CS5231 - AY1617

Final Project

Privilege Escalation By Exploiting Vulnerability In Handling Stack Segment Exception

Group: 6
Advisor: Professor Liang Zhenkai

SINGAPORE, NOVEMBER 2016

Privilege Escalation By Exploiting Vulnerability In Handling Stack Segment Exception

Clément Fischer, Nguyen Hoang Vu, Nguyen Quoc Binh, Sébastien Iooss

***Abstract* - There are many well known privilege escalation techniques in Linux environment, for example: exploiting setuid programs, exploiting Time-of-check to Time-of-use (TOCTOU), etc. Those techniques are based on exploiting vulnerabilities happen when principle of least privilege is not enforced fully. In this paper, we present a privilege escalation techniques by exploiting the vulnerability when a process switches back from kernel space to user space after executing an interrupt. By analyzing CVE-2014-9322 (to the best of our knowledge, it is the only CVE mentions the vulnerability), the nature of vulnerability as well as the techniques for exploitation and mitigation are explained. A demonstration on how to use this vulnerability to gain root access is also covered in order to clarify the details.**

Keywords: Privilege Escalation, SMAP, Stack Segment Exception, Stack Segment Register

I. Introduction

Android Security bulletin of April 2016 fixed a Privilege Escalation vulnerability in Kernel. The vulnerability came from the Linux kernel, and is present on all versions from 3.0 to 3.17.5. It happens when a process switches back from kernel space to user space after executing an interrupt. As Android suffers from fragmentation, many devices has not been fully patched. Hence, understanding the mechanism behind the vulnerability as well as the exploitation and defense techniques is very important.
We have studied the Linux kernel to understand how the vulnerability can lead to arbitrary code execution. We were able to adapt the Proof of Concept on different kernels and systems and successfully port the exploitation to new kernels on Fedora via Loadable Kernel Module. We also investigated how the defense techniques like SMAP or UDEREF can completely prevent the vulnerability.

II. Background

II.1 Concepts of the vulnerability

A. Inter-privilege return with IRET instruction

A running process at user space reaches kernel space through an interrupt. After finishing the requested service, kernel has to return control to the process and restore its state so that it can continue from the next instruction after the interrupt. The return from an interrupt or exception handler is initiated with the IRET (Interrupt Return) instruction. IRET pops CS, the flags register, and the instruction pointer from the stack and resumes the routine that was interrupted.

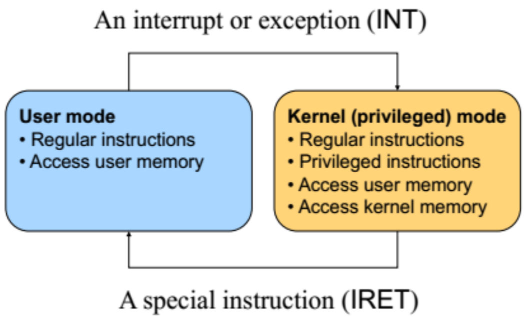


Fig. 1. Inter-privilege return with IRET instruction

To be more specific, when returning from an interrupt or exception handler, IRET performs these actions:

1. Performs a privilege check.
2. Restores the CS and EIP registers to their values prior to the interrupt or exception.
3. Restores the EFLAGS register.
4. Restores the SS and ESP registers to their values prior to the interrupt or exception, resulting in a stack switch back to the stack of the interrupted procedure.
5. Resumes execution of the interrupted procedure.

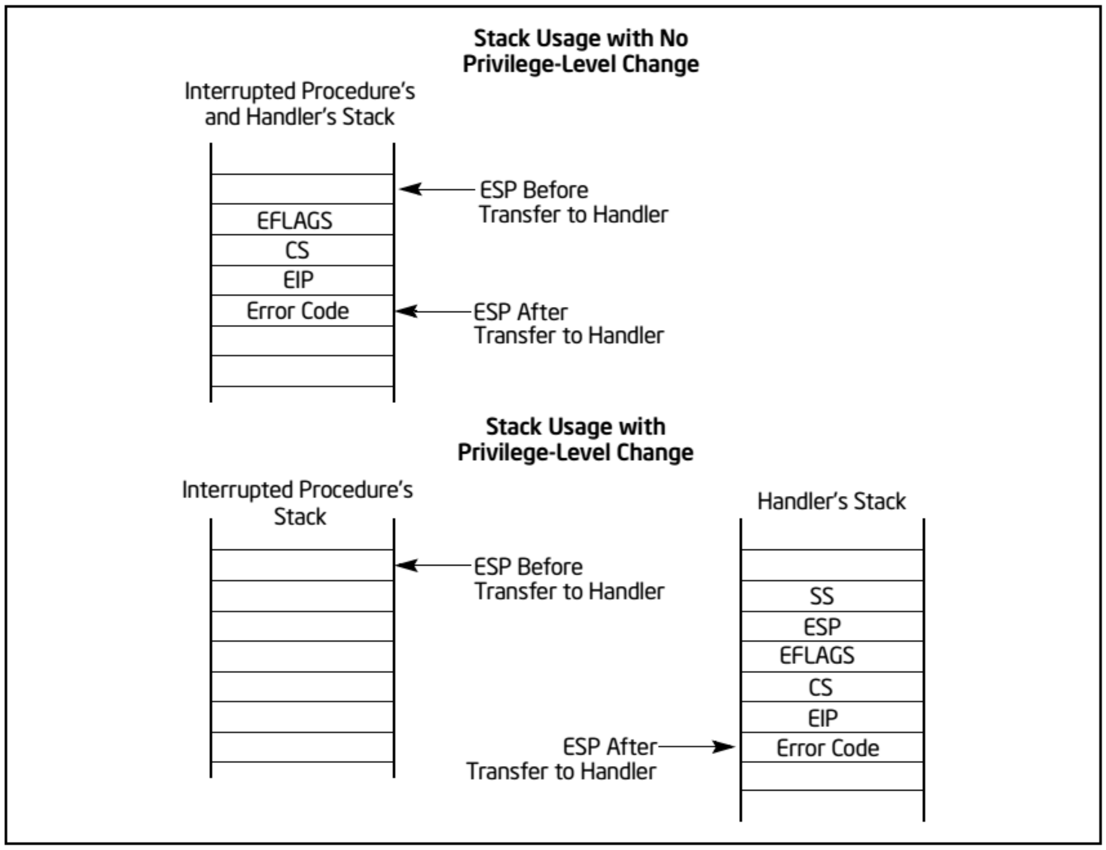

Stack Usage with No Privilege-Level Change
Interrupted Procedure's and Handler's Stack
EFLAGS
CS
EIP
Error Code
ESP Before Transfer to Handler
ESP After Transfer to Handler
Stack Usage with Privilege-Level Change
Interrupted Procedure's Stack
ESP Before Transfer to Handler
Handler's Stack
SS
ESP
EFLAGS
CS
EIP
Error Code
ESP After Transfer to Handler

Fig. 2. Stack Usage on Transfers to Interrupt and Exception Handling Routines

B. The segment register GS and SWAPGS instruction

There are six segment registers (CS, DS, SS, ES, FS, and GS). Each of those registers hold a segment selectors (16 bis). A segment selector is a special pointer that points to a segment in memory. To access a particular segment in memory, the segment selector of that segment must be presented in the appropriate segment register.

