Linux Livepatch Documentation

The kernel development community

Jun 10, 2024

CONTENTS

CHAPTER

ONE

LIVEPATCH

This document outlines basic information about kernel livepatching.

1.1 1. Motivation

There are many situations where users are reluctant to reboot a system. It may be because their system is performing complex scientific computations or under heavy load during peak usage. In addition to keeping systems up and running, users want to also have a stable and secure system. Livepatching gives users both by allowing for function calls to be redirected; thus, fixing critical functions without a system reboot.

1.2 2. Kprobes, Ftrace, Livepatching

There are multiple mechanisms in the Linux kernel that are directly related to redirection of code execution; namely: kernel probes, function tracing, and livepatching:

- The kernel probes are the most generic. The code can be redirected by putting a breakpoint instruction instead of any instruction.
- The function tracer calls the code from a predefined location that is close to the function entry point. This location is generated by the compiler using the '-pg' gcc option.
- Livepatching typically needs to redirect the code at the very beginning of the function entry before the function parameters or the stack are in any way modified.

All three approaches need to modify the existing code at runtime. Therefore they need to be aware of each other and not step over each other's toes. Most of these problems are solved by using the dynamic ftrace framework as a base. A Kprobe is registered as a ftrace handler when the function entry is probed, see CONFIG_KPROBES_ON_FTRACE. Also an alternative function from a live patch is called with the help of a custom ftrace handler. But there are some limitations, see below.

1.3 3. Consistency model

Functions are there for a reason. They take some input parameters, get or release locks, read, process, and even write some data in a defined way, have return values. In other words, each function has a defined semantic.

Many fixes do not change the semantic of the modified functions. For example, they add a NULL pointer or a boundary check, fix a race by adding a missing memory barrier, or add some locking around a critical section. Most of these changes are self contained and the function presents itself the same way to the rest of the system. In this case, the functions might be updated independently one by one.

But there are more complex fixes. For example, a patch might change ordering of locking in multiple functions at the same time. Or a patch might exchange meaning of some temporary structures and update all the relevant functions. In this case, the affected unit (thread, whole kernel) need to start using all new versions of the functions at the same time. Also the switch must happen only when it is safe to do so, e.g. when the affected locks are released or no data are stored in the modified structures at the moment.

The theory about how to apply functions a safe way is rather complex. The aim is to define a so-called consistency model. It attempts to define conditions when the new implementation could be used so that the system stays consistent.

Livepatch has a consistency model which is a hybrid of kGraft and kpatch: it uses kGraft's per-task consistency and syscall barrier switching combined with kpatch's stack trace switching. There are also a number of fallback options which make it quite flexible.

Patches are applied on a per-task basis, when the task is deemed safe to switch over. When a patch is enabled, livepatch enters into a transition state where tasks are converging to the patched state. Usually this transition state can complete in a few seconds. The same sequence occurs when a patch is disabled, except the tasks converge from the patched state to the unpatched state.

An interrupt handler inherits the patched state of the task it interrupts. The same is true for forked tasks: the child inherits the patched state of the parent.

Livepatch uses several complementary approaches to determine when it's safe to patch tasks:

1. The first and most effective approach is stack checking of sleeping tasks. If no affected functions are on the stack of a given task, the task is patched. In most cases this will patch most or all of the tasks on the first try. Otherwise it'll keep trying periodically. This option is only available if the architecture has reliable stacks (HAVE_RELIABLE_STACKTRACE).

2. The second approach, if needed, is kernel exit switching. A task is switched when it returns to user space from a system call, a user space IRQ, or a signal. It's useful in the following cases:

   a) Patching I/O-bound user tasks which are sleeping on an affected function. In this case you have to send SIGSTOP and SIGCONT to force it to exit the kernel and be patched.

   b) Patching CPU-bound user tasks. If the task is highly CPU-bound then it will get patched the next time it gets interrupted by an IRQ.

3. For idle "swapper" tasks, since they don't ever exit the kernel, they instead have a klp_update_patch_state() call in the idle loop which allows them to be patched before the CPU enters the idle state.

   (Note there's not yet such an approach for kthreads.)

Architectures which don't have HAVE_RELIABLE_STACKTRACE solely rely on the second approach. It's highly likely that some tasks may still be running with an old version of the function, until that function returns. In this case you would have to signal the tasks. This especially applies to kthreads. They may not be woken up and would need to be forced. See below for more information.

Unless we can come up with another way to patch kthreads, architectures without HAVE_RELIABLE_STACKTRACE are not considered fully supported by the kernel livepatching.

The /sys/kernel/livepatch/<patch>/transition file shows whether a patch is in transition. Only a single patch can be in transition at a given time. A patch can remain in transition indefinitely, if any of the tasks are stuck in the initial patch state.

A transition can be reversed and effectively canceled by writing the opposite value to the /sys/kernel/livepatch/<patch>/enabled file while the transition is in progress. Then all the tasks will attempt to converge back to the original patch state.

There's also a /proc/<pid>/patch_state file which can be used to determine which tasks are blocking completion of a patching operation. If a patch is in transition, this file shows 0 to indicate the task is unpatched and 1 to indicate it's patched. Otherwise, if no patch is in transition, it shows -1. Any tasks which are blocking the transition can be signaled with SIGSTOP and SIGCONT to force them to change their patched state. This may be harmful to the system though. Sending a fake signal to all remaining blocking tasks is a better alternative. No proper signal is actually delivered (there is no data in signal pending structures). Tasks are interrupted or woken up, and forced to change their patched state. The fake signal is automatically sent every 15 seconds.

Administrator can also affect a transition through /sys/kernel/livepatch/<patch>/force attribute. Writing 1 there clears TIF_PATCH_PENDING flag of all tasks and thus forces the tasks to the patched state. Important note! The force attribute is intended for cases when the transition gets stuck for a long time because of a blocking task. Administrator is expected to collect all necessary data (namely stack traces of such blocking tasks) and request a clearance from a

patch distributor to force the transition. Unauthorized usage may cause harm to the system. It depends on the nature of the patch, which functions are (un)patched, and which functions the blocking tasks are sleeping in (/proc/<pid>/stack may help here). Removal (rmmod) of patch modules is permanently disabled when the force feature is used. It cannot be guaranteed there is no task sleeping in such module. It implies unbounded reference count if a patch module is disabled and enabled in a loop.

Moreover, the usage of force may also affect future applications of live patches and cause even more harm to the system. Administrator should first consider to simply cancel a transition (see above). If force is used, reboot should be planned and no more live patches applied.

1.3.1 3.1 Adding consistency model support to new architectures

For adding consistency model support to new architectures, there are a few options:

1) Add CONFIG_HAVE_RELIABLE_STACKTRACE. This means porting objtool, and for non-DWARF unwinders, also making sure there's a way for the stack tracing code to detect interrupts on the stack.

2) Alternatively, ensure that every kthread has a call to klp_update_patch_state() in a safe location. Kthreads are typically in an infinite loop which does some action repeatedly. The safe location to switch the kthread's patch state would be at a designated point in the loop where there are no locks taken and all data structures are in a well-defined state.

   The location is clear when using workqueues or the kthread worker API. These kthreads process independent actions in a generic loop.

