Address Space Isolation in the Linux Kernel

Mike Rapoport, James Bottomley
<{rppt,jejb}@linux.ibm.com>

This project has received funding from the European Union’s Horizon 2020 research and innovation programme under grant agreement No 825377
Containers, clouds and security

- **From chroot to cloud-native**
  - Containers are everywhere

- Often containers run inside VMs

- But why?
  - VMs provide isolation
  - Containers are easy for DevOps

- Is this nesting really necessary?
Hardware isolation

- VMs isolation is enforced by hardware

- For containers we have MMU!
  - Address space isolation is one of the best protection methods since the invention of the virtual memory.
  - Vulnerabilities are inevitable, how can we minimize the damage
  - Make parts of the Linux kernel use a restricted address space for better security
Securing containers with MMU

- System call interface is a large attack surface
  - Can we restrict kernel mappings during system call execution?

- Major container isolation are namespaces
  - Can we protect namespaces with page tables?
Related work

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

- WIP: improve mitigation for HyperThreading leaks
  - KVM address space isolation
    - Restricted context for KVM VMExit handlers
  - Process local memory
    - Kernel memory visible only in the context of a specific process
System Call Isolation (SCI)

- Execute system calls in a restricted address space
  - System calls run with **very** limited page tables
  - Accesses to most of the kernel code and data cause page faults

- Ability to inspect and verify memory accesses
  - For code: only allow calls and jumps to known symbols to prevent ROP attacks
  - For data: TBD?

https://lore.kernel.org/lkml/1556228754-12996-1-git-send-email-rppt@linux.ibm.com/
SCI page tables

Kernel Page Table
- User space
- Kernel entry
- Kernel space

System call Page Table
- User space
- Kernel entry
- Syscall entry

User Page Table
- User space
- Kernel entry
SCI flow

1. System call
2. Switch address space
3. Access unmapped code
4. Page fault
5. Switch address space
6. Map the page
7. Is access safe?
   - Yes
   - No
8. Kill process
SCI in practice

- **Weakness**
  - Cannot verify RET targets
  - Performance degradation
  - Page granularity
  - Intel CET makes SCI irrelevant

- **Follow up possibility**
  - Use ftrace to construct shadow stack
  - Utilize compiler return thunk to verify RET targets
Exclusive mappings

• Memory region mapped only in a single process page table
  ○ Excluded from the direct map

• Use-cases
  ○ Store secrets
  ○ Protect the entire VM memory
mmap(MAP_EXCLUSIVE)

- Memory region in a process is isolated from the rest of the system
- Can be used to store secrets in memory:

```c
void *addr = mmap(MAP_EXCLUSIVE, ...);
struct iovec iov = {
    .base = addr,
    .len = PAGE_SIZE,
};

fd = open_and_decrypt("/path/to/secret.file", O_RDONLY);
readv(fd, &iov, 1);
```

https://lore.kernel.org/lkml/1572171452-7958-1-git-send-email-rppt@kernel.org/
mmap(MAP_EXCLUSIVE)

+ Convenient mmap()/mpropect()/madvise() interfaces
  ● Plugable into existing allocators
  ● Can be used at post-allocation time
+ Simple implementation

- Requires page flag and VMA flag
  ● We have ran out long time ago
- Multiple modifications to core mm core

— Fragmentation of the direct map
Extension to `memfd_create()` system call

```c
int fd, ret;
void *p;

fd = memfd_create("secure", MFD_CLOEXEC | MFD_SECRET);
if (fd < 0)
    perror("open"), exit(1);
if (ioctl(fd, MFD_SECRET_EXCLUSIVE))
    perror("ioctl"), exit(1);

p = mmap(NULL, PAGE_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
if (p == MAP_FAILED)
    perror("mmap"), exit(1);

secure_page = p;
```

https://lore.kernel.org/lkml/20200130162340.GA14232@rapoport-lnx/
memfd_create (MFD_SECRET)

+ Black magic is behind a file descriptor
  - .mmap() and .fault() hide the details from core mm
+ May use memory preallocated at boot
  - Yet to be implemented

- Auditing of core mm core is still required
- May introduce complexity into page cache and mount APIs

— Fragmentation of the direct map
Demo

```c
exit();
}
```

```c
void _attribute_(__constructor) preload_setup(void) {
    int fd = memfd_create("secure", MFD_CLOEXEC|MFD_SECRET);
    int ret;
    void *p;
    check(fd < 0, "memfd_create");
    ret = ioctl(fd, MFD_SECRET_EXCLUSIVE);
    check(ret < 0, "ioctl");

    ret = truncate(fd, PAGE_SIZE);
    check(ret < 0, "truncate");

    p = mmap(NULL, PAGE_SIZE, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
    check(p == MAP_FAILED, "mmap");
    secure_page = p;
}
```
Most objects in a namespace are private
  ○ No need to map them in other namespaces

Per-namespace page tables improve isolation
  ○ Shared between processes in a namespace
  ○ Private objects are mapped exclusively by owning namespace page table
Address space for netns

- Netns is an independent network stack
  - Network devices, sockets, protocol data
- Objects inside the network namespace are private
  - Except skb’s that cross namespace boundaries
- Exclusive mappings of netns objects effectively creates isolated networking stack, just like in a VM
Restricted Mappings Framework

1. Create a restricted mapping from an existing mapping
2. Switch to the restricted mapping when entering a particular execution context
3. Switch to the unrestricted mapping when leaving that execution context
4. Keep track of the state

* From tglx comment to KVM ASI patches:

https://lore.kernel.org/kvm/alpine.DEB.2.21.1907122059430.1669@nanos.tec.linutronix.de/
APIs for Kernel Page Table Management

- Create first class abstraction for page tables
  - Break the assumption ‘page table == struct mm_struct’
  - Introduce struct pg_table to represent page table
- Clone and populate restricted page tables
  - Copy page table entries at a specified level
- Drop mappings from the restricted page tables
- On-demand memory mapping and unmapping
- Tear down restricted page tables
Restricted 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
Context Switch

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

- **Implicit transitions**
  - Interrupt/exception, process context switch

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

- **No change for processes with private memory**
Freeing Restricted Page Tables

- Integration with existing TLB management infrastructure
  - Avoid excessive TLB shootdowns
- Special care for shared page table levels
  - Avoid freeing main kernel page tables
- Proper accounting of page table pages
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
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
Address space for netns

- **Kernel page table per namespace**
  
  ```c
  struct net {
    pg_table *pgt; /* namespace private page table */
    passive; /* To decide when the network */
  } /* namespace should be freed. */
  ```

- **Processes in a namespace share view of the kernel mappings**
  - Switch page table at `clone()`, `unshare()`, `setns()` time.

- **Private kernel objects are mapped only in the namespace PGD**
  - Enforced at object allocation time
Proof of concept implementation

- Private memory allocations with `kmalloc()`
  - Mapped only in processes in a single netns
  - Still visible in init_mm address space
- Socket objects, protocol data and skb’s are allocated using `__GFP_EXCLUSIVE`
- Backdoor syscall for testing
- Surprisingly, there is network traffic inside a netns ;-)
Putting it all together

User-exclusive memory

Namespaces isolation

KVM isolation

Page cache extensions

Private allocations

SL*B

Page Allocator

Page Table Management API
Conclusions

- Using restricted contexts reduces the attack surface
- Complexity vs security benefits are yet to be evaluated
- Reworking kernel address space management is a major challenge
Thank You