Kernel Address Space Isolation

Tweaking the Kernel Page Table to Limit Host Data Exposure

Alexandre Chartre
Oracle

Mike Rapoport
James Bottomley
Joel Nider
IBM

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 825377
Oracle - Safe Harbor Statement

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, timing and pricing of any features or functionality described for Oracle’s products may change and remains at the sole discretion of Oracle Corporation.
Host Kernel Data Exposure

- VMs and containers rely on a secure host kernel
  - Host kernel has access to all memory
  - Can store secrets (e.g. encryption keys)
  - Can access data from other VMs/containers

- But I am safe! VMs and containers can’t access host kernel data
  - Yes, but sometimes they can…
  - Example: Spectre attack derivatives (L1TF, MDS)

- Can we limit damages if host kernel data access is compromised?
  - Restrict kernel host data visibility?
What You Don’t Know Won’t Hurt You

- Kernel uses a single address space
  - Same address space used by all kernel components
- Unmapped memory is harder to access
- Page tables ensure hardware enforced context boundary
  - Known, reliable solution to isolate kernel and processes
  - Enforced even during speculative execution
- Can we change kernel mappings for certain contexts?
  - Protect data by having restricted mappings
  - Have more than one PGD for the kernel?
Use Cases

- Page Table Isolation (PTI)
  - Restricted context for kernel-mode code on entry boundary

- KVM Address Space Isolation
  - Restricted context for KVM VMExit handlers

- Process-Local Memory
  - Kernel memory visible only in the context of a specific `process`

- Namespace-Local Memory
  - Kernel memory visible only in the context of a specific `namespace`
KVM Address Space Isolation
CPU Hyper-Threading Data Leakage

- Data can leak between sibling CPU hyper-threads (example: speculative attacks like L1TF or MDS)
- Major issue for virtualization
  - Allow guest-to-guest attacks
  - And guest-to-host attacks
- Basic mitigation: disable CPU hyper-threading
  - Significant performance impact
  - Can we do better?
Address Space Isolation (ASI)

- Original idea from Liran Alon (Oracle)
- Define a page table with limited kernel mappings (typically with no secret/sensitive data)
- Switch to ASI page table
  - Explicitly (ASI Enter), to run with non-sensitive data
- Switch back to full kernel page table
  - Explicitly (ASI Exit), when full kernel access is needed
  - Implicity (ASI Abort), on interrupt, exception, context switch
• Include core kernel mappings  
  (kernel text, GDT, current stack)

• Minimum kernel mappings to:
  ○ Enter/Exit ASI
  ○ Enter interrupt/exception handlers

• Optional kernel mappings  
  (stack canary, CPU offsets, current task…)

• Should not map kernel secrets or sensitive data
KVM Address Space Isolation

- Use ASI when running a guest VCPU
  - KVM_RUN ioctl
  - enter KVM ASI
  - Run VM (VMEnter)
  - Run (most) VMExit handlers with ASI
  - Loop

- ASI page table with KVM data
  - KVM guest data limited to a specified guest or vcpu

- Mitigation for HT data leakage between guest and host
  - Guest can run while sibling thread runs VMExit handlers in host
  - Host has no sensitive data to leak
KVM ASI Expectations

● Benefits
  ○ Prevent guest-to-host attacks
  ○ Including guest-to-guest attack through host

● But only when the ASI page table is used
  ○ ASI Exit requires sibling HT to exit guest

● Limitations
  ○ Does not prevent direct guest-to-guest attacks
  ○ Should pin each VM to a distinct CPU cores
AS1 Challenges

- Any use case other than KVM?
- Difficult to identify all data to map
  - Code analyzer to find all memory access?
- Map data granularity (4K) can cause adjacent data to leak
- Interaction with memory allocation
- Need some integration with the scheduler
  - Need to interact with sibling when entering/exiting ASI
- Unassessed performance impact
Process-Local Memory
Process-Local Memory Overview

- Proposal from Julian Stecklina and Marius Hillenbrand (Amazon)
- Make some kernel data non-globally visible
- Data visible only in the context of a specified process
  - More precisely of a specified mm
  - mm-local memory is a better name
- Secrets should be explicitly stored in that region
- Prevent other processes to peak at secrets
Process-Local Memory Details

- Add private memory mapping to the process kernel page table
  - Use a process exclusive PGD entry
- Memory allocated with specific functions
  - kmalloc_proclocal(), kfree_proclocal()
- Allocated memory is removed from direct map
  - Memory not visible from regular kernel address space
Using Process-Local Memory with KVM

- Store KVM guest data in process-local memory
  - General-purpose registers, FPU registers…

- Benefits
  - Prevent KVM from sharing data among all guests
  - Prevent guest-to-guest attack through host

- Limitations
  - Does not prevent direct guest-to-guest attacks
  - Should pin each VM to a distinct CPU cores
  - Guest-to-host attack still possible (to collect non-guest data)
Namespace-Local Memory
Address Space for Namespaces

