Talos Vulnerability Report

TALOS-2021-1243

Linux Kernel Arm SIGPAGE information disclosure vulnerability

MAY 28, 2021

CVE NUMBER

CVE-2021-21781

SUMMARY

An information disclosure vulnerability exists in the ARM SIGPAGE functionality of Linux Kernel v5.4.66 and v5.4.54. The latest version (5.11-rc4) seems to still be vulnerable. A userland application can read the contents of the sigpage, which can leak kernel memory contents. An attacker can read a process's memory at a specific offset to trigger this vulnerability.

CONFIRMED VULNERABLE VERSIONS

The versions below were either tested or verified to be vulnerable by Talos or confirmed to be vulnerable by the vendor.

Linux Kernel v5.4.54
Linux Kernel v5.4.66

PRODUCT URLS

Kernel - https://github.com/torvalds/linux

CVSSV3 SCORE

4.0 - CVSS:3.0/AV:L/AC:L/PR:N/UI:N/S:U/C:L/I:N/A:N

CWE

CWE-908 - Use of Uninitialized Resource

DETAILS

The Linux Kernel is the free and open-source core of Unix-like operating systems.

When examining a given Linux process' virtual memory space on an ARMv7 processor, the quickest ways to look are `info proc map` in gdb, and also by reading a process' `/proc/self/maps` file. The output thereof might look approximately like so:

```
[^.^]> info proc map
process 37
Mapped address spaces:

        Start Addr   End Addr       Size     Offset objfile
        0xbee54000 0xbee7c000    0x28000        0x0 /lib/libgcc_s.so.1
        0xbee7c000 0xbee7d000     0x1000    0x18000 /lib/libgcc_s.so.1
        0xbee7d000 0xbeedb000    0x5e000        0x0 /usr/lib/libc.so
        0xbeeea000 0xbeeec000     0x2000    0x5d000 /usr/lib/libc.so
        0xbeeec000 0xbeeed000     0x1000        0x0
        0xbeeed000 0xbeeee000     0x1000        0x0 /mnt/apps/[...]/app
        0xbeefd000 0xbeefe000     0x1000        0x0 /mnt/apps/[...]/app
        0xbeefe000 0xbeff000     0x1000     0x1000 /mnt/apps/[...]/app
        0xbefe2000 0xbefe3000     0x1000        0x0 [sigpage]  // [1]
        0xbefe5000 0xbeffe000    0x19000        0x0 [stack]
        0xffff0000 0xffff1000     0x1000        0x0 [vectors]
```

Nothing really out of place, but let us examine the `[sigpage]` segment of memory at [1], which is used for storing signal handler information in userland:

```
[o.o]> x/200wx 0xbefe2000
0xbefe2000:     0x00000005      0x00000000      0x00000000      0xa649a76f
0xbefe2010:     0x00000000      0x00000000      0x00000000      0x00000000
0xbefe2020:     0x00000000      0xa64868c9      0xa649a7c5      0xa64c48f7
0xbefe2030:     0x00000000      0x00000000      0x00000000      0x00000002
0xbefe2040:     0x00000000      0x00000000      0x00000000      0x00000000
0xbefe2050:     0xffffffff      0x00000000      0x00000000      0x00000000
0xbefe2060:     0x00000000      0x00000000      0x00000000      0x00000000
0xbefe2070:     0x00000000      0x00000000      0x00000000      0x00000000
0xbefe2080:     0x00000000      0x00000000      0x00000000      0x00000000
0xbefe2090:     0x00000009      0x00000000      0x00000000      0xa649a76f
0xbefe20a0:     0x00000000      0x00000000      0x00000000      0x00000000
0xbefe20b0:     0xa6486819      0x00000000      0xa649a7c5      0xa64c48ff
0xbefe20c0:     0x00000100      0x00000000      0x00000000      0x00000000
0xbefe20d0:     0x00000000      0x00000000      0x00000000      0x00000000
0xbefe20e0:     0x00000000      0x00000000      0x00000000      0x00000000
0xbefe20f0:     0x00000000      0x00000000      0x00000000      0x00000000
0xbefe2100:     0x00000000      0x00000000      0x00000000      0x00000000
0xbefe2110:     0x00000000      0x00000000      0x00000000      0x00000000
0xbefe2120:     0x00000005      0x00000000      0x00000000      0xa649a76f
0xbefe2130:     0xa64c4b07      0xa64c4a07      0x00000000      0xa64c4a07
0xbefe2140:     0x00000000      0xa64868c9      0xa649a7c5      0xa64c4a07
// [...]
```