   It's much more complicated with kthreads which have a custom loop. There the safe location must be carefully selected on a case-by-case basis.

   In that case, arches without HAVE_RELIABLE_STACKTRACE would still be able to use the non-stack-checking parts of the consistency model:

   a) patching user tasks when they cross the kernel/user space boundary; and

   b) patching kthreads and idle tasks at their designated patch points.

   This option isn't as good as option 1 because it requires signaling user tasks and waking kthreads to patch them. But it could still be a good backup option for those architectures which don't have reliable stack traces yet.

1.4 4. Livepatch module

Livepatches are distributed using kernel modules, see samples/livepatch/livepatch-sample.c.

The module includes a new implementation of functions that we want to replace. In addition, it defines some structures describing the relation between the original and the new implementation. Then there is code that makes the kernel start using the new code when the livepatch module is loaded. Also there is code that cleans up before the livepatch module is removed. All this is explained in more details in the next sections.

1.4.1 4.1. New functions

New versions of functions are typically just copied from the original sources. A good practice is to add a prefix to the names so that they can be distinguished from the original ones, e.g. in a backtrace. Also they can be declared as static because they are not called directly and do not need the global visibility.

The patch contains only functions that are really modified. But they might want to access functions or data from the original source file that may only be locally accessible. This can be solved by a special relocation section in the generated livepatch module, see *Livepatch module ELF format* for more details.

1.4.2 4.2. Metadata

The patch is described by several structures that split the information into three levels:

- *struct klp_func* is defined for each patched function. It describes the relation between the original and the new implementation of a particular function.

  The structure includes the name, as a string, of the original function. The function address is found via kallsyms at runtime.

  Then it includes the address of the new function. It is defined directly by assigning the function pointer. Note that the new function is typically defined in the same source file.

  As an optional parameter, the symbol position in the kallsyms database can be used to disambiguate functions of the same name. This is not the absolute position in the database, but rather the order it has been found only for a particular object (vmlinux or a kernel module). Note that kallsyms allows for searching symbols according to the object name.

- *struct klp_object* defines an array of patched functions (*struct klp_func*) in the same object. Where the object is either vmlinux (NULL) or a module name.

  The structure helps to group and handle functions for each object together. Note that patched modules might be loaded later than the patch itself and the relevant functions might be patched only when they are available.

- *struct klp_patch* defines an array of patched objects (*struct klp_object*).

  This structure handles all patched functions consistently and eventually, synchronously. The whole patch is applied only when all patched symbols are found. The only exception are symbols from objects (kernel modules) that have not been loaded yet.

  For more details on how the patch is applied on a per-task basis, see the “Consistency model” section.

1.5 5. Livepatch life-cycle

Livepatching can be described by five basic operations: loading, enabling, replacing, disabling, removing.

Where the replacing and the disabling operations are mutually exclusive. They have the same result for the given patch but not for the system.

1.5.1 5.1. Loading

The only reasonable way is to enable the patch when the livepatch kernel module is being loaded. For this, *klp_enable_patch()* has to be called in the module_init() callback. There are two main reasons:

First, only the module has an easy access to the related *struct klp_patch*.

Second, the error code might be used to refuse loading the module when the patch cannot get enabled.

1.5.2 5.2. Enabling

The livepatch gets enabled by calling *klp_enable_patch()* from the module_init() callback. The system will start using the new implementation of the patched functions at this stage.

First, the addresses of the patched functions are found according to their names. The special relocations, mentioned in the section “New functions”, are applied. The relevant entries are created under /sys/kernel/livepatch/<name>. The patch is rejected when any above operation fails.

Second, livepatch enters into a transition state where tasks are converging to the patched state. If an original function is patched for the first time, a function specific struct klp_ops is created and an universal ftrace handler is registered[1]. This stage is indicated by a value of ‘1’ in /sys/kernel/livepatch/<name>/transition. For more information about this process, see the “Consistency model” section.

Finally, once all tasks have been patched, the ‘transition’ value changes to ‘0’.

1.5.3 5.3. Replacing

All enabled patches might get replaced by a cumulative patch that has the .replace flag set.

Once the new patch is enabled and the ‘transition’ finishes then all the functions (*struct klp_func*) associated with the replaced patches are removed from the corresponding struct klp_ops. Also the ftrace handler is unregistered and the struct klp_ops is freed when the related function is not modified by the new patch and func_stack list becomes empty.

See *Atomic Replace & Cumulative Patches* for more details.

[1] Note that functions might be patched multiple times. The ftrace handler is registered only once for a given function. Further patches just add an entry to the list (see field *func_stack*) of the struct klp_ops. The right implementation is selected by the ftrace handler, see the “Consistency model” section.

That said, it is highly recommended to use cumulative livepatches because they help keeping the consistency of all changes. In this case, functions might be patched two times only during the transition period.

1.5.4 5.4. Disabling

Enabled patches might get disabled by writing '0' to /sys/kernel/livepatch/<name>/enabled.

First, livepatch enters into a transition state where tasks are converging to the unpatched state. The system starts using either the code from the previously enabled patch or even the original one. This stage is indicated by a value of '1' in /sys/kernel/livepatch/<name>/transition. For more information about this process, see the "Consistency model" section.

Second, once all tasks have been unpatched, the 'transition' value changes to '0'. All the functions (*struct klp_func*) associated with the to-be-disabled patch are removed from the corresponding struct klp_ops. The ftrace handler is unregistered and the struct klp_ops is freed when the func_stack list becomes empty.

Third, the sysfs interface is destroyed.

1.5.5 5.5. Removing

Module removal is only safe when there are no users of functions provided by the module. This is the reason why the force feature permanently disables the removal. Only when the system is successfully transitioned to a new patch state (patched/unpatched) without being forced it is guaranteed that no task sleeps or runs in the old code.

1.6 6. Sysfs

Information about the registered patches can be found under /sys/kernel/livepatch. The patches could be enabled and disabled by writing there.

/sys/kernel/livepatch/<patch>/force attributes allow administrator to affect a patching operation.

See Documentation/ABI/testing/sysfs-kernel-livepatch for more details.

1.7 7. Limitations

The current Livepatch implementation has several limitations:

- Only functions that can be traced could be patched.

  Livepatch is based on the dynamic ftrace. In particular, functions implementing ftrace or the livepatch ftrace handler could not be patched. Otherwise, the code would end up in an infinite loop. A potential mistake is prevented by marking the problematic functions by "notrace".

- Livepatch works reliably only when the dynamic ftrace is located at the very beginning of the function.

  The function need to be redirected before the stack or the function parameters are modified in any way. For example, livepatch requires using -fentry gcc compiler option on x86_64.

  One exception is the PPC port. It uses relative addressing and TOC. Each function has to handle TOC and save LR before it could call the ftrace handler. This operation has to be

reverted on return. Fortunately, the generic ftrace code has the same problem and all this is handled on the ftrace level.

- Kretprobes using the ftrace framework conflict with the patched functions.

  Both kretprobes and livepatches use a ftrace handler that modifies the return address. The first user wins. Either the probe or the patch is rejected when the handler is already in use by the other.

- Kprobes in the original function are ignored when the code is redirected to the new implementation.

