Contribution ID: **46** Type: **not specified**

Restricted kernel address spaces

Thursday, 27 August 2020 09:00 (45 minutes)

This proposal is recycled from the one I've suggested to LSF/MM/BPF [0]. Unfortunately, LSF/MM/BPF was cancelled, but I think it is still relevant.

Restricted mappings in the kernel mode may improve mitigation of hardware speculation vulnerabilities and minimize the damage exploitable kernel bugs can cause.

There are several ongoing efforts to use restricted address spaces in Linux kernel for various use cases:
* speculation vulnerabilities mitigation in KVM [1]
* support for memory areas with more restrictive protection that the defaults ("secret", or "protected" memory) [2], [3], [4]
* hardening of the Linux containers [no reference yet :)]

Last year we had vague ideas and possible directions, this year we have several real challenges and design decisions we'd like to discuss:

- "Secret" memory userspace APIs

Should such API follow "native" MM interfaces like mmap(), mprotect(), madvise() or it would be better to use a file descriptor , e.g. like memfd-create does?

MM "native" APIs would require VM_something flag and probably a page flag or page_ext. With file-descriptor VM_SPECIAL and custom implementation of .mmap() and .fault() would suffice. On the other hand, mmap() and mprotect() seem better fit semantically and they could be more easily adopted by the userspace.

- Direct/linear map fragmentation

Whenever we want to drop some mappings from the direct map or even change the protection bits for some memory area, the gigantic and huge pages that comprise the direct map need to be broken and there's no THP for the kernel page tables to collapse them back. Moreover, the existing API defined in <asm/set$_memory.h > by several architectures do not really presume it would be widely used.

For the "secret" memory use-case the fragmentation can be minimized by caching large pages, use them to satisfy smaller "secret" allocations and than collapse them back once the "secret" memory is freed. Another possibility is to pre-allocate physical memory at boot time.

Yet another idea is to make page allocator aware of the direct map layout.

- Kernel page table management

Currently we presume that only one kernel page table exists (well, mostly) and the page table abstraction is required only for the user page tables. As such, we presume that 'page table == struct mm_struct' and the mm_struct is used all over by the operations that manage the page tables.

The management of the restricted address space in the kernel requires ability to create, update and remove kernel contexts the same way we do for the userspace.

One way is to overload the mm_struct, like EFI and text poking did. But it is quite an overkill, because most of the mm_struct contains information required to manage user mappings.

My suggestion is to introduce a first class abstraction for the page table and then it could be used in the same way for user and kernel context management. For now I have a very basic POC that slitted several fields from the mm_struct into a new 'struct pg_table' [5]. This new abstraction can be used e.g. by PTI implementation of the page table cloning and the KVM ASI work.

[0] https://lore.kernel.org/linux-mm/20200206165900.GD17499@linux.ibm.com/
[1] https://lore.kernel.org/lkml/20200504145810.11882-1-alexandre.chartre@oracle.com
[2] https://lore.kernel.org/lkml/20190612170834.14855-1-mhillenb@amazon.de/
[3] https://lore.kernel.org/lkml/20200130162340.GA14232@rapoport-lnx/
[4] https://lore.kernel.org/lkml/20200522125214.31348-1-kirill.shutemov@linux.intel.com
[5] https://git.kernel.org/pub/scm/linux/kernel/git/rppt/linux.git/log/?h=pg_table/v0.0

I agree to abide by the anti-harassment policy

I agree

Primary author: RAPOPORT, Mike (IBM)

Presenter: RAPOPORT, Mike (IBM)

Session Classification: Kernel Summit

Track Classification: Kernel Summit