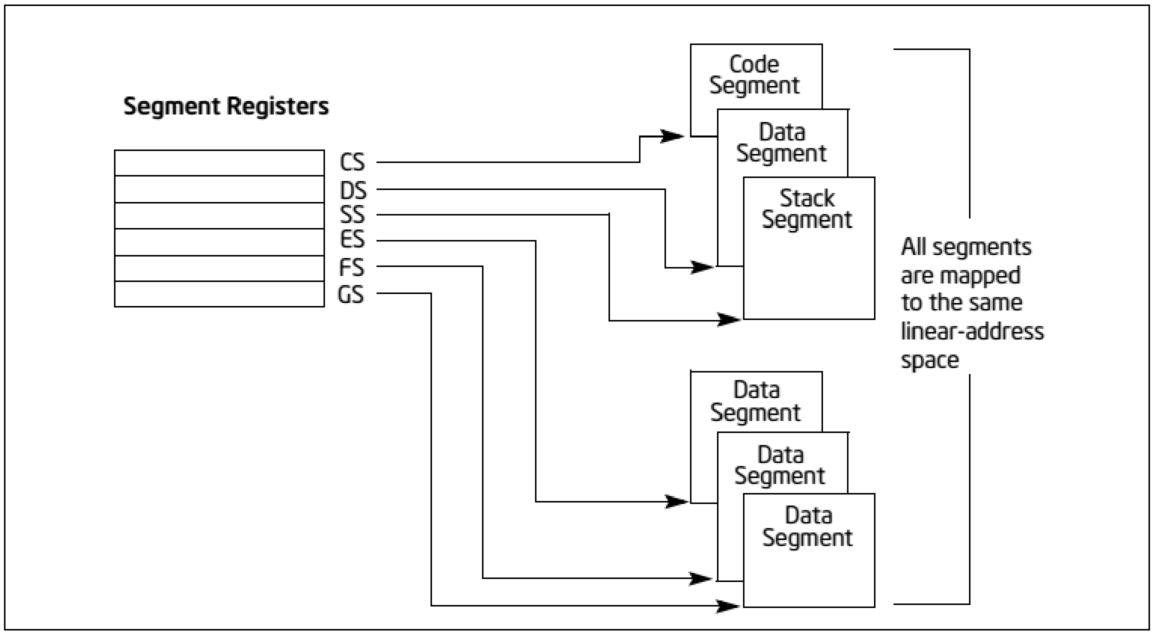


Fig. 3. Use of Segment Registers in Segmented Memory Model

The segment register GS is used by both user processes to access per-thread state data and by the kernel to access per-processor state data. As the processor enters and leaves the kernel it uses the SWAPGS instruction to toggle between the kernel and user values for the GS register. The SWAPGS instruction needs to be executed before IRET. The purpose of this instruction is swapping the content of the GS register with a value contained in one of the machine-specific registers (MSRs).

The usual pattern of syscall handler is:

1. SWAPGS
2. access Per-CPU kernel data structures via memory instructions with gs prefix
3. SWAPGS
4. return to user mode

C. The vulnerability

When the memory is accessed with gs segment prefix, the memory at the linear address: Logical Address + Base Address is accessed. The base address is derived from Global Descriptor Table, as illustrated in Fig. 4.

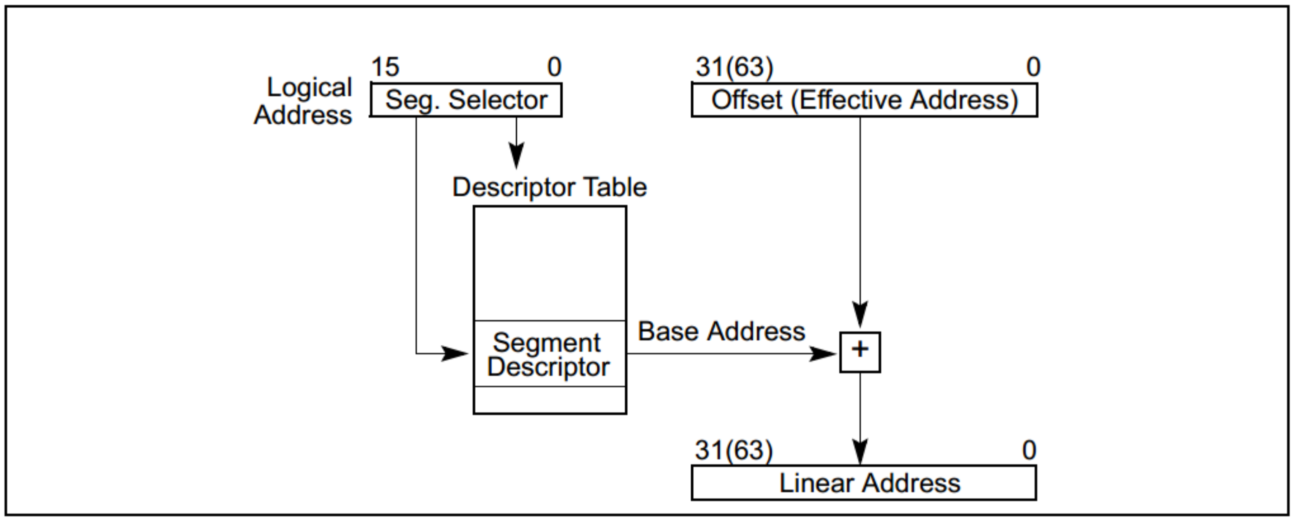


Fig. 4. Logical Address to Linear Address Translation

However, there is a situation in which gs segment base address can be changed without involving GDT. It means that kernel will access important data structures with a wrong gs base address, that can be controlled by the attacker.

According to Intel 64 and IA-32 Architectures Software Developer's Manual Vol. 2B, the section about SWAPGS instruction:

> "...The SWAPGS instruction is a privileged instruction intended for use by system software. When using SYSCALL to implement system calls, there is no kernel stack at the OS entry point. Neither is there a straightforward method to obtain a pointer to kernel structures from which the kernel stack pointer could be read. Thus, the kernel cannot save general purpose registers or reference memory. By design, SWAPGS does not require any general purpose registers or memory operands. No registers need to be saved before using the instruction. SWAPGS exchanges the CPL 0 data pointer from the IA32_KERNEL_GS_BASE MSR with the GS base register. The kernel can then use the GS prefix on normal memory references to access kernel data structures."

It can be inferred that, to ensure gs base address points to kernel memory, kernel code must ensure that whenever it accesses Per-CPU data with gs prefix, the number of SWAPGS instructions executed by the kernel since entry from user mode is non even.

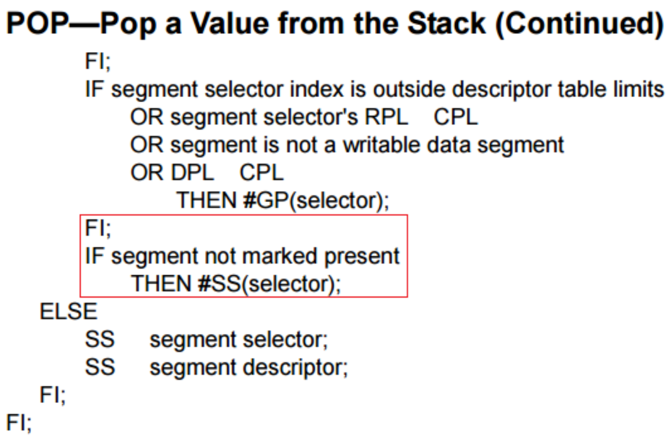
There is an important connection between the number of SWAPGS instruction and IRET. In some cases, IRET throws an exception after being executed. The exception handler returns execution to bad_iret function. If the exception is Stack-Segment Fault (#SS), the exception handler will make mistake by doing one extra SWAPGS instruction. It means the GS register will switch to the usermode GS base while the kernel still expects it to be the kernel GS. By exploiting the vulnerability caused by one extra SWAPGS in the vulnerable code path, the attacker can make a local Denial of Service (DoS) attack or gain root privilege from user mode.

D. Existing protections

It has been known that to exploit the vulnerability, a Stack-Segment Fault Exception (#SS) must be thrown by IRET. Hence, the main task for the exploitation is investigating in which cases that type of exception happens. According to Intel Manual, to raise the #SS exception, those conditions must be satisfied:

```
#SS(0)
If an attempt to pop a value off the stack violates the SS limit.
If an attempt to pop a value off the stack causes a non-canonical address to be referenced.
```

It is impossible to force one of the two above conditions happens in user mode. However, by referring POP instruction, it can be seen that when the segment defined by the return frame is not present, we can invoke the exception. Hence, #SS can be raised by setting ss register to something not present.



POP—Pop a Value from the Stack (Continued)

```
            FI;
            IF segment selector index is outside descriptor table limits
                OR segment selector's RPL    CPL
                OR segment is not a writable data segment
                OR DPL    CPL
                    THEN #GP(selector);
            FI;
            IF segment not marked present
                THEN #SS(selector);
        ELSE
            SS    segment selector;
            SS    segment descriptor;
        FI;
    FI;
```

Fig. 5. Snippet from Intel Architecture Software Developer's Manual about POP instruction

To input a value into SS register in user mode, there are some barriers caused by existing protection mechanisms. Those are:

- Setting the value of SS register by using gdb or ptrace is disallowed.
- Using mov instructions pattern:

```
mov $nonpresent_seg_selector, %eax
mov %ax, %ss
```

  is still failed, because the second instruction will generate General Protection Exception (#GP), not the Stack-Segment Fault Exception that we want.

II.2 Exploitation Techniques

A. Level 1: Denial of Service (Kernel Panic)

Another approach to raise #SS in user mode without using tools or system calls is utilizing the inter-thread communication. To be more specific, by setup the communication between two threads (both in the same process) like Fig. 6.

