Virtually Mapped Kernel Stack Support

Author: Shuah Khan <skhan@linuxfoundation.org>

Overview

This is a compilation of information from the code and original patch series that introduced the *Virtually Mapped Kernel Stacks feature <https://lwn.net/Articles/694348/>*

Introduction

Kernel stack overflows are often hard to debug and make the kernel susceptible to exploits. Problems could show up at a later time making it difficult to isolate and root-cause.

Virtually-mapped kernel stacks with guard pages causes kernel stack overflows to be caught immediately rather than causing difficult to diagnose corruptions.

HAVE_ARCH_VMAP_STACK and VMAP_STACK configuration options enable support for virtually mapped stacks with guard pages. This feature causes reliable faults when the stack overflows. The usability of the stack trace after overflow and response to the overflow itself is architecture dependent.

> **Note**
>
> As of this writing, arm64, powerpc, riscv, s390, um, and x86 have support for VMAP_STACK.

HAVE_ARCH_VMAP_STACK

Architectures that can support Virtually Mapped Kernel Stacks should enable this bool configuration option. The requirements are:

- vmalloc space must be large enough to hold many kernel stacks. This may rule out many 32-bit architectures.
- Stacks in vmalloc space need to work reliably. For example, if vmap page tables are created on demand, either this mechanism needs to work while the stack points to a virtual address with unpopulated page tables or arch code (switch_to() and switch_mm(), most likely) needs to ensure that the stack's page table entries are populated before running on a possibly unpopulated stack.
- If the stack overflows into a guard page, something reasonable should happen. The definition of "reasonable" is flexible, but instantly rebooting without logging anything would be unfriendly.

VMAP_STACK

VMAP_STACK bool configuration option when enabled allocates virtually mapped task stacks. This option depends on HAVE_ARCH_VMAP_STACK.

- Enable this if you want the use virtually-mapped kernel stacks with guard pages. This causes kernel stack overflows to be caught immediately rather than causing difficult-to-diagnose corruption.

> **Note**
>
> Using this feature with KASAN requires architecture support for backing virtual mappings with real shadow memory, and KASAN_VMALLOC must be enabled.

> **Note**
>
> VMAP_STACK is enabled, it is not possible to run DMA on stack allocated data.

Kernel configuration options and dependencies keep changing. Refer to the latest code base:

Kconfig <https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/arch/Kconfig>

Allocation

When a new kernel thread is created, thread stack is allocated from virtually contiguous memory pages from the page level allocator. These pages are mapped into contiguous kernel virtual space with PAGE_KERNEL protections.

alloc_thread_stack_node() calls __vmalloc_node_range() to allocate stack with PAGE_KERNEL protections.

- Allocated stacks are cached and later reused by new threads, so memcg accounting is performed manually on assigning/releasing stacks to tasks. Hence, __vmalloc_node_range is called without __GFP_ACCOUNT.
- vm_struct is cached to be able to find when thread free is initiated in interrupt context. free_thread_stack() can be called in interrupt context.
- On arm64, all VMAP's stacks need to have the same alignment to ensure that VMAP'd stack overflow detection works correctly. Arch specific vmap stack allocator takes care of this detail.
- This does not address interrupt stacks - according to the original patch

Thread stack allocation is initiated from clone(), fork(), vfork(), kernel_thread() via kernel_clone(). Leaving a few hints for searching the code base to understand when and how thread stack is allocated.

Bulk of the code is in: *kernel/fork.c <https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/kernel/fork.c>*.

stack_vm_area pointer in task_struct keeps track of the virtually allocated stack and a non-null stack_vm_area pointer serves as a indication that the virtually mapped kernel stacks are enabled.

```
struct vm_struct *stack_vm_area;
```

Stack overflow handling

Leading and trailing guard pages help detect stack overflows. When stack overflows into the guard pages, handlers have to be careful not overflow the stack again. When handlers are called, it is likely that very little stack space is left.

On x86, this is done by handling the page fault indicating the kernel stack overflow on the double-fault stack.

Testing VMAP allocation with guard pages

How do we ensure that VMAP_STACK is actually allocating with a leading and trailing guard page? The following lkdtm tests can help detect any regressions.

```
void lkdtm_STACK_GUARD_PAGE_LEADING()
void lkdtm_STACK_GUARD_PAGE_TRAILING()
```

Conclusions

- A percpu cache of vmalloced stacks appears to be a bit faster than a high-order stack allocation, at least when the cache hits.
- THREAD_INFO_IN_TASK gets rid of arch-specific thread_info entirely and simply embed the thread_info (containing only flags) and 'int cpu' into task_struct.
- The thread stack can be free'ed as soon as the task is dead (without waiting for RCU) and then, if vmapped stacks are in use, cache the entire stack for reuse on the same cpu.