Reliable Stacktrace

This document outlines basic information about reliable stacktracing.

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.

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 futher 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.

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.

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.

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).

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.

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.

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:

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\livepatch\[linux-master][Documentation][livepatch]reliable-stacktrace.rst, line 183)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

   SYM_CODE_START(return_to_handler)
           UNWIND_HINT_EMPTY
           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.

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:

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\livepatch\[linux-master][Documentation][livepatch]reliable-stacktrace.rst, line 239)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

   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.

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:

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\livepatch\[linux-master][Documentation][livepatch]reliable-stacktrace.rst, line 266)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none
```

```
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.

System Message: WARNING/2 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\livepatch\[linux-master][Documentation][livepatch]reliable-stacktrace.rst, line 291)

Cannot analyze code. No Pygments lexer found for "none".

```
.. code-block:: none

   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).