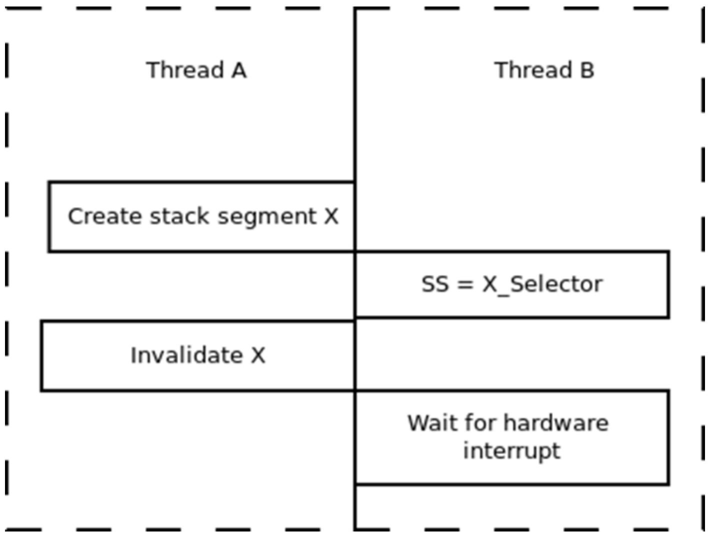


Fig. 6. Threads communication in the exploitation

Thread A creates a custom segment X in Local Descriptor Table by using a syscall which include sys_modify_ldt. The return of that syscall is done via sysret instruction that hard-codes the value of SS. If X is invalidated in the same thread that set "SS := X instruction", SS would be undone. After this step, the system is freezed (kernel panic).

B. Level 2: Privilege Escalation by overwriting a kernel code pointer

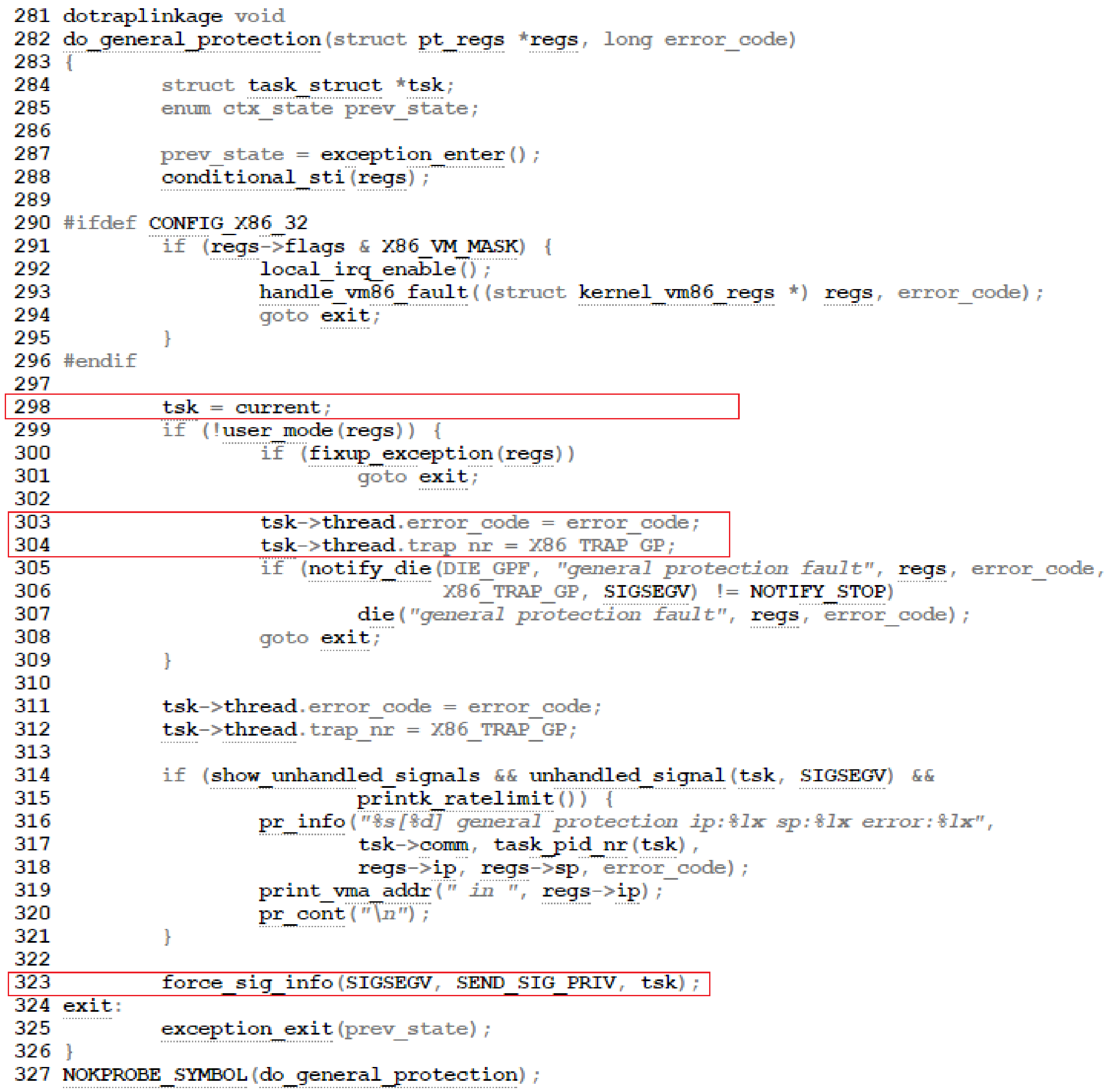
The higher level of exploitation is overwriting a kernel code pointer. The task requires finding a kernel function which has a pointer pointing to the data structure describing the whole Linux process. However, there are many potential aspects make the found kernel function not effective. Hence, we should try a new one until success. Moreover, to formalize the exploitation is very challenging since it depends much on randomness. In the below steps, we explained the most important steps in the original exploitation done by the researcher who discovered this vulnerability.

As stated before, the vulnerability happens after an interrupt. Hence, we examined kernel code (version 3.16) in the file trap.c (/arch/x86/kernel/traps.c), which handle hardware traps and faults. Specially, the function **do_general_protection**.

The noticeable lines is (298), (311), (312), (323). Line (298) can be disassembled into a mov instruction (ARBITRARY_ADDR is a random address):

```
mov    %gs: ARBITRARY_ADDR, %rbx
```

Line (311) and (312) writes to controllable addresses (at some fixed offset from the beginning of the task struct). We can utilize them to write value to an custom address X.



```
281 dotraplinkage void
282 do_general_protection(struct pt_regs *regs, long error_code)
283 {
284         struct task_struct *tsk;
285         enum ctx_state prev_state;
286
287         prev_state = exception_enter();
288         conditional_sti(regs);
289
290 #ifdef CONFIG_X86_32
291         if (regs->flags & X86_VM_MASK) {
292                 local_irq_enable();
293                 handle_vm86_fault((struct kernel_vm86_regs *) regs, error_code);
294                 goto exit;
295         }
296 #endif
297
298         tsk = current;
299         if (!user_mode(regs)) {
300                 if (fixup_exception(regs))
301                         goto exit;
302
303                 tsk->thread.error_code = error_code;
304                 tsk->thread.trap_nr = X86_TRAP_GP;
305                 if (notify_die(DIE_GPF, "general protection fault", regs, error_code,
306                                X86_TRAP_GP, SIGSEGV) != NOTIFY_STOP)
307                         die("general protection fault", regs, error_code);
308                 goto exit;
309         }
310
311         tsk->thread.error_code = error_code;
312         tsk->thread.trap_nr = X86_TRAP_GP;
313
314         if (show_unhandled_signals && unhandled_signal(tsk, SIGSEGV) &&
315                         printk_ratelimit()) {
316                 pr_info("%s[%d] general protection ip:%lx sp:%lx error:%lx",
317                         tsk->comm, task_pid_nr(tsk),
318                         regs->ip, regs->sp, error_code);
319                 print_vma_addr(" in ", regs->ip);
320                 pr_cont("\n");
321         }
322
323         force_sig_info(SIGSEGV, SEND_SIG_PRIV, tsk);
324 exit:
325         exception_exit(prev_state);
326 }
327 NOKPROBE_SYMBOL(do_general_protection);
```

Fig. 8. A code snipped of Linux kernel version 3.16, function do_general_protection

The problem to make Line (311) or Line (312) writes to an controllable address X is the values being written are constants, error_code equals to 0 and X86_TRAP_GP equals to 13 (General Protection Fault). However, we can overcome the difficulty about by making these steps:

