Using ftrace to hook to functions

Written for: 4.14

Introduction

The ftrace infrastructure was originally created to attach callbacks to the beginning of functions in order to record and trace the flow of the kernel. But callbacks to the start of a function can have other use cases. Either for live kernel patching, or for security monitoring. This document describes how to use ftrace to implement your own function callbacks.

The ftrace context

> **Warning**
>
> The ability to add a callback to almost any function within the kernel comes with risks. A callback can be called from any context (normal, softirq, irq, and NMI). Callbacks can also be called just before going to idle, during CPU bring up and takedown, or going to user space. This requires extra care to what can be done inside a callback. A callback can be called outside the protective scope of RCU.

There are helper functions to help against recursion, and making sure RCU is watching. These are explained below.

The ftrace_ops structure

To register a function callback, a ftrace_ops is required. This structure is used to tell ftrace what function should be called as the callback as well as what protections the callback will perform and not require ftrace to handle.

There is only one field that is needed to be set when registering an ftrace_ops with ftrace:

```
struct ftrace_ops ops = {
      .func                    = my_callback_func,
      .flags                   = MY_FTRACE_FLAGS
      .private                 = any_private_data_structure,
};
```

Both .flags and .private are optional. Only .func is required.

To enable tracing call:

```
register_ftrace_function(&ops);
```

To disable tracing call:

```
unregister_ftrace_function(&ops);
```

The above is defined by including the header:

```
#include <linux/ftrace.h>
```

The registered callback will start being called some time after the register_ftrace_function() is called and before it returns. The exact time that callbacks start being called is dependent upon architecture and scheduling of services. The callback itself will have to handle any synchronization if it must begin at an exact moment.

The unregister_ftrace_function() will guarantee that the callback is no longer being called by functions after the unregister_ftrace_function() returns. Note that to perform this guarantee, the unregister_ftrace_function() may take some time to finish.

The callback function

The prototype of the callback function is as follows (as of v4.14):

```
void callback_func(unsigned long ip, unsigned long parent_ip,
                   struct ftrace_ops *op, struct pt_regs *regs);
```

@ip

This is the instruction pointer of the function that is being traced. (where the fentry or mcount is within the function)

@parent_ip

This is the instruction pointer of the function that called the the function being traced (where the call of the function occurred).

@op

This is a pointer to ftrace_ops that was used to register the callback. This can be used to pass data to the callback via the private pointer.

@regs

If the FTRACE_OPS_FL_SAVE_REGS or FTRACE_OPS_FL_SAVE_REGS_IF_SUPPORTED flags are set in the ftrace_ops structure, then this will be pointing to the pt_regs structure like it would be if an breakpoint was placed at the start of the function where ftrace was tracing. Otherwise it either contains garbage, or NULL.

Protect your callback

As functions can be called from anywhere, and it is possible that a function called by a callback may also be traced, and call that same callback, recursion protection must be used. There are two helper functions that can help in this regard. If you start your code with:

```
int bit;

bit = ftrace_test_recursion_trylock(ip, parent_ip);
if (bit < 0)
        return;
```

and end it with:

```
ftrace_test_recursion_unlock(bit);
```

The code in between will be safe to use, even if it ends up calling a function that the callback is tracing. Note, on success, ftrace_test_recursion_trylock() will disable preemption, and the ftrace_test_recursion_unlock() will enable it again (if it was previously enabled). The instruction pointer (ip) and its parent (parent_ip) is passed to ftrace_test_recursion_trylock() to record where the recursion happened (if CONFIG_FTRACE_RECORD_RECURSION is set).

Alternatively, if the FTRACE_OPS_FL_RECURSION flag is set on the ftrace_ops (as explained below), then a helper trampoline will be used to test for recursion for the callback and no recursion test needs to be done. But this is at the expense of a slightly more overhead from an extra function call.

If your callback accesses any data or critical section that requires RCU protection, it is best to make sure that RCU is "watching", otherwise that data or critical section will not be protected as expected. In this case add:

```
if (!rcu_is_watching())
        return;
```

Alternatively, if the FTRACE_OPS_FL_RCU flag is set on the ftrace_ops (as explained below), then a helper trampoline will be used to test for rcu_is_watching for the callback and no other test needs to be done. But this is at the expense of a slightly more overhead from an extra function call.

The ftrace FLAGS

The ftrace_ops flags are all defined and documented in include/linux/ftrace.h. Some of the flags are used for internal infrastructure of ftrace, but the ones that users should be aware of are the following:

FTRACE_OPS_FL_SAVE_REGS

If the callback requires reading or modifying the pt_regs passed to the callback, then it must set this flag. Registering a ftrace_ops with this flag set on an architecture that does not support passing of pt_regs to the callback will fail.

FTRACE_OPS_FL_SAVE_REGS_IF_SUPPORTED

Similar to SAVE_REGS but the registering of a ftrace_ops on an architecture that does not support passing of regs will not fail with this flag set. But the callback must check if regs is NULL or not to determine if the architecture supports it.