  There is a work in progress to add warnings about this situation.

CHAPTER

TWO

(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.

2.1 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.

2.2 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

2.3 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.

2.4 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.

2.4.1 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.

2.4.2 __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.)

CHAPTER

THREE

ATOMIC REPLACE & CUMULATIVE PATCHES

There might be dependencies between livepatches. If multiple patches need to do different changes to the same function(s) then we need to define an order in which the patches will be installed. And function implementations from any newer livepatch must be done on top of the older ones.

This might become a maintenance nightmare. Especially when more patches modified the same function in different ways.

An elegant solution comes with the feature called "Atomic Replace". It allows creation of so called "Cumulative Patches". They include all wanted changes from all older livepatches and completely replace them in one transition.

3.1 Usage

The atomic replace can be enabled by setting "replace" flag in *struct klp_patch*, for example:

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

All processes are then migrated to use the code only from the new patch. Once the transition is finished, all older patches are automatically disabled.

Ftrace handlers are transparently removed from functions that are no longer modified by the new cumulative patch.

As a result, the livepatch authors might maintain sources only for one cumulative patch. It helps to keep the patch consistent while adding or removing various fixes or features.

Users could keep only the last patch installed on the system after the transition to has finished. It helps to clearly see what code is actually in use. Also the livepatch might then be seen as a "normal" module that modifies the kernel behavior. The only difference is that it can be updated at runtime without breaking its functionality.

3.2 Features

The atomic replace allows:

- Atomically revert some functions in a previous patch while upgrading other functions.
- Remove eventual performance impact caused by core redirection for functions that are no longer patched.
- Decrease user confusion about dependencies between livepatches.

3.3 Limitations:

- Once the operation finishes, there is no straightforward way to reverse it and restore the replaced patches atomically.

  A good practice is to set .replace flag in any released livepatch. Then re-adding an older livepatch is equivalent to downgrading to that patch. This is safe as long as the livepatches do _not_ do extra modifications in (un)patching callbacks or in the module_init() or module_exit() functions, see below.

  Also note that the replaced patch can be removed and loaded again only when the transition was not forced.

- Only the (un)patching callbacks from the _new_ cumulative livepatch are executed. Any callbacks from the replaced patches are ignored.

  In other words, the cumulative patch is responsible for doing any actions that are necessary to properly replace any older patch.

  As a result, it might be dangerous to replace newer cumulative patches by older ones. The old livepatches might not provide the necessary callbacks.

  This might be seen as a limitation in some scenarios. But it makes life easier in many others. Only the new cumulative livepatch knows what fixes/features are added/removed and what special actions are necessary for a smooth transition.

  In any case, it would be a nightmare to think about the order of the various callbacks and their interactions if the callbacks from all enabled patches were called.

- There is no special handling of shadow variables. Livepatch authors must create their own rules how to pass them from one cumulative patch to the other. Especially that they should not blindly remove them in module_exit() functions.

  A good practice might be to remove shadow variables in the post-unpatch callback. It is called only when the livepatch is properly disabled.

CHAPTER

FOUR

LIVEPATCH MODULE ELF FORMAT

This document outlines the ELF format requirements that livepatch modules must follow.

4.1 1. Background and motivation

Formerly, livepatch required separate architecture-specific code to write relocations. However, arch-specific code to write relocations already exists in the module loader, so this former approach produced redundant code. So, instead of duplicating code and re-implementing what the module loader can already do, livepatch leverages existing code in the module loader to perform the all the arch-specific relocation work. Specifically, livepatch reuses the apply_relocate_add() function in the module loader to write relocations. The patch module ELF format described in this document enables livepatch to be able to do this. The hope is that this will make livepatch more easily portable to other architectures and reduce the amount of arch-specific code required to port livepatch to a particular architecture.

Since apply_relocate_add() requires access to a module's section header table, symbol table, and relocation section indices, ELF information is preserved for livepatch modules (see section 5). Livepatch manages its own relocation sections and symbols, which are described in this document. The ELF constants used to mark livepatch symbols and relocation sections were selected from OS-specific ranges according to the definitions from glibc.

4.1.1 Why does livepatch need to write its own relocations?

A typical livepatch module contains patched versions of functions that can reference non-exported global symbols and non-included local symbols. Relocations referencing these types of symbols cannot be left in as-is since the kernel module loader cannot resolve them and will therefore reject the livepatch module. Furthermore, we cannot apply relocations that affect modules not yet loaded at patch module load time (e.g. a patch to a driver that is not loaded). Formerly, livepatch solved this problem by embedding special "dynrela" (dynamic rela) sections in the resulting patch module ELF output. Using these dynrela sections, livepatch could resolve symbols while taking into account its scope and what module the symbol belongs to, and then manually apply the dynamic relocations. However this approach required livepatch to supply arch-specific code in order to write these relocations. In the new format, livepatch manages its own SHT_RELA relocation sections in place of dynrela sections, and the symbols that the relas reference are special livepatch symbols (see section 2 and 3). The arch-specific livepatch relocation code is replaced by a call to apply_relocate_add().

4.2 2. Livepatch modinfo field

Livepatch modules are required to have the "livepatch" modinfo attribute. See the sample livepatch module in samples/livepatch/ for how this is done.

Livepatch modules can be identified by users by using the 'modinfo' command and looking for the presence of the "livepatch" field. This field is also used by the kernel module loader to identify livepatch modules.

4.2.1 Example:

Modinfo output:

```
% modinfo livepatch-meminfo.ko
filename:               livepatch-meminfo.ko
livepatch:              Y
license:                GPL
depends:
vermagic:               4.3.0+ SMP mod_unload
```

4.3 3. Livepatch relocation sections

A livepatch module manages its own ELF relocation sections to apply relocations to modules as well as to the kernel (vmlinux) at the appropriate time. For example, if a patch module patches a driver that is not currently loaded, livepatch will apply the corresponding livepatch relocation section(s) to the driver once it loads.

Each "object" (e.g. vmlinux, or a module) within a patch module may have multiple livepatch relocation sections associated with it (e.g. patches to multiple functions within the same object). There is a 1-1 correspondence between a livepatch relocation section and the target section (usually the text section of a function) to which the relocation(s) apply. It is also possible for a livepatch module to have no livepatch relocation sections, as in the case of the sample livepatch module (see samples/livepatch).

Since ELF information is preserved for livepatch modules (see Section 5), a livepatch relocation section can be applied simply by passing in the appropriate section index to apply_relocate_add(), which then uses it to access the relocation section and apply the relocations.

Every symbol referenced by a rela in a livepatch relocation section is a livepatch symbol. These must be resolved before livepatch can call apply_relocate_add(). See Section 3 for more information.

4.4 3.1 Livepatch relocation section format

Livepatch relocation sections must be marked with the SHF_RELA_LIVEPATCH section flag. See include/uapi/linux/elf.h for the definition. The module loader recognizes this flag and will avoid applying those relocation sections at patch module load time. These sections must also be marked with SHF_ALLOC, so that the module loader doesn't discard them on module load (i.e. they will be copied into memory along with the other SHF_ALLOC sections).