- Kernel objects in a namespace are logically “private”
- Ensure they are visible only to processes belonging to the namespace
- Example: netns
  - Most objects are bound to `struct net` in some way
  - `struct sk_buff` is trickier as packets cross namespace boundaries
Namespace PGDs

- Kernel page-table per namespace
- Processes in a namespace share view of the kernel mappings
- Some kernel objects are mapped only in the namespace PGD
Private memory allocations

- Extend `alloc_page()` and `kmalloc()` with context awareness
- Pages and objects are visible in a single context
  - Can be a process or all processes in a namespace
- Special care for objects traversing context boundaries
Kernel Page Table and Context Consolidation
Kernel Page Table Management

- Improve API for managing kernel page table
- Dissociate page table and mm
  - struct pg_table
- Clone and populate page table
  - Copy page table entries at a specified level
- Drop page table mappings
- On-demand map/unmap
Kernel Context Creation

- Pre-built at boot time (PTI)
- When creating process
  - During clone()
  - PTI page table, process-local page table
- When specifying namespace
  - During unshare() or setns()
  - Namespace-local page table
- When creating VM or virtual CPU
  - During KVM vcpu_create() or vm_create()
  - KVM ASI page table
Kernel Context Switch

- Explicit transitions
  - Syscall boundary (PTI)
  - ASI enter/exit

- Implicit transitions
  - Interrupt/exception, context switch

- Need unified mechanism to switch kernel page table
  - Same mechanism for PTI and ASI

- No change for processes with private memory
struct pg_table
{
    spinlock_t    page_table_lock;
    atomic_long_t pgtables_bytes;
    pgt_t         *pgd;
    /* more fields */
    pgt_context_t context;
};

struct mm_struct {
    struct pg_table pgt;
    ...
};
Tracking Page Table Pages

- Required for freeing page tables
  - Especially when lower levels are shared with `init_mm.pgd`
- Two longs in `struct page` and/or `page_ext`

```c
struct {
    /* Page table pages */
    unsigned long _pt_pad_1;  /* compound_head */
    pgtable_t pmd_huge_pte;  /* protected by page->ptl */
    unsigned long _pt_pad_2;  /* mapping */
}
```

- Ensure `PageTable(page)` is consistent for all page table pages
  - Currently it’s not the case
Per-Context Allocations

* Allow per-context allocations
  - __GFP_EXCLUSIVE – for pages
  - SLAB_EXCLUSIVE – for slabs
  - PG_exclusive page type

* Drop pages from the direct map on allocation
  - set_memory_np()/set_pages_np()

* Put them back on freeing
  - set_memory_p()/set_pages_p()

* Only allowed in a context of a process with non-default page table
  - if (current->mm && &current->mm.pgt != &init_mm.pgt)
Private SL*B Caches

- First per-context allocation creates a new cache
  - Similar to memcg child caches
    - kmalloc-1k
      - cgroup
        - kmalloc-1k(108:A)
      - kmalloc-1k(1)
    - cgroup

- Allocate pages for cache with \_\_GFP\_EXCLUSIVE

- Map/unmap pages for out-of-context accesses
  - SLUB debugging
  - SLAB freeing from other context, e.g. workqueue
mmap(SECRET) / madvise(SECRET)

- Pages in a secret mapping are dropped from the direct map
- Visible only in the “owning” context
Conclusion

- Reducing kernel mapping can help reduce scope of attacks
  - Especially for cloud environment, virtualization
- Page table and context management can be improved
- Solutions more or less intrusive and simple
  - process/namespace-local memory: just add PGD entries
  - ASI: define an entire page table
Next Steps

- First, improve kernel page table and context management
  - And provide building blocks to create custom kernel page tables

- Next, implement kernel local memory
  - Unify process and namespace-local memory
  - Could be used to refactor LDT

- Finally, continue evaluating ASI
  - Still many hurdles before ASI is production ready
  - Need more investigations
  - Is the complexity worth the benefit?
Appendix
References

- **Address Space Isolation and KVM ASI**
  - RFC
    - [https://lore.kernel.org/r/1562855138-19507-1-git-send-email-alexandre.chartre@oracle.com](https://lore.kernel.org/r/1562855138-19507-1-git-send-email-alexandre.chartre@oracle.com)
  - Blog

- **System Calls Address Space Isolation RFC**
  - [https://lore.kernel.org/r/1556228754-12996-1-git-send-email-rppt@linux.ibm.com](https://lore.kernel.org/r/1556228754-12996-1-git-send-email-rppt@linux.ibm.com)

- **Process-Local Memory RFC**
  - [https://lore.kernel.org/r/20190612170834.14855-1-mhillenb@amazon.de](https://lore.kernel.org/r/20190612170834.14855-1-mhillenb@amazon.de)