could first be changed to read sysctl_tcp_challenge_ack_limit with READ_ONCE.

__init and probe function patches support

Although __init and probe functions are not directly livepatch-able, it may be possible to implement similar updates via pre/post-patch callbacks.

The commit `48900cb6af42 ("virtio-net: drop NETIF_F_FRAGLIST")` change the way that virtnet_probe() initialized its driver's net_device features. A pre/post-patch callback could iterate over all such devices, making a similar change to their hw_features value. (Client functions of the value may need to be updated accordingly.)