The name of a livepatch relocation section must conform to the following format:

```
.klp.rela.objname.section_name
^        ^^     ^ ^          ^
|________||_____| |__________|
   [A]      [B]        [C]
```

[A]
The relocation section name is prefixed with the string ".klp.rela."

[B]
The name of the object (i.e. "vmlinux" or name of module) to which the relocation section belongs follows immediately after the prefix.

[C]
The actual name of the section to which this relocation section applies.

4.4.1 Examples:

Livepatch relocation section names:

```
.klp.rela.ext4.text.ext4_attr_store
.klp.rela.vmlinux.text.cmdline_proc_show
```

`readelf --sections` output for a patch module that patches vmlinux and modules 9p, btrfs, ext4:

```
Section Headers:
[Nr] Name                              Type                    Address           ␣
 ↪Off    Size   ES Flg Lk Inf Al
[ snip ]
[29] .klp.rela.9p.text.caches.show RELA                    0000000000000000␣
 ↪002d58 0000c0 18 AIo 64   9  8
[30] .klp.rela.btrfs.text.btrfs.feature.attr.show RELA     0000000000000000␣
 ↪002e18 000060 18 AIo 64  11  8
[ snip ]
```

```
[34] .klp.rela.ext4.text.ext4.attr.store RELA              0000000000000000␣
↪002fd8 0000d8 18 AIo 64  13  8
[35] .klp.rela.ext4.text.ext4.attr.show RELA               0000000000000000␣
↪0030b0 000150 18 AIo 64  15  8
[36] .klp.rela.vmlinux.text.cmdline.proc.show RELA         0000000000000000␣
↪003200 000018 18 AIo 64  17  8
[37] .klp.rela.vmlinux.text.meminfo.proc.show RELA         0000000000000000␣
↪003218 0000f0 18 AIo 64  19  8
[ snip ]                                    ^                                 ␣
↪              ^
                                            |                                 ␣
↪              |
                                           [*]                                ␣
↪             [*]
```

[*]
Livepatch relocation sections are SHT_RELA sections but with a few special characteristics. Notice that they are marked SHF_ALLOC ("A") so that they will not be discarded when the module is loaded into memory, as well as with the SHF_RELA_LIVEPATCH flag ("o" - for OS-specific).

`readelf --relocs` output for a patch module:

```
Relocation section '.klp.rela.btrfs.text.btrfs_feature_attr_show' at offset␣
↪0x2ba0 contains 4 entries:
    Offset             Info             Type               Symbol's Value ␣
↪Symbol's Name + Addend
000000000000001f  0000005e00000002 R_X86_64_PC32          0000000000000000 .
↪klp.sym.vmlinux.printk,0 - 4
0000000000000028  0000003d0000000b R_X86_64_32S           0000000000000000 .
↪klp.sym.btrfs.btrfs_ktype,0 + 0
0000000000000036  0000003b00000002 R_X86_64_PC32          0000000000000000 .
↪klp.sym.btrfs.can_modify_feature.isra.3,0 - 4
000000000000004c  0000004900000002 R_X86_64_PC32          0000000000000000 .
↪klp.sym.vmlinux.snprintf,0 - 4
[ snip ]                                                                    ^
                                                                            |
                                                                           [*]
```

[*]
Every symbol referenced by a relocation is a livepatch symbol.

4.5 4. Livepatch symbols

Livepatch symbols are symbols referred to by livepatch relocation sections. These are symbols accessed from new versions of functions for patched objects, whose addresses cannot be resolved by the module loader (because they are local or unexported global syms). Since the module loader only resolves exported syms, and not every symbol referenced by the new patched functions is exported, livepatch symbols were introduced. They are used also in cases where we cannot immediately know the address of a symbol when a patch module loads. For example, this is the case when livepatch patches a module that is not loaded yet. In this case, the relevant livepatch symbols are resolved simply when the target module loads. In any case, for any livepatch relocation section, all livepatch symbols referenced by that section must be resolved before livepatch can call apply_relocate_add() for that reloc section.

Livepatch symbols must be marked with SHN_LIVEPATCH so that the module loader can identify and ignore them. Livepatch modules keep these symbols in their symbol tables, and the symbol table is made accessible through module->symtab.

4.6 4.1 A livepatch module's symbol table

Normally, a stripped down copy of a module's symbol table (containing only "core" symbols) is made available through module->symtab (See layout_symtab() in kernel/module/kallsyms.c). For livepatch modules, the symbol table copied into memory on module load must be exactly the same as the symbol table produced when the patch module was compiled. This is because the relocations in each livepatch relocation section refer to their respective symbols with their symbol indices, and the original symbol indices (and thus the symtab ordering) must be preserved in order for apply_relocate_add() to find the right symbol.

For example, take this particular rela from a livepatch module::

```
Relocation section '.klp.rela.btrfs.text.btrfs_feature_attr_show' at offset␣
↪0x2ba0 contains 4 entries:
    Offset             Info             Type               Symbol's Value ␣
↪Symbol's Name + Addend
000000000000001f  0000005e00000002 R_X86_64_PC32          0000000000000000 .
↪klp.sym.vmlinux.printk,0 - 4

This rela refers to the symbol '.klp.sym.vmlinux.printk,0', and the symbol␣
↪index is encoded
in 'Info'. Here its symbol index is 0x5e, which is 94 in decimal, which refers␣
↪to the
symbol index 94.
And in this patch module's corresponding symbol table, symbol index 94 refers␣
↪to that very symbol:
[ snip ]
94: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT OS [0xff20] .klp.sym.vmlinux.
↪printk,0
[ snip ]
```

4.7 4.2 Livepatch symbol format

Livepatch symbols must have their section index marked as SHN_LIVEPATCH, so that the module loader can identify them and not attempt to resolve them. See include/uapi/linux/elf.h for the actual definitions.

Livepatch symbol names must conform to the following format:

```
.klp.sym.objname.symbol_name,sympos
^       ^^     ^ ^          ^ ^
|_______||_____| |__________| |
   [A]     [B]       [C]    [D]
```

[A]
The symbol name is prefixed with the string ".klp.sym."

[B]
The name of the object (i.e. "vmlinux" or name of module) to which the symbol belongs follows immediately after the prefix.

[C]
The actual name of the symbol.

[D]
The position of the symbol in the object (as according to kallsyms) This is used to differentiate duplicate symbols within the same object. The symbol position is expressed numerically (0, 1, 2...). The symbol position of a unique symbol is 0.

4.7.1 Examples:

Livepatch symbol names:

```
.klp.sym.vmlinux.snprintf,0
.klp.sym.vmlinux.printk,0
.klp.sym.btrfs.btrfs_ktype,0
```

`readelf --symbols` output for a patch module:

```
Symbol table '.symtab' contains 127 entries:
   Num:    Value          Size Type    Bind   Vis      Ndx          Name
   [ snip ]
    73: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT OS [0xff20] .klp.sym.
↪vmlinux.snprintf,0
    74: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT OS [0xff20] .klp.sym.
↪vmlinux.capable,0
    75: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT OS [0xff20] .klp.sym.
↪vmlinux.find_next_bit,0
    76: 0000000000000000     0 NOTYPE  GLOBAL DEFAULT OS [0xff20] .klp.sym.
↪vmlinux.si_swapinfo,0
  [ snip ]                                                ^
                                                          |
                                                         [*]
```

[*]
Note that the ‘Ndx’ (Section index) for these symbols is SHN_LIVEPATCH (0xff20). “OS” means OS-specific.

