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

IBM

- 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

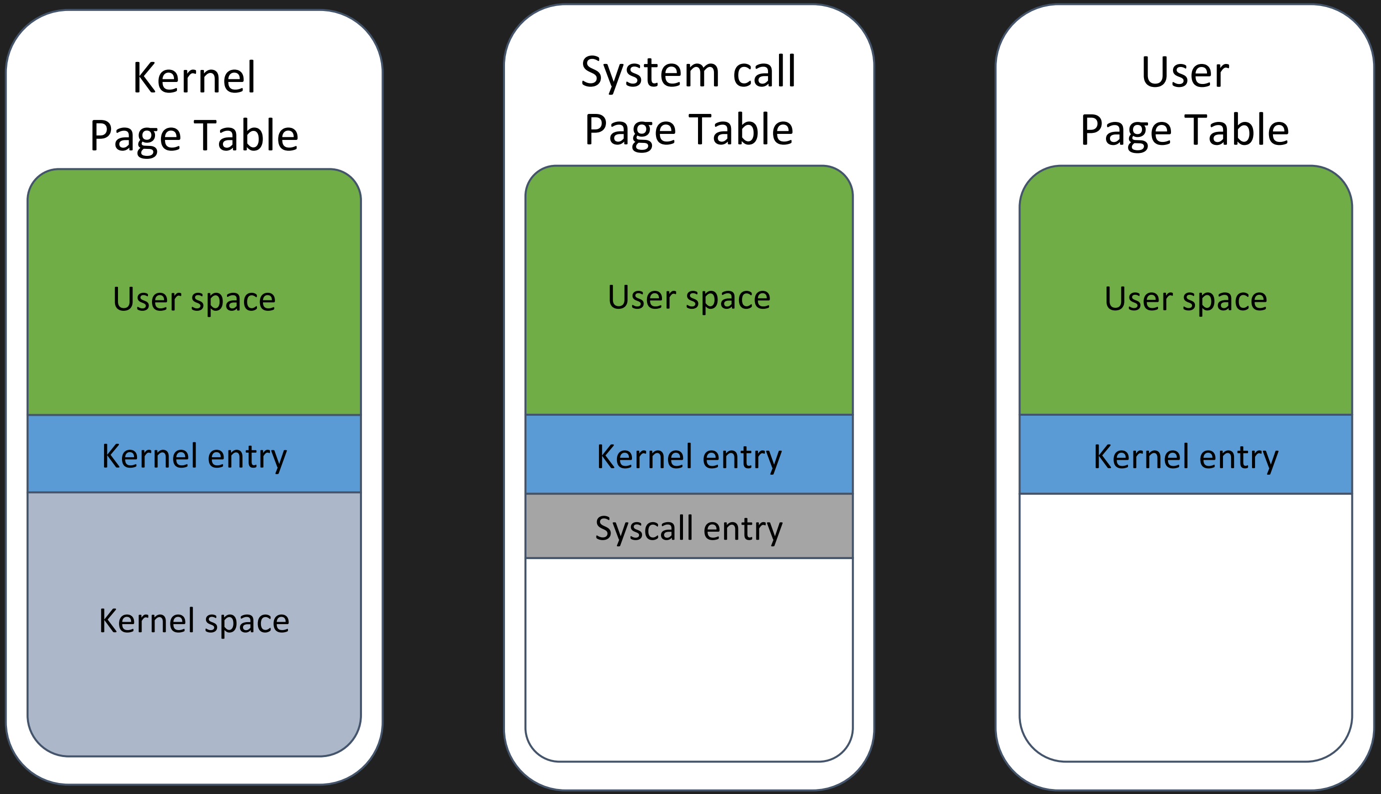
System Call Isolation (SCI)