1. Prepare an user mode memory at FAKE_PERCPU and set GS base to it

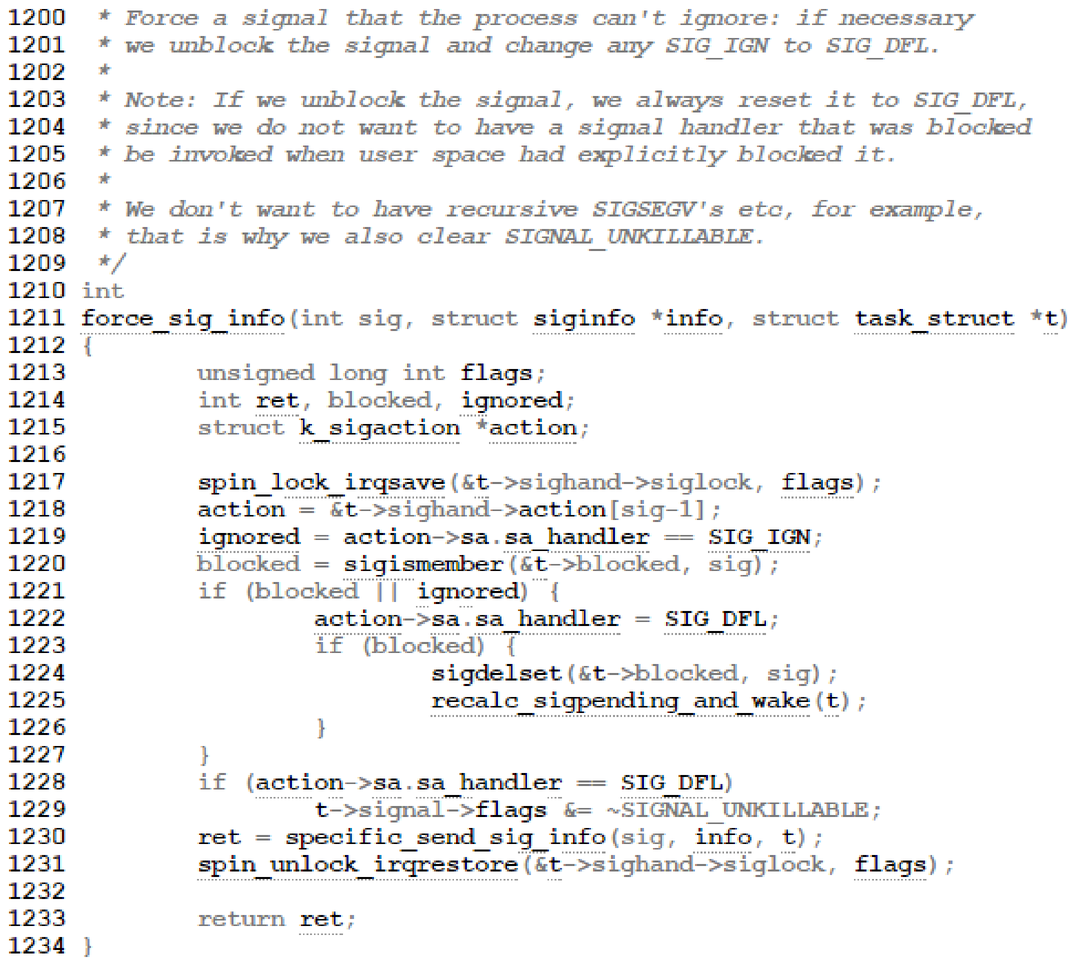
2. Make the location
   FAKE_PERCPU + ARBITRARY_ADDR

hold the pointer
FAKE_CURRENT_WITH_OFFSET,
such that:
FAKE_CURRENT_WITH_OFFSET = X – offsetof (struct task_struct, thread.error_code)

3. Trigger the vulnerability as the same way as Level 1 (threads communication)

By using those setup, do_general_protection() can be hijacked to write to the address X. However, after writing to the address X, the function will try to access other fields in task_struct. Since we have no control beyond X, the result will be a page fault in kernel.

Hence, we must move to another function. It is the function **force_sig_info** at line (323)



```
 * Force a signal that the process can't ignore: if necessary
 * we unblock the signal and change any SIG_IGN to SIG_DFL.
 *
 * Note: If we unblock the signal, we always reset it to SIG_DFL,
 * since we do not want to have a signal handler that was blocked
 * be invoked when user space had explicitly blocked it.
 *
 * We don't want to have recursive SIGSEGV's etc, for example,
 * that is why we also clear SIGNAL_UNKILLABLE.
 */
int
force_sig_info(int sig, struct siginfo *info, struct task_struct *t)
{
        unsigned long int flags;
        int ret, blocked, ignored;
        struct k_sigaction *action;

        spin_lock_irqsave(&t->sighand->siglock, flags);
        action = &t->sighand->action[sig-1];
        ignored = action->sa.sa_handler == SIG_IGN;
        blocked = sigismember(&t->blocked, sig);
        if (blocked || ignored) {
                action->sa.sa_handler = SIG_DFL;
                if (blocked) {
                        sigdelset(&t->blocked, sig);
                        recalc_sigpending_and_wake(t);
                }
        }
        if (action->sa.sa_handler == SIG_DFL)
                t->signal->flags &= ~SIGNAL_UNKILLABLE;
        ret = specific_send_sig_info(sig, info, t);
        spin_unlock_irqrestore(&t->sighand->siglock, flags);

        return ret;
}
```

Fig. 9. A code snipped of Linux kernel version 3.16, function force_sig_info

To overwrite the location X in the kernel. What needed is a fake task_struct t which contains sighand field, so that:

X = address of t→sighand→action[sig-1].sa.sa_handler.

The necessary flows is:

1. Prepare FAKE_CURRENT so that t->sighand->siglock points to a locked spinlock in user mode, at SPINLOCK_USERMODE

2. force_sig_info() will hang in spin_lock_irqsave

   at this moment, another user mode thread running on another CPU will change t->sighand,

   so that t->sighand->action[sig-1].sa.sa_handler is our overwrite target, and then unlock SPINLOCK_USERMODE

3. spin_lock_irqsave will return.

4. force_sig_info() will reload t->sighand, and perform the desired write.

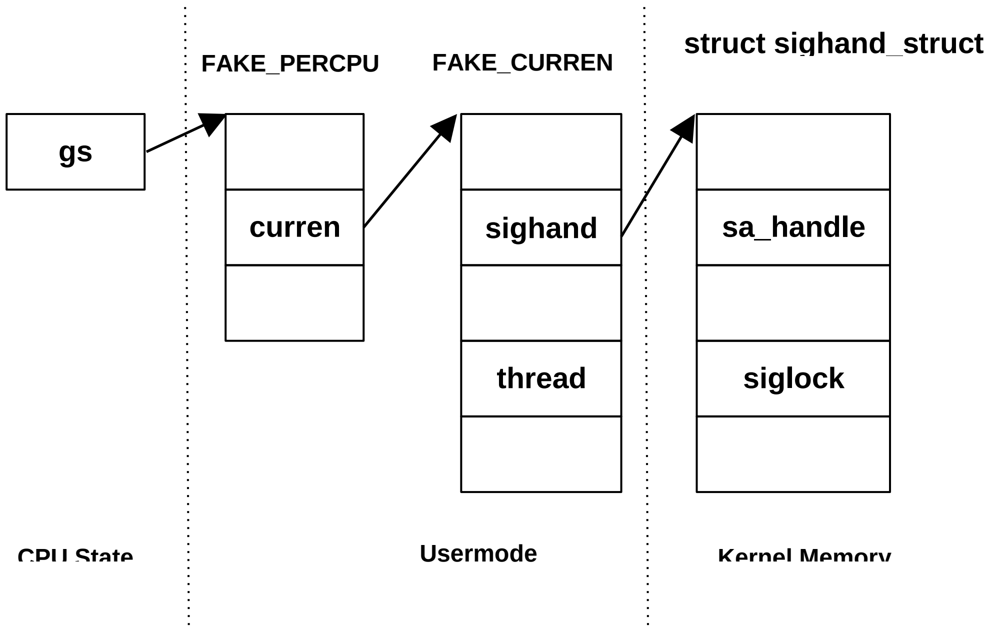


Fig. 10. The fake task_struct

However, as stated at the beginning, the successful possibility of the exploitation for gaining higher privilege depends much on randomness. Even if a kernel function which seems suitable for the exploitation found, there may be some potential aspects which make the exploitation failed.

C. Level 3: Combination with Returned-oriented Programming (ROP) techniques

So far, we know how to overwrite the location X in the kernel by changing 8 bits to 0. How can it be useful? If we succeed to build a pointer to a structure with code pointer, then we can achieve code execution. To do so we can overwrite top bits of an existing pointer to a structure so that the new address is in the user space. If we recreate the structure in the userspace, we can fully control the content and set the code pointers to specific addresses.