4.8 5. Symbol table and ELF section access

A livepatch module’s symbol table is accessible through module->symtab.

Since apply_relocate_add() requires access to a module’s section headers, symbol table, and relocation section indices, ELF information is preserved for livepatch modules and is made accessible by the module loader through module->klp_info, which is a `klp_modinfo` struct. When a livepatch module loads, this struct is filled in by the module loader.

CHAPTER

FIVE

SHADOW VARIABLES

Shadow variables are a simple way for livepatch modules to associate additional “shadow” data with existing data structures. Shadow data is allocated separately from parent data structures, which are left unmodified. The shadow variable API described in this document is used to allocate/add and remove/free shadow variables to/from their parents.

The implementation introduces a global, in-kernel hashtable that associates pointers to parent objects and a numeric identifier of the shadow data. The numeric identifier is a simple enumeration that may be used to describe shadow variable version, class or type, etc. More specifically, the parent pointer serves as the hashtable key while the numeric id subsequently filters hashtable queries. Multiple shadow variables may attach to the same parent object, but their numeric identifier distinguishes between them.

5.1 1. Brief API summary

(See the full API usage docbook notes in livepatch/shadow.c.)

A hashtable references all shadow variables. These references are stored and retrieved through a <obj, id> pair.

- The klp_shadow variable data structure encapsulates both tracking meta-data and shadow-data:
  - meta-data
    - obj - pointer to parent object
    - id - data identifier
  - data[] - storage for shadow data

It is important to note that the *klp_shadow_alloc()* and *klp_shadow_get_or_alloc()* are zeroing the variable by default. They also allow to call a custom constructor function when a non-zero value is needed. Callers should provide whatever mutual exclusion is required.

Note that the constructor is called under klp_shadow_lock spinlock. It allows to do actions that can be done only once when a new variable is allocated.

- *klp_shadow_get()* - retrieve a shadow variable data pointer - search hashtable for <obj, id> pair
- *klp_shadow_alloc()* - allocate and add a new shadow variable - search hashtable for <obj, id> pair
  - if exists

        * WARN and return NULL
    - if <obj, id> doesn't already exist
        * allocate a new shadow variable
        * initialize the variable using a custom constructor and data when provided
        * add <obj, id> to the global hashtable
- `klp_shadow_get_or_alloc()` - get existing or alloc a new shadow variable - search hashtable for <obj, id> pair
    - if exists
        * return existing shadow variable
    - if <obj, id> doesn't already exist
        * allocate a new shadow variable
        * initialize the variable using a custom constructor and data when provided
        * add <obj, id> pair to the global hashtable
- `klp_shadow_free()` - detach and free a <obj, id> shadow variable - find and remove a <obj, id> reference from global hashtable
    - if found
        * call destructor function if defined
        * free shadow variable
- `klp_shadow_free_all()` - detach and free all <_, id> shadow variables - find and remove any <_, id> references from global hashtable
    - if found
        * call destructor function if defined
        * free shadow variable

5.2 2. Use cases

(See the example shadow variable livepatch modules in samples/livepatch/ for full working demonstrations.)

For the following use-case examples, consider commit 1d147bfa6429 ("mac80211: fix AP powersave TX vs. wakeup race"), which added a spinlock to net/mac80211/sta_info.h :: struct sta_info. Each use-case example can be considered a stand-alone livepatch implementation of this fix.

5.2.1 Matching parent's lifecycle

If parent data structures are frequently created and destroyed, it may be easiest to align their shadow variables lifetimes to the same allocation and release functions. In this case, the parent data structure is typically allocated, initialized, then registered in some manner. Shadow variable allocation and setup can then be considered part of the parent's initialization and should be completed before the parent "goes live" (ie, any shadow variable get-API requests are made for this <obj, id> pair.)

For commit 1d147bfa6429, when a parent sta_info structure is allocated, allocate a shadow copy of the ps_lock pointer, then initialize it:

```
#define PS_LOCK 1
struct sta_info *sta_info_alloc(struct ieee80211_sub_if_data *sdata,
                                const u8 *addr, gfp_t gfp)
{
      struct sta_info *sta;
      spinlock_t *ps_lock;

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

      /* Attach a corresponding 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);
      ...
```

When requiring a ps_lock, query the shadow variable API to retrieve one for a specific struct sta_info::

```
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(sta, PS_LOCK);
      if (ps_lock)
              spin_lock(ps_lock);
      ...
```

When the parent sta_info structure is freed, first free the shadow variable:

```
void sta_info_free(struct ieee80211_local *local, struct sta_info *sta)
{
      klp_shadow_free(sta, PS_LOCK, NULL);
      kfree(sta);
      ...
```

5.2.2 In-flight parent objects

Sometimes it may not be convenient or possible to allocate shadow variables alongside their parent objects. Or a livepatch fix may require shadow variables for only a subset of parent object instances. In these cases, the *klp_shadow_get_or_alloc()* call can be used to attach shadow variables to parents already in-flight.

For commit 1d147bfa6429, a good spot to allocate a shadow spinlock is inside ieee80211_sta_ps_deliver_wakeup():

```
int ps_lock_shadow_ctor(void *obj, void *shadow_data, void *ctor_data)
{
      spinlock_t *lock = shadow_data;

      spin_lock_init(lock);
      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);

      if (ps_lock)
              spin_lock(ps_lock);
      ...
```

This usage will create a shadow variable, only if needed, otherwise it will use one that was already created for this <obj, id> pair.

Like the previous use-case, the shadow spinlock needs to be cleaned up. A shadow variable can be freed just before its parent object is freed, or even when the shadow variable itself is no longer required.

5.2.3 Other use-cases

Shadow variables can also be used as a flag indicating that a data structure was allocated by new, livepatched code. In this case, it doesn't matter what data value the shadow variable holds, its existence suggests how to handle the parent object.

5.3 3. References

- https://github.com/dynup/kpatch

  The livepatch implementation is based on the kpatch version of shadow variables.

- http://files.mkgnu.net/files/dynamos/doc/papers/dynamos_eurosys_07.pdf

  Dynamic and Adaptive Updates of Non-Quiescent Subsystems in Commodity Operating System Kernels (Kritis Makris, Kyung Dong Ryu 2007) presented a datatype update technique called "shadow data structures".

CHAPTER

SIX

SYSTEM STATE CHANGES

Some users are really reluctant to reboot a system. This brings the need to provide more livepatches and maintain some compatibility between them.

Maintaining more livepatches is much easier with cumulative livepatches. Each new livepatch completely replaces any older one. It can keep, add, and even remove fixes. And it is typically safe to replace any version of the livepatch with any other one thanks to the atomic replace feature.

The problems might come with shadow variables and callbacks. They might change the system behavior or state so that it is no longer safe to go back and use an older livepatch or the original kernel code. Also any new livepatch must be able to detect what changes have already been done by the already installed livepatches.