- 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

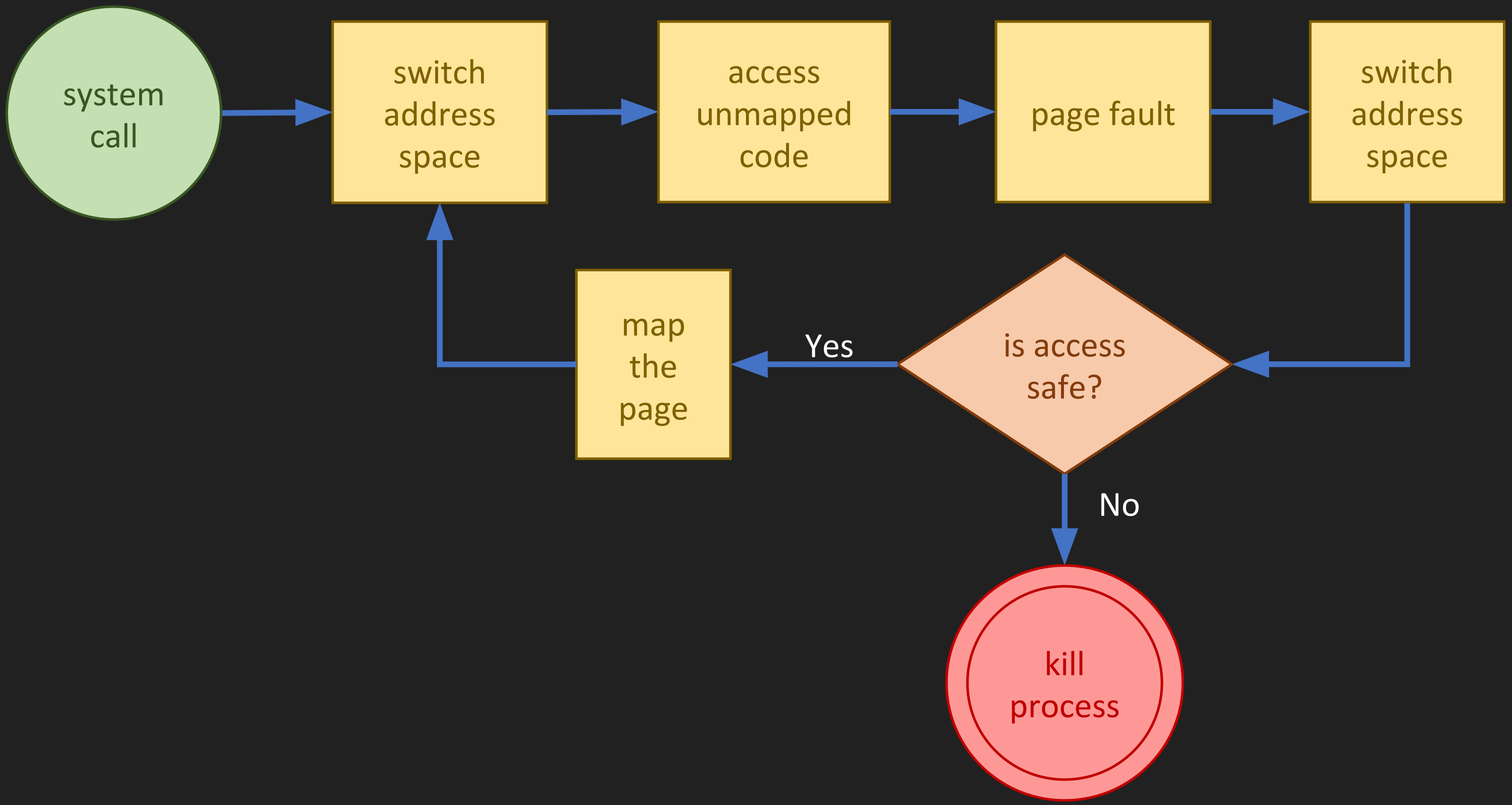
- Ability to inspect and verify memory accesses
  - For code: only allow calls and jumps to known symbols to prevent ROP attacks
  - For data: TBD?

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

SCI page tables



SCI flow


system call
switch address space
access unmapped code
page fault
switch address space
is access safe?
Yes
map the page
No
kill process