FTRACE_OPS_FL_RECURSION

By default, it is expected that the callback can handle recursion. But if the callback is not that worried about overehead, then setting this bit will add the recursion protection around the callback by calling a helper function that will do the recursion protection and only call the callback if it did not recurse.

Note, if this flag is not set, and recursion does occur, it could cause the system to crash, and possibly reboot via a triple fault.

Not, if this flag is set, then the callback will always be called with preemption disabled. If it is not set, then it is possible (but not guaranteed) that the callback will be called in preemptable context.

FTRACE_OPS_FL_IPMODIFY

Requires FTRACE_OPS_FL_SAVE_REGS set. If the callback is to "hijack" the traced function (have another function called instead of the traced function), it requires setting this flag. This is what live kernel patches uses. Without this flag the pt_regs->ip can not be modified.

Note, only one ftrace_ops with FTRACE_OPS_FL_IPMODIFY set may be registered to any given function at a time.

FTRACE_OPS_FL_RCU

If this is set, then the callback will only be called by functions where RCU is "watching". This is required if the callback function performs any rcu_read_lock() operation.

RCU stops watching when the system goes idle, the time when a CPU is taken down and comes back online, and when entering from kernel to user space and back to kernel space. During these transitions, a callback may be executed and RCU synchronization will not protect it.

FTRACE_OPS_FL_PERMANENT

If this is set on any ftrace ops, then the tracing cannot disabled by writing 0 to the proc sysctl ftrace_enabled. Equally, a callback with the flag set cannot be registered if ftrace_enabled is 0.

Livepatch uses it not to lose the function redirection, so the system stays protected.

Filtering which functions to trace

If a callback is only to be called from specific functions, a filter must be set up. The filters are added by name, or ip if it is known.

```
int ftrace_set_filter(struct ftrace_ops *ops, unsigned char *buf,
                      int len, int reset);
```

@ops

The ops to set the filter with

@buf

The string that holds the function filter text.

@len

The length of the string.

@reset

Non-zero to reset all filters before applying this filter.

Filters denote which functions should be enabled when tracing is enabled. If @buf is NULL and reset is set, all functions will be enabled for tracing.

The @buf can also be a glob expression to enable all functions that match a specific pattern.

See Filter Commands in :file:`Documentation/trace/ftrace.rst`.

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\trace\(linux-master) (Documentation) (trace)ftrace-uses.rst`, line 256); ***backlink***

Unknown interpreted text role "file".

To just trace the schedule function:

```
ret = ftrace_set_filter(&ops, "schedule", strlen("schedule"), 0);
```

To add more functions, call the ftrace_set_filter() more than once with the @reset parameter set to zero. To remove the current filter set and replace it with new functions defined by @buf, have @reset be non-zero.

To remove all the filtered functions and trace all functions:

```
ret = ftrace_set_filter(&ops, NULL, 0, 1);
```

Sometimes more than one function has the same name. To trace just a specific function in this case, ftrace_set_filter_ip() can be used.

```
ret = ftrace_set_filter_ip(&ops, ip, 0, 0);
```

Although the ip must be the address where the call to fentry or mcount is located in the function. This function is used by perf and kprobes that gets the ip address from the user (usually using debug info from the kernel).

If a glob is used to set the filter, functions can be added to a "notrace" list that will prevent those functions from calling the callback. The "notrace" list takes precedence over the "filter" list. If the two lists are non-empty and contain the same functions, the callback will not be called by any function.

An empty "notrace" list means to allow all functions defined by the filter to be traced.

```
int ftrace_set_notrace(struct ftrace_ops *ops, unsigned char *buf,
                       int len, int reset);
```

This takes the same parameters as ftrace_set_filter() but will add the functions it finds to not be traced. This is a separate list from the filter list, and this function does not modify the filter list.

A non-zero @reset will clear the "notrace" list before adding functions that match @buf to it.

Clearing the "notrace" list is the same as clearing the filter list

```
ret = ftrace_set_notrace(&ops, NULL, 0, 1);
```

The filter and notrace lists may be changed at any time. If only a set of functions should call the callback, it is best to set the filters before registering the callback. But the changes may also happen after the callback has been registered.

If a filter is in place, and the @reset is non-zero, and @buf contains a matching glob to functions, the switch will happen during the time of the ftrace_set_filter() call. At no time will all functions call the callback.

```
ftrace_set_filter(&ops, "schedule", strlen("schedule"), 1);

register_ftrace_function(&ops);

msleep(10);

ftrace_set_filter(&ops, "try_to_wake_up", strlen("try_to_wake_up"), 1);
```

is not the same as:

```
ftrace_set_filter(&ops, "schedule", strlen("schedule"), 1);

register_ftrace_function(&ops);

msleep(10);

ftrace_set_filter(&ops, NULL, 0, 1);

ftrace_set_filter(&ops, "try_to_wake_up", strlen("try_to_wake_up"), 0);
```

As the latter will have a short time where all functions will call the callback, between the time of the reset, and the time of the new setting of the filter.