While this above output does not have any clear structure or meaning to it, we can examine the kernel code to explain exactly what we are seeing. The initialization of this page occurs within `arch_setup_additional_pages` of `arch/arm/kernel/process.c`:

```
static struct page *signal_page;
extern struct page *get_signal_page(void);

int arch_setup_additional_pages(struct linux_binprm *bprm, int uses_interp)
{
    struct mm_struct *mm = current->mm;
    struct vm_area_struct *vma;
    unsigned long npages;
    unsigned long addr;
    unsigned long hint;
    int ret = 0;

    if (!signal_page)
        signal_page = get_signal_page(); // [1]
    if (!signal_page)
        return -ENOMEM;

   // [...]
```

Since `static struct page *signal_page` is static, the page can only be initialized once, which occurs at [1] with `get_signal_page()`: The pointer to sigpage is assigned to `static struct page *signal_page` at [1], via function `get_signal_page()`:

```
struct page *get_signal_page(void)
{
    unsigned long ptr;
    unsigned offset;
    struct page *page;
    void *addr;

    page = alloc_pages(GFP_KERNEL, 0); // [1]

    if (!page)
        return NULL;

    addr = page_address(page);

    /* Give the signal return code some randomness */
    offset = 0x200 + (get_random_int() & 0x7fc);
    signal_return_offset = offset;

    /*
     * Copy signal return handlers into the vector page, and
     * set sigreturn to be a pointer to these.
     */
    memcpy(addr + offset, sigreturn_codes, sizeof(sigreturn_codes));   //[2]

    ptr = (unsigned long)addr + offset;
    flush_icache_range(ptr, ptr + sizeof(sigreturn_codes));

    return page;
}
```

At [1], the buddy allocator grabs a single page of memory, and at [2], a set of instructions are copied from `extern const unsigned long sigreturn_codes[17];` into a random spot inside of our sigpage. After this, nothing else of import materially occurs on the page, and it's returned back up to `arch_setup_additional_pages`. Continuing therein:

```
    if (!signal_page)
        signal_page = get_signal_page();  //[1]
    if (!signal_page)
        return -ENOMEM;

    npages = 1; /* for sigpage */
    npages += vdso_total_pages;

    if (down_write_killable(&mm->mmap_sem))
        return -EINTR;
    hint = sigpage_addr(mm, npages);
    addr = get_unmapped_area(NULL, hint, npages << PAGE_SHIFT, 0, 0);
    if (IS_ERR_VALUE(addr)) {
        ret = addr;
        goto up_fail;
    }

    vma = _install_special_mapping(mm, addr, PAGE_SIZE, // [2]
        VM_READ | VM_EXEC | VM_MAYREAD | VM_MAYWRITE | VM_MAYEXEC,
        &sigpage_mapping);
```