This is where the livepatch system state tracking gets useful. It allows to:

- store data needed to manipulate and restore the system state
- define compatibility between livepatches using a change id and version

6.1 1. Livepatch system state API

The state of the system might get modified either by several livepatch callbacks or by the newly used code. Also it must be possible to find changes done by already installed livepatches.

Each modified state is described by *struct klp_state*, see include/linux/livepatch.h.

Each livepatch defines an array of struct klp_states. They mention all states that the livepatch modifies.

The livepatch author must define the following two fields for each *struct klp_state*:

- *id*
  - Non-zero number used to identify the affected system state.
- *version*
  - Number describing the variant of the system state change that is supported by the given livepatch.

The state can be manipulated using two functions:

- *klp_get_state()*
  - Get *struct klp_state* associated with the given livepatch and state id.

- *klp_get_prev_state()*
  - Get *struct klp_state* associated with the given feature id and already installed livepatches.

6.2 2. Livepatch compatibility

The system state version is used to prevent loading incompatible livepatches. The check is done when the livepatch is enabled. The rules are:

- Any completely new system state modification is allowed.
- System state modifications with the same or higher version are allowed for already modified system states.
- Cumulative livepatches must handle all system state modifications from already installed livepatches.
- Non-cumulative livepatches are allowed to touch already modified system states.

6.3 3. Supported scenarios

Livepatches have their life-cycle and the same is true for the system state changes. Every compatible livepatch has to support the following scenarios:

- Modify the system state when the livepatch gets enabled and the state has not been already modified by a livepatches that are being replaced.
- Take over or update the system state modification when is has already been done by a livepatch that is being replaced.
- Restore the original state when the livepatch is disabled.
- Restore the previous state when the transition is reverted. It might be the original system state or the state modification done by livepatches that were being replaced.
- Remove any already made changes when error occurs and the livepatch cannot get enabled.

6.4 4. Expected usage

System states are usually modified by livepatch callbacks. The expected role of each callback is as follows:

pre_patch()

- Allocate *state->data* when necessary. The allocation might fail and *pre_patch()* is the only callback that could stop loading of the livepatch. The allocation is not needed when the data are already provided by previously installed livepatches.
- Do any other preparatory action that is needed by the new code even before the transition gets finished. For example, initialize *state->data*.

The system state itself is typically modified in *post_patch()* when the entire system is able to handle it.

- Clean up its own mess in case of error. It might be done by a custom code or by calling *post_unpatch()* explicitly.

post_patch()

- Copy *state->data* from the previous livepatch when they are compatible.
- Do the actual system state modification. Eventually allow the new code to use it.
- Make sure that *state->data* has all necessary information.
- Free *state->data* from replaces livepatches when they are not longer needed.

pre_unpatch()

- Prevent the code, added by the livepatch, relying on the system state change.
- Revert the system state modification..

post_unpatch()

- Distinguish transition reverse and livepatch disabling by checking `klp_get_prev_state()`.
- In case of transition reverse, restore the previous system state. It might mean doing nothing.
- Remove any not longer needed setting or data.

Note: *pre_unpatch()* typically does symmetric operations to *post_patch()*. Except that it is called only when the livepatch is being disabled. Therefore it does not need to care about any previously installed livepatch.

post_unpatch() typically does symmetric operations to *pre_patch()*. It might be called also during the transition reverse. Therefore it has to handle the state of the previously installed livepatches.

CHAPTER

SEVEN

RELIABLE STACKTRACE

This document outlines basic information about reliable stacktracing.

7.1 1. Introduction

The kernel livepatch consistency model relies on accurately identifying which functions may have live state and therefore may not be safe to patch. One way to identify which functions are live is to use a stacktrace.

Existing stacktrace code may not always give an accurate picture of all functions with live state, and best-effort approaches which can be helpful for debugging are unsound for livepatching. Livepatching depends on architectures to provide a *reliable* stacktrace which ensures it never omits any live functions from a trace.

7.2 2. Requirements

Architectures must implement one of the reliable stacktrace functions. Architectures using CONFIG_ARCH_STACKWALK must implement 'arch_stack_walk_reliable', and other architectures must implement 'save_stack_trace_tsk_reliable'.

Principally, the reliable stacktrace function must ensure that either:

- The trace includes all functions that the task may be returned to, and the return code is zero to indicate that the trace is reliable.
- The return code is non-zero to indicate that the trace is not reliable.

Note: In some cases it is legitimate to omit specific functions from the trace, but all other functions must be reported. These cases are described in further detail below.

Secondly, the reliable stacktrace function must be robust to cases where the stack or other unwind state is corrupt or otherwise unreliable. The function should attempt to detect such cases and return a non-zero error code, and should not get stuck in an infinite loop or access memory in an unsafe way. Specific cases are described in further detail below.

7.3 3. Compile-time analysis

To ensure that kernel code can be correctly unwound in all cases, architectures may need to verify that code has been compiled in a manner expected by the unwinder. For example, an unwinder may expect that functions manipulate the stack pointer in a limited way, or that all functions use specific prologue and epilogue sequences. Architectures with such requirements should verify the kernel compilation using objtool.

In some cases, an unwinder may require metadata to correctly unwind. Where necessary, this metadata should be generated at build time using objtool.

7.4 4. Considerations

The unwinding process varies across architectures, their respective procedure call standards, and kernel configurations. This section describes common details that architectures should consider.

7.4.1 4.1 Identifying successful termination

Unwinding may terminate early for a number of reasons, including:

- Stack or frame pointer corruption.
- Missing unwind support for an uncommon scenario, or a bug in the unwinder.
- Dynamically generated code (e.g. eBPF) or foreign code (e.g. EFI runtime services) not following the conventions expected by the unwinder.

To ensure that this does not result in functions being omitted from the trace, even if not caught by other checks, it is strongly recommended that architectures verify that a stacktrace ends at an expected location, e.g.

- Within a specific function that is an entry point to the kernel.
- At a specific location on a stack expected for a kernel entry point.
- On a specific stack expected for a kernel entry point (e.g. if the architecture has separate task and IRQ stacks).

7.4.2 4.2 Identifying unwindable code

Unwinding typically relies on code following specific conventions (e.g. manipulating a frame pointer), but there can be code which may not follow these conventions and may require special handling in the unwinder, e.g.

- Exception vectors and entry assembly.
- Procedure Linkage Table (PLT) entries and veneer functions.
- Trampoline assembly (e.g. ftrace, kprobes).
- Dynamically generated code (e.g. eBPF, optprobe trampolines).
- Foreign code (e.g. EFI runtime services).

To ensure that such cases do not result in functions being omitted from a trace, it is strongly recommended that architectures positively identify code which is known to be reliable to unwind from, and reject unwinding from all other code.

Kernel code including modules and eBPF can be distinguished from foreign code using '__kernel_text_address()'. Checking for this also helps to detect stack corruption.

There are several ways an architecture may identify kernel code which is deemed unreliable to unwind from, e.g.

- Placing such code into special linker sections, and rejecting unwinding from any code in these sections.
- Identifying specific portions of code using bounds information.

7.4.3 4.3 Unwinding across interrupts and exceptions

At function call boundaries the stack and other unwind state is expected to be in a consistent state suitable for reliable unwinding, but this may not be the case part-way through a function. For example, during a function prologue or epilogue a frame pointer may be transiently invalid, or during the function body the return address may be held in an arbitrary general purpose register. For some architectures this may change at runtime as a result of dynamic instrumentation.