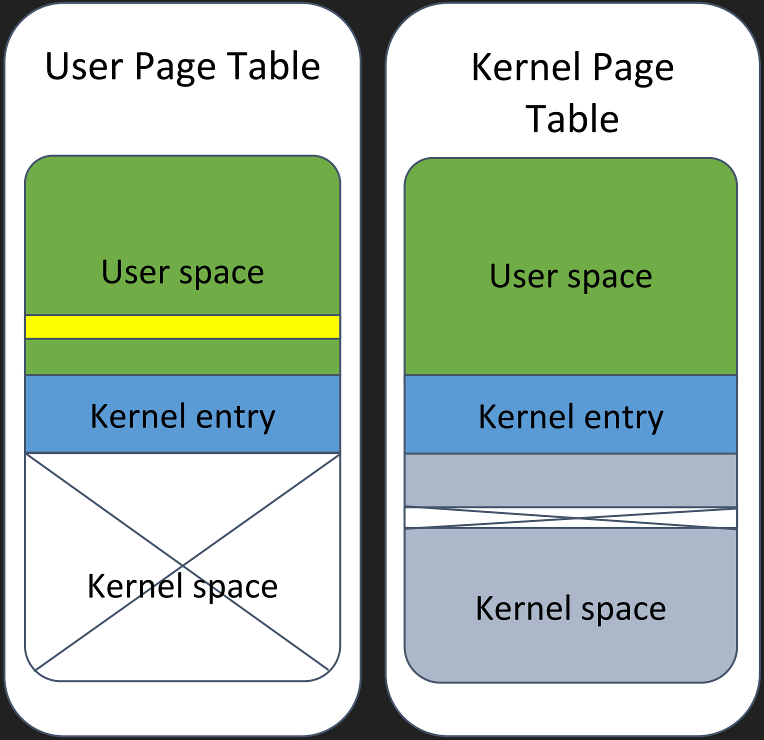
SCI in practice

- 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:

```
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

memfd_create(MFD_SECRET)

- Extension to [memfd_create()](#) system call

```
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

memfd

```
[  206.399484] RIP: 0033:0x7fa6b1a63acf
[  206.400705] Code: c7 44 24 10 08 00 00 00 48 89 44 24 18 48 8d 44 24 30 8b 70 08 48 8b 50 10
 4c 0f 43 50 18 48 89 44 24 20 b8 65 00 00 00 0f 05 <48> 3d 00 f0 ff ff 77 41 48 85 c0 78 06 41
 83 f8 02 76 1e 48 8b 4c
[  206.406200] RSP: 002b:00007ffcecb61470 EFLAGS: 00000293 ORIG_RAX: 0000000000000065
[  206.408621] RAX: ffffffffffffffda RBX: 00007f6f98135000 RCX: 00007fa6b1a63acf
[  206.410785] RDX: 00007f6f98135000 RSI: 0000000000000f27 RDI: 0000000000000001
[  206.412945] RBP: 0000000000000f27 R08: 0000000000000000 R09: 00007f6f98135000
[  206.415122] R10: 00007ffcecb61478 R11: 0000000000000293 R12: 0000000000000008
[  206.417298] R13: 00007fa6b173c718 R14: 0000000000000000 R15: 0000000000000000
[  206.419472] Modules linked in:
[  206.420532] CR2: ffff8e55273c5000
[  206.421737] ---[ end trace b1454be259cdc0c0 ]---
[  206.423243] RIP: 0010:__access_remote_vm+0x22e/0x320
[  206.424868] Code: 83 e7 f8 48 89 02 44 89 f0 49 8b 4c 04 f8 48 89 4c 02 f8 48 29 fa 41 8d 0c
 16 48 29 d6 c1 e9 03 89 c9 f3 48 a5 e9 08 ff ff ff <48> 8b 02 49 8d 7c 24 08 48 89 d6 48 83 e7
 f8 49 89 04 24 44 89 f0
[  206.430352] RSP: 0018:ffffa6be80493de0 EFLAGS: 00010246
[  206.432009] RAX: 0000000000000000 RBX: 0000000000000008 RCX: 0000000000000008
[  206.434204] RDX: ffff8e55273c5000 RSI: 00007f6f98136000 RDI: ffffa6be80493e18
[  206.436414] RBP: 0000000000000008 R08: ffffa6be80493e10 R09: 0000000000000000
[  206.438591] R10: 80000004273c5067 R11: 0000000000000067 R12: ffffa6be80493e98
[  206.440861] R13: 00007f6f98135000 R14: 0000000000000008 R15: 0000000000000000
[  206.443019] FS:  00007fa6b173c7c0(0000) GS:ffff8e552fd40000(0000) knlGS:0000000000000000
[  206.445685] CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[  206.447526] CR2: ffff8e55273c5000 CR3: 000000042ab22002 CR4: 00000000000606e0
```

```
                exit(1);
        }
}

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


        p = mmap(NULL, PAGE_SIZE, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
        check(p == MAP_FAILED, "mmap");

        secure_page = p;
}
                                                              54,1-8        80%
```

```
debian@debian:~$ cd openssl-test/
debian@debian:~/openssl-test$ LD_PRELOAD=./preload.so ./openssl_test
in crypto malloc from openssl_test.c:9
secure_ptr holds "this is a secret"
insecure_ptr holds "another secret"
```

```
Reading symbols from /lib64/ld-linux-x86-64.so.2...
Reading symbols from /usr/lib/debug/.build-id/bd/c4116b3193146db4945c36329513df492b14ab.debug.
..
0x00007f6f97d4eaf4 in __libc_pause () at ../sysdeps/unix/sysv/linux/pause.c:29
29      ../sysdeps/unix/sysv/linux/pause.c: No such file or directory.
(gdb) up
#1  0x000055e499baf226 in main (argc=1, argv=0x7ffd1468ddf8) at openssl_test.c:18
18              pause();
(gdb) l
13              strcpy(insecure_ptr, "another secret");
14
15              printf("secure_ptr holds \"%s\"\n", secure_ptr);
16              printf("insecure_ptr holds \"%s\"\n", insecure_ptr);
17
18              pause();
19
20              exit(0);
21      }
(gdb) p insecure_ptr
$1 = 0x55e49b06a6b0 "another secret"
(gdb) p secure_ptr
Killed
debian@debian:~/openssl-test$
```

```
[4] 0:ssh*                                                    "aquarius" 16:21 29-Jan-20
```