A researcher who worked on the PoC chose the **proc_root** variable. It's a structure of type **proc_dir_entry.** This structure represent an entry in the /proc directory and more specifically, the proc_root variable represent the /proc directory. This is exactly the kind of pointer we need because each time something try to access a file in the /proc directory, the starting from proc_root the subdir pointers are follow until the file of the file we were looking for matches. Then, pointers from **proc_iops** are called. Hence, the basic idea is:

- Overwriting the top bits of **proc_root.subdir** to point to a structure **proc_dir_entry** in userspace which name is "A".
- In this structure in user space (that we totally control), we set the **inode_operations** field pointing to user space address FAKE_IOPS

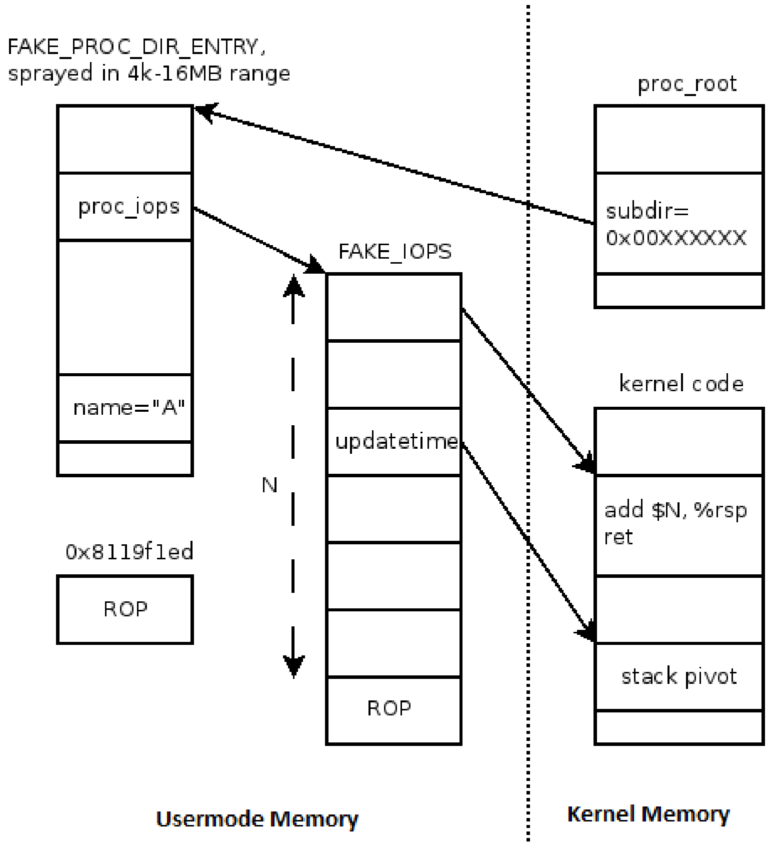


Fig. 11. The fake task_struct when combination with ROP technique

Then as soon as our process call open("/proc/A", …), pointers from FAKE_IOPS (which we control) will be called. Then we will use the stack pivoting technique to chain ROP gadgets: we will make the stack pointer pointing to a buffer we control. The ROP chain will clear the SMEP bit from the cr4 register and then run the shellcode.

II.3 Mitigation Techniques

A. Patching Linux kernel

As explained in previous sections, the vulnerability arises when there is an #SS Exception raised by IRET. When that happens, the #SS exception handler is called. In the vulnerable kernels, the stack_segment interrupt is handled in a "paranoid" way instead of the normal "non-paranoid" way, as can be seen from a snippet of the **kernel/entry_64.S** code:

```
idtentry stack_segment do_stack_segment has_error_code=1 paranoid=1
#ifdef CONFIG_XEN
idtentry xen_debug do_debug has_error_code=0
idtentry xen_int3 do_int3 has_error_code=0
idtentry xen_stack_segment do_stack_segment has_error_code=1
#endif
idtentry general_protection do_general_protection has_error_code=1
```

We can see that the general_protection is non-paranoid while stack_segment is paranoid. Here, being "paranoid" means the kernel will always check the MSR_GS_BASE value to see if it is in kernel space or not, so as to do a SWAPGS if necessary. However, in the case of the #SS exception raised by IRET, this results in an extra SWAPGS.

As such, we just need to make this IDT Entry for stack_segment interrupt non-paranoid. This also means that there is no need for an entry for stack segment fault on the Interrupt Stack Table (IST) which is used for processing "paranoid" IDT entries. Hence, the patch just involves modifying any code related to make the IDT Entry for stack_segment interrupt non-paranoid and remove the entry for stack segment fault on the IDT. Also, we do not need the do_stack_segment function in **kernel/traps.c** now that the interrupt is non-paranoid.

B. SMAP and UDEREF

Supervisor Mode Access Prevention **(**SMAP) is a CPU-based security mechanism in Intel processors and is enabled by default in Linux if available. SMAP prevents unintended supervisor mode accesses to data on user pages. If the SMAP bit is set in the CR4 register, explicit supervisor mode data accesses to user mode pages are allowed if and only if this bit is 1. In other words, SMAP will prevent unintended data accesses to user space memory, but care must be taken because it has to be disabled/enabled around legitimate access functions in the kernel, for example, copy_to_user(), copy_from_user() functions.

The user mode memory accessed in unintended ways by kernel mode will be prohibited , meaning that the attacker controlled pointers can no longer target user mode memory directly. Moreover, simple kernel bugs such as NULL pointer based dereferences will just trigger a SMAP access violation (page fault , #PF) instead of letting the attacker take over kernel data flow. In the our exploitation, the memory access dereferenced by NULL pointer is located at user mode address, the kernel code cannot write/read the crafted data on that memory address when SMAP is active.

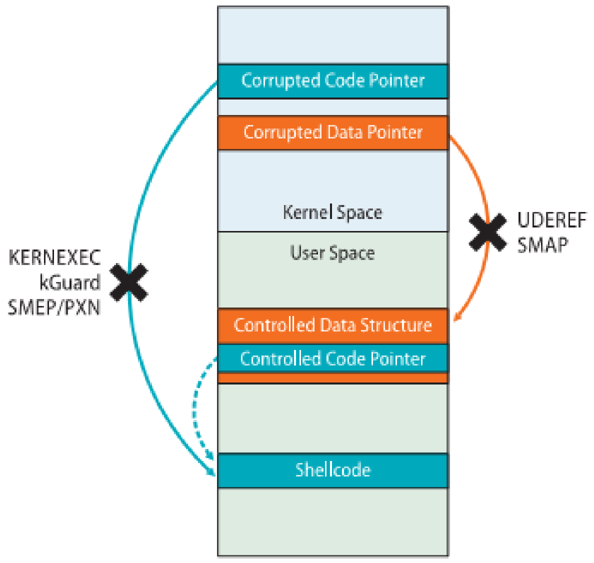


Fig. 12. SMAP/UDEREF prevention mechanism

UDEREF prevents control flow transfers and dereferences from kernel to userspace. In x86, UDEREF rely on memory segmentation to map the kernel space into a 1GB segment that returns a memory fault whenever privileged code tries to dereference pointers to, or fetch instructions from, non-kernel addresses. In x86-64, due to the lack of segmentation support, UDEREF/amd64 remaps user space memory into a different (shadow), non-executable area when execution enters the kernel (and restores it on exit), to prevent user-space dereferences. UDEREF was also ported to ARM architecture.

III. Our objective and approach

As mentioned in the above section, there are mainly two levels of exploitation for this vulnerability: denial of service (DOS) and gaining root privilege shell.

Our objective is twofold:

- The technique used to gain a root privilege shell requires specific information about the kernel. Currently, there has only been an implementation of the exploit on Fedora 20, kernel version 3.11.10-301.fc20.x86_64 done by Adam Zabrocki, Senior Security Engineer at Microsoft. The author also noted that this was the only version where the exploit worked reliably. Our first objective is to investigate how this exploit will work on other Linux kernel versions.
- Investigate how the exploits might work in Android devices.

To elaborate more on the first objective, it will basically involve:

- Finding a method to find the necessary information on a Linux kernel:
    - The appropriate sizes and offsets in the task_struct, signal_struct, signhand_struct
    - The memory address of the proc_root.subdir structure

    - The addresses of appropriate ROP gadgets to use in the exploit.

- Test on different different linux kernels to see if it works.