If an interrupt or other exception is taken while the stack or other unwind state is in an inconsistent state, it may not be possible to reliably unwind, and it may not be possible to identify whether such unwinding will be reliable. See below for examples.

Architectures which cannot identify when it is reliable to unwind such cases (or where it is never reliable) must reject unwinding across exception boundaries. Note that it may be reliable to

unwind across certain exceptions (e.g. IRQ) but unreliable to unwind across other exceptions (e.g. NMI).

Architectures which can identify when it is reliable to unwind such cases (or have no such cases) should attempt to unwind across exception boundaries, as doing so can prevent unnecessarily stalling livepatch consistency checks and permits livepatch transitions to complete more quickly.

7.4.4 4.4 Rewriting of return addresses

Some trampolines temporarily modify the return address of a function in order to intercept when that function returns with a return trampoline, e.g.

- An ftrace trampoline may modify the return address so that function graph tracing can intercept returns.
- A kprobes (or optprobes) trampoline may modify the return address so that kretprobes can intercept returns.

When this happens, the original return address will not be in its usual location. For trampolines which are not subject to live patching, where an unwinder can reliably determine the original return address and no unwind state is altered by the trampoline, the unwinder may report the original return address in place of the trampoline and report this as reliable. Otherwise, an unwinder must report these cases as unreliable.

Special care is required when identifying the original return address, as this information is not in a consistent location for the duration of the entry trampoline or return trampoline. For example, considering the x86_64 'return_to_handler' return trampoline:

```
SYM_CODE_START(return_to_handler)
        UNWIND_HINT_UNDEFINED
        subq  $24, %rsp

        /* Save the return values */
        movq %rax, (%rsp)
        movq %rdx, 8(%rsp)
        movq %rbp, %rdi

        call ftrace_return_to_handler

        movq %rax, %rdi
        movq 8(%rsp), %rdx
        movq (%rsp), %rax
        addq $24, %rsp
        JMP_NOSPEC rdi
SYM_CODE_END(return_to_handler)
```

While the traced function runs its return address on the stack points to the start of return_to_handler, and the original return address is stored in the task's cur_ret_stack. During this time the unwinder can find the return address using ftrace_graph_ret_addr().

When the traced function returns to return_to_handler, there is no longer a return address on the stack, though the original return address is still stored in the task's cur_ret_stack. Within ftrace_return_to_handler(), the original return address is removed from cur_ret_stack

and is transiently moved arbitrarily by the compiler before being returned in rax. The return_to_handler trampoline moves this into rdi before jumping to it.

Architectures might not always be able to unwind such sequences, such as when ftrace_return_to_handler() has removed the address from cur_ret_stack, and the location of the return address cannot be reliably determined.

It is recommended that architectures unwind cases where return_to_handler has not yet been returned to, but architectures are not required to unwind from the middle of return_to_handler and can report this as unreliable. Architectures are not required to unwind from other trampolines which modify the return address.

7.4.5 4.5 Obscuring of return addresses

Some trampolines do not rewrite the return address in order to intercept returns, but do transiently clobber the return address or other unwind state.

For example, the x86_64 implementation of optprobes patches the probed function with a JMP instruction which targets the associated optprobe trampoline. When the probe is hit, the CPU will branch to the optprobe trampoline, and the address of the probed function is not held in any register or on the stack.

Similarly, the arm64 implementation of DYNAMIC_FTRACE_WITH_REGS patches traced functions with the following:

```
MOV X9, X30
BL <trampoline>
```

The MOV saves the link register (X30) into X9 to preserve the return address before the BL clobbers the link register and branches to the trampoline. At the start of the trampoline, the address of the traced function is in X9 rather than the link register as would usually be the case.

Architectures must either ensure that unwinders either reliably unwind such cases, or report the unwinding as unreliable.

7.4.6 4.6 Link register unreliability

On some other architectures, 'call' instructions place the return address into a link register, and 'return' instructions consume the return address from the link register without modifying the register. On these architectures software must save the return address to the stack prior to making a function call. Over the duration of a function call, the return address may be held in the link register alone, on the stack alone, or in both locations.

Unwinders typically assume the link register is always live, but this assumption can lead to unreliable stack traces. For example, consider the following arm64 assembly for a simple function:

```
function:
        STP X29, X30, [SP, -16]!
        MOV X29, SP
        BL <other_function>
        LDP X29, X30, [SP], #16
        RET
```

At entry to the function, the link register (x30) points to the caller, and the frame pointer (X29) points to the caller's frame including the caller's return address. The first two instructions create a new stackframe and update the frame pointer, and at this point the link register and the frame pointer both describe this function's return address. A trace at this point may describe this function twice, and if the function return is being traced, the unwinder may consume two entries from the fgraph return stack rather than one entry.

The BL invokes 'other_function' with the link register pointing to this function's LDR and the frame pointer pointing to this function's stackframe. When 'other_function' returns, the link register is left pointing at the BL, and so a trace at this point could result in 'function' appearing twice in the backtrace.

Similarly, a function may deliberately clobber the LR, e.g.

```
caller:
        STP X29, X30, [SP, -16]!
        MOV X29, SP
        ADR LR, <callee>
        BLR LR
        LDP X29, X30, [SP], #16
        RET
```

The ADR places the address of 'callee' into the LR, before the BLR branches to this address. If a trace is made immediately after the ADR, 'callee' will appear to be the parent of 'caller', rather than the child.

Due to cases such as the above, it may only be possible to reliably consume a link register value at a function call boundary. Architectures where this is the case must reject unwinding across exception boundaries unless they can reliably identify when the LR or stack value should be used (e.g. using metadata generated by objtool).

CHAPTER

EIGHT

LIVEPATCHING APIS

8.1 Livepatch Enablement

int **klp_enable_patch**(struct *klp_patch* *patch)

enable the livepatch

Parameters

struct klp_patch *patch
patch to be enabled

Description

Initializes the data structure associated with the patch, creates the sysfs interface, performs the needed symbol lookups and code relocations, registers the patched functions with ftrace.

This function is supposed to be called from the livepatch module_init() callback.

Return

0 on success, otherwise error

8.2 Shadow Variables

void ***klp_shadow_get**(void *obj, unsigned long id)

retrieve a shadow variable data pointer

Parameters

void *obj
pointer to parent object

unsigned long id
data identifier

Return

the shadow variable data element, NULL on failure.

void ***klp_shadow_alloc**(void *obj, unsigned long id, size_t size, gfp_t gfp_flags,
klp_shadow_ctor_t ctor, void *ctor_data)

allocate and add a new shadow variable

Parameters

void *obj
pointer to parent object

unsigned long id
data identifier

size_t size
size of attached data

gfp_t gfp_flags
GFP mask for allocation

klp_shadow_ctor_t ctor
custom constructor to initialize the shadow data (optional)

void *ctor_data
pointer to any data needed by **ctor** (optional)

Description

Allocates **size** bytes for new shadow variable data using **gfp_flags**. The data are zeroed by default. They are further initialized by **ctor** function if it is not NULL. The new shadow variable is then added to the global hashtable.