Protecting namespaces with page tables

- 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

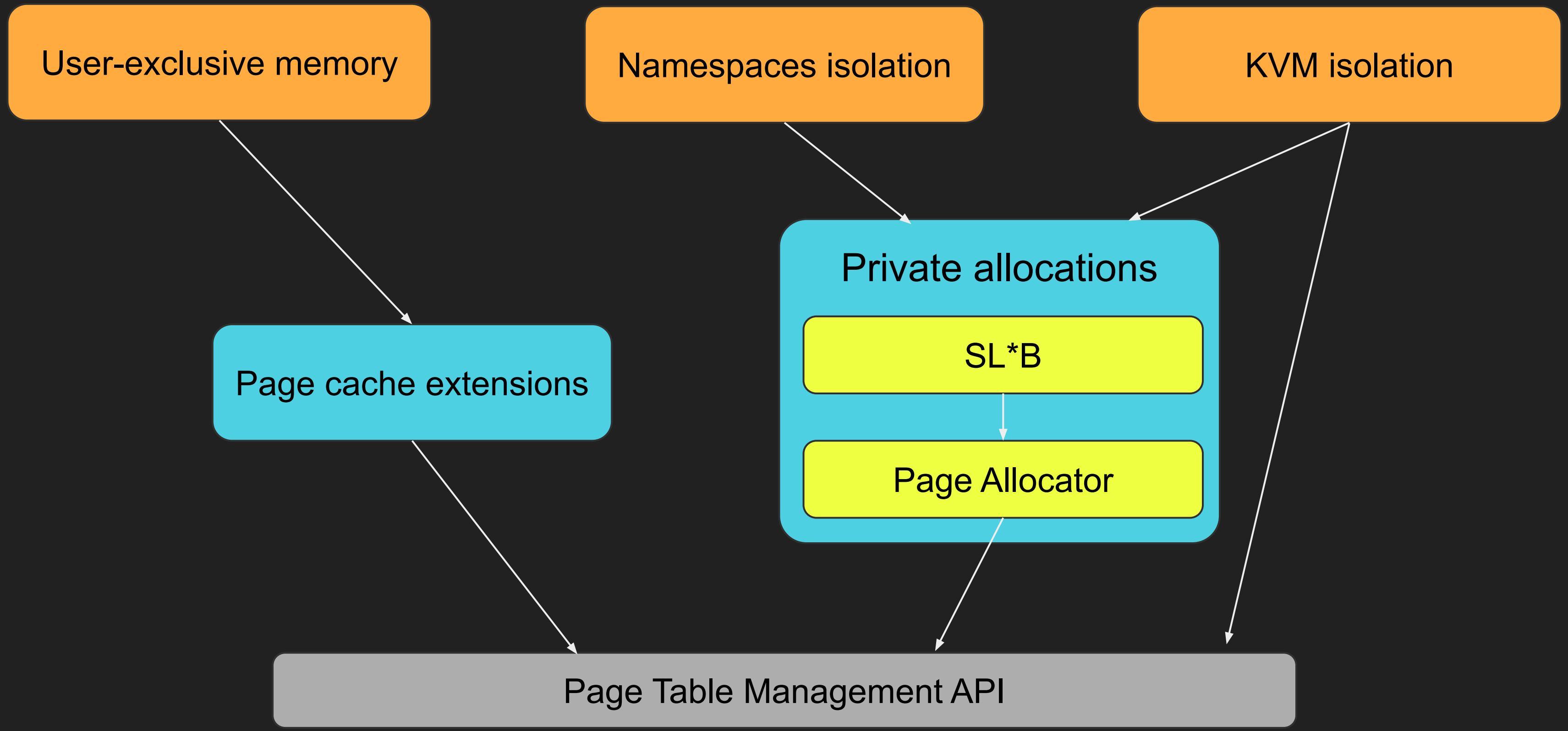
```
@@ -52,6 +52,7 @@ struct bpf_prog;
 #define NETDEV_HASHENTRIES (1 << NETDEV_HASHBITS)

 struct net {
+    pg_table                    *pgt;          /* namespace private page table */
     refcount_t                  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



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