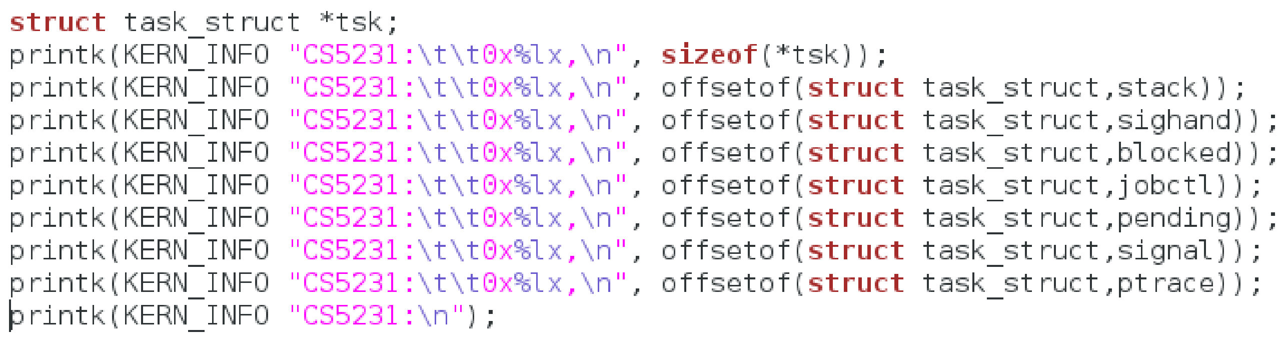
To find the sizes and offsets of the structures, we wrote a kernel module to print out the values.
As for finding out the addresses of the proc_root.subdir, we use information in /proc/kallsyms as well as employ the kernel module.
Finally for ROP gadgets, we used two different methods: the gdb method and the kernel module method.

As for the second objective, we use Android Emulator to emulate Android devices and use adb to transfer programs and run the exploits.

IV. Implementation

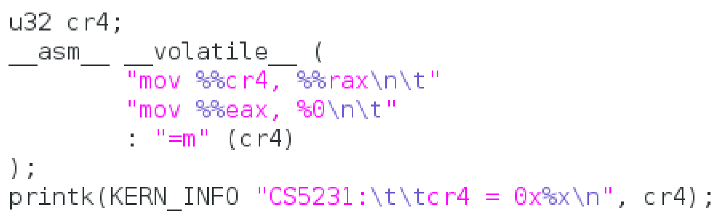
A. Finding sizes, offsets and cr4 value

To find the offsets of various structures, we created a Loadable Kernel Module (LKM) and automatically load it using a bash script. To find the sizes of various structures, we just need to declare them and find the size using the `sizeof` operator. To find the offset of a structure inside a structure, we just need to use the function `offsetof(type, member).` Below is a snippet of the LKM's code:



```
struct task_struct *tsk;
printk(KERN_INFO "CS5231:\t\t0x%lx,\n", sizeof(*tsk));
printk(KERN_INFO "CS5231:\t\t0x%lx,\n", offsetof(struct task_struct,stack));
printk(KERN_INFO "CS5231:\t\t0x%lx,\n", offsetof(struct task_struct,sighand));
printk(KERN_INFO "CS5231:\t\t0x%lx,\n", offsetof(struct task_struct,blocked));
printk(KERN_INFO "CS5231:\t\t0x%lx,\n", offsetof(struct task_struct,jobctl));
printk(KERN_INFO "CS5231:\t\t0x%lx,\n", offsetof(struct task_struct,pending));
printk(KERN_INFO "CS5231:\t\t0x%lx,\n", offsetof(struct task_struct,signal));
printk(KERN_INFO "CS5231:\t\t0x%lx,\n", offsetof(struct task_struct,ptrace));
printk(KERN_INFO "CS5231:\n");
```

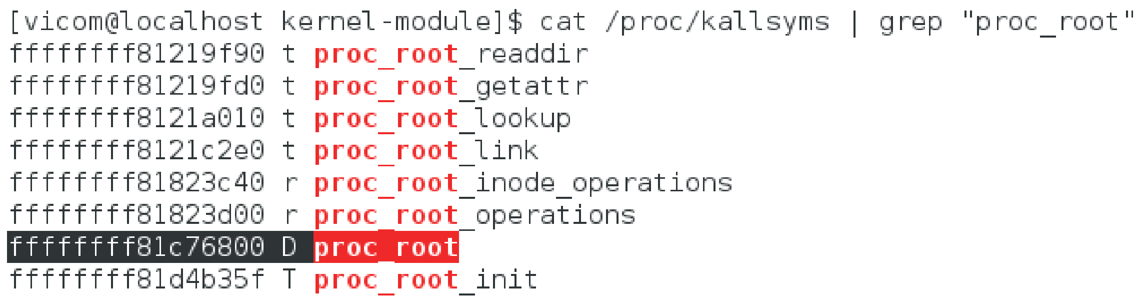
Since the exploitation involves rewriting the cr4 register to disable SMEP, we are also interested in finding out the cr4 register value in the current system. To find out the CR4 register value, we tried different ways and decided to just use a few assembly codes in our LKM to get the CR4 register. The code is as such:



```
u32 cr4;
__asm__ __volatile__ (
        "mov %%cr4, %%rax\n\t"
        "mov %%eax, %0\n\t"
        : "=m" (cr4)
);
printk(KERN_INFO "CS5231:\t\tcr4 = 0x%x\n", cr4);
```

B. Finding addresses of proc_root.subdir

The address of proc_root can be obtained by running "sudo cat /proc/kallsyms | grep proc_root".



```
[vicom@localhost kernel-module]$ cat /proc/kallsyms | grep "proc_root"
ffffffff81219f90 t proc_root_readdir
ffffffff81219fd0 t proc_root_getattr
ffffffff8121a010 t proc_root_lookup
ffffffff8121c2e0 t proc_root_link
ffffffff81823c40 r proc_root_inode_operations
ffffffff81823d00 r proc_root_operations
ffffffff81c76800 D proc_root
ffffffff81d4b35f T proc_root_init
```

Given the address of proc_root, which is basically a struct proc_dir_entry, we just need to add the offset of subdir in struct proc_dir_entry. Again, we can just use the kernel module to print out the offset, similarly to the previous section. In practice, we found that proc_root.subdir is always proc_root+0x40 for all tested kernels.

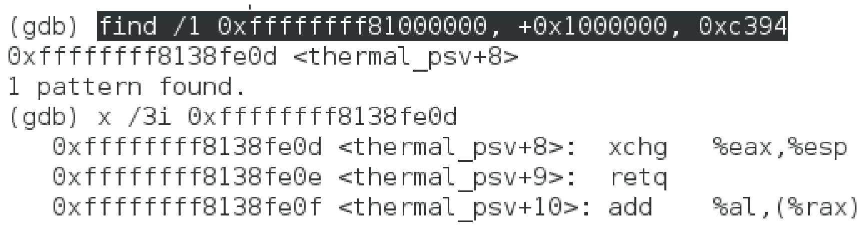
Finding ROP gadgets

- **The gdb method**

First, we get a "kernel image with debug symbols" of the kernel version we are interested in. For Fedora kernels, this can be obtained from installing the kernel debug package kernel-debuginfo-<release>, either using "yum install kernel-debuginfo-<release>" command or downloading and installing the rpm package from koji. For Ubuntu, this can be obtained from installing the linux-image-<release>-dbgsym package.

Then, we use gdb to load the kernel image, located in /usr/lib/debug/lib/modules/<release> for Fedora or /usr/lib/debug/boot for Ubuntu.

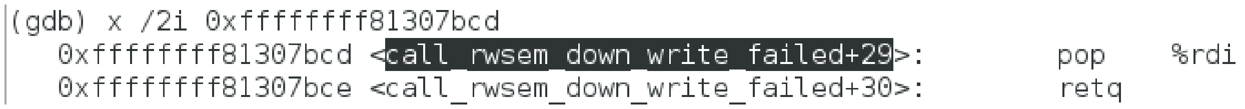
In general, to find a certain specific ROP gadget, we can just search for the exact sequence of bytes representing the targeted instruction sequence using the "find" command. For example, to find the stack pivot sequence, which is "xchg %esp, %eax; ret" or 0xc394 , we can just run this command:



```
(gdb) find /1 0xffffffff81000000, +0x1000000, 0xc394
0xffffffff8138fe0d <thermal_psv+8>
1 pattern found.
(gdb) x /3i 0xffffffff8138fe0d
   0xffffffff8138fe0d <thermal_psv+8>:  xchg   %eax,%esp
   0xffffffff8138fe0e <thermal_psv+9>:  retq
   0xffffffff8138fe0f <thermal_psv+10>: add    %al,(%rax)
```

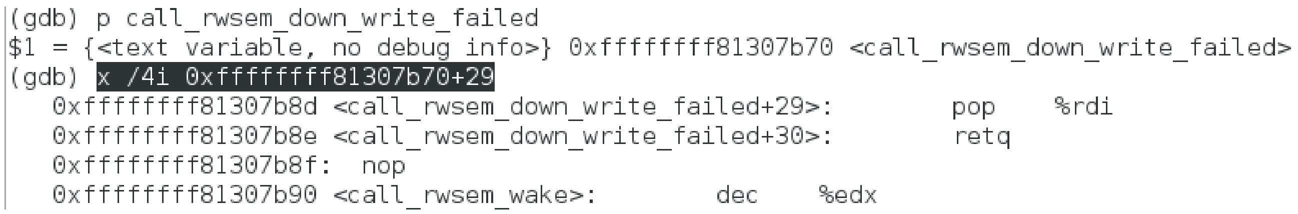
Note that in practice, we find the gadgets starting from the memory address 0xffffffff81000000.