We grab our sigpage at [1] (assuming it didn't already exist), and insert it into an appropriate spot of the current memory map at [2]. To reiterate, while the signal page is only initialized once (when the `init` binary is run), it is mapped into every process that is created. This brings us back to the initial question of what exactly is within the `[sigpage]` mapping of our userland process:

```
[o.o]> x/200wx 0xbefe2000
0xbefe2000:     0x00000005      0x00000000      0x00000000      0xa649a76f
0xbefe2010:     0x00000000      0x00000000      0x00000000      0x00000000
0xbefe2020:     0x00000000      0xa64868c9      0xa649a7c5      0xa64c48f7
// [...]
0xbefe25b0:     0x00000000      0x00000000      0x00000000      0x00000000
0xbefe25c0:     0x00000000      0x00000000      0xbeaa3008      0xbeaa3008
0xbefe25d0:     0xe3a07077      0xef900077      0xdf002777      0xe3a070ad  //[1]
0xbefe25e0:     0xef9000ad      0xdf0027ad      0xe59d32f4      0xe8930208
0xbefe25f0:     0xe12fff13      0xcb0c9bbd      0x47104699      0xe59d3374
0xbefe2600:     0xe8930208      0xe12fff13      0xcb0c9bdd      0x47104699
0xbefe2610:     0x00000000      0x00000000      0x00000000      0x00000000
0xbefe2620:     0x00000000      0x00000000      0x00000000      0x00000000
//[...]
```

At [1], we do actually see the signal handler instructions that got copied in initially, disassembled they look like so:

```
[-.-]> x/40i 0xc04004ac
   0xc04004ac <sigreturn_codes>:        mov     r7, #119        ; 0x77
   0xc04004b0 <sigreturn_codes+4>:      svc     0x00900077
   0xc04004b4 <sigreturn_codes+8>:      movs    r7, #119        ; 0x77
   0xc04004b6 <sigreturn_codes+10>:     svc     0
   0xc04004b8 <sigreturn_codes+12>:     mov     r7, #173        ; 0xad
   0xc04004bc <sigreturn_codes+16>:     svc     0x009000ad
   0xc04004c0 <sigreturn_codes+20>:     movs    r7, #173        ; 0xad
   0xc04004c2 <sigreturn_codes+22>:     svc     0
   0xc04004c4 <sigreturn_codes+24>:     ldr     r3, [sp, #756]  ; 0x2f4
   0xc04004c8 <sigreturn_codes+28>:     ldm     r3, {r3, r9}
   0xc04004cc <sigreturn_codes+32>:     bx      r3
   0xc04004d0 <sigreturn_codes+36>:     ldr     r3, [sp, #756]  ; 0x2f4
   0xc04004d2 <sigreturn_codes+38>:     ldmia   r3, {r2, r3}
   0xc04004d4 <sigreturn_codes+40>:     mov     r9, r3
   0xc04004d6 <sigreturn_codes+42>:     bx      r2
   0xc04004d8 <sigreturn_codes+44>:     ldr     r3, [sp, #884]  ; 0x374
   0xc04004dc <sigreturn_codes+48>:     ldm     r3, {r3, r9}
   0xc04004e0 <sigreturn_codes+52>:     bx      r3
   0xc04004e4 <sigreturn_codes+56>:     ldr     r3, [sp, #884]  ; 0x374
   0xc04004e6 <sigreturn_codes+58>:     ldmia   r3, {r2, r3}
   0xc04004e8 <sigreturn_codes+60>:     mov     r9, r3
   0xc04004ea <sigreturn_codes+62>:     bx      r2
   0xc04004ec <sigreturn_codes+64>:     andeq   r0, r0, r0
```

However as might be obvious, the rest of the memory is uninitialized, having not been zeroed after being grabbed by `page = alloc_pages(GFP_KERNEL, 0);` inside `get_signal_page()`.

Thus, any userland process can read the `[sigpage]` mapping within their own virtual memory space to leak kernel data (that does not change until the device reboots). It's worth noting that this page's contents depend entirely on the device itself, and potentially might contain data from a previous boot if the device is not shut down for too long. To determine the freshness of the memory, one can count how many instances of the `sigreturn_codes` have been copied in.

TIMELINE

2021-01-28 - Vendor Disclosure
2021-02-05 - Vendor Patched
2021-06-25 - Public Release

CREDIT

Discovered by Lilith >_> of Cisco Talos.