If an existing <obj, id> shadow variable can be found, this routine will issue a WARN, exit early and return NULL.

This function guarantees that the constructor function is called only when the variable did not exist before. The cost is that **ctor** is called in atomic context under a spin lock.

Return

the shadow variable data element, NULL on duplicate or failure.

void *klp_shadow_get_or_alloc(void *obj, unsigned long id, size_t size, gfp_t gfp_flags,
klp_shadow_ctor_t ctor, void *ctor_data)

get existing or allocate a new shadow variable

Parameters

void *obj
pointer to parent object

unsigned long id
data identifier

size_t size
size of attached data

gfp_t gfp_flags
GFP mask for allocation

klp_shadow_ctor_t ctor
custom constructor to initialize the shadow data (optional)

void *ctor_data
pointer to any data needed by **ctor** (optional)

Description

Returns a pointer to existing shadow data if an <obj, id> shadow variable is already present. Otherwise, it creates a new shadow variable like *klp_shadow_alloc()*.

This function guarantees that only one shadow variable exists with the given **id** for the given **obj**. It also guarantees that the constructor function will be called only when the variable did not exist before. The cost is that **ctor** is called in atomic context under a spin lock.

Return

the shadow variable data element, NULL on failure.

void **klp_shadow_free**(void *obj, unsigned long id, klp_shadow_dtor_t dtor)
detach and free a <obj, id> shadow variable

Parameters

void *obj
pointer to parent object

unsigned long id
data identifier

klp_shadow_dtor_t dtor
custom callback that can be used to unregister the variable and/or free data that the shadow variable points to (optional)

Description

This function releases the memory for this <obj, id> shadow variable instance, callers should stop referencing it accordingly.

void **klp_shadow_free_all**(unsigned long id, klp_shadow_dtor_t dtor)
detach and free all <_, id> shadow variables

Parameters

unsigned long id
data identifier

klp_shadow_dtor_t dtor
custom callback that can be used to unregister the variable and/or free data that the shadow variable points to (optional)

Description

This function releases the memory for all <_, id> shadow variable instances, callers should stop referencing them accordingly.

8.3 System State Changes

struct *klp_state* ***klp_get_state**(struct *klp_patch* *patch, unsigned long id)
get information about system state modified by the given patch

Parameters

struct klp_patch *patch
livepatch that modifies the given system state

unsigned long id
custom identifier of the modified system state

Description

Checks whether the given patch modifies the given system state.

The function can be called either from pre/post (un)patch callbacks or from the kernel code added by the livepatch.

Return

pointer to *struct klp_state* when found, otherwise NULL.

struct *klp_state* ***klp_get_prev_state**(unsigned long id)
get information about system state modified by the already installed livepatches

Parameters

unsigned long id
custom identifier of the modified system state

Description

Checks whether already installed livepatches modify the given system state.

The same system state can be modified by more non-cumulative livepatches. It is expected that the latest livepatch has the most up-to-date information.

The function can be called only during transition when a new livepatch is being enabled or when such a transition is reverted. It is typically called only from pre/post (un)patch callbacks.

Return

pointer to the latest struct klp_state from already
installed livepatches, NULL when not found.

8.4 Object Types

struct **klp_func**
function structure for live patching

Definition:

```
struct klp_func {
    const char *old_name;
    void *new_func;
    unsigned long old_sympos;
    void *old_func;
    struct kobject kobj;
    struct list_head node;
    struct list_head stack_node;
    unsigned long old_size, new_size;
    bool nop;
    bool patched;
    bool transition;
};
```

Members

old_name
name of the function to be patched

new_func
pointer to the patched function code

old_sympos
a hint indicating which symbol position the old function can be found (optional)

old_func
pointer to the function being patched

kobj
kobject for sysfs resources

node
list node for klp_object func_list

stack_node
list node for klp_ops func_stack list

old_size
size of the old function

new_size
size of the new function

nop
temporary patch to use the original code again; dyn. allocated

patched
the func has been added to the klp_ops list

transition
the func is currently being applied or reverted

Description

The patched and transition variables define the func's patching state. When patching, a func is always in one of the following states:

> patched=0 transition=0: unpatched patched=0 transition=1: unpatched, temporary starting state patched=1 transition=1: patched, may be visible to some tasks patched=1 transition=0: patched, visible to all tasks

And when unpatching, it goes in the reverse order:

> patched=1 transition=0: patched, visible to all tasks patched=1 transition=1: patched, may be visible to some tasks patched=0 transition=1: unpatched, temporary ending state patched=0 transition=0: unpatched

struct **klp_callbacks**
pre/post live-(un)patch callback structure

Definition:

```
struct klp_callbacks {
    int (*pre_patch)(struct klp_object *obj);
    void (*post_patch)(struct klp_object *obj);
```

```
    void (*pre_unpatch)(struct klp_object *obj);
    void (*post_unpatch)(struct klp_object *obj);
    bool post_unpatch_enabled;
};
```

Members

pre_patch
executed before code patching

post_patch
executed after code patching

pre_unpatch
executed before code unpatching

post_unpatch
executed after code unpatching

post_unpatch_enabled
flag indicating if post-unpatch callback should run

Description

All callbacks are optional. Only the pre-patch callback, if provided, will be unconditionally executed. If the parent klp_object fails to patch for any reason, including a non-zero error status returned from the pre-patch callback, no further callbacks will be executed.

struct **klp_object**
kernel object structure for live patching

Definition:

```
struct klp_object {
    const char *name;
    struct klp_func *funcs;
    struct klp_callbacks callbacks;
    struct kobject kobj;
    struct list_head func_list;
    struct list_head node;
    struct module *mod;
    bool dynamic;
    bool patched;
};
```

Members

name
module name (or NULL for vmlinux)

funcs
function entries for functions to be patched in the object

callbacks
functions to be executed pre/post (un)patching

kobj
kobject for sysfs resources

func_list
dynamic list of the function entries

node
list node for klp_patch obj_list

mod
kernel module associated with the patched object (NULL for vmlinux)

dynamic
temporary object for nop functions; dynamically allocated

patched
the object's funcs have been added to the klp_ops list

struct **klp_state**
state of the system modified by the livepatch

Definition:

```
struct klp_state {
    unsigned long id;
    unsigned int version;
    void *data;
};
```

Members

id
system state identifier (non-zero)

version
version of the change

data
custom data

struct **klp_patch**
patch structure for live patching

Definition:

```
struct klp_patch {
    struct module *mod;
    struct klp_object *objs;
    struct klp_state *states;
    bool replace;
    struct list_head list;
    struct kobject kobj;
    struct list_head obj_list;
    bool enabled;
    bool forced;
    struct work_struct free_work;
```

```
    struct completion finish;
};
```

Members

mod
reference to the live patch module

objs
object entries for kernel objects to be patched

states
system states that can get modified

replace
replace all actively used patches

list
list node for global list of actively used patches

kobj
kobject for sysfs resources

obj_list
dynamic list of the object entries

enabled
the patch is enabled (but operation may be incomplete)

forced
was involved in a forced transition

free_work
patch cleanup from workqueue-context

finish
for waiting till it is safe to remove the patch module

INDEX

K