However, there can be a more convenient way to find gadgets. We notice that many functions in the kernel's code remain relatively unchanged. Therefore, if a gadget can be found at `some_function+some_offset` in kernel 3.11.10, we can often find the same gadget at the same function and offset in kernel 3.11.9. For example, from Adam's code we already know that in kernel 3.11.10, we can find "pop %rdi; ret" at 0xffffffff81307bcd. Now at the kernel 3.11.10 image, we find out which function it is:



```
(gdb) x /2i 0xffffffff81307bcd
   0xffffffff81307bcd <call_rwsem_down_write_failed+29>:        pop    %rdi
   0xffffffff81307bce <call_rwsem_down_write_failed+30>:        retq
```

Thus, we can find the address of the same function in kernel 3.11.9 image and take a look at the instructions at the same offsets:



```
(gdb) p call_rwsem_down_write_failed
$1 = {<text variable, no debug info>} 0xffffffff81307b70 <call_rwsem_down_write_failed>
(gdb) x /4i 0xffffffff81307b70+29
   0xffffffff81307b8d <call_rwsem_down_write_failed+29>:        pop    %rdi
   0xffffffff81307b8e <call_rwsem_down_write_failed+30>:        retq
   0xffffffff81307b8f:  nop
   0xffffffff81307b90 <call_rwsem_wake>:        dec    %edx
```

As can be seeen above, we have found the same gadget at the same function in kernel 3.11.9.

- **The kernel module method**

Although the gdb method is straight-forward and reliable, it is relatively slow and troublesome. We would need to find and download a kernel image of the same kernel version, which is usually large (around 300MB). We also need to manually use gdb to find and check the addresses of each of the ROP gadgets. In addition, there could be instances where obtaining a debug kernel image is troublesome too.

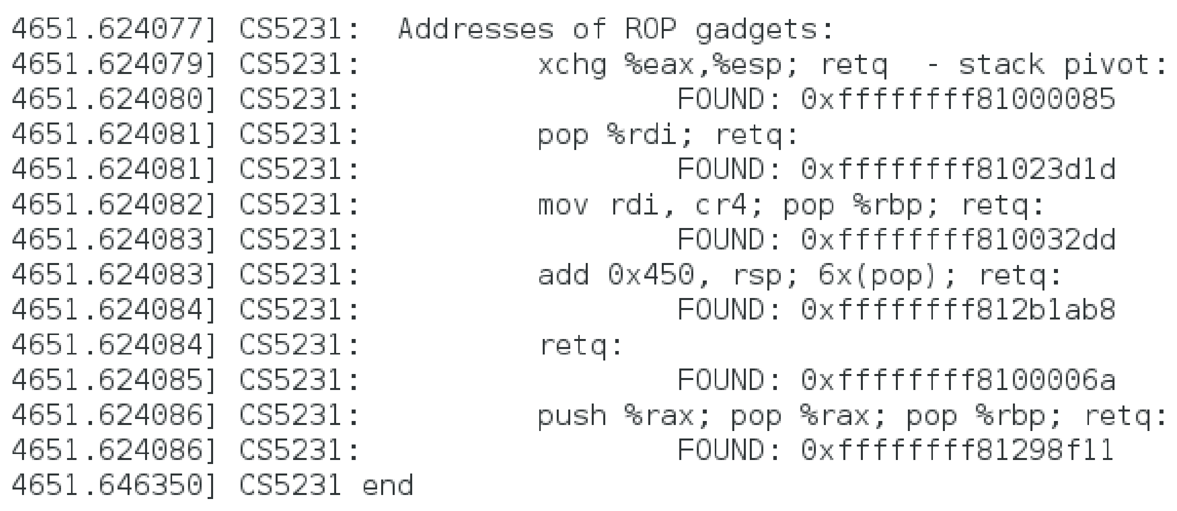
As such, we came up with another approach to get the ROP gadgets automatically: a kernel module to directly scan the kernel memory for ROP gadgets.

To do that, we first need to find out the exact hex values of the ROP gadgets and keep them in an array of arrays of bytes as such:

```
unsigned char gadgets[][17] = {
        {0x94, 0xc3},   //xchg %eax,%esp; retq  -> stack pivot
        {0x5f, 0xc3},   //pop %rdi; retq
        {0x0f, 0x22, 0xe7, 0x5d, 0xc3}, //mov rdi, cr4; pop %rbp; retq
```

With such an array, we just need to search the kernel memory byte by byte and print out locations where the sequence of bytes matches exactly with a gadget. In practice, we start searching from memory location BASE_ADDR = 0xffffffff81000000, and stops the search when we found all the gadgets or we have reached BASE_ADDR + SEARCH_RANGE, where SEARCH_RANGE = 0x1000000. With such settings, it only takes a few seconds and we could find all the ROP gadgets for most of the tested kernels.

This is the result for kernel 3.11.10:

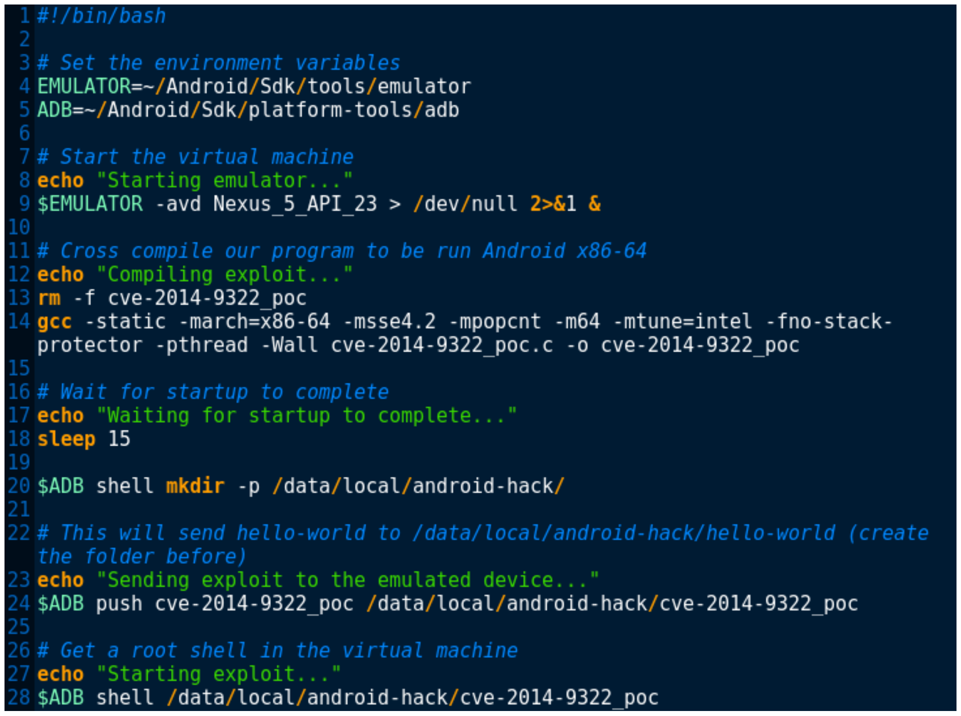


```
4651.624077] CS5231:  Addresses of ROP gadgets:
4651.624079] CS5231:          xchg %eax,%esp; retq  - stack pivot:
4651.624080] CS5231:                  FOUND: 0xffffffff81000085
4651.624081] CS5231:          pop %rdi; retq:
4651.624081] CS5231:                  FOUND: 0xffffffff81023d1d
4651.624082] CS5231:          mov rdi, cr4; pop %rbp; retq:
4651.624083] CS5231:                  FOUND: 0xffffffff810032dd
4651.624083] CS5231:          add 0x450, rsp; 6x(pop); retq:
4651.624084] CS5231:                  FOUND: 0xffffffff812b1ab8
4651.624084] CS5231:          retq:
4651.624085] CS5231:                  FOUND: 0xffffffff8100006a
4651.624086] CS5231:          push %rax; pop %rax; pop %rbp; retq:
4651.624086] CS5231:                  FOUND: 0xffffffff81298f11
4651.646350] CS5231 end
```

Fig. 13. Addresses of ROP gadgets

C. Testing on Android

We have tested the DOS attack on Android 5.1 as well as Android 6.0 and it freezed the device for both of the cases. The detailed commands to compile, transfer and run the binary is as follows:



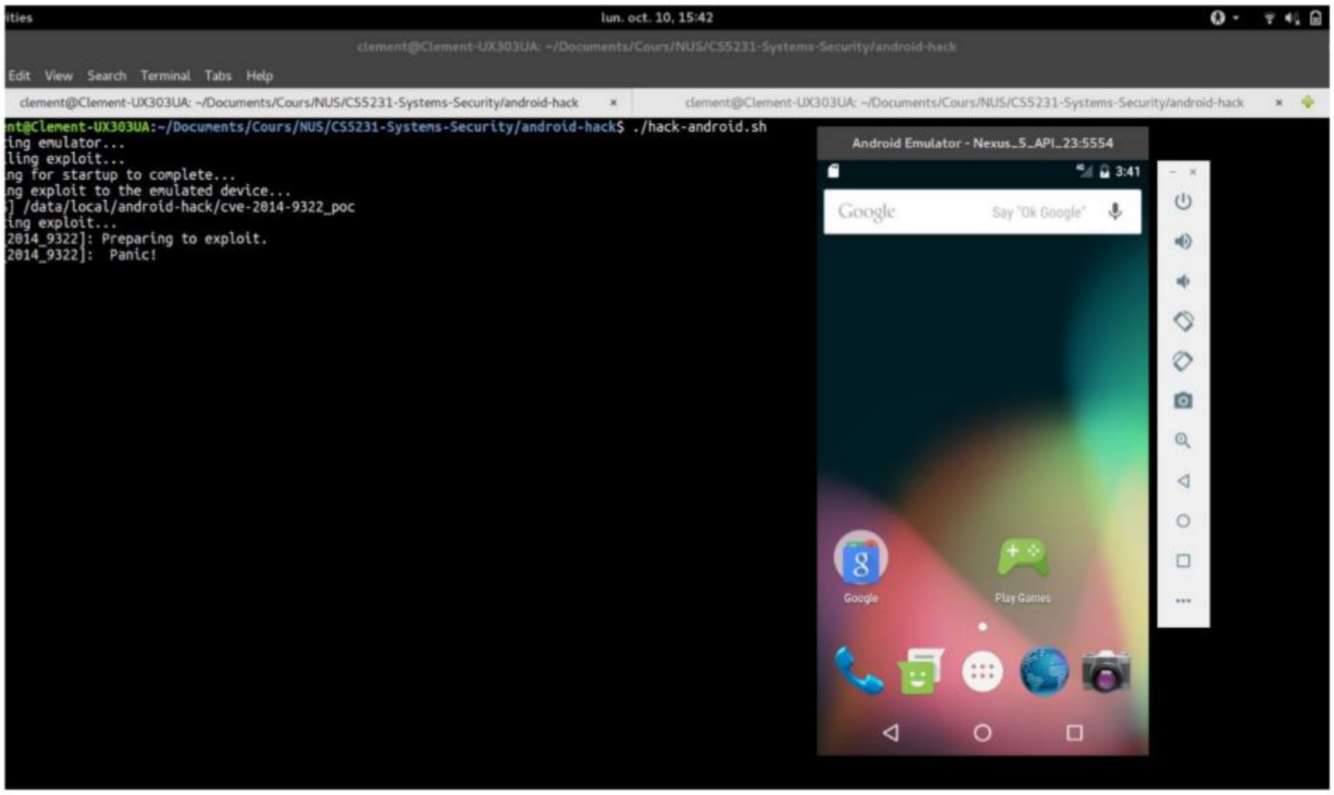
```bash
#!/bin/bash

# Set the environment variables
EMULATOR=~/Android/Sdk/tools/emulator
ADB=~/Android/Sdk/platform-tools/adb

# Start the virtual machine
echo "Starting emulator..."
$EMULATOR -avd Nexus_5_API_23 > /dev/null 2>&1 &

# Cross compile our program to be run Android x86-64
echo "Compiling exploit..."
rm -f cve-2014-9322_poc
gcc -static -march=x86-64 -msse4.2 -mpopcnt -m64 -mtune=intel -fno-stack-protector -pthread -Wall cve-2014-9322_poc.c -o cve-2014-9322_poc

# Wait for startup to complete
echo "Waiting for startup to complete..."
sleep 15

$ADB shell mkdir -p /data/local/android-hack/

# This will send hello-world to /data/local/android-hack/hello-world (create the folder before)
echo "Sending exploit to the emulated device..."
$ADB push cve-2014-9322_poc /data/local/android-hack/cve-2014-9322_poc

# Get a root shell in the virtual machine
echo "Starting exploit..."
$ADB shell /data/local/android-hack/cve-2014-9322_poc
```

Fig. 14. Shell script to compile, transfer and run the PoC on Android

V. Experimental Results

V.1 Denial of Service attack on Android

The DoS attack successfully freezes the Android device for both Android 5.1 and Android 6.0.


ities
lun. oct. 10, 15:42
clement@Clement-UX303UA: ~/Documents/Cours/NUS/CS5231-Systems-Security/android-hack
Edit View Search Terminal Tabs Help
Android Emulator - Nexus_5_API_23:5554
Google
Play Games

```
nt@Clement-UX303UA:~/Documents/Cours/NUS/CS5231-Systems-Security/android-hack$ ./hack-android.sh
ing emulator...
lling exploit...
ng for startup to complete...
ng exploit to the emulated device...
] /data/local/android-hack/cve-2014-9322_poc
ing exploit...
2014_9322]: Preparing to exploit.
2014_9322]:  Panic!
```

Fig. 15. Kernel panic caused in Android Simulator via adb

Causing kernel panic can be considered very serious since the Integrity and Availability of the system are violated. Attacker who exploit the vulnerability can stop an important service.

V.2 Gaining root shell in different kernels

Other than the kernel which is provided in the code for Fedora kernel 3.11.10, we have tested on 6 other kernels, one of which is a Ubuntu kernel while the other 5 are Fedora kernels. The results can be summarized in the table below:

Kernel	**Results**
Fedora 20, kernel 3.11.0	Kernel is fine, successfully gained root shell
Fedora 20, kernel 3.11.7	Kernel is fine, successfully gained root shell
Fedora 20, kernel 3.11.9	Kernel is fine, successfully gained root shell
Fedora 20, kernel 3.11.10 (done previously)	*Kernel is fine, successfully gained root shell*
Fedora 20, kernel 3.12.5	Kernel freezes, no root shell
Fedora 20, kernel 3.13.0	Kernel freezes, no root shell

Ubuntu 14.04, kernel 3.13.0	Kernel freezes, no root shell

Whether the exploit works or not also depends on the processors. For processors with Sandy Bridge and later the exploit simply does not work.

V.3 Conclusion

Although the vulnerability is present in both Fedora 20 kernel 3.12.5 and kernel 3.13.0, we did not manage to gain a root privilege shell (while the DOS exploit still works). This could be due to some slight differences in the kernels' behaviour when such a fault happens, leading to some part of Adam Zabrocki's approach either not working or needing some slight changes (for example, in the size of the buffers). This could also be a direction for future works regarding this vulnerability.

Since the gaining root shell exploit did not even work for Fedora 20 kernel 3.12 and above, we did not try compiling and running the exploit on Android devices, whose kernels are even more different. Although it is highly unlikely that it will work in Android with the same code, there might be some different tweaks that could make it work for Android. This is also another possible direction for future works.

VI. Conclusion

The vulnerability caused in handling stack segment exception is serious since it affects the whole CIA triad of exploited system. Although it was discovered nearly 2 years ago, the vulnerability had still not gained much attention as well as is not fully patched in potential vulnerable systems. Our work can be considered as a comprehensive technical report about the concepts of vulnerability as well as the attacking and preventing techniques. We greatly appreciate our instructor, Professor Liang Zhenkai, for his invaluable explanations on the progress of the project.

References

[1] System Security Lecture Notes, NUS AY-1617
[2] Intel 64 and IA-32 Architectures Software Developer's Manual
[3] Linux Kernel Version 3.16. http://lxr.free-electrons.com/source/arch/x86/kernel/traps.c?v=3.16
[4] CVE-2014-9322. https://labs.bromium.com/2015/02/02/exploiting-badiret-vulnerability-cve-2014-9322-linux-kernel-privilege-escalation/
[5] Follow-up on Exploiting "BadIRET" vulnerability, Adam Zabrocki. http://blog.pi3.com.pl/?p=509
[6] Fedora Kernel. http://koji.fedoraproject.org/koji/packageinfo?packageID=8
[7] Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. ret2dir: Rethinking Kernel Isolation. In Proceedings of the 23rd USENIX Conference on Security Symposium, SEC'14, pages 957–972, